

CodeWarrior™

Development Tools

MSL C++ Reference

Revised: 020319

Metrowerks, the Metrowerks insignia, and CodeWarrior are registered trademarks of Metrowerks Corp. in the US and/or other countries. All other trade names, trademarks and registered trademarks are the property of their respective owners.

© Copyright. 2002. Metrowerks Corp. ALL RIGHTS RESERVED.

Metrowerks reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Metrowerks does not assume any liability arising out of the application or use of any product described herein. Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft or automobile navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.

Documentation stored on electronic media may be printed for personal use only. Except for the forgoing, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

How to Contact Metrowerks:

| | |
|---|---|
| Corporate Headquarters | Metrowerks Corporation 9801 Metric Blvd. Austin, TX 78758 U.S.A. |
| World Wide Web | http://www.metrowerks.com |
| Ordering & Technical Support | Voice: (800) 377-5416 Fax: (512) 997-4901 |

Table of Contents

| | |
|---|-----------|
| 1 Introduction | 21 |
| About the MSL C++ Library Reference Manual | 21 |
| 2 The C++ Library | 25 |
| The MSL C++ Library Overview (clause 17) | 25 |
| 17.1 Definitions. | 25 |
| 17.2 Additional Definitions | 29 |
| Multi-Thread Safety | 29 |
| 17.3 Methods of Descriptions. | 30 |
| 17.3.1 Structure of each sub-clause | 31 |
| 17.3.2 Other Conventions | 31 |
| 17.4 Library-wide Requirements | 33 |
| 17.4.1 Library contents and organization | 33 |
| 17.4.2 Using the library | 35 |
| 17.4.3 Constraints on programs. | 35 |
| 17.4.4 Conforming Implementations | 37 |
| 17.4.4.5 Reentrancy | 38 |
| 17.4.4.8 Restrictions On Exception Handling | 38 |
| Features not implemented in MSL C++ | 38 |
| Template Functionality | 38 |
| ANSI/ISO Library Functionality | 38 |
| 3 Language Support Library | 41 |
| The Language Support Library (clause 18) | 41 |
| 18.1 Types | 41 |
| 18.2 Implementation properties. | 42 |
| 18.2.1 Numeric limits | 42 |
| 18.2.2 C Library | 50 |
| 18.3 Start and termination | 51 |
| 18.4 Dynamic Memory Management | 52 |
| 18.4.1 Storage Allocation and Deallocation. | 53 |
| 18.4.1.2 Array Forms | 54 |
| 18.4.1.3 Placement Forms | 54 |
| 18.4.2 Storage Allocation Errors | 55 |
| 18.5 Type identification | 57 |

| | |
|--|-----------|
| 18.5.1 Class <code>type_info</code> | 57 |
| 18.5.2 Class <code>bad_cast</code> | 58 |
| 18.5.3 Class <code>bad_typeid</code> | 59 |
| 18.6 Exception Handling. | 60 |
| 18.6.1 Class <code>Exception</code> | 61 |
| 18.6.2 Violating Exception Specifications. | 62 |
| 18.6.3 Abnormal Termination | 63 |
| 18.6.4 <code>uncaught_exception</code> | 64 |
| 18.7 Other Runtime Support | 64 |
| 4 Diagnostics Library | 67 |
| The Diagnostics library (clause 19) | 67 |
| 19.1 Exception Classes. | 67 |
| 19.1.1 Class <code>Logic_error</code> | 68 |
| 19.1.2 Class <code>domain_error</code> | 68 |
| 19.1.3 Class <code>Invalid_argument</code> | 69 |
| 19.1.4 Class <code>Length_error</code> | 69 |
| 19.1.5 Class <code>Out_of_range</code> | 70 |
| 19.1.6 Class <code>Runtime_error</code> | 70 |
| 19.1.7 Class <code>Range_error</code> | 71 |
| 19.1.8 Class <code>Overflow_error</code> | 71 |
| 19.1.9 Class <code>Underflow_error</code> | 72 |
| 19.2 Assertions | 73 |
| 19.3 Error Numbers. | 73 |
| 5 General Utilities Libraries | 75 |
| The General Utilities Library (clause 20) | 75 |
| 20.1 Requirements | 75 |
| 20.1.1 Equality Comparisons. | 75 |
| 20.1.2 Less Than Comparison | 76 |
| 20.1.3 Copy Construction | 76 |
| 20.1.4 Default Construction | 76 |
| 20.1.5 Allocator Requirements | 76 |
| 20.2 Utility Components. | 78 |
| 20.2.1 Operators | 79 |
| 20.2.2 Pairs | 80 |
| 20.3 Function objects | 81 |

| | |
|---|------------|
| 20.3.1 Base. | 83 |
| 20.3.2 Arithmetic operations | 84 |
| 20.3.3 Comparisons. | 85 |
| 20.3.4 Logical operations | 87 |
| 20.3.5 Negators | 87 |
| 20.3.6 Binders | 89 |
| 20.3.7 Adaptors for Pointers to Functions | 91 |
| 20.3.8 Adaptors for Pointers to Members | 92 |
| 20.4 Memory. | 97 |
| 20.4.1 The default allocator | 98 |
| 20.4.1.1 allocator members. | 99 |
| 20.4.1.2 allocator globals. | 100 |
| 20.4.2 Raw storage iterator. | 100 |
| 20.4.3 Temporary buffers | 102 |
| 20.4.4 Specialized Algorithms | 102 |
| 20.4.5 Template Class Auto_ptr. | 103 |
| 20.4.5.2 Auto_ptr Members | 107 |
| 20.4.5.3 auto_ptr conversions | 108 |
| 20.4.6 C Library | 109 |
| 20.5 Date and Time | 109 |
| 6 Strings Library | 111 |
| The Strings Library (clause 21) | 111 |
| 21.1 Character traits. | 111 |
| 21.1.1 Character Trait Definitions | 112 |
| 21.1.2 Character Trait Requirements. | 112 |
| 21.1.3 Character Trait Type Definitions | 114 |
| 21.1.4 struct <code>char_traits<T></code> | 115 |
| 21.2 String Classes | 116 |
| 21.3 Class <code>Basic_string</code> | 121 |
| 21.3.1 Constructors and Assignments | 128 |
| 21.3.2 Iterator Support | 130 |
| 21.3.3 Capacity. | 131 |
| 21.3.4 Element Access. | 132 |
| 21.3.5 Modifiers | 133 |
| 21.3.6 String Operations. | 136 |
| 21.3.7 Non-Member Functions and Operators | 141 |

| | |
|---|------------|
| 21.3.7.9 Inserters and extractors | 146 |
| 23.4 Null Terminated Sequence Utilities | 147 |
| 7 Localization Library | 151 |
| The Localization library (clause 22) | 151 |
| Supported Locale Names | 151 |
| Strings and Characters in Locale Data Files | 153 |
| 22.1 Locales | 155 |
| 22.1.1 Class locale | 157 |
| 22.1.2 Locale Globals | 164 |
| 22.1.3 Convenience Interfaces | 165 |
| 22.2 Standard Locale Categories | 166 |
| 22.2.1 The Ctype Category | 166 |
| 22.2.1.5 Template Class Codecvt | 182 |
| 22.2.2 The Numeric Category | 190 |
| 22.2.3 The Numeric Punctuation Facet | 195 |
| 22.2.4 The Collate Category | 202 |
| 22.2.5 The Time Category | 213 |
| 22.2.6 The Monetary Category | 241 |
| 22.2.7 The Message Retrieval Category | 260 |
| 22.2.8 Program-defined Facets | 269 |
| 22.3 C Library Locales | 269 |
| 8 Containers Library | 271 |
| The Containers library (clause 23) | 271 |
| 23.1 Container Requirements | 271 |
| 23.1.1 Sequences Requirements | 272 |
| 23.1.2 Associative Containers Requirements | 274 |
| 23.2 Sequences | 274 |
| 23.2.1 Template Class Deque | 280 |
| 23.2.2 Template Class List | 284 |
| 23.2.3 Container adaptors | 291 |
| 23.2.3.1 Template Class Queue | 291 |
| 23.2.3.2 Template Class Priority_queue | 292 |
| 23.2.3.3 Template Class Stack | 294 |
| 23.2.4 Template Class Vector | 296 |
| 23.2.5 Class Vector<bool> | 301 |

| | |
|---|------------|
| 23.3 Associative Containers | 304 |
| 23.3.1 Template Class Map. | 307 |
| 23.3.2 Template Class Multimap | 313 |
| 23.3.3 Template Class Set | 318 |
| 23.3.4 Template Class Multiset | 321 |
| 23.3.5 Template Class Bitset | 325 |
| Template Class Bitset | 327 |
| 9 Iterators Library | 335 |
| The Iterators library (clause 24) | 335 |
| 24.1 Requirements | 336 |
| 24.1.1 Input Iterators | 336 |
| 24.1.2 Output Iterators | 336 |
| 24.1.3 Forward Iterators | 336 |
| 24.1.4 Bidirectional Iterators. | 337 |
| 24.1.5 Random Access Iterators. | 337 |
| 24.2 Header <iterator> | 337 |
| 24.3 Iterator Primitives | 339 |
| 24.3.1 Iterator Traits. | 339 |
| 24.3.2 Basic Iterator | 340 |
| 24.3.3 Standard Iterator Tags. | 340 |
| 24.4 Predefined Iterators. | 342 |
| 24.4.1 Reverse iterators | 342 |
| 24.4.2 Insert Iterators | 347 |
| 24.5 Stream Iterators | 351 |
| 24.5.1 Template Class Istream_iterator. | 352 |
| 24.5.2 Template Class Ostream_iterator | 354 |
| 24.5.3 Template Class Istreambuf_iterator | 355 |
| 24.5.4 Template Class Ostreambuf_iterator. | 357 |
| 10 Algorithms Library | 361 |
| The Algorithms Library (clause 25) | 361 |
| Header <algorithm> | 361 |
| 25.1 Non-modifying Sequence Operations | 372 |
| 25.2 Mutating Sequence Operators | 377 |
| 25.3 Sorting And Related Operations | 385 |
| 25.4 C library algorithms | 396 |

| | |
|--|------------|
| 11 Numerics Library | 399 |
| The Numerics Library (clause 26) | 399 |
| 26.1 Numeric type requirements | 399 |
| 26.3 Numeric arrays. | 400 |
| 26.3.1 Header <valarray> | 401 |
| 26.3.2 Template Class Valarray | 405 |
| 26.3.3 Valarray Non-member Operations | 411 |
| 26.3.4 Class slice | 416 |
| 26.3.5 Template Class Slice_array | 418 |
| 26.3.6 Class Gslice | 419 |
| 26.3.7 Template Class Gslice_array | 421 |
| 26.3.8 Template Class Mask_array | 423 |
| 26.3.9 Template Class Indirect_array | 425 |
| 26.4 Generalized Numeric Operations | 427 |
| Header <numeric> | 427 |
| 26.5 C Library | 430 |
| <cmath> | 430 |
| <cstdlib> | 431 |
| 12 Complex Class | 433 |
| The Complex Class Library (clause 26.2) | 433 |
| __MSL_CX_LIMITED_RANGE | 433 |
| Header <complex> | 434 |
| 26.2.3 Complex Specializations | 438 |
| Complex Template Class. | 440 |
| Constructors and Assignments. | 440 |
| Complex Member Functions. | 441 |
| Operators | 441 |
| Overloaded Operators and Functions | 443 |
| Complex Operators. | 443 |
| Complex Value Operations | 446 |
| Complex Transcendentals | 447 |
| 13 Input and Output Library | 451 |
| The Input and Output Library (clause 27.1) | 451 |
| Input and Output Library Summary | 451 |
| 27.1 Iostreams requirements | 452 |

| | |
|--|------------|
| 27.1.1 Definitions | 452 |
| 27.1.2 Type requirements | 453 |
| 27.1.2.5 Type SZ_T | 453 |
| 14 Forward Declarations | 455 |
| The Streams and String Forward Declarations (clause 27.2) | 455 |
| Header <iostreamfwd> | 455 |
| Header <stringfwd> | 458 |
| 15 iostream Objects | 461 |
| The Standard Input and Output Stream Library (clause 27.3) | 461 |
| Header <iostream> | 461 |
| Stream Buffering | 462 |
| 27.3.1 Narrow stream objects. | 462 |
| 27.3.2 Wide stream objects | 463 |
| 16 iostreams Base Classes | 465 |
| Input and Output Stream Base Library (clause 27.4) | 465 |
| Header <ios> | 465 |
| Template Class fpos | 467 |
| 27.4.1 Typedef Declarations | 467 |
| 27.4.2 Class ios_base | 468 |
| 27.4.3.1 Typedef Declarations | 470 |
| 27.4.3.1.1 failure | 470 |
| 27.4.3.1.1.1 failure | 471 |
| failure::what. | 471 |
| 27.4.3.1.2 Type fmtflags | 471 |
| 27.4.3.1.3 Type iostate | 472 |
| 27.4.3.1.4 Type openmode | 473 |
| 27.4.3.1.5 Type seekdir | 473 |
| 27.4.3.1.6 Class Init | 474 |
| Class Init Constructor. | 474 |
| 27.4.3.2 ios_base fmtflags state functions | 475 |
| flags | 475 |
| setf. | 477 |
| unsetf. | 478 |
| precision | 479 |
| width. | 480 |

| | |
|---|-----|
| 27.4.3.3 <code>ios_base</code> locale functions | 482 |
| <code>imbue</code> | 482 |
| <code>getloc</code> | 482 |
| 27.4.3.4 <code>ios_base</code> storage function. | 482 |
| <code>xalloc</code> | 482 |
| <code>iword</code> | 482 |
| <code>pword</code> | 483 |
| <code>register_callback</code> | 483 |
| <code>sync_with_stdio</code> | 484 |
| 27.4.3.5 <code>ios_base</code> Constructor. | 484 |
| 27.4.4 Template class <code>basic_ios</code> | 484 |
| 27.4.4.1 <code>basic_ios</code> Constructor | 486 |
| 27.4.4.2 Member Functions | 487 |
| <code>tie</code> | 487 |
| <code>rdbuf</code> | 488 |
| <code>imbue</code> | 490 |
| <code>fill</code> | 490 |
| <code>copyfmt</code> | 491 |
| 27.4.4.3 <code>basic_ios</code> iostate flags functions | 491 |
| <code>operator bool</code> | 491 |
| <code>operator !</code> | 491 |
| <code>rdstate</code> | 491 |
| <code>clear</code> | 493 |
| <code>setstate</code> | 495 |
| <code>good</code> | 496 |
| <code>eof</code> | 496 |
| <code>fail</code> | 497 |
| <code>bad</code> | 498 |
| <code>exceptions</code> | 500 |
| 27.4.5 <code>ios_base</code> manipulators | 500 |
| 27.4.5.1 <code>fmtflags</code> manipulators | 501 |
| 27.4.5.2 <code>adjustfield</code> manipulators | 502 |
| 27.4.5.3 <code>basefield</code> manipulators. | 502 |
| 27.4.5.4 <code>floatfield</code> manipulators. | 503 |
| Overloading Manipulators | 504 |

| | |
|---|------------|
| 17 Stream Buffers | 507 |
| The Stream Buffers Library (clause 27.5) | 507 |
| Header <streambuf> | 507 |
| 27.5.1 Stream buffer requirements. | 507 |
| 27.5.2 Template class <code>basic_streambuf<charT, traits></code> | 508 |
| 27.5.2.1 <code>basic_streambuf</code> Constructor | 511 |
| 27.5.2.2 <code>basic_streambuf</code> Public Member Functions | 511 |
| 27.5.2.2.1 Locales | 512 |
| <code>basic_streambuf::pubimbue</code> | 512 |
| <code>basic_streambuf::getloc</code> | 512 |
| 27.5.2.2.2 Buffer Management and Positioning | 512 |
| <code>basic_streambuf::pubsetbuf</code> | 512 |
| <code>basic_streambuf::pubseekoff</code> | 513 |
| <code>basic_streambuf::pubseekpos</code> | 514 |
| <code>basic_streambuf::pubsync</code> | 516 |
| 27.5.2.2.3 Get Area | 517 |
| <code>basic_streambuf::in_avail</code> | 517 |
| <code>basic_streambuf::snextc</code> | 517 |
| <code>basic_streambuf::sbumpc</code> | 518 |
| <code>basic_streambuf::sgetc</code> | 519 |
| <code>basic_streambuf::sgetn</code> | 520 |
| 27.5.2.2.4 Putback | 520 |
| <code>basic_streambuf::sputback</code> | 520 |
| <code>basic_streambuf::sungetc</code> | 522 |
| 27.5.2.2.5 Put Area | 522 |
| <code>basic_streambuf::sputc</code> | 522 |
| <code>basic_streambuf::sputn</code> | 523 |
| 27.5.2.3 <code>basic_streambuf</code> Protected Member Functions. . . | 523 |
| 27.5.2.3.1 Get Area Access | 523 |
| <code>basic_streambuf::eback</code> | 524 |
| <code>basic_streambuf::gptr</code> | 524 |
| <code>basic_streambuf::egptr</code> | 524 |
| <code>basic_streambuf::gbump</code> | 524 |
| <code>basic_streambuf::setg</code> | 524 |
| 27.5.2.3.2 Put Area Access | 525 |
| <code>basic_streambuf::pbase</code> | 525 |
| <code>basic_streambuf::pptr</code> | 525 |

| | |
|--|-----|
| basic_streambuf::epptr | 525 |
| basic_streambuf::pbump | 525 |
| basic_streambuf::setp | 525 |
| 27.5.2.4 basic_streambuf Virtual Functions | 526 |
| 27.5.2.4.1 Locales | 526 |
| basic_streambuf::imbue | 526 |
| 27.5.2.4.2 Buffer Management and Positioning | 526 |
| basic_streambuf::setbuf | 526 |
| basic_streambuf::seekoff | 527 |
| basic_streambuf::seekpos | 527 |
| basic_streambuf::sync | 527 |
| 27.5.2.4.3 Get Area | 527 |
| basic_streambuf::showmanc | 528 |
| basic_streambuf::xsgetn | 528 |
| basic_streambuf::underflow | 528 |
| basic_streambuf::uflow | 529 |
| 27.5.2.4.4 Putback | 529 |
| basic_streambuf::pbackfail | 529 |
| 27.5.2.4.5 Put Area | 530 |
| basic_streambuf::xsputn | 530 |
| basic_streambuf::overflow | 530 |

| | |
|---|------------|
| 18 Formatting and Manipulators | 533 |
| The Formatting and Manipulators Library (clause 27.6) | 533 |
| Headers | 533 |
| Header <iostream> | 534 |
| Header <ostream> | 534 |
| Header <iomanip> | 535 |
| 27.6.1 Input Streams | 535 |
| 27.6.1.1 Template class basic_istream | 536 |
| 27.6.1.1.1 basic_istream Constructors | 538 |
| 27.6.1.1.2 Class basic_istream::sentry | 539 |
| Class basic_istream::sentry Constructor | 540 |
| sentry::Operator bool | 540 |
| 27.6.1.2 Formatted input functions | 540 |
| 27.6.1.2.1 Common requirements | 540 |
| 27.6.1.2.2 Arithmetic Extractors Operator >> | 541 |

| | |
|--|-----|
| 27.6.1.2.3 basic_istream extractor operator >> | 542 |
| Overloading Extractors: | 544 |
| 27.6.1.3 Unformatted input functions | 547 |
| basic_istream::gcount | 547 |
| basic_istream::get | 549 |
| basic_istream::getline | 551 |
| basic_istream::ignore | 553 |
| basic_istream::peek | 554 |
| basic_istream::read | 555 |
| basic_istream::readsome | 556 |
| basic_istream::putback | 558 |
| basic_istream::unget | 559 |
| basic_istream::sync | 561 |
| basic_istream::tellg | 562 |
| basic_istream::seekg | 562 |
| 27.6.1.4 Standard basic_istream manipulators | 564 |
| basic_ifstream::ws | 564 |
| 27.6.1.4.1 basic_iostream Constructor | 566 |
| 27.6.2 Output streams | 566 |
| 27.6.2.1 Template class basic_ostream | 567 |
| 27.6.2.2 basic_ostream Constructor | 568 |
| 27.6.2.3 Class basic_ostream::sentry | 570 |
| Class basic_ostream::sentry Constructor | 570 |
| sentry::Operator bool | 571 |
| 27.6.2.4 Formatted output functions | 571 |
| 27.6.2.4.1 Common requirements | 571 |
| 27.6.2.4.2 Arithmetic Inserter Operator << | 571 |
| 27.6.2.4.3 basic_ostream::operator<< | 573 |
| Overloading Inserters | 575 |
| 27.6.2.5 Unformatted output functions | 577 |
| basic_ostream::telliop | 577 |
| basic_ostream::seekp | 577 |
| basic_ostream::put | 579 |
| basic_ostream::write | 580 |
| basic_ostream::flush | 581 |
| 27.6.2.6 Standard basic_ostream manipulators | 584 |
| basic_ostream:: endl | 584 |

| | |
|---|------------|
| basic_ostream::ends | 584 |
| basic_ostream::flush | 585 |
| 27.6.3 Standard manipulators. | 588 |
| Standard Manipulator Instantiations | 588 |
| resetiosflags | 588 |
| setiosflags | 589 |
| :setbase | 589 |
| setfill | 590 |
| setprecision | 591 |
| setw | 592 |
| Overloaded Manipulator | 593 |
| 19 String Based Streams | 597 |
| The String Based Stream Library (clause 27.7) | 597 |
| Header <sstream> | 597 |
| 27.7.1 Template class basic_stringbuf. | 598 |
| 27.7.1.1 basic_stringbuf constructors | 599 |
| 27.7.1.2 Member functions | 601 |
| basic_stringbuf::str | 601 |
| 27.7.1.3 Overridden virtual functions | 602 |
| basic_stringbuf::underflow | 602 |
| basic_stringbuf::pbackfail | 603 |
| basic_stringbuf::overflow | 603 |
| basic_stringbuf::seekoff | 603 |
| basic_stringbuf::seekpos. | 604 |
| 27.7.2 Template class basic_istringstream. | 604 |
| 27.7.2.1 basic_istringstream constructors. | 605 |
| 27.7.2.2 Member functions. | 607 |
| basic_istringstream::rdbuf | 607 |
| basic_istringstream::str | 608 |
| 27.7.2.3 Class basic_ostringstream. | 609 |
| 27.7.2.4 basic_ostringstream constructors. | 610 |
| 27.7.2.5 Member functions. | 611 |
| basic_ostringstream::rdbuf | 611 |
| basic_ostringstream::str | 613 |
| 27.7.3 Class basic_stringstream | 614 |
| 27.7.3.4 basic_stringstream constructors | 615 |

| | |
|--|------------|
| 27.7.3.5 Member functions | 617 |
| basic_stringstream::rdbuf | 617 |
| basic_stringstream::str | 618 |
| 20 File Based Streams | 621 |
| The File Based Streams Library (clause 27.8) | 621 |
| Header <fstream> | 621 |
| 27.8.1 File streams | 621 |
| 27.8.1.1 Template class basic_filebuf | 622 |
| 27.8.1.2 basic_filebuf Constructors | 624 |
| 27.8.1.3 Member functions | 625 |
| basic_filebuf::is_open | 625 |
| basic_filebuf::open | 625 |
| basic_filebuf::close | 627 |
| 27.8.1.4 Overridden virtual functions | 627 |
| basic_filebuf::showmany | 627 |
| basic_filebuf::underflow | 628 |
| basic_filebuf::pbackfail | 628 |
| basic_filebuf::overflow | 628 |
| basic_filebuf::seekoff | 628 |
| basic_filebuf::seekpos | 629 |
| basic_filebuf::setbuf | 629 |
| basic_filebuf::sync | 629 |
| basic_filebuf::imbue | 629 |
| 27.8.1.5 Template class basic_ifstream | 630 |
| 27.8.1.6 basic_ifstream Constructor | 631 |
| 27.8.1.7 Member functions | 632 |
| basic_ifstream::rdbuf | 632 |
| basic_ifstream::is_open | 633 |
| basic_ifstream::open | 634 |
| basic_ifstream::close | 635 |
| 27.8.1.8 Template class basic_ofstream | 636 |
| 27.8.1.9 basic_ofstream constructor | 637 |
| 27.8.1.10 Member functions | 638 |
| basic_ofstream::rdbuf | 638 |
| basic_ofstream::is_open | 639 |
| basic_ofstream::open | 640 |

| | |
|--|------------|
| basic_ofstream::close | 642 |
| 27.8.1.11 Template class basic_fstream | 642 |
| 27.8.1.12 basic_fstream Constructor | 643 |
| 27.8.1.13 Member Functions | 644 |
| basic_fstream::rdbuf | 644 |
| basic_fstream::is_open | 646 |
| basic_fstream::open. | 646 |
| basic_fstream::close. | 647 |
| 21 C Library Files | 649 |
| The C Library Files (clause 27.9) | 649 |
| <cstdio> Macros | 649 |
| <cstdio> Types. | 649 |
| <cstdio> Functions | 650 |
| 22 Strstream | 651 |
| The Strstream Class Library (Annex D) | 651 |
| Header <strstream>. | 652 |
| Strstreambuf Class | 654 |
| Strstreambuf constructors and Destructors. | 656 |
| Strstreambuf Public Member Functions | 657 |
| freeze | 657 |
| pcount | 658 |
| str | 659 |
| Protected Virtual Member Functions | 660 |
| setbuf. | 660 |
| seekoff | 660 |
| seekpos | 661 |
| underflow. | 661 |
| pbackfail | 661 |
| overflow | 662 |
| Istrstream Class | 662 |
| Constructors and Destructor | 663 |
| Constructors. | 663 |
| Destructor. | 664 |
| Public Member Functions | 664 |
| rdbuf | 664 |

| | |
|--|------------|
| str | 665 |
| Ostrstream Class | 666 |
| Constructors and Destructor | 666 |
| Constructors. | 666 |
| Destructor. | 667 |
| Public Member Functions | 668 |
| freeze | 668 |
| pcount | 669 |
| rdbuf | 670 |
| str | 670 |
| Strstream Class | 670 |
| Strstream Types | 671 |
| Constructors and Destructor | 671 |
| Constructors. | 671 |
| Destructor. | 672 |
| Public Member Functions | 672 |
| freeze | 672 |
| pcount | 672 |
| rdbuf | 672 |
| str | 673 |
| 23 Msl_mutex.h | 675 |
| The Msl_mutex Support Library | 675 |
| Header <msl_mutex.h> | 675 |
| Mutex | 676 |
| Mutex_lock | 678 |
| 24 MSL_Utility | 681 |
| The MSL Utilities Library | 681 |
| The <msl_utlity> Header | 681 |
| Basic Compile-Time Transformations | 682 |
| Type Query | 684 |
| CV Query | 685 |
| Type Classification | 685 |
| POD classification | 688 |
| Miscellaneous | 689 |

| | |
|--|------------|
| 25 MSL C++ Debug Mode | 699 |
| Overview of MSL C++ Debug Mode | 699 |
| Types of Errors Detected | 699 |
| How to Enable Debug Mode. | 699 |
| Debug Mode Implementations | 700 |
| Debug Mode Containers. | 703 |
| deque. | 703 |
| list | 704 |
| string. | 705 |
| vector. | 706 |
| tree-based containers - map, multimap, set, multiset | 707 |
| cdequeue | 707 |
| slist. | 708 |
| hash-based containers - map, multimap, set, multiset | 709 |
| Invariants | 710 |
| 26 Hash Libraries | 713 |
| The Hash Containers Library. | 713 |
| General Hash Issues | 713 |
| Introduction | 713 |
| Namespace Issues | 714 |
| Fully Qualified Reference: | 714 |
| Namespace Alias. | 714 |
| Using Declaration | 715 |
| Using Directive | 716 |
| Compatibility Headers | 716 |
| Constructors. | 717 |
| Iterator Issues | 717 |
| Capacity | 718 |
| Insert For Unique Hashed Containers | 720 |
| Insert For Multi Hashed Containers. | 721 |
| Incompatibility with Previous versions Metrowerks Hash Containers | 723 |
| Hash_set | 723 |
| Introduction. | 724 |
| Old HashSet Headers | 724 |
| Template Parameters | 725 |

| | |
|--|------------|
| Nested Types | 725 |
| Iterator Issues | 726 |
| hash_set | 726 |
| Hash_map | 732 |
| Introduction | 732 |
| Old Hashmap Headers | 733 |
| Template Parameters | 733 |
| Nested Types | 734 |
| Iterator Issues | 735 |
| Hash_fun | 742 |
| 27 Mslconfig | 745 |
| C++ Switches, Flags and Defines | 745 |
| _CSTD | 746 |
| _Inhibit_Container_Optimization | 746 |
| _Inhibit_Optimize_RB_bit | 746 |
| _MSL_DEBUG | 747 |
| _msl_error | 747 |
| _MSL_ARRAY_AUTO_PTR | 747 |
| __MSL_CPP__ | 748 |
| _MSL_EXTENDED_BINDERS | 748 |
| _MSL_EXTENDED_PRECISION_OUTP | 749 |
| _MSL_FORCE_ENABLE_BOOL_SUPPORT | 749 |
| _MSL_FORCE_ENUMS_ALWAYS_INT | 750 |
| _MSL_IMP_EXP | 751 |
| __MSLONGLONG_SUPPORT__ | 752 |
| _MSL_MINIMUM_NAMED_LOCALE | 752 |
| _MSL_MULTITHREAD | 753 |
| _MSL_NO_BOOL | 753 |
| _MSL_NO_CONSOLE_IO | 754 |
| _MSL_NO_CPP_NAMESPACE | 754 |
| _MSL_NO_EXCEPTIONS | 754 |
| _MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG | 754 |
| _MSL_NO_FILE_IO | 755 |
| _MSL_NO_IO | 755 |
| _MSL_NO_LOCALE | 755 |
| _MSL_NO_REFCOUNT_STRING | 756 |

| | |
|---------------------------------------|------------|
| _MSL_NO_VECTOR_BOOL | 756 |
| _MSL_NO_WCHART. | 756 |
| _MSL_NO_WCHART_LANG_SUPPORT | 757 |
| _MSL_NO_WCHART_C_SUPPORT | 757 |
| _MSL_NO_WCHART_CPP_SUPPORT | 757 |
| _MSL_USE_AUTO_PTR_96 | 757 |
| _STD | 757 |
| Index | 759 |

Introduction

This reference manual describes the contents of the C++ standard library and what Metrowerks' library provides for its users. The C++ Standard library provides an extensible framework, and contains components for: language support, diagnostics, general utilities, strings, locales, containers, iterators, algorithms, numerics, and input/output.

About the MSL C++ Library Reference Manual

This section describes each chapter in this manual. The various chapter's layout mirrors that of the ISO (the International Organization for Standardization) C++ Standard.

[“The MSL C++ Library Overview \(clause 17\)” on page 25](#), of this manual describes the language support library that provides components that are required by certain parts of the C++ language, such as memory allocation and exception processing.

[“The Language Support Library \(clause 18\)” on page 41](#), discusses the ANSI/ISO language support library.

[“The Diagnostics library \(clause 19\)” on page 67](#), elaborates on the diagnostics library that provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.

[“The General Utilities Library \(clause 20\)” on page 75](#), discusses the general utilities library, which includes components used by other library elements, such as predefined storage allocator for dynamic storage management.

[“The Strings Library \(clause 21\)” on page 111](#), discusses the strings components provided for manipulating text represented as

sequences of type `char`, sequences of type `wchar_t`, or sequences of any other “*character-like*” type.

[“The Localization library \(clause 22\)” on page 151](#), covers the localization components extend internationalization support for character classification, numeric, monetary, and date/time formatting and parsing among other things.

[“The Containers library \(clause 23\)” on page 271](#), discusses container classes: lists, vectors, stacks, and so forth. These classes provide a C++ program with access to a subset of the most widely used algorithms and data structures.

[“The Iterators library \(clause 24\)” on page 335](#), discusses iterator classes.

[“The Algorithms Library \(clause 25\)” on page 361](#), discusses the algorithms library. This library provides sequence, sorting, and general numerics algorithms.

[“The Numerics Library \(clause 26\)” on page 399](#), discusses the numerics library. It describes numeric arrays, generalized numeric algorithms and facilities included from the ISO C library.

[“The Complex Class Library \(clause 26.2\)” on page 433](#), describes the components for complex number types

[“The Input and Output Library \(clause 27.1\)” on page 451](#), overviews the input and output class libraries.

[“The Streams and String Forward Declarations \(clause 27.2\)” on page 455](#), discusses the input and output streams forward declarations.

[“The Standard Input and Output Stream Library \(clause 27.3\)” on page 461](#), discusses the initialized input and output objects.

[“Input and Output Stream Base Library \(clause 27.4\)” on page 465](#), discusses the `iostream_base` class.

[“The Stream Buffers Library \(clause 27.5\)” on page 507](#), discusses the stream buffer classes.

[“The Formatting and Manipulators Library \(clause 27.6\)” on page 533.](#) discusses the formatting and manipulator classes.

[“The String Based Stream Library \(clause 27.7\)” on page 597.](#) discusses the string based stream classes.

[“The File Based Streams Library \(clause 27.8\)” on page 621.](#) discusses the file based stream classes.

[“The C Library Files \(clause 27.9\)” on page 649.](#) discusses the namespace C Library functions.

[“The Msl_mutex Support Library” on page 675.](#) discusses the mutex classes.

[“The Strstream Class Library \(Annex D\)” on page 651.](#) a non-templatized array based stream class.

[“The MSL Utilities Library” on page 681.](#) utilities used for non standard headers.

[“Overview of MSL C++ Debug Mode” on page 699.](#) describes the Metrowerks Standard C++ library debug mode facilities.

[“The Hash Containers Library” on page 713.](#) nonstandard “hash” libraries.

[“C++ Switches, Flags and Defines” on page 745.](#) is a chapter on the various flags that you can use to create a customized version of the MSL C++ Library

Introduction

About the MSL C++ Library Reference Manual

The C++ Library

This chapter is an introduction to the Metrowerks Standard C++ library.

The MSL C++ Library Overview (clause 17)

This section introduces you to the definitions, conventions, terminology, and other aspects of the MSL C++ library.

The chapter is constructed in the following sub sections and mirrors clause 17 of the ISO (the International Organization for Standardization) C++ Standard :

- [“17.1 Definitions” on page 25](#) standard C++ terminology
- [“17.2 Additional Definitions” on page 29](#) additional terminology
- [“Multi-Thread Safety” on page 29](#) multi-threaded policy
- [“17.3 Methods of Descriptions” on page 30](#) standard conventions
- [“17.4 Library-wide Requirements” on page 33](#) library requirements
- [“Features not implemented in MSL C++” on page 38](#) Standard C++ features not yet implemented in Metrowerks Standard Libraries

17.1 Definitions

This section discusses the meaning of certain terms in the MSL C++ library.

- [“17.1.1 Arbitrary-Positional Stream” on page 26](#)
- [“17.1.2 Character” on page 26](#)
- [“17.1.3 Character Sequences” on page 26](#)

- [“17.1.4 Comparison Function” on page 26](#)
- [“17.1.5 Component” on page 27](#)
- [“17.1.6 Default Behavior” on page 27](#)
- [“17.1.7 Handler Function” on page 27](#)
- [“17.1.8 Iostream Class Templates” on page 27](#)
- [“17.1.9 Modifier Function” on page 27](#)
- [“17.1.10 Object State” on page 27](#)
- [“17.1.11 Narrow-oriented Iostream Classes” on page 27](#)
- [“17.1.12 NTCTS” on page 27](#)
- [“17.1.13 Observer Function” on page 28](#)
- [“17.1.14 Replacement Function” on page 28](#)
- [“17.1.15 Required Behavior” on page 28](#)
- [“17.1.16 Repositional Stream” on page 28](#)
- [“17.1.17 Reserved Function” on page 28](#)
- [“17.1.18 Traits” on page 28](#)
- [“17.1.19 Wide-oriented Iostream Classes” on page 28](#)

17.1.1 Arbitrary-Positional Stream

A stream that can seek to any position within the length of the stream. An arbitrary-positional stream is also a repositional stream

17.1.2 Character

Any object which, when treated sequentially, can represent text. A character can be represented by any type that provides the definitions specified.

17.1.3 Character Sequences

A class or a type used to represent a character. A character container class shall be a `POD` type.

17.1.4 Comparison Function

An operator function for equality or relational operators.

17.1.5 Component

A group of library entities directly related as members, parameters, or return types. For example, a class and a related non-member template function entity would be referred to as a component.

17.1.6 Default Behavior

The specific behavior provided by the implementation, for replacement and handler functions.

17.1.7 Handler Function

A non-reserved function that may be called at various points with a program through supplying a pointer to the function. The definition may be provided by a C++ program.

17.1.8 Iostream Class Templates

Templates that take two template arguments: `charT` and `traits`. `CharT` is a character container class, and `traits` is a structure which defines additional characteristics and functions of the character type.

17.1.9 Modifier Function

A class member function other than constructors, assignment, or destructor, that alters the state of an object of the class.

17.1.10 Object State

The current value of all non-static class members of an object.

17.1.11 Narrow-oriented Iostream Classes

The instantiations of the `iostream` class templates on the character container class. Traditional `iostream` classes are regarded as the narrow-oriented `iostream` classes.

17.1.12 NTCTS

Null Terminated Character Type Sequences. Traditional `char` strings are NTCTS.

17.1.13 Observer Function

A const member function that accesses the state of an object of the class, but does not alter that state.

17.1.14 Replacement Function

A non-reserved C++ function whose definition is provided by a program. Only one definition for such a function is in effect for the duration of the program's execution.

17.1.15 Required Behavior

The behavior for any replacement or handler function definition in the program replacement or handler function. If a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined.

17.1.16 Repositional Stream

A stream that can seek only to a position that was previously encountered.

17.1.17 Reserved Function

A function, specified as part of the C++ Standard Library, that must be defined by the implementation. If a C++ program provides a definition for any reserved function, the results are undefined.

17.1.18 Traits

A class that encapsulates a set of types and functions necessary for template classes and template functions to manipulate objects of types for which they are instantiated.

17.1.19 Wide-oriented `iostream` Classes

The instantiations of the `iostream` class templates on the character container class `wchar_t` and the default value of the traits parameter.

17.2 Additional Definitions

Metrowerks Standard Libraries have additional definitions.

- [.“Multi-Thread Safety” on page 29](#) precautions used with multi-threaded systems.

Multi-Thread Safety

MSL C++ Library is multi-thread safe provided that the operating system supports thread-safe system calls. Library has locks at appropriate places in the code for thread safety. The locks are implemented as a mutex class -- the implementation of which may differ from platform to platform.

This ensures that the library is MT-Safe internally. For example, if a buffer is shared between two string class objects (via an internal refcount), then only one string object will be able to modify the shared buffer at a given time.

Thus the library will work in the presence of multiple threads in the same way as in single thread provided the user does not share objects between threads or locks between accesses to objects that are shared.

MSL C++ Thread Safety Policy

MSL C++ is Level-1 thread safe. That is:

1. It is safe to simultaneously call const and non-const methods from different threads to distinct objects.
2. It is safe to simultaneously call const methods, and methods otherwise guaranteed not to alter the state of an object (or invalidate outstanding references and iterators of a container) from different threads to the same object.
3. It is not safe for different threads to simultaneously access the same object when at least one thread calls non-const methods, or methods that invalidate outstanding references or iterators to the object. The programmer is responsible for using thread synchronization primitives (e.g. mutex) to avoid such situations.

Simultaneous use of allocators such as new and malloc are thread safe.

Simultaneous use of global objects such as cin and cout is not safe. The programmer is responsible for using thread synchronization primitives to avoid such situations. MSL C++ provides an extension to standard C++ (std::mutex) to aid in such code. For example:

```
#include <iostream>
#include <iomanip>
#include <mutex.h>

std::mutex cout_lock;

int main()
{
    cout_lock.lock();
    std::cout << "The number is " <<
    std::setw(5) << 20 << '\n';
    cout_lock.unlock();
}
```

Note that if only one thread is accessing a standard stream then no synchronization is necessary. For example, one could have one thread handling input from cin, and another thread handling output to cout, without worrying about mutex objects.

The thread safety of MSL C++ can be controlled by the flag [“MSL MULTITHREAD” on page 753](#) in <mslconfig>. If you explicitly use std::mutex objects in your code, then they will become empty do-nothing objects when multi-threading is turned off (`_MSL_MULTITHREAD` is undefined). Thus the same source can be used in both single thread and multi-thread projects.

The `_MSL_MULTITHREAD` flag causes some mutex objects to be set up in the library internally to protect data that is not obviously shared..

See [“MSL MULTITHREAD” on page 753](#) for a fuller description.

17.3 Methods of Descriptions

Conventions used to describe the C++ Standard Library.

17.3.1 Structure of each sub-clause

This document follows the Standard C++ convention and numbers the library chapters by the sub-clause numbers.

Table 2.1 Chapter Descriptions

| Chapter | Description | Chapter | Description |
|----------------|--------------------|----------------|--------------------|
| 18 | Language Support | 23 | Containers |
| 19 | Diagnostics | 24 | Iterators |
| 20 | General utilities | 25 | Algorithms |
| 21 | Strings | 26 | Numerics |
| 22 | Localizations | 27 | Input/Output |

17.3.1.1 Summary

The Metrowerks Standard Library descriptions include a short description, notes, remarks, cross references and examples of usage.

17.3.2 Other Conventions

Some other terminology and conventions used in this reference are:

17.3.2.1.1 Character sequences

A letter is any of the 26 lowercase or 26 uppercase letters

The decimal-point character is represented by a period, ‘.’

A character sequence is an array object of the types `char`, `unsigned char`, or `signed char`.

A character sequence can be designated by a pointer value `S` that points to its first element.

17.3.2.1.3.1 Byte strings

A null-terminated byte string, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the terminating null character).

The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.

The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.

A static NTBS is an NTBS with static storage duration.

17.3.2.1.3.2 Multibyte strings

A null-terminated multibyte string, or NTMBS, is an NTBS that consists of multibyte characters,

A static NTMBS is an NTMBS with static storage duration.

17.3.2.1.3.3 Wide-character sequences

A wide-character sequence is an array object of type `wchar_t`

A wide character sequence can be designated by a pointer value that designates its first element.

A null-terminated wide-character string, or NTWCS, is a wide-character sequence whose highest addressed element has the value zero.

The length of an NTWCS is the number of elements that precede the terminating null wide character.

An empty NTWCS has a length of zero.

The value of an NTWCS is the sequence of values of the elements up to and including the terminating null character.

A static NTWCS is an NTWCS with static storage duration.

17.3.2.2 Functions within classes

Copy constructors, assignment operators, (non-virtual) destructors or virtual destructors that can be generated by default may not be described

17.3.2.3 Private members

To simplify understanding, where objects of certain types are required by the external specifications of their classes to store data. The declarations for such member objects are enclosed in a comment that ends with `exposition only`, as in:

```
// streambuf* sb; exposition only
```

17.4 Library-wide Requirements

The requirements that apply to the entire C++ Standard library.

- [“17.4.1 Library contents and organization” on page 33](#)
- [“17.4.2 Using the library” on page 35](#)
- [“17.4.3 Constraints on programs” on page 35](#)
- [“17.4.4 Conforming Implementations” on page 37](#)
- [“17.4.4.5 Reentrancy” on page 38](#)

17.4.1 Library contents and organization

The Metrowerks Standard Libraries are organized in the same fashion as the ANSI/ISO C++ Standard.

17.4.1.1 Library Contents

Definitions are provided for Macros, Values, Types, Templates, Classes, Function and, Objects.

All library entities except macros, operator new and operator delete are defined within the namespace std or `namespace` nested within namespace std.

17.4.1.2 Headers

The components of the MSL C++ Library are declared or defined in various headers.

Table 2.2 MSL C++ Library headers:

| C++ | Headers | C++ | Headers |
|--------------------------------|-------------------------------|---------------------------------|------------------------------|
| <code><algorithm></code> | <code><bitset></code> | <code><complex></code> | <code><deque></code> |
| <code><exception></code> | <code><fstream></code> | <code><functional></code> | <code><iomanip></code> |
| <code><ios></code> | <code><iostream></code> | <code><iostream></code> | <code><istream></code> |
| <code><iterator></code> | <code><limits></code> | <code><list></code> | <code><locale></code> |

| | | | |
|----------------|----------------|----------------|---------------|
| <map> | <memory> | <new> | <numeric> |
| <ostream> | <queue> | <set> | <sstream> |
| <stack> | <stdexcept> | <streambuf> | <string> |
| <typeinfo> | <utility> | <valarray> | <vector> |
| C Style | Headers | C Style | Header |
| <cassert> | <cctype> | <cerrno> | <cfloat> |
| <ciso646> | <climits> | <clocale> | <cmath> |
| <csetjmp> | <csignal> | <cstdarg> | <cstddef> |
| <cstdio> | <cstdlib> | <cstring> | <ctime> |
| <cwchar> | <cwctype> | | |

Except as may be noted, the contents of each C style header `cname` shall be the same as that of the corresponding header `name.h`. In the MSL C++ Library the declarations and definitions (except for names which are defined as macros in C) are within namespace scope of the namespace `std`.

| | |
|-------------|--|
| NOTE | The names defined as macros in C include: <code>assert</code> , <code>errno</code> , <code>offsetof</code> , <code>setjmp</code> , <code>va_arg</code> , <code>va_end</code> , and <code>va_start</code> . |
|-------------|--|

17.4.1.3 Freestanding Implementations

A freestanding implementation has an implementation-defined set of headers. This set shall include at least the following headers.

Table 2.3 MSL C++ Freestanding Implementation Headers

| Header | Description |
|-----------|---------------------------|
| <cstddef> | Types |
| <limits> | Implementation properties |
| <cstdlib> | Start and termination |
| <new> | Dynamic memory management |

| Header | Description |
|-------------|-----------------------|
| <typeinfo> | Type identification |
| <exception> | Exception handling |
| <cstdarg> | Other runtime support |

The Metrowerks Standard Library header <cstdlib> includes the functions `abort()`, `atexit()`, and `exit()`.

17.4.2 Using the library

A description of how a C++ program gains access to the facilities of the C++ Standard Library.

17.4.2.1 Headers

A header's contents are made available to a translation unit when it contains the appropriate `#include` preprocessing directive.

A translation unit shall include a header only outside of any external declaration or definition, and shall include the header lexically before the first reference to any of the entities it declares or first defines in that translation unit.

17.4.2.2 Linkage

The Metrowerks Standard C++ Library has external “C++” linkage unless otherwise specified

Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

17.4.3 Constraints on programs

Restrictions on C++ programs that use the facilities of the Metrowerks Standard C++ Library.

17.4.3.1 Reserved Names

Metrowerks Standard Library reserves certain sets of names and function signatures for its implementation.

Names that contain a double underscore (_ _) or begins with an underscore followed by an upper-case letter is reserved to the MSL library for its use.

Names that begin with an underscore are reserved to the library for use as a name in the global namespace.

User code can safely use macros that are all uppercase characters and underscores, except for leading underscores. Library code will either be in namespace std or in namespace Metrowerks. Implementation details in namespace std will be prefixed by a double underscore or an underscore followed by an uppercase character. Implementation details in namespace Metrowerks are nested in a nested namespace, for example:

```
Metrowerks::details.
```

17.4.3.1.3 External Linkage

Each name from the Metrowerks Standard C library declared with external linkage is reserved to the implementation for use as a name with extern “C” linkage, both in namespace std and in the global namespace.

17.4.3.2 Headers

The behavior of any header file with the same name as a Metrowerks Standard Library public or private header is undefined.

17.4.3.3 Derived classes

Virtual member function signatures defined for a base class in the C++ Standard Library may be overridden in a derived class defined in the program.

17.4.3.4 Replacement Functions

If replacement definition occurs prior to the program startup then replacement functions are allowed.

A C++ program may provide the definition for any of eight dynamic memory allocation function signatures declared in header <new>.

```
operator new(size_t)
```

```
operator new(size_t, const std::nothrow_t&)
operator new[](size_t)
operator new[](size_t, const std::nothrow_t&)
operator delete(void*)
operator delete(void*, const std::nothrow_t&)
operator delete[](void*)
operator delete[](void*, const std::nothrow_t&)
```

17.4.3.5 Handler functions

MSL Standard C++ Library provides default versions of the following handler functions:

```
unexpected_handler
terminate_handler
```

A C++ program may install different handler functions during execution, by supplying a pointer to a function defined in the program or the library as an argument to:

```
set_new_handler
set_unexpected
set_terminate
```

17.4.3.6 Other functions

In certain cases the Metrowerks Standard C++ Library depends on components supplied by a C++ program. If these components do not meet their requirements, the behavior is undefined.

17.4.3.7 Function arguments

If a C++ library function is passed incorrect but legal arguments the behavior is undefined.

17.4.4 Conforming Implementations

Metrowerks Standard Library is an ANSI/ISO Conforming implementation as described by the ANSI/ISO Standards in section 17.4.4

17.4.4.5 Reentrancy

In Metrowerks Standard Library, Multi-Threaded safety, as described in the section [“Multi-Thread Safety” on page 29](#) is a driving force in the design of this efficient C++ library

17.4.4.8 Restrictions On Exception Handling

Any of the functions defined in the Metrowerks Standard C++ Library may report a failure by throwing an exception. No destructor operation defined in the Metrowerks Standard C++ Library will throw an exception.

The C Style library functions all have a `throw()` exception-specification. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

The functions `qsort()` and `bsearch()` meet this condition. In particular, they can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc`.

Features not implemented in MSL C++

The following sections of Standard C++ will not be implemented in the MSL C++ product.

- [“Template Functionality” on page 38](#)
- [“ANSI/ISO Library Functionality” on page 38](#)

Template Functionality

There are minimal areas of the Standard library that depend on C++ features that have so far not been implemented.

ANSI/ISO Library Functionality

The following standard library features are not part of the current release, but are being implemented:

- Messages in Locale

The C++ Library

Features not implemented in MSL C++

Language Support Library

This chapter includes the implicit functions and objects generated during the execution of some C++ programs. It also contains the headers for those function, objects and defined types.

The Language Support Library (clause 18)

The chapter is constructed in the following sub sections and mirrors clause 18 of the ISO (the International Organization for Standardization) C++ Standard :

- [“18.1 Types” on page 41](#) covers predefined types
- [“18.2 Implementation properties” on page 42](#) covers implementation defined properties
- [“18.3 Start and termination” on page 51](#) covers functions used for starting and termination of a program
- [“18.4 Dynamic Memory Management” on page 52](#) covers operators used for dynamic allocation and release of memory.
- [“18.5 Type identification” on page 57](#) covers objects and functions used for runtime type identification.
- [“18.6 Exception Handling” on page 60](#) covers objects and functions used for exception handling and errors in exception handling.
- [“18.7 Other Runtime Support” on page 64](#) covers variations from standard C library support functions.

18.1 Types

The header <cstddef> contains the same types and definitions as the standard C stddef.h with the following changes as show in [“Header <cstddef>” on page 42](#).

Table 3.1 Header <cstddef>

| | |
|-----------|---|
| NULL | The macro NULL is an implementation-defined C++ constant value. MSL defines this as 0L. |
| offsetof | This macro accepts a restricted set of type arguments that shall be a POD structure or a POD union. The result of applying the offsetof macro to a field that is a static data member or a function member is undefined. |
| ptrdiff_t | No change from standard C. An signed integral type large enough to hold the difference between two pointers. |
| size_t | No change from standard C. An unsigned integral type large enough to hold the result of the sizeof operator. |

18.2 Implementation properties

The headers <limits>, <climits>, and <cfloat> supply implementation dependent characteristics for fundamental types.

18.2.1 Numeric limits

The numeric_limits component provides a C++ program with information about various properties of the implementation's representation of the fundamental types.

Specializations including floating point and integer types are provided.

The member `is_specialized` shall be true for specializations of numeric_limits.

Members declared static const in the numeric_limits template, specializations are usable as integral constant expressions.

Non-fundamental standard types, do not have specializations.

Listing 3.1 Header <limits>

```
namespace std {
template<class T> class numeric_limits;
enum float_round_style;
```

```
enum float_denorm_style;
template<> class numeric_limits<bool>;
template<> class numeric_limits<char>;
template<> class numeric_limits<signed char>;
template<> class numeric_limits<unsigned char>;
template<> class numeric_limits<wchar_t>;
template<> class numeric_limits<short>;
template<> class numeric_limits<int>;
template<> class numeric_limits<long>;
template<> class numeric_limits<unsigned short>;
template<> class numeric_limits<unsigned int>;
template<> class numeric_limits<unsigned long>;
template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;


---


// Metrowerks extensions


---


template<> class numeric_limits<long long>;
template<> class numeric_limits<unsigned long long>;
}

Template class numeric_limits
namespace std {
template<class T> class numeric_limits {
public:
    static const bool is_specialized = false;
    static T min() throw();
    static T max() throw();
    static const int digits = 0;
    static const int digits10 = 0;
    static const bool is_signed = false;
    static const bool is_integer = false;
    static const bool is_exact = false;
    static const int radix = 0;
    static T epsilon() throw();
    static T round_error() throw();
    static const int min_exponent = 0;
    static const int min_exponent10 = 0;
    static const int max_exponent = 0;
    static const int max_exponent10 = 0;
    static const bool has_infinity = false;
    static const bool has_quiet_NaN = false;
```

```
static const bool has_signaling_NaN = false;
static const float_denorm_style has_denorm = denorm_absent;
static const bool has_denorm_loss = false;
static T infinity() throw();
static T quiet_NaN() throw();
static T signaling_NaN() throw();
static T denorm_min() throw();
static const bool is_iec559 = false;
static const bool is_bounded = false;
static const bool is_modulo = false;
static const bool traps = false;
static const bool tinytess_before = false;
static const float_round_style round_style = round_toward_zero;
};

}
```

18.2.1.2 Numeric_limits Static Members

All static members shall be provided but they do not need to make sense. see 18.2.1.5

is_specialized

The data member for distinguishing specializations. The default value is false.

Prototype static const bool is_specialized = false;

min

The minimum finite value for floating point types with denormalization.

Prototype static T min() throw();

Return The maximum positive normalized value is returned.

max

The minimum finite value for floating point types with denormalization.

Prototype static T max() throw();

Return The maximum positive normalized value is returned.

digits

Designates the number of non-signed digits that can be represented for integral types. The number of radix digits in the mantissa for floating point types

Prototype static const int digits = 0;

is_signed

True if the number is signed.

Prototype static const bool is_signed = false;

is_integer

True if the number is an integer.

Prototype static const bool is_integer = false;

is_exact

True if the number is exact.

Prototype static const bool is_exact = false;

Remarks All integer types are exact, but not all floating point types are exact.

radix

Specifies the base or radix of the exponent of a floating point type or base of an integral type.

Prototype static const int radix = 0;

epsilon

The difference between 1 and the least value greater than 1.

Prototype static T epsilon() throw();

round_error

A function to measure the rounding error.

Prototype static T round_error() throw();

Return The maximum rounding error.

min_exponent

Holds the minimum exponent so that the radix raised to one less than this would be normalized.

Prototype static const int min_exponent;

min_exponent10

Stores the minimum negative exponent that 10 raised to that power would be a normalized floating point type.

Prototype static const int min_exponent10 = 0;

max_exponent

The maximum positive integer so that the radix raised to the power one less than this is representable.

Prototype static const int max_exponent = 0;

max_exponent10

The maximum positive integer so that the 10 raised to this power is representable.

Prototype static const int max_exponent10 = 0;

has_infinity

True if is positive for infinity.

Prototype static const bool has_infinity = false;

has_quiet_NaN

True if the number has a quiet “Not a Number”.

Prototype static const bool has_quiet_NaN = false;

has_signaling_NaN

True if the number is a signaling “Not a Number”.

Prototype static const bool has_signaling_NaN = false;

has_denorm

Distinguishes if the floating point number has the ability to be denormalized.

Prototype static const float_denorm_style
has_denorm = denorm_absent;

Remarks The static variable has_denorm equals denorm_present if the type allows denormalized values or denorm_absent if the type does not and denorm_ineterminate if i is indeterminate

has_denorm_loss

Is true if there is a loss of accuracy because of a denormalization loss.

Prototype static const bool has_denorm_loss = false;

infinity

Determines a positive infinity.

Prototype static T infinity() throw();

Return Returns a positive infinity if available.

quiet_NaN

Determines if there is a quiet “Not a Number”.

Prototype static T quiet_NaN() throw();

Return Returns a quiet “Not a Number” if available.

signaling_NaN

Determines if there is a signaling “Not a Number”.

Prototype static T signaling_NaN() throw();

Return Returns a signaling “Not a Number” if available.

denorm_min

Determines the minimum positive denormalized value.

Prototype `static T denorm_min() throw();`

Return Returns the minimum positive denormalized value.

is_iec559

The values is true if and only if the type adheres to IEC 559 standard

Prototype `static const bool is_iec559 = false;`

is_bounded

The value is true if the set of values representable by the type is finite.

Prototype `static const bool is_bounded = false;`

Remarks All predefined data types are bounded.

is_modulo

This value is true if the type is modulo. A type is modulo if it is possible to add two positive numbers and have a result that wraps around to a third number that is less.

Prototype `static const bool is_modulo = false;`

Remarks This value is generally true for unsigned integral types and false for floating point types.

traps

The value is true if trapping is implemented for the type.

Prototype `static const bool traps = false;`

tinyness_before

This value is true if tinyness is detected before rounding.

Prototype `static const bool tinyness_before = false;`

round_style

This value is the rounding style as a type float_round_style.

Prototype static const float_round_style round_style
 = round_toward_zero;

See Also [“Floating Point Rounding Styles” on page 49](#)

8.2.1.3 Type float_round_style

An enumerated type in std namespace used to determine the characteristics for rounding floating point numbers.

Table 3.2 Floating Point Rounding Styles

| Enumerated Type | Value | Meaning |
|---------------------------|--------------|--------------------------------------|
| round_ineterminate | -1 | The rounding is indeterminable |
| round_toward_zero | 0 | The rounding is toward zero |
| round_to_nearest | 1 | Round is to the nearest value |
| round_toward_infinity | 2 | The rounding is to infinity |
| round_toward_neg_infinity | 3 | The rounding is to negative infinity |

18.2.1.4 Type float_denorm_style

The presence of denormalization is represented by the std namespace enumerated type float_denorm_style.

Table 3.3 Floating Point Denorm Styles

| Enumerated Type | Value | Meaning |
|------------------------|--------------|-----------------------------------|
| denorm_ineterminate | -1 | Denormalization is indeterminable |
| denorm_absent | 0 | Denormalization is absent |
| denorm_present | 1 | Denormalization is present |

18.2.1.5 numeric_limits specializations

All member have specializations but these values are not required to be meaningful. Any value that is not meaningful is set to 0 or false.

18.2.2 C Library

The contents of <climits> are the same as standard C's limits.h and the contents of <cfloat> are the same as standard C's float.h.

Table 3.4 Header <climits>

| | | | |
|-----------|-----------|-----------|------------|
| CHAR_BIT | CHAR_MAX | CHAR_MIN | INT_MAX |
| INT_MIN | LONG_MAX | LONG_MIN | MB_LEN_MAX |
| SCHAR_MAX | SCHAR_MIN | SHRT_MAX | SHRT_MIN |
| UCHAR_MAX | UINT_MAX | ULONG_MAX | USHRT_MAX |

The header <cfloat> is the same as standard C float.h

Table 3.5 Header <cfloat>

| | | |
|-----------------|----------------|--------------|
| DBL_DIG | DBL_EPSILON | DBL_MANT_DIG |
| DBL_MAX | DBL_MAX_10_EXP | DBL_MAX_EXP |
| DBL_MIN | DBL_MIN_10_EXP | DBL_MIN_EXP |
| FLT_DIG | FLT_EPSILON | FLT_MANT_DIG |
| FLT_MAX | FLT_MAX_10_EXP | FLT_MAX_EXP |
| FLT_MIN | FLT_MIN_10_EXP | FLT_MIN_EXP |
| FLT_RADIX | FLT_ROUNDS | LDBL_DIG |
| LDBL_EPSILON | LDBL_MANT_DIG | LDBL_MAX |
| LDBL_MAX_10_EXP | LDBL_MAX_EXP | LDBL_MIN |
| LDBL_MIN_10_EXP | LDBL_MIN_EXP | |

18.3 Start and termination

The header <cstdlib> has the same functionality as the standard C header stdlib.h in regards to start and termination functions except for the functions and macros as described below.

Table 3.6 Start and Termination Differences

| Macro | Value | Meaning |
|--------------|-------|---|
| EXIT_FAILURE | 1 | This macro is used to signify a failed return |
| EXIT_SUCCESS | 0 | This macro is used to signify a successful return |

The return from the `main` function is ignored on the Macintosh operating system and is return using the native event processing method on other operating systems.

abort

Terminates the Program with abnormal termination.

Prototype `abort(void)`

Remarks The program is terminated without executing destructors for objects of automatic or static storage duration and without calling the functions passed to `atexit`.

atexit

The `atexit` function registers functions to be called when [exit](#) is called in normal program termination.

Prototype `extern "C" int atexit(void (* f)(void))`
`extern "C++" int atexit(void (* f)(void))`

Remarks If there is no handler for a thrown exception `terminate` is called
The registration of at least 32 functions is allowed.

4. Functions registered with `atexit` are called in reverse order.
5. A function registered with `atexit` before an object of static storage duration will not be called until the object's destruction.

6. A function registered with atexit after an object of static storage duration is initialized will be called before the object's destruction.

Return The atexit() function returns zero if the registration succeeds, non zero if it fails.

exit

Terminates the program with normal cleanup actions.

Prototype `exit(int status)`

Remarks The function exit() has additional behavior in order:

1. Objects with static storage duration are destroyed and functions registered by calling atexit are called.
2. Objects with static storage duration are destroyed in the reverse order of construction. If main contains no automatic objects control can be transferred to main if an exception thrown is caught in main.
3. Functions registered with atexit are called
4. All open C streams with unwritten buffered data are flushed, closed, including streams associated with cin and cout. All `tmpfile()` files are removed.
5. Control is returned to the host environment.

If status is zero or EXIT_SUCCESS, a successful termination is returned to the host environment.

If status is EXIT_FAILURE, an unsuccessful termination is returned to the host environment.

Otherwise the status returned to the host environment is implementation-defined.

18.4 Dynamic Memory Management

The header `<new>` defines procedures for the management of dynamic allocation and error reporting of dynamic allocation errors.

Listing 3.2 Header <new>

```
namespace std {
class bad_alloc;
struct nothrow_t {};
extern const nothrow_t noexcept;
typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();
}

void* operator new(std::size_t size) throw(std::bad_alloc);
void* operator new(std::size_t size, const std::nothrow_t&) throw();
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
void* operator new[](std::size_t size) throw(std::bad_alloc);
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
void operator delete[](void* ptr) throw();
void operator delete[](void* ptr, const std::nothrow_t&) throw();
void* operator new(std::size_t size, void* ptr) throw();
void* operator new[](std::size_t size, void* ptr) throw();
void operator delete(void* ptr, void*) throw();
void operator delete[](void* ptr, void*) throw();
```

18.4.1 Storage Allocation and Deallocation

This clause covers storage allocation and deallocation functions and error management.

18.4.1.1 Single Object Forms

Dynamic allocation and freeing of single object data types.

operator new

Dynamically allocates signable objects.

Prototype void* operator new
 (std::size_t size) throw(std::bad_alloc);
 void* operator new
 (std::size_t size, const
 std::nothrow_t&) throw();

Remarks The `nothrow` version of `new` returns a `null` pointer on failure.
The normal version throws a `bad_alloc` exception on error.

Return Returns a pointer to the allocated memory.

operator delete

Frees memory allocated with operator `new`.

Prototype `void operator delete(void* ptr) throw();`
`void operator delete(void* ptr, const std::nothrow_t&) throw();`

18.4.1.2 Array Forms

Dynamic allocation and freeing of array based data types.

operator new[]

Used for dynamic allocation or array based data types.

Prototype `void* operator new[]`
 `(std::size_t size) throw(std::bad_alloc);`
`void* operator new[]`
 `(std::size_t size, const std::nothrow_t&) throw();`

Remarks The default operator `new` will throw an exception upon failure. The `nothrow` version will return `NULL` upon failure.

operator delete[]

Operator `delete[]` is used in conjunction with operator `new[]` for array allocations.

Prototype `void operator delete[]`
 `(void* ptr) throw();`
`void operator delete[]`
 `(void* ptr, const std::nothrow_t&) throw();`

18.4.1.3 Placement Forms

Placement operators are reserved and may not be overloaded by a C++ program.

placement operator new

Allocates memory at a specific memory address.

Prototype `void* operator new (std::size_t size, void* ptr)
throw();
void* operator new[] (std::size_t size, void* ptr)
throw();`

placement operator delete

The placement delete operators are used in conjunction with the corresponding placement new operators.

Prototype `void operator delete (void* ptr, void*) throw();
void operator delete[] (void* ptr, void*) throw();`

18.4.2 Storage Allocation Errors

C++ provides for various objects, functions and types for management of allocation errors.

18.4.2.1 Class Bad_alloc

A class used to report a failed memory allocation attempt.

Listing 3.3 Class bad_alloc

```
namespace std {  
class bad_alloc : public exception {  
public:  
    bad_alloc() throw();  
    bad_alloc(const bad_alloc&) throw();  
    bad_alloc& operator=(const bad_alloc&) throw();  
    virtual ~bad_alloc() throw();  
    virtual const char* what() const throw();  
};  
}
```

Constructor

Constructs a `bad_alloc` object.

Prototype `bad_alloc() throw();`
`bad_alloc(const bad_alloc&) throw();`

Assignment Operator

Assigns one `bad_alloc` object to another `bad_alloc` object.

Prototype `bad_alloc& operator=(const bad_alloc&) throw();`

destructor

Destroys the `bad_alloc` object.

Prototype `virtual ~bad_alloc() throw();`

what

An error message describing the allocation exception.

Prototype `virtual const char* what() const throw();`

Return A null terminated byte string “`bad_alloc`”.

type new_handler

The type of a handler function that is called by operator `new` or operator `new[]`.

Prototype `typedef void (*new_handler)();`

If `new` requires more memory allocation, the `new_handler` will:

Allocate more memory and return.

Throw an exception of type `bad_alloc` or `bad_alloc` derived class.

Either call `abort` or `exit`.

set_new_handler

Sets the new handler function.

Prototype `new_handler set_new_handler
(new_handler new_p) throw();`

Return Zero on the first call and the previous `new_handler` upon further calls.

18.5 Type identification

The header <typeinfo> defines three types for type identification and type identification errors.

The three classes are:

- [18.5.1 Class type_info](#)
- [18.5.2 Class bad_cast](#)
- [18.5.3 Class bad_typeid](#)

18.5.1 Class type_info

Class type_info contains functions and operations to obtain information about a type.

Listing 3.4 Class type_info

```
namespace std {
    class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
};
```

operator==

Returns true if types are the same.

Prototype `bool operator==(const type_info& rhs) const;`

Return Returns true if the objects are the type.

operator!=

C.compares for inequality.

Prototype `bool operator!=(const type_info& rhs) const;`

Return Returns true if the objects are not the same type.

before

Is true if this object precedes the argument in collation order.

Prototype `bool before(const type_info& rhs) const;`

Returns Returns true if *this precedes the argument the collation order.

name

Returns the name of the class.

Prototype `const char* name() const;`

Return Returns the type name.

Constructors

Private constructor so copying to prevent copying of this object.

Prototype `type_info(const type_info& rhs);`

Assignment Operator

Private assignment to prevent copying of this object.

Prototype `type_info& operator=(const type_info& rhs);`

18.5.2 Class `bad_cast`

A class for exceptions thrown in runtime casting.

Listing 3.5 Class `bad_cast`

```
namespace std {
class bad_cast : public exception {
public:
bad_cast() throw();
```

```
bad_cast(const bad_cast&) throw();
bad_cast& operator=(const bad_cast&) throw();
virtual ~bad_cast() throw();
virtual const char* what() const throw();
};
```

Constructors

Constructs an object of class `bad_cast`.

Prototype `bad_cast() throw();`
 `bad_cast(const bad_cast&) throw();`

Assignment Operator

Copies an object of class `bad_cast`.

Prototype `bad_cast& operator=(const bad_cast&) throw();`

what

An error message describing the casting exception.

Prototype `virtual const char* what() const throw();`
Return Returns the null terminated byte string "bad_cast".

18.5.3 Class `bad_typeid`

Defines a type used for handling `bad typeid` exceptions.

Listing 3.6 Class `bad_typeid`

```
namespace std {
class bad_typeid : public exception {
public:
bad_typeid() throw();
bad_typeid(const bad_typeid&) throw();
bad_typeid& operator=(const bad_typeid&) throw();
virtual ~bad_typeid() throw();
virtual const char* what() const throw();
```

```
} ;  
}
```

Constructors

Constructs an object of class `bad_typeid`.

Prototype `bad_typeid() throw();`
 `bad_typeid(const bad_typeid&) throw();`

Assignment Operator

Copies a class `bad_typeid` object.

Prototype `bad_typeid& operator=(const bad_typeid&) throw();`

what

An error message describing the typeid exception.

Prototype `virtual const char* what() const throw();`

Return Returns the null terminated byte string "bad_typeid".

18.6 Exception Handling

The header `<exception>` defines types and procedures necessary for the handling of exceptions.

Listing 3.7 Header `<exception>`

```
namespace std {  
class exception;  
class bad_exception;  
typedef void (*unexpected_handler)();  
unexpected_handler set_unexpected(unexpected_handler f) throw();  
void unexpected();  
typedef void (*terminate_handler)();  
terminate_handler set_terminate(terminate_handler f) throw();  
void terminate();  
bool uncaught_exception();  
}
```

18.6.1 Class Exception

A base class for objects thrown as exceptions.

Listing 3.8 Class exception

```
namespace std {
    class exception {
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
```

Constructors

Constructs and object of the exception class.

Prototype `exception() throw();`
 `exception(const exception&) throw();`

Assignment Operator

Copies an object of exception class.

Prototype `exception& operator=(const exception&) throw();`

destructor

Destroys an exception object.

Prototype `virtual ~exception() throw();`

what

An error message describing the exception.

Prototype `virtual const char* what() const throw();`

Return Returns the null terminated byte string "exception".

18.6.2 Violating Exception Specifications

Defines objects used for exception violations.

18.6.2.1 Class `bad_exception`

A type used for information and reporting of a bad exceptions.

Listing 3.9 Class `bad_exception`

```
namespace std {
class bad_exception : public exception {
public:
bad_exception() throw();
bad_exception(const bad_exception&) throw();
bad_exception& operator=(const bad_exception&) throw();
virtual ~bad_exception() throw();
virtual const char* what() const throw();
};
}
```

Constructors

Constructs an object of class `bad_exception`.

Prototype `bad_exception() throw();`
 `bad_exception(const bad_exception&) throw();`

Assignment Operator

Copies an object of class `bad_exception`

Prototype `bad_exception& operator=`
 `(const bad_exception&) throw();`

what

An error message describing the bad exception.

Prototype `virtual const char* what() const throw();`

Return Returns the null terminated byte string "bad_exception".

type unexpected_handler

A type of handler called by the `unexpected` function.

Prototype `typedef void (*unexpected_handler)();`

Remarks The `unexpected_handler` calls `terminate()`.

set_unexpected

Sets the unexpected handler function.

Prototype `unexpected_handler set_unexpected(unexpected_handler f) throw();`

Return Returns the previous `unexpected_handler`.

unexpected

Called when a function ends by an exception not allowed in the specifications.

Prototype `void unexpected();`

Remarks May be called directly by the program.

18.6.3 Abnormal Termination

Types and functions used for abnormal program termination.

type terminate_handler

A type of handler called by the function `terminate` when terminating an exception.

Prototype `typedef void (*terminate_handler)();`

Remarks The `terminate_handler` calls `abort()`.

set_terminate

Sets the function for terminating an exception.

Prototype `terminate_handler set_terminate(terminate_handler f) throw();`

Remarks The `terminate_handler` shall not be a null pointer.

Return The previous `terminate_handler` is returned.

terminate

A function called when exception handling is abandoned.

Prototype `void terminate();`

Remarks Exception handling may be abandoned by the implementation or may be called directly by the program.

18.6.4 uncaught_exception

Determines an uncaught exception

Prototype `bool uncaught_exception();`

Remarks Throwing an exception while `uncaught_exception` is true can result in a call of `terminate`.

Return Returns true if an exception is uncaught.

18.7 Other Runtime Support

The C++ headers `<cstdarg>`, `<csetjmp>`, `<ctime>`, `<csignal>` and `<cstdlib>` contain macros, types and functions that vary from the corresponding standard C headers.

Table 3.7 Header `<cstdarg>`

| | |
|-----------------------|-------------------------------------|
| <code>va_arg</code> | A macro used in C++ Runtime support |
| <code>va_end</code> | A macro used in C++ Runtime support |
| <code>va_start</code> | A macro used in C++ Runtime support |
| <code>va_list</code> | A type used in C++ Runtime support |

If the second parameter of `va_start` is declared with a function, array, or reference type, or with a type for which there is no parameter, the behavior is undefined

Table 3.8 Header <csetjmp>

| | |
|---------|--|
| setjmp | A macro used in C++ Runtime support |
| jmp_buf | A type used in C++ Runtime support |
| longjmp | A function used in C++ Runtime support |

The function longjmp is more restricted than in the standard C implementation.

Table 3.9 Header <ctime>

| | |
|----------------|--|
| CLOCKS_PER_SEC | A macro used in C++ Runtime support |
| clock_t | A type used in C++ Runtime support |
| clock | A function used in C++ Runtime support |

If a signal handler attempts to use exception handling the result is undefined.

Table 3.10 Header <csignal>

| | |
|--------------|--|
| SIGABRT | A macro used in C++ Runtime support |
| SIGILL | A macro used in C++ Runtime support |
| SIGSEGV | A macro used in C++ Runtime support |
| SIG_DFL | A macro used in C++ Runtime support |
| SIG_IGN | A macro used in C++ Runtime support |
| SIGFPE | A macro used in C++ Runtime support |
| SIGINT | A macro used in C++ Runtime support |
| SIGTERM | A macro used in C++ Runtime support |
| SIG_ERR | A macro used in C++ Runtime support |
| sig_atomic_t | A macro used in C++ Runtime support |
| raise | A type used in C++ Runtime support |
| signal | A function used in C++ Runtime support |

All signal handlers should have C linkage.

Table 3.11 Header <cstdlib>

| | |
|--------|--|
| getenv | A function used in C++ Runtime support |
| system | A function used in C++ Runtime support |

Diagnostics Library

This chapter describes objects and facilities used to report error conditions.

The Diagnostics library (clause 19)

The chapter is constructed in the following sub sections and mirrors clause 19 of the ISO (the International Organization for Standardization) C++ Standard :

- [“19.1 Exception Classes” on page 67](#)
- [“19.2 Assertions” on page 73](#)
- [“19.3 Error Numbers” on page 73](#)

19.1 Exception Classes

The library provides for exception classes for use with logic errors and runtime errors. Logic errors in theory can be predicted in advance while runtime errors can not. The header `<stdexcept>` predefines several types of exceptions for C++ error reporting.

Listing 4.1 Header `<stdexcept>`

```
namespace std {  
    class logic_error;  
    class domain_error;  
    class invalid_argument;  
    class length_error;  
    class out_of_range;  
    class runtime_error;  
    class range_error;  
    class overflow_error;
```

```
class underflow_error;  
}
```

19.1.1 Class Logic_error

The `logic_error` class is derived from the [“18.6.1 Class Exception” on page 61](#), and is used for exceptions that are detectable before program execution.

Listing 4.2 Class logic_error

```
namespace std {  
class logic_error : public exception {  
public:  
    explicit logic_error(const string& what_arg);  
};  
}
```

Constructors

Constructs an object of class `logic_error`. Initializes `exception::what` to the `what_arg` argument.

Prototype `logic_error(const string& what_arg);`

19.1.2 Class domain_error

A derived class of logic error the `domain_error` object is used for exceptions of domain errors.

Listing 4.3 Class domain_error

```
namespace std {  
class domain_error : public logic_error {  
public:  
    explicit domain_error(const string& what_arg);  
};  
}
```

Constructors

Constructs an object of `domain_error`. Initializes `exception::what` to the `what_arg` argument

Prototype `domain_error(const string& what_arg);`

19.1.3 Class Invalid_argument

A derived class of `logic_error` the `invalid_argument` is used for exceptions of invalid arguments.

Listing 4.4 Class invalid_argument

```
namespace std {  
class invalid_argument : public logic_error {  
public:  
    explicit invalid_argument(const string& what_arg);  
};  
}
```

Constructors

Constructs an object of class `invalid_argument`. Initializes `exception::what` to the `what_arg` argument

Prototype `invalid_argument(const string& what_arg);`

19.1.4 Class Length_error

A derived class of `logic_error` the `length_error` is use to report exceptions when an object exceeds allowed sizes.

Listing 4.5 Class length_error

```
namespace std {  
class length_error : public logic_error {  
public:  
    explicit length_error(const string& what_arg);  
};  
}
```

Constructors

Constructs an object of class `length_error`. Initializes `exception::what` to the `what_arg` argument

Prototype `length_error(const string& what_arg);`

19.1.5 Class Out_of_range

A derived class of `logic_error` an object of `out_of_range` is used for exceptions for out of range errors.

Listing 4.6 Class out_of_range

```
namespace std {  
class out_of_range : public logic_error {  
public:  
    explicit out_of_range(const string& what_arg);  
};  
}
```

Constructors

Constructs an object of the class `out_of_range`. Initializes `exception::what` to the `what_arg` argument

Prototype `out_of_range(const string& what_arg);`

19.1.6 Class Runtime_error

Derived from the ["18.6.1 Class Exception" on page 61](#), the `runtime_error` object is used to report errors detectable only during runtime.

Listing 4.7 Class runtime_error

```
namespace std {  
class runtime_error : public exception {  
public:  
    explicit runtime_error(const string& what_arg);  
};  
}
```

```
};  
}
```

Constructors

Constructs an object of the class `runtime_error`. Initializes `exception::what` to the `what_arg` argument

Prototype `runtime_error(const string& what_arg);`

19.1.7 Class Range_error

Derived from the `runtime_error` class, an object of `range_error` is used for exceptions due to runtime out of range errors.

Listing 4.8 Class range_error

```
namespace std {  
class range_error : public runtime_error {  
public:  
    explicit range_error(const string& what_arg);  
};  
}
```

Constructors

Constructs an object of the class `range_error`. Initializes `exception::what` to the `what_arg` argument

Prototype `range_error(const string& what_arg);`

19.1.8 Class Overflow_error

The `overflow_error` object is derived from the class `runtime_error` and is used to report arithmetical overflow errors.

Listing 4.9 Class overflow_error

```
namespace std {
class overflow_error : public runtime_error {
public:
explicit overflow_error(const string& what_arg);
};
}
```

Constructors

Constructs an object of the class `overflow_error`. Initializes `exception::what` to the `what_arg` argument

Prototype `overflow_error(const string& what_arg);`

19.1.9 Class Underflow_error

The class `underflow_error` is derived from the class `runtime_error` and is used to report the arithmetical underflow error.

Listing 4.10 Class underflow_error

```
namespace std {
class underflow_error : public runtime_error {
public:
explicit underflow_error(const string& what_arg);
};
}
```

Constructors

Constructs an object of the class `underflow_error`. Initializes `exception::what` to the `what_arg` argument

Prototype `underflow_error(const string& what_arg);`

19.2 Assertions

The header `<cassert>` provides for the `assert` macro and is used the same as the standard C header `assert.h`

19.3 Error Numbers

The header `<cerrno>` provides macros: EDOM ERANGE and `errno` to be used for domain and range errors reported by using the `errno` facility. The `<cerrno>` header is used the same as standard C header `errno.h`

General Utilities Libraries

This clause describes components used by other elements of the Standard C++ library. These components may also be used by C++ programs.

The General Utilities Library (clause 20)

The chapter is constructed in the following sub sections and mirrors clause 20 of the ISO (the International Organization for Standardization) C++ Standard :

- [“20.1 Requirements” on page 75](#)
- [“20.2 Utility Components” on page 78](#)
- [“20.3 Function objects” on page 81](#)
- [“20.4 Memory” on page 97](#)
- [“20.5 Date and Time” on page 109](#)

20.1 Requirements

This section describes the requirements for template arguments, types used to instantiate templates and storage allocators used as general utilities.

20.1.1 Equality Comparisons

The equality comparison operator is required. The `(==)` expression has a `bool` return type and specifies that for `x == y` and `y == z` that `x` will equal `z`. In addition the reciprocal is also true. That is, if `x == y` then `y` equals `x`. Also if `x == y` and `y == z` then `z` will be equal to `x`.

20.1.2 Less Than Comparison

A less than operator is required. The `(<)` expression has a `bool` return type and states that if `x < y` that `x` is less than `y` and that `y` is not less than `x`.

20.1.3 Copy Construction

A copy constructor for the general utilities library has the following requirements:

- If the copy constructor is `TYPE(t)` then the argument must be an equivalent of `TYPE`.
- If the copy constructor is `TYPE(const t)` then the argument must be the equivalent of `const TYPE`.
- `&T`, denotes the address of `T`.
- `&const T`, denotes the address of `const T`.

20.1.4 Default Construction

A default constructor is not necessary. However, some container class members may specify a default constructor as a default argument. In that case when a default constructor is used as a default argument there must be a default constructor defined.

20.1.5 Allocator Requirements

The general utilities library requirements include requirements for allocators. Allocators are objects that contain information about the container. This includes information concerning pointer types, the type of their difference, the size of objects in this allocation, also the memory allocation and deallocation information. All of the standard containers are parameterized in terms of allocators.

The allocator class includes the following members

Table 5.1 Allocator Members

| Expression | Meaning |
|-------------------------|--|
| pointer | A pointer to a type |
| const_pointer | A pointer to a const type |
| reference | A reference of a type |
| const_reference | A reference to a const type |
| value_type | A type identical to the type |
| size_type | An unsigned integer that can represent the largest object in the allocator |
| difference_type | A signed integer that can represent the difference between any two pointers in the allocator |
| rebind | The template member is effectively a typedef of the type to which the allocator is bound |
| address(type) | Returns the address of type |
| address(const type) | Returns the address of the const type |
| allocate(size) | Returns the allocation of size |
| allocate(size, address) | Returns the allocation of size at the address |
| max_size | The largest value that can be passed to allocate |
| Ax == Ay | Returns a bool true if the storage of each allocator can be deallocated by the other |
| Ax != Ay | Returns a bool true if the storage of each allocator can not be deallocated by the other |
| T() | Constructs an instance of type |
| T x(y) | x is constructed with the values of y |

Allocator template parameters must meet additional requirements

- All instances of an allocator are interchangeable and compare equal to each other
- Members must meet the requirements in [“The Typedef Members Requirements” on page 78](#)

Implementation-defined allocators are allowed.

Table 5.2 The Typedef Members Requirements

| Member | Type |
|-----------------|-----------|
| pointer | T* |
| const_pointer | T const* |
| size_type | size_t |
| difference_type | ptrdiff_t |

20.2 Utility Components

This sub-clause contains some basic template functions and classes that are used throughout the rest of the library.

Listing 5.1 Header <utility>

```
namespace std {
namespace rel_ops {
template<class T> bool operator!=(const T&, const T&);
template<class T> bool operator> (const T&, const T&);
template<class T> bool operator<=(const T&, const T&);
template<class T> bool operator>=(const T&, const T&);
}

template <class T1, class T2> struct pair;
template <class T1, class T2>
    bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
    bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
    bool operator!=(const pair<T1,T2>&, const pair<T1,T2>&);
template <class T1, class T2>
```

```
    bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);  
template <class T1, class T2>  
    bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);  
template <class T1, class T2>  
    bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);  
template <class T1, class T2> pair<T1,T2>  
    make_pair(const T1&, const T2&);  
}
```

20.2.1 Operators

The Standard C++ library provides general templated comparison operators that are based on operator== and operator<.

operator!=

This operator determines if the first argument is not equal to the second argument.

Prototype template <class T> bool operator!=(const T& x, const T& y);

operator>

This operator determines if the first argument is less than the second argument.

Prototype template <class T> bool operator>(const T& x, const T& y);

operator<=

This operator determines if the first argument is less than or equal to the second argument.

Prototype template <class T> bool operator<=(const T& x, const T& y);

operator>=

This operator determines if the first argument is greater than or equal to the second argument.

Prototype template <class T> bool operator>=(const T& x, const T& y);

20.2.2 Pairs

The utility library includes support for paired values.

Listing 5.2 Struct Pair

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1& x, const T2& y);
    template<class U, class V> pair(const pair< U, V> & p);
};
```

Constructors

The pair class contains various constructors to fit each pairs needs.

Prototype `pair();`

Initializes its members as with default type constructors.

Prototype `template<class U, class V> pair(const pair< U, V> & p);`

Initializes and does any implicit conversions if necessary.

operator ==

The pair equality operator returns true if each pair argument is equal to the other.

Prototype `template <class T1, class T2>
 bool operator==(const pair<T1, T2>& x,
 const pair<T1, T2>& y);`

operator <

The pair less than operator returns true if the second pair argument is less than the first pair argument.

Prototype `template <class T1, class T2> bool operator<`

```
(const pair<T1, T2>& x,  
 const pair<T1, T2>& y);
```

make_pair

Makes a pair of the two arguments.

Prototype template <class T1, class T2>
 pair<T1, T2> make_pair(const T1& x, const T2& y);

Return Returns a pair of the two arguments.

20.3 Function objects

Function objects have the operator() defined and used for more effective use of the library. When a pointer to a function would normally be passed to an algorithm function the library is specified to accept an object with operator() defined. The use of function objects with function templates increases the power and efficiency of the library

NOTE In order to manipulate function objects that take one or two arguments it is required that their function objects provide the defined types. If the function object takes one argument then argument_type and result_type are defined. If the function object takes two arguments then the first_argument_type, second_argument_type, and result_type must be defined.

Listing 5.3 Header <functional> Synopsis

```
namespace std {  
template <class Arg, class Result> struct unary_function;  
template <class Arg1, class Arg2,  
          class Result> struct binary_function;  
  
template <class T> struct plus;  
template <class T> struct minus;  
template <class T> struct multiplies;  
template <class T> struct divides;  
template <class T> struct modulus;  
template <class T> struct negate;
```

```
template <class T> struct equal_to;
template <class T> struct not_equal_to;
template <class T> struct greater;
template <class T> struct less;
template <class T> struct greater_equal;
template <class T> struct less_equal;

template <class T> struct logical_and;
template <class T> struct logical_or;
template <class T> struct logical_not;

template <class Predicate> struct unary_negate;
template <class Predicate>
    unary_negate<Predicate> not1(const Predicate&);
template <class Predicate> struct binary_negate;
template <class Predicate>
    binary_negate<Predicate> not2(const Predicate&);

template <class Operation> class binder1st;
template <class Operation, class T>
    binder1st<Operation> bind1st(const Operation&, const T&);
template <class Operation> class binder2nd;
template <class Operation, class T>
binder2nd<Operation> bind2nd(const Operation&, const T&);

template <class Arg, class Result> class
pointer_to_unary_function;
template <class Arg, class Result>
    pointer_to_unary_function<Arg,Result>
    ptr_fun(Result (*)(Arg));
template <class Arg1, class Arg2, class Result>
    class pointer_to_binary_function;
template <class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*)(Arg1,Arg2));

template<class S, class T> class mem_fun_t;
template<class S, class T, class A> class mem_funl_t;
template<class S, class T>
    mem_fun_t<S,T> mem_fun(S (T::*f)());
template<class S, class T, class A>
```

```

mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A));
template<class S, class T> class mem_fun_ref_t;
template<class S, class T, class A> class mem_fun1_ref_t;
template<class S, class T>
    mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
template<class S, class T, class A>
    mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A));
template <class S, class T> class const_mem_fun_t;
template <class S, class T, class A> class const_mem_fun1_t;
template <class S, class T>
    const_mem_fun_t<S,T> mem_fun(S (T::*f)() const);
template <class S, class T, class A>
    const_mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A) const);
template <class S, class T> class const_mem_fun_ref_t;
template <class S, class T, class A> class const_mem_fun1_ref_t;
template <class S, class T>
    const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const);
template <class S, class T, class A>
    const_mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A)
const);
}

```

20.3.1 Base

Classes are provided to simplify the `typedef` of the argument and result types.

Listing 5.4 Struct Unary_function

```

template <class Arg, class Result>
    struct unary_function {
        typedef Arg argument_type;
        typedef Result result_type;
    };

```

Listing 5.5 Struct Binary_function

```

template <class Arg1, class Arg2, class Result>
    struct binary_function {
        typedef Arg1 first_argument_type;

```

```
typedef Arg2 second_argument_type;
typedef Result result_type;
};
```

20.3.2 Arithmetic operations

The utility library provides function object classes with operator() defined for the arithmetic operations.

plus

Adds the first and the second and returns that sum.

Prototype `template <class T> struct plus :
binary_function<T,T,T> {
 T operator()(const T& x, const T& y) const;
};`

minus

Subtracts the second from the first and returns the difference.

Prototype `template <class T> struct minus :
binary_function<T,T,T> {
 T operator()(const T& x, const T& y) const;
};`

multiplies

Multiplies the first times the second and returns the resulting value.

Prototype `template <class T> struct multiplies :
binary_function<T,T,T> {
 T operator()(const T& x, const T& y) const;
};`

divides

Divides the first by the second and returns the resulting value.

Prototype `template <class T> struct divides :
binary_function<T,T,T> {
 T operator()(const T& x, const T& y) const;
};`

modulus

Determines the modulus of the first by the second argument and returns the result.

Prototype

```
template <class T> struct modulus :  
binary_function<T,T,T> {  
    T operator()(const T& x, const T& y) const;  
};
```

negate

This function returns the negative value of the argument.

Prototype

```
template <class T> struct negate :  
unary_function<T,T> {  
    T operator()(const T& x) const;  
};
```

20.3.3 Comparisons

The utility library provides function object classes with operator() defined for the comparison operations.

NOTE

For the greater, less, greater_equal and less_equal template classes specializations for pointers yield a total order.

equal_to

Returns true if the first argument is equal to the second argument.

Prototype

```
template <class T> struct equal_to :  
binary_function<T,T,bool> {  
    bool operator()(const T& x, const T& y) const;  
};
```

not_equal_to

Returns true if the first argument is not equal to the second argument.

Prototype

```
template <class T> struct not_equal_to :  
binary_function<T,T,bool> {
```

```
    bool operator()(const T& x, const T& y) const;  
};
```

greater

Returns true if the first argument is greater than the second argument.

Prototype `template <class T> struct greater :
binary_function<T,T,bool> {
 bool operator()(const T& x, const T& y) const;
};`

less

Returns true if the first argument is less than the second argument.

Prototype `template <class T> struct less :
binary_function<T,T,bool> {
 bool operator()(const T& x, const T& y) const;
};`

`5 operator()` returns $x < y$.

greater_equal

Returns true if the first argument is greater than or equal to the second argument.

Prototype `template <class T> struct greater_equal :
binary_function<T,T,bool> {
 bool operator()(const T& x, const T& y) const;
};`

less_equal

Returns true if the first argument is less than or equal to the second argument.

Prototype `template <class T> struct less_equal :
binary_function<T,T,bool> {
 bool operator()(const T& x, const T& y) const;
};`

20.3.4 Logical operations

The utility library provides function object classes with operator() defined for the logical operations.

logical_and

Returns true if the first and the second argument are true.

Prototype `template <class T> struct logical_and :
binary_function<T,T,bool> {
 bool operator()(const T& x, const T& y) const;
};`

logical_or

Returns true if the first or the second argument are true.

Prototype `template <class T> struct logical_or :
binary_function<T,T,bool> {
 bool operator()(const T& x, const T& y) const;
};`

logical_not

Returns true if the argument is zero

Prototype `template <class T> struct logical_not :
unary_function<T,bool> {
 bool operator()(const T& x) const;
};`

20.3.5 Negators

The utility library provides negators `not1` and `not2` that return the complement of the unary or binary predicate. A predicate is an object that takes one or two arguments and returns something convertible to `bool`.

Unary_negate

In the template class `unary_negate` the operator() returns the compliment of the predicate argument.

Listing 5.6 Unary_negate

```
template <class Predicate>
class unary_negate
    public unary_function<typename Predicate::argument_type, bool>
{
public:
    explicit unary_negate(const Predicate& pred);
    bool operator()(const typename Predicate::argument_type& x) const;
};
```

not1

The template function `not1` returns the `unary_predicate` of the predicate argument.

Prototype `template <class Predicate>`
 `unary_negate<Predicate>`
 `not1(const Predicate& pred);`

binary_negate

In the template class `binary_negate` the `operator()` returns the compliment of the predicate arguments.

Listing 5.7 Binary_negate

```
template <class Predicate>
class binary_negate
    public binary_function<typename
Predicate::first_argument_type,
    typename Predicate::second_argument_type, bool> {
public:
    explicit binary_negate(const Predicate& pred);
    bool operator()(const typename Predicate::first_argument_type&
x,
                     const typename Predicate::second_argument_type& y) const;
};
```

not2

The template function `not2` returns the `binary_predicate` of the predicate arguments.

Prototype `template <class Predicate>`
`binary_negate<Predicate>`
`not2(const Predicate& pred);`

20.3.6 Binders

The binders classes, `bind1st` and `bind2nd` take a function object and a value and return a function object constructed out of the function bound to the value.

20.3.6.1 Template class `binder1st`

Listing 5.8 `class binder1st`

```
template <class Operation>
class binder1st
    public unary_function<typename
Operation::second_argument_type,
    typename Operation::result_type> {
protected:
    Operation op;
    typename Operation::first_argument_type value;
public:
    binder1st(const Operation& x,
        const typename Operation::first_argument_type& y);
    typename Operation::result_type operator()
        (const typename Operation::second_argument_type& x) const;
};
```

Remarks The constructor initializes the `Operation op` with `x` and `value` with `y`.

`operator()`

Return Returns `op(value, x)`.

bind1st

Binds the first.

Prototype template <class Operation, class T>
 binder1st<Operation> bind1st(const Operation&
 op, const T& x);

Return binder1st<Operation>(op,
 typename Operation::first_argument_type(x)).

20.3.6.3 Template class binder2nd

Template binder class.

Listing 5.9 class binder2nd

```
template <class Operation>
class binder2nd
{
public unary_function<typename Operation::first_argument_type,
                     typename Operation::result_type> {
protected:
    Operation op;
    typename Operation::second_argument_type value;
public:
    binder2nd(const Operation& x,
              const typename Operation::second_argument_type& y);
    typename Operation::result_type
    operator()(const typename Operation::first_argument_type& x)
const;
};
```

Remarks The constructor initializes op with x and value with y.

operator()

Grouping operator

Prototype operator()(const typename
 Operation::first_argument_type& x) const;

Return returns op(x,value).

20.3.6.4 bind2nd

Binds the second.

Prototype template <class Operation, class T>
 binder2nd<Operation> bind2nd
 (const Operation& op, const T& x);

Return binder2nd<Operation>(op,
 typename Operation::second_argument_type(x)).

20.3.7 Adaptors for Pointers to Functions

Adaptors for pointers to pointers to both unary and binary functions work with adaptors.

Listing 5.10 class pointer_to_unary_function

```
template <class Arg, class Result>
class pointer_to_unary_function :
    public unary_function<Arg, Result> {
public:
    explicit pointer_to_unary_function(Result (* f)(Arg));
    Result operator()(Arg x) const;
};
```

operator()

Prototype Result operator()(Arg x) const;

Return Returns f(x).

pointer_to_unary_function

Prototype template <class Arg, class Result>
 pointer_to_unary_function<Arg, Result>
 ptr_fun(Result (* f)(Arg));

Return pointer_to_unary_function<Arg, Result>(f).

class pointer_to_binary_function

A class for pointer to binary binding.

Listing 5.11 class pointer_to_binary_function

```
template <class Arg1, class Arg2, class Result>
    class pointer_to_binary_function :
public binary_function<Arg1,Arg2,Result> {
public:
    explicit pointer_to_binary_function(Result (* f)(Arg1, Arg2));
    Result operator()(Arg1 x, Arg2 y) const;
};
```

operator()

Grouping operator.

Prototype Result operator()(Arg1 x, Arg2 y) const;

Return returns f(x, y).

pointer_to_binary_function

Prototype template <class Arg1, class Arg2, class Result>
 pointer_to_binary_function<Arg1,Arg2,Result
 ptr_fun(Result (* f)(Arg1, Arg2));

Return pointer_to_binary_function<Arg1,Arg2,Result>(f).

20.3.8 Adaptors for Pointers to Members

Adaptors for pointer members are adaptors that allow you to call member functions for elements within a collection.

class mem_fun_t

An adaptor for pointers to member functions.

Listing 5.12 mem_fun_t

```
template <class S, class T> class mem_fun_t
: public unary_function<T*, S> {
public:
    explicit mem_fun_t(S (T::*p)());
    S operator()(T* p) const;
};
```

Remarks The constructor for `mem_fun_t` calls the member function it is initialized with a given pointer argument and an appropriate additional argument.

class mem_fun1_t

A class for binding a member function.

Listing 5.13 Class Mem_fun1_t

```
template <class S, class T, class A> class mem_fun1_t
: public binary_function<T*, A, S> {
public:
    explicit mem_fun1_t(S (T::*p)(A));
    S operator()(T* p, A x) const;
};
```

Remarks The constructor for `mem_fun1_t` calls the member function it is initialized with given a pointer argument and an appropriate additional argument.

mem_fun_t<>, mem_fun1_t<>

Member to function template functions.

Prototype `template<class S, class T> mem_fun_t<S,T>`
`mem_fun(S (T::*f)());`
`template<class S, class T, class A>`
`mem_fun1_t<S,T,A>`
`mem_fun(S (T::*f)(A));`

Remarks The functions return an object through which X::f can be called given a pointer to an X followed by the argument required for f (if any).

class mem_fun_ref_t

Member to function reference object.

Listing 5.14 class mem_fun_ref_t

```
template <class S, class T> class mem_fun_ref_t
public unary_function<T, S> {
```

```
public:  
    explicit mem_fun_ref_t(S (T::*p)());  
    S operator()(T& p) const;  
};
```

Remarks The function `mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

class mem_fun1_ref_t

Member to function reference object.

Listing 5.15 class mem_fun1_ref_t

```
template <class S, class T, class A> class mem_fun1_ref_t  
    public binary_function<T, A, S> {  
public:  
    explicit mem_fun1_ref_t(S (T::*p)(A));  
    S operator()(T& p, A x) const;  
};
```

Remarks The constructor for `mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

mem_fun_ref_t

Template function for member to reference.

Prototype `template<class S, class T> mem_fun_ref_t<S,T>
 mem_fun_ref(S (T::*f)());
template<class S, class T, class A>
 mem_fun1_ref_t<S,T,A>
 mem_fun_ref(S (T::*f)(A));`

Remarks The template function `mem_fun_ref` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

class const_mem_fun_t

Provides a constant member to function object.

Listing 5.16 class const_mem_fun_t

```
template <class S, class T> class const_mem_fun_t
: public unary_function<T*, S> {
public:
    explicit const_mem_fun_t(S (T::*p)() const);
    S operator()(const T* p) const;
};
```

Remarks The constructor for `const_mem_fun_t` calls the member function it is initialized with given a pointer argument.

class const_mem_fun1_t

Provides a const to member function object.

Listing 5.17 class const_mem_fun1_t

```
template <class S, class T, class A> class const_mem_fun1_t
    public binary_function<T*, A, S> {
public:
    explicit const_mem_fun1_t(S (T::*p)(A) const);
    S operator()(const T* p, A x) const;
};
```

Remarks The constructor for `const_mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

const_mem_fun_t

A constant member to function adaptor.

Prototype `template<class S, class T> const_mem_fun_t<S,T>`
 `mem_fun(S (T::*f)() const);`
 `template<class S, class T, class A>`
 `const_mem_fun1_t<S,T,A>`
 `mem_fun(S (T::*f)(A) const);`

Remarks The template function `mem_fun` returns an object through which X::f can be called given a pointer to an X followed by the argument required for f (if any).

class const_mem_fun_ref_t

A constant member to function class.

Listing 5.18 class const_mem_fun_ref_t

```
template <class S, class T> class const_mem_fun_ref_t
    public unary_function<T, S> {
public:
    explicit const_mem_fun_ref_t(S (T::*p)() const);
    S operator()(const T& p) const;
};
```

Remarks The constructor for `const_mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

class const_mem_fun1_ref_t<>

A constant member to function reference adaptor object.

Listing 5.19 class const_mem_fun1_ref_t

```
template <class S, class T, class A> class const_mem_fun1_ref_t
    public binary_function<T, A, S> {
public:
    explicit const_mem_fun1_ref_t(S (T::*p)(A) const);
    S operator()(const T& p, A x) const;
};
```

Remarks The constructor for `const_mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

const_mem_fun_ref_t<>

A constant member function reference adaptor.

Prototype `template<class S, class T>`
 `const_mem_fun_ref_t<S,T>`
 `mem_fun_ref(S (T::*f)() const);`
 `template<class S, class T, class A>`
 `const_mem_fun1_ref_t<S,T,A>`

```
mem_fun_ref(S (T::*f)(A) const);
```

Remarks The template functions `mem_fun_ref` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

20.4 Memory

The header `<memory>` includes functions and classes for the allocation and deallocation of memory.

Listing 5.20 Header `<memory>`

```
namespace std {  
    template <class T> class allocator;  
    template <> class allocator<void>;  
    template <class T, class U>  
        bool operator==(const allocator<T>&, const allocator<U>&)  
            throw();  
    template <class T, class U>  
        bool operator!=(const allocator<T>&, const allocator<U>&)  
            throw();  
  
    template <class OutputIterator, class T>  
        class raw_storage_iterator;  
  
    template <class T>  
        pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);  
    template <class T>  
        void return_temporary_buffer(T* p);  
  
    template <class InputIterator, class ForwardIterator>  
        ForwardIterator uninitialized_copy  
            (InputIterator first, InputIterator last, ForwardIterator  
            result);  
    template <class ForwardIterator, class T> void uninitialized_fill  
        (ForwardIterator first, ForwardIterator last, const T& x);  
    template <class ForwardIterator, class Size, class T>  
        void uninitialized_fill_n  
            (ForwardIterator first, Size n, const T& x);
```

```
template<class X> class auto_ptr;  
}
```

20.4.1 The default allocator

The default allocation classes.

Listing 5.21 class allocator

```
namespace std {  
template <class T> class allocator;  
  
template <> class allocator<void> {  
    public:  
    typedef void* pointer;  
    typedef const void* const_pointer;  
  
    typedef void value_type;  
    template <class U> struct rebind { typedef allocator<U> other; };  
};  
  
template <class T> class allocator {  
    public:  
    typedef size_t size_type;  
    typedef ptrdiff_t difference_type;  
    typedef T* pointer;  
    typedef const T* const_pointer;  
    typedef T& reference;  
    typedef const T& const_reference;  
    typedef T value_type;  
    template <class U> struct rebind { typedef allocator<U> other; };  
    allocator() throw();  
    allocator(const allocator&) throw();  
    template <class U> allocator(const allocator<U>&) throw();  
    ~allocator() throw();  
    pointer address(reference x) const;  
    const_pointer address(const_reference x) const;  
    pointer allocate  
        (size_type, allocator<void>::const_pointer hint = 0);  
    void deallocate(pointer p, size_type n);  
    size_type max_size() const throw();
```

```
void construct(pointer p, const T& val);  
void destroy(pointer p);  
};  
}
```

20.4.1.1 allocator members

Allocator class members.

address

Determine the address of the allocation.

Prototype pointer address(reference x) const;
 const_pointer address(const_reference x) const;

Return Returns the address of the allocation.

allocate

Create an allocation and return a pointer to it.

Prototype pointer allocate(size_type n,
 allocator<void>::const_pointer hint=0);

Return A pointer to the initial element of an array of storage.

Exception Allocate throw a `bad_alloc` exception if the storage cannot be obtained.

deallocate

Remove an allocation from memory.

Prototype void deallocate(pointer p, size_type n);

Deallocates the storage referenced by p.

max_size

Determines the Maximum size for an allocation.

Prototype size_type max_size() const throw();

Return Returns the largest size of memory that may be.

construct

Determines the Maximum size for an allocation.

Prototype `void construct(pointer p, const_reference val);`

Return A pointer to the allocated memory.

destroy

Destroys the memory allocated

Prototype `void destroy(pointer p);`

20.4.1.2 allocator globals

Provides globals operators in memory allocation.

operator==

Equality operator.

Prototype `template <class T1, class T2> bool operator==
 (const allocator<T1>&,
 const allocator<T2>&) throw();`

Return Returns true if the arguments are equal.

operator!=

Inequality operator

Prototype `template <class T1, class T2> bool operator!=
 (const allocator<T1>&,
 const allocator<T2>&) throw();`

Return Returns true if the arguments are not equal.

20.4.2 Raw storage iterator

A means of storing the results of un-initialized memory.

NOTE The formal template parameter OutputIterator is required to have its operator* return an object for which operator& is defined and returns

a pointer to T, and is also required to satisfy the requirements of an output iterator.

Listing 5.22 class raw_storage_iterator

```
namespace std {
template <class OutputIterator, class T> class
raw_storage_iterator
public iterator<output_iterator_tag,void,void,void,void> {
    public:
explicit raw_storage_iterator(OutputIterator x);
raw_storage_iterator<OutputIterator,T>& operator*();
raw_storage_iterator<OutputIterator,T>& operator=
    (const T& element);
raw_storage_iterator<OutputIterator,T>& operator++();
raw_storage_iterator<OutputIterator,T> operator++(int);
} ;
}
```

Constructors

A constructor for the `raw_storage_iterator` class.

Prototype `raw_storage_iterator(OutputIterator x);`

Initializes the iterator.

operator *

A dereference operator.

Prototype `raw_storage_iterator<OutputIterator,T>&`
`operator*();`

Return The dereference operator return `*this`.

operator=

The `raw_storage_iterator` assignment operator.

Prototype `raw_storage_iterator<OutputIterator,T>&`
`operator=(const T& element);`

4 Effects: Constructs a value from element at the location to which the iterator points.

Return A reference to the iterator.

operator++

Post and Pre-increment operators for `raw_storage_iterator`.

Prototype `raw_storage_iterator<OutputIterator, T>&`
`operator++(); // Pre-increment`
`raw_storage_iterator<OutputIterator, T>`
`operator++(int); // Post-increment`

Return Returns the old value of the iterator.

20.4.3 Temporary buffers

Methods for storing and retrieving temporary allocations.

get_temporary_buffer

Retrieves a pointer to store temporary objects.

Prototype `template <class T> pair<T*, ptrdiff_t>`
`get_temporary_buffer(ptrdiff_t n);`

Return An address for the buffer and its size or zero if unsuccessful.

return_temporary_buffer

Deallocation for the `get_temporary_buffer` procedure.

Prototype `template <class T>`
`void return_temporary_buffer(T* p);`

The buffer must have been previously allocated by `get_temporary_buffer`.

20.4.4 Specialized Algorithms

Algorithm necessary to fulfill iterator requirements.

uninitialized_copy

An uninitialized copy.

Prototype `template <class InputIterator,
 class ForwardIterator>
ForwardIterator uninitialized_copy
 (InputIterator first, InputIterator
 last, ForwardIterator result);`

Return Returns a `ForwardIterator` to the result argument.

uninitialized_fill

An uninitialized fill.

Prototype `template <class ForwardIterator, class T>
 void uninitialized_fill
 (ForwardIterator first,
 ForwardIterator last, const T& x);`

uninitialized_fill_n

An uninitialized fill with a size limit.

Prototype `template <class ForwardIterator,
 class Size, class T>
 void uninitialized_fill_n
 (ForwardIterator first, Size n, const T& x);`

20.4.5 Template Class Auto_ptr

The `auto_ptr` class stores a pointer to an object obtained using `new` and deletes that object when it is destroyed. For example when a local allocation goes out of scope.

The template `auto_ptr_ref` holds a reference to an `auto_ptr`, and is used by the `auto_ptr` conversions. This allows `auto_ptr` objects to be passed to and returned from functions.

```
namespace std {  
  
template <class Y> struct auto_ptr_ref {};
```

```
template<class X> class auto_ptr {
    public:
typedef X element_type;

explicit auto_ptr(X* p =0) throw();
auto_ptr(auto_ptr&) throw();
template<class Y> auto_ptr(auto_ptr<Y>&) throw();
auto_ptr& operator=(auto_ptr&) throw();
auto_ptr& operator=(auto_ptr_ref<X> r) throw();
template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
~auto_ptr() throw();

X& operator*() const throw();
X* operator->() const throw();
X* get() const throw();
X* release() throw();
void reset(X* p =0) throw();

auto_ptr(auto_ptr_ref<X>) throw();
template<class Y> operator auto_ptr_ref<Y>() throw();
template<class Y> operator auto_ptr<Y>() throw();
};

}
```

NOTE An `auto_ptr` owns the object it holds a pointer to. When copying an `auto_ptr` the pointer transfers ownership to the destination.

If more than one `auto_ptr` owns the same object at the same time the behavior of the program is undefined.

See the example of using `std::auto_ptr` and extension version for arrays in [Listing 5.23](#)

This extension can be turned off by commenting out `#define _MSL_ARRAY_AUTO_PTR` in `<mslconfig>`. No recompile of the C++ lib is necessary, but do rebuild any precompiled headers when making this change.

The functionality provided by the extended `std::auto_ptr` is very similar to that provided by the newer Metrowerks::alloc_ptr found in `<msl_utility>`.

Listing 5.23 Using Auto_ptr

```
#include <iostream>
#include <memory>

using std::auto_ptr;
using std::_Array;

struct A
{
    A() {std::cout << "construct A\n";}
    virtual ~A() {std::cout << "destruct A\n";}
};

struct B
: A
{
    B() {std::cout << "construct B\n";}
    virtual ~B() {std::cout << "destruct B\n";}
};

auto_ptr<B> source();
void sink_b(auto_ptr<B>);
void sink_a(auto_ptr<A>);

auto_ptr<B, _Array<B> > array_source();
void array_sink(auto_ptr<B, _Array<B> >);

auto_ptr<B>
source()
{
    return auto_ptr<B>(new B);
}

void
sink_b(auto_ptr<B>)
{
}

void
```

General Utilities Libraries

20.4 Memory

```
sink_a(auto_ptr<A>)

{
}

auto_ptr<B, _Array<B> >
array_source()
{
    return auto_ptr<B, _Array<B> >(new B [2]);
}

void
array_sink(auto_ptr<B, _Array<B> >)
{
}

int main()
{
{
    auto_ptr<B> b(new B);
    auto_ptr<B> b2(b);
    b = b2;
    auto_ptr<B> b3(source());
    auto_ptr<A> a(b);
    a = b3;
    b3 = source();
    sink_b(source());
    auto_ptr<A> a2(source());
    a2 = source();
    sink_a(source());
}
{
    auto_ptr<B, _Array<B> > b(new B [2]);
    auto_ptr<B, _Array<B> > b2(b);
    b = b2;
    auto_ptr<B, _Array<B> > b3(array_source());
    b3 = array_source();
    array_sink(array_source());
//    auto_ptr<A, _Array<A> > a(b3); // Should not compile
//    a = b3; // Should not
compile
```

```
}
```

auto_ptr Constructors

Constructs an auto_ptr object.

Prototype `explicit auto_ptr(X* p =0) throw();
auto_ptr(auto_ptr& a) throw();
template<class Y> auto_ptr(auto_ptr<Y>& a)
throw();`

operator =

An auto_ptr assignment operator.

Prototype `template<class Y> auto_ptr& operator=(
 auto_ptr<Y>& a) throw();
auto_ptr& operator=
 (auto_ptr& a) throw();`

Return Returns the this pointer.

destructor

Destroys the auto_ptr object.

Prototype `~auto_ptr() throw();`

20.4.5.2 Auto_ptr Members

Member of the auto_ptr class.

operator*

The de-reference operator.

Prototype `X& operator*() const throw();`

Return Returns what the dereferenced this pointer holds.

operator->

The pointer dereference operator.

| | |
|---|---|
| Prototype | <code>X* operator->() const throw();</code> |
| Return | Returns what the dereferenced this pointer holds. |
| get | |
| Gets the value that the pointer points to. | |
| Prototype | <code>X* get() const throw();</code> |
| Return | Returns what the dereferenced this pointer holds. |
| release | |
| Releases the auto_ptr object. | |
| Prototype | <code>X* release() throw();</code> |
| Return | Returns what the dereferenced this pointer holds. |
| reset | |
| Resets the auto_ptr to zero or another pointer. | |
| Prototype | <code>void reset(X* p=0) throw();</code> |

20.4.5.3 auto_ptr conversions

Conversion functionality for the auto_ptr class for copying and converting.

Conversion Constructor

A conversion constructor.

| | |
|-----------|---|
| Prototype | <code>auto_ptr(auto_ptr_ref<X> r) throw();</code> |
|-----------|---|

operator auto_ptr_ref

Provides a convert to lvalue process.

| | |
|-----------|--|
| Prototype | <code>template<class Y> operator auto_ptr_ref<Y>() throw();</code> |
|-----------|--|

| | |
|--------|--|
| Return | Returns a reference that holds the this pointer. |
|--------|--|

operator auto_ptr

Releases the auto_ptr and returns the pointer held.

Prototype template<class Y> operator auto_ptr<Y>() throw();

Return Returns the pointer held.

20.4.6 C Library

The MSL C++ memory libraries use the C library memory functions. See the MSL C Reference for <stdlib.h> functions calloc, malloc, free, realloc for more information.

20.5 Date and Time

The header <ctime> has the same contents as the Standard C library header <time.h> but within namespace std.

General Utilities Libraries

20.5 Date and Time

Strings Library

This chapter is a reference guide to the ANSI/ISO String class that describes components for manipulating sequences of characters, where characters may be of type `char`, `wchar_t`, or of a type defined in a C++ program.

The Strings Library (clause 21)

The chapter is constructed in the following sub sections and mirrors clause 21 of the ISO (the International Organization for Standardization) C++ Standard :

- [“21.1 Character traits” on page 111](#) defines types and facilities for character manipulations
- [“21.2 String Classes” on page 116](#) lists string and character structures and classes
- [β²“21.3 Class Basic_string” on page 121](#) defines facilities for character sequence manipulations.
- [“23.4 Null Terminated Sequence Utilities” on page 147](#) lists facilities for Null terminated character sequence strings.

21.1 Character traits

This section defines a class template `char_traits<charT>` and two specializations for `char` and `wchar_t` types. These types are required by string and stream classes and are passed to these classes as formal parameters `charT` and `traits`.

The topics in this section are:

- [“21.1.1 Character Trait Definitions” on page 112](#)
- [“21.1.2 Character Trait Requirements” on page 112](#)
- [“21.1.3 Character Trait Type Definitions” on page 114](#)

- [“21.1.4 struct char_traits<T>” on page 115](#)

21.1.1 Character Trait Definitions

character Any object when treated sequentially can represent text. This term is not restricted to just `char` and `wchar_t` types

character container type A class or type used to represent a character. This object must be POD (Plain Old Data).

traits A class that defines types and functions necessary for handling characteristics.

NTCTS A null character termination string is a character sequence that proceeds the null character value `charT(0)`.

21.1.2 Character Trait Requirements

These types are required by string and stream classes and are passed to these classes as formal parameters `charT` and `traits`.

assign

Used for character type assignment.

Prototype `static void assign
 (char_type, const char_type);`

eq

Used for `bool` equality checking.

Prototype `static bool eq
 (const char_type&, const char_type&);`

lt

Used for `bool` less than checking.

Prototype `static bool lt(const char_type&, const
 char_type&);`

compare

Used for NTCTS comparison.

Prototype `static int compare
 (const char_type*, const char_type*, size_t n);`

length

Used when determining the length of a NTCTS.

Prototype `static size_t length
 (const char_type*);`

find

Used to find a character type in an array

Prototype `static const char_type* find
 (const char_type*, int n, const char_type&);`

move

Used to move one NTCTS to another even if the receiver contains the sting already.

Prototype `static char_type* move
 (char_type*, const char_type*, size_t);`

copy

Used for copying a NTCTS that does not contain the NTCTS already.

Prototype `static char_type* copy
 (char_type*, const char_type*, size_t);`

not_eof

Used for bool inequality checking.

Prototype `static int_type not_eof
 (const int_type&);`

to_char_type

Used to convert to a char type from an int_type

Prototype `static char_type to_char_type
 (const int_type&);`

to_int_type

Used to convert from a char type to an int_type.

Prototype static int_type to_int_type
 (const char_type&);

eq_int_type

Used to test for deletion.

Prototype static bool eq_int_type
 (const int_type&, const int_type&);

get_state

Used to store the state of the file buffer.

Prototype static state_type get_state
 (pos_type pos);

eof

Used to test for the end of a file.

Prototype static int_type eof();

21.1.3 Character Trait Type Definitions

There are several types defined in the char_traits structure for both wide and conventional char types.

Table 6.1 The functions are:

| Type | Defined | Use |
|-----------|------------|--|
| char | char_type | char values |
| int | int_type | integral values of char types including eof |
| streamoff | off_type | stream offset values |
| streampos | pos_type | stream position values |
| mbstate_t | state_type | file state values |

21.1.4 struct `char_traits`<T>

The template structure is overloaded for both the `wchar_t` type `struct char_traits<wchar_t>`. This specialization is used for string and stream usage.

NOTE The `assign`, `eq` and `lt` are the same as the `=`, `==` and `<` operators.

```
namespace std {
template<>
// struct char_traits<wchar_t>    or
struct char_traits<char> {
    typedef char char_type;
    typedef int int_type;
    typedef streamoff off_type;
    typedef streampos pos_type;
    typedef mbstate_t state_type;

    static void assign(char_type, const char_type);
    static bool eg(const char_type&, const
char_type&);
    static bool lt(const char_type&, const
char_type&);

    static int compar(const char_type*, const
char_type*, size_t n);
    static size_t length(const char_type*);
    static const char_type* find(const char_type*, 
int n, const char_type&);

    static char_type* move(char_type*, const
char_type*, size_t);
    static char_type* copy(char_type*, const
char_type*, size_t);
    static char_type* assign(char_type*, size_t,
char_type);

    static int_type not_eof(const int_type&);
    static char_type to_char_type(const int_type&);
    static int_type to_int_type(const char_type&);
    static bool eq_int_type(const int_type&, const
int_type& );
```

```
    static state_type get_state(pos_type pos);
    static int_type eof();
}
}
```

21.2 String Classes

The header <string> define string and trait classes used to manipulate character and wide character like template arguments.

```
Prototype namespace std {

    template<class charT> struct char_traits;
    template <> struct char_traits<char>;
    template <> struct char_traits<wchar_t>;

    template
    <class charT,
     class traits =char_traits<charT>,
     class Allocator = allocator<charT> >
    class basic_string;

    template
    <class charT, class traits, class Allocator>
    basic_string <charT,traits, Allocator>
    operator+
        (const basic_string
         <charT,traits,Allocator>& lhs,
        const basic_string
         <charT, traits, Allocator>& rhs);

    template
    <class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
    operator+
        const charT* lhs,
        const basic_string
         <charT, traits, Allocator>& rhs);

    template
```

```
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
    (charT lhs, const basic_string
        <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
    (const basic_string
        <charT,traits,Allocator>& lhs,
     const charT* rhs);

template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
    (const basic_string
        <charT,traits,Allocator>& lhs, charT rhs);

template
<class charT, class traits, class Allocator>
bool operator==
    (const basic_string
        <charT,traits,Allocator>& lhs,
     const basic_string
        <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator==
    (const charT* lhs,
     const basic_string
        <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator==
    (const basic_string<charT,
     traits,Allocator>& lhs,
     const charT* rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator!=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
     <charT,traits, Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator!=
    (const charT* lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator!=
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);

template
<class charT, class traits, class Allocator>
bool operator<
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const basic_string
     <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator<
    (const basic_string
     <charT,traits,Allocator>& lhs,
     const charT* rhs);

template
<class charT, class traits, class Allocator>
bool operator<
    (const charT* lhs,
```

```
    const basic_string
        <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator>
    (const basic_string
        <charT,traits,Allocator>& lhs,
     const basic_string
        <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator>=
    (const basic_string
        <charT,traits,Allocator>& lhs,
     const basic_string
        <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator<=
    (const basic_string
        <charT,traits,Allocator>& lhs,
     const basic_string
        <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator<
    (const basic_string
        <charT,traits,Allocator>& lhs,
     const charT* rhs);

template
<class charT, class traits, class Allocator>
bool operator==
```

```
bool operator<=
  (const charT* lhs,
  const basic_string
    <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator>=
  (const basic_string
    <charT,traits,Allocator>& lhs,
  const basic_string
    <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator>=
  (const basic_string
    <charT,traits,Allocator>& lhs,
  const charT* rhs);

template
<class charT, class traits, class Allocator>
bool operator>=
  (const charT* lhs,
  const basic_string
    <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
void swap
  (basic_string<charT,traits,Allocator>& lhs,
  basic_string<charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
operator>>
  (basic_istream<charT,traits>& is,
  basic_string<charT,traits,Allocator>& str);

template
<class charT, class traits, class Allocator>
```

```
basic_ostream<charT, traits>&
operator<<
    (basic_ostream<charT, traits>& os,
     const basic_string
         <charT,traits,Allocator>& str);

template
<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
getline
    (basic_istream<charT,traits>& is,
     basic_string<charT,traits,Allocator>& str,
     charT delim);

template
<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
getline
    (basic_istream<charT,traits>& is,
     basic_string<charT,traits,Allocator>& str);

typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
};
```

```
}
```

21.3 Class *Basic_string*

The `class basic_string` is used to store and manipulate a sequence of character like types of varying length known as strings.

Memory for a string is allocated and deallocated as necessary by member functions.

The first element of the sequence is at position zero.

The iterators used by `basic_string` are random iterators and as such qualifies as a reversible container

The topics in this section include:

- [“21.3.1 Constructors and Assignments” on page 128](#)

-
- “[21.3.2 Iterator Support” on page 130](#)
 - “[21.3.3 Capacity” on page 131](#)
 - “[21.3.4 Element Access” on page 132](#)
 - “[21.3.5 Modifiers” on page 133](#)
 - “[21.3.6 String Operations” on page 136](#)
 - “[23.3.7 Non-Member Functions and Operators” on page 141](#)
-

NOTE In general, the string size can be constrained by memory restrictions.

Prototype

```
namespace std {
    template <class charT, class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    class basic_string {

        public:
        typedef traits traits_type;
        typedef typename traits::char_type value_type;
        typedef Allocator allocator_type;
        typedef typename Allocator::size_type size_type;
        typedef typename Allocator::difference_type
            difference_type;
        typedef typename Allocator::reference reference;
        typedef typename Allocator::const_reference
            const_reference;
        typedef typename Allocator::pointer pointer;
        typedef typename Allocator::const_pointer
            const_pointer;
        // typedef implementation defined iterator;
        // typedef implementation defined const_iterator;
        typedef std::reverse_iterator<iterator>
            reverse_iterator;
        typedef std::reverse_iterator<const_iterator>
            const_reverse_iterator;

        static const size_type npos = -1;

        explicit basic_string
            (const Allocator& a = Allocator());
    };
}
```

```
basic_string(const basic_string& str);
basic_string(const basic_string& str,
             size_type pos, size_type n = npos,
             const Allocator& a = allocator_type());
basic_string
    (const charT* s,
     size_type n,
     const Allocator& a = Allocator());
basic_string
    (const charT* s,
     const Allocator& a = Allocator());
basic_string
    (size_type n,
     charT c,
     const Allocator& a = Allocator());
template<class InputIterator>
basic_string
    (InputIterator begin,
     InputIterator end,
     const Allocator& a = Allocator());
~basic_string();

basic_string& operator= (const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

size_type size() const;
size_type length() const;
size_type max_size() const;
void resize(size_type n, charT c);
void resize(size_type n);
```

Strings Library

21.3 Class Basic_string

```
size_type capacity() const;
void reserve(size_type res_arg = 0);
void clear();
bool empty() const;

const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const_reference at(size_type n) const;
reference at(size_type n);

basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& append(const basic_string& str);
basic_string& append(
    const basic_string& str,
    size_type pos, size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
basic_string& append
    (InputIterator first, InputIterator last);
void push_back(const charT);
basic_string& assign(const basic_string& );
basic_string& assign
    (const basic_string& str,
     size_type pos, size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
basic_string& assign
    (InputIterator first, InputIterator last);
basic_string& insert
    (size_type pos1, const basic_string& str);
basic_string& insert
    (size_type pos1, const basic_string& str,
     size_type pos2, size_type n);
basic_string& insert
    (size_type pos, const charT* s, size_type n);
basic_string& insert
```

```
(size_type pos, const charT* s);
basic_string& insert
    (size_type pos, size_type n, charT c);
iterator insert(iterator p, charT c = charT());
void insert(iterator p, size_type n, charT c);
template<class InputIterator>
void insert
    (iterator p, InputIterator first,
     InputIterator last);
basic_string& erase
    (size_type pos = 0, size_type n = npos);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
basic_string& replace
    (size_type pos1, size_type n1,
     const basic_string& str);
basic_string& replace
    (size_type pos1, size_type n1,
     const basic_string& str,
     size_type pos2, size_type n2);
basic_string& replace
    (size_type pos, size_type n1,
     const charT* s, size_type n2);
basic_string& replace
    (size_type pos, size_type n1, const charT* s);
basic_string& replace
    (size_type pos, size_type n1,
     size_type n2, charT c);
basic_string& replace
    (iterator i1, iterator i2,
     const basic_string& str);
basic_string& replace
    (iterator i1, iterator i2,
     const charT* s, size_type n);
basic_string& replace(
    iterator i1, iterator i2, const charT* s);
basic_string& replace
    (iterator i1, iterator i2,
     size_type n, charT c);
template<class InputIterator>
basic_string& replace
    (iterator i1, iterator i2,
```

Strings Library

21.3 Class Basic_string

```
    InputIterator j1, InputIterator j2);

    size_type copy
        (charT* s, size_type n,
         size_type pos = 0) const;
    void swap(basic_string<charT,traits,Allocator>&);

    const charT* c_str() const;
    const charT* data() const;
    allocator_type get_allocator() const;
    size_type find
        (const basic_string& str,
         size_type pos = 0) const;
    size_type find
        (const charT* s, size_type pos,
         size_type n) const;
    size_type find
        (const charT* s, size_type pos = 0) const;
    size_type find (charT c, size_type pos = 0) const;
    size_type rfind
        (const basic_string& str,
         size_type pos = npos) const;
    size_type rfind
        (const charT* s, size_type pos,
         size_type n) const;
    size_type rfind
        (const charT* s, size_type pos = npos) const;
    size_type rfind
        (charT c, size_type pos = npos) const;
    size_type find_first_of
        (const basic_string& str,
         size_type pos = 0) const;
    size_type find_first_of
        (const charT* s, size_type pos,
         size_type n) const;
    size_type find_first_of
        (const charT* s, size_type pos = 0) const;
    size_type find_first_of
        (charT c, size_type pos = 0) const;
    size_type find_last_of
        (const basic_string& str,
         size_type pos = npos) const;
```

```
size_type find_last_of
    (const charT* s,
     size_type pos, size_type n) const;
size_type find_last_of
    (const charT* s, size_type pos = npos) const;
size_type find_last_of
    (charT c, size_type pos = npos) const;
size_type find_first_not_of
    (const basic_string& str,
     size_type pos = 0) const;
size_type find_first_not_of
    (const charT* s, size_type pos,
     size_type n) const;
size_type find_first_not_of
    (const charT* s, size_type pos = 0) const;
size_type find_first_not_of
    (charT c, size_type pos = 0) const;
size_type find_last_not_of
    (const basic_string& str,
     size_type pos = npos) const;
size_type find_last_not_of
    (const charT* s,
     size_type pos, size_type n) const;
size_type find_last_not_of
    (const charT* s, size_type pos = npos) const;
size_type find_last_not_of
    (charT c, size_type pos = npos) const;
basic_string substr
    (size_type pos = 0, size_type n = npos) const;

int compare(const basic_string& str) const;
int compare(
    size_type pos1, size_type n1,
    const basic_string& str) const;
int compare
    (size_type pos1, size_type n1,
     const basic_string& str,
     size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1,
            const charT* s) const;
int compare(size_type pos1, size_type n1,
```

```
        const charT* s, size_type n2) const;  
    };  
}
```

The class basic_string can have either of two implementations:

- Refcounted.
- Non-refcounted.

The interface and functionality are identical with both implementations. The only difference is performance. Which performs best is dependent upon usage patterns in each application.

The refcounted implementation ships as the default.

NOTE To enable the non-refcounted implementation un-comment `#define _MSL_NO_REFCOUNT_STRING` in `<mslconfig>`. The C++ library and precompiled headers must be rebuilt after making this change.

21.3.1 Constructors and Assignments

Constructor, destructor and assignment operators and functions.

Constructors

The various basic_string constructors construct a string object for character sequence manipulations. All constructors include an Allocator argument that is used for memory allocation.

Prototype `explicit basic_string
(const Allocator& a = Allocator());`

Remarks This default constructor, constructs an empty string. A zero sized string that may be copied to is created.

Prototype `basic_string
(const basic_string& str,
size_type pos = 0,
size_type n = npos,
const Allocator& a = Allocator());`

| | |
|-----------|--|
| Remarks | This constructor takes a string class argument and creates a copy of that string, with size of the length of that string and a capacity at least as large as that string. |
| | <ul style="list-style-type: none"> • An exception is thrown upon failure |
| Prototype | <pre>basic_string (const charT* s, size_type n, const Allocator& a = Allocator());</pre> |
| Remarks | This constructor takes a <code>const char</code> array argument and creates a copy of that array with the size limited to the <code>size_type</code> argument. |
| | <ul style="list-style-type: none"> • The <code>charT*</code> argument shall not be a null pointer • An exception is thrown upon failure |
| Prototype | <pre>basic_string (const charT* s, const Allocator& a = Allocator());</pre> |
| Remarks | This constructor takes an <code>const char</code> array argument. The size is determined by the size of the <code>char</code> array. |
| | <ul style="list-style-type: none"> • The <code>charT*</code> argument shall not be a null pointer |
| Prototype | <pre>basic_string (size_type n, charT c, const Allocator& a = Allocator());</pre> |
| Remarks | This constructor creates a string of <code>size_type n</code> size repeating <code>charT c</code> as the filler. |
| | <ul style="list-style-type: none"> • A <code>length_error</code> is thrown if <code>n</code> is less than <code>npos</code>. |
| Prototype | <pre>template<class InputIterator> basic_string (InputIterator begin, InputIterator end, const Allocator& a = Allocator());</pre> |
| Remarks | This iterator string takes <code>InputIterator</code> arguments and creates a string with its first position starts with <code>begin</code> and its ending position is <code>end</code> . Size is the distance between <code>beginning</code> and <code>end</code> . |

Destructor

Deallocates the memory referenced by the `basic_string` object.

Prototype `~basic_string () ;`

Assignment Operator

Assigns the input string, char array or char type to the current string.

Prototype `basic_string& operator= (const basic_string& str);`

Remarks If `*this` and `str` are the same object has it has no effect.

Prototype `basic_string& operator=(const charT* s);`

Remarks Used to assign a NCTCS to a string.

Prototype `basic_string& operator=(charT c);`

Remarks Used to assign a single char type to a string.

Assignment & Addition Operator `basic_string`

Appends the string `rhs` to the current string.

Prototype `string& operator+= (const string& rhs);`
`string& operator+= (const charT* s);`
`string& operator+= (charT s);`

Both of the overloaded functions construct a string object from the input `s`, and append it to the current string.

Return The assignment operator returns the `this` pointer.

21.3.2 Iterator Support

Member functions for string iterator support.

begin

Returns an iterator to the first character in the string

Prototype `iterator begin();`
`const_iterator begin() const;`

end

Returns an iterator that is past the end value.

```
iterator end();  
const_iterator end() const;
```

rbegin

Returns an iterator that is equivalent to

Prototype `reverse_iterator(end()).
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;`

rend

Returns an iterator that is equivalent to

Prototype `reverse_iterator(begin()).
reverse_iterator rend();
const_reverse_iterator rend() const;`

21.3.3 Capacity

Member functions for determining a strings capacity.

size

Returns the size of the string.

Prototype `size_type size() const;`

length

Returns the length of the string

Prototype `size_type length() const;`

max_size

Returns the maximum size of the string.

Prototype `size_type max_size() const;`

resize

Resizes the string to size `n`.

Prototype `void resize(size_type n);`
`void resize(size_type n, charT c);`

If the size of the string is longer than `size_type n`, it shortens the string to `n`, if the size of the string is shorter than `n` it appends the string to size `n` with `charT c` or `charT()` if no filler is specified.

capacity

Returns the memory storage capacity.

Prototype `size_type capacity() const;`

reserve

A directive that indicates a planned change in memory size to allow for better memory management.

Prototype `void reserve(size_type res_arg = 0);`

clear

Erases from `begin()` to `end()`.

Prototype `void clear();`

empty

Empties the string stored.

Prototype `bool empty() const;`

Return Returns `true` if the size is equal to zero, otherwise `false`.

21.3.4 Element Access

Member functions and operators for accessing individual string elements.

operator[]

An operator used to access an indexed element of the string.

Prototype `const_reference operator[](size_type pos) const;`
`reference operator[](size_type pos);`

at

A function used to access an indexed element of the string.

Prototype `const_reference at(size_type n) const;`
`reference at(size_type n);`

21.3.5 Modifiers

Operators for appending a string.

21.3.5.1 operator+=

An Operator used to append to the end of a string.

Prototype `basic_string& operator+=(const basic_string& str);`
`basic_string& operator+=(const charT* s);`
`basic_string& operator+=(charT c);`

21.3.5.2 append

Member functions for appending to the end of a string.

A function used to append to the end of a string.

Prototype `basic_string& append(const basic_string& str);`
`basic_string& append(`
 `const basic_string& str,`
 `size_type pos, size_type n);`

`basic_string& append(const charT* s, size_type n);`

`basic_string& append(const charT* s);`

`basic_string& append(size_type n, charT c);`

`template<class InputIterator>`
`basic_string& append`

```
(InputIterator first, InputIterator last);
```

21.5.3.3 assign

Assigns a string, Null Terminated Character Type Sequence or char type to the string.

Prototype

```
basic_string& assign(const basic_string&);

basic_string& assign
    (const basic_string& str,
     size_type pos, size_type n);

basic_string& assign(const charT* s, size_type n);

basic_string& assign(const charT* s);

basic_string& assign(size_type n, charT c);

template<class InputIterator>
basic_string& assign
    (InputIterator first, InputIterator last);
```

Remarks If there is a size argument whichever is smaller the string size or argument value will be assigned.

21.3.5.4 insert

Inserts a string, Null Terminated Character Type Sequence or char type into the string.

Prototype

```
basic_string& insert
    (size_type pos1, const basic_string& str);

basic_string& insert
    (size_type pos1, const basic_string& str,
     size_type pos2, size_type n);

basic_string& insert
    (size_type pos, const charT* s, size_type n);

basic_string& insert
    (size_type pos, const charT* s);

basic_string& insert
    (size_type pos, size_type n, charT c);
iterator insert(iterator p, charT c = charT());
```

```
void insert(iterator p, size_type n, charT c);  
  
template<class InputIterator>  
void insert  
    (iterator p, InputIterator first,  
     InputIterator last);
```

Remarks May throw an exception,

21.3.5.5 **erase**

Erases the string

Prototype basic_string& erase
 (size_type pos = 0, size_type n = npos);
 iterator erase(iterator position);
 iterator erase(iterator first, iterator last);

Remarks May throw an exception,

21.3.5.6 **replace**

Replaces the string with a string, Null Terminated Character Type Sequence or char type.

Prototype basic_string& replace
 (size_type pos1, size_type n1,
 const basic_string& str);

basic_string& replace
 (size_type pos1, size_type n1,
 const basic_string& str,
 size_type pos2, size_type n2);

basic_string& replace
 (size_type pos, size_type n1,
 const charT* s, size_type n2);

basic_string& replace
 (size_type pos, size_type n1, const charT* s);

basic_string& replace
 (size_type pos, size_type n1,
 size_type n2, charT c);

basic_string& replace

```
(iterator il, iterator i2,
const basic_string& str);

basic_string& replace
(iterator il, iterator i2,
const charT* s, size_type n);

basic_string& replace(
iterator il, iterator i2, const charT* s);

basic_string& replace
(iterator il, iterator i2,
size_type n, charT c);

template<class InputIterator>
basic_string& replace
(iterator il, iterator i2,
InputIterator j1, InputIterator j2);
```

Remarks May throw an exception,

21.3.5.7 copy

Copies a Null Terminated Character Type Sequence to a string up to the size designated.

Prototype `size_type copy`
 `(charT* s, size_type n,`
 `size_type pos = 0) const;`

Remarks The function `copy` does not pad the string with Null characters.

21.3.5.8 swap

Swaps one string for another.

Prototype `void swap(basic_string<chart,traits,Allocator>&);`

21.3.6 String Operations

Member functions for sequences of character operations.

c_str

Returns the string as a Null terminated character type sequence.

Prototype `const charT* c_str() const;`

data

Returns the string as an array without a Null terminator.

Prototype `const charT* data() const;`

get_allocator

Returns a copy of the allocator object used to create the string.

Prototype `allocator_type get_allocator() const;`

21.3.6.1 find

Finds a string, Null Terminated Character Type Sequence or char type in a string starting from the beginning.

Prototype `size_type find
 (const basic_string& str,
 size_type pos = 0) const;`

`size_type find
 (const charT* s, size_type pos,
 size_type n) const;`

`size_type find
 (const charT* s, size_type pos = 0) const;`

`size_type find (charT c, size_type pos = 0) const;`

Return The found position or npos if not found.

21.3.6.2 rfind

Finds a string, Null Terminated Character Type Sequence or char type in a string testing backwards from the end.

Prototype `size_type rfind
 (const basic_string& str,
 size_type pos = npos) const;`

`size_type rfind
 (const charT* s, size_type pos,
 size_type n) const;`

`size_type rfind
 (const charT* s, size_type pos = npos) const;`

`size_type rfind
 (charT c, size_type pos = npos) const;`

Return The found position or `npos` if not found.

21.6.3.3 `find_first_of`

Finds the first position of one of the elements in the function's argument starting from the beginning.

Prototype `size_type find_first_of
 (const basic_string& str,
 size_type pos = 0) const;`

`size_type find_first_of
 (const charT* s, size_type pos,
 size_type n) const;`

`size_type find_first_of
 (const charT* s, size_type pos = 0) const;`

`size_type find_first_of
 (charT c, size_type pos = 0) const;`

Return The found position or `npos` if not found.

21.3.6.4 `find_last_of`

Finds the last position of one of the elements in the function's argument starting from the beginning.

Prototype `size_type find_last_of
 (const basic_string& str,
 size_type pos = npos) const;`

`size_type find_last_of`

```
(const charT* s,
 size_type pos, size_type n) const;

size_type find_last_of
(const charT* s, size_type pos = npos) const;

size_type find_last_of
(charT c, size_type pos = npos) const;
```

Return The found position or `npos` if not found.

21.3.6.5 `find_first_not_of`

Finds the first position that is not one of the elements in the function's argument starting from the beginning.

Prototype

```
size_type find_first_not_of
(const basic_string& str,
 size_type pos = 0) const;

size_type find_first_not_of
(const charT* s, size_type pos,
 size_type n) const;

size_type find_first_not_of
(const charT* s, size_type pos = 0) const;

size_type find_first_not_of
(charT c, size_type pos = 0) const;
```

Return The found position or `npos` if not found.

21.3.6.6 `find_last_not_of`

Finds the last position that is not one of the elements in the function's argument starting from the beginning.

Prototype

```
size_type find_last_not_of
(const basic_string& str,
 size_type pos = npos) const;

size_type find_last_not_of
(const charT* s,
 size_type pos, size_type n) const;

size_type find_last_not_of
(const charT* s, size_type pos = npos) const;
```

```
size_type find_last_not_of  
    (charT c, size_type pos = npos) const;
```

Return The found position or `npos` if not found.

21.3.6.7 substr

Returns a string if possible from beginning at the first arguments position to the last position.

Prototype `basic_string substr
 (size_type pos = 0, size_type n = npos) const;`

Remarks May throw an exception,

21.3.6.8 compare

Compares a string, substring or Null Terminated Character Type Sequence with a lexicographical comparison.

```
int compare(const basic_string& str) const;  
  
int compare(  
    size_type pos1, size_type n1,  
    const basic_string& str) const;  
  
int compare  
    (size_type pos1, size_type n1,  
    const basic_string& str,  
    size_type pos2, size_type n2) const;  
  
int compare(const charT* s) const;  
  
int compare  
    (size_type pos1, size_type n1, const charT* s,  
    size_type n2 = npos) const;
```

Return Less than zero if the string is smaller than the argument lexicographically, zero if the string is the same size as the argument lexicographically and greater than zero if the string is larger than the argument lexicographically.

23.3.7 Non-Member Functions and Operators

21.3.7.1 operator+

Appends one string to another.

Prototype

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
(const basic_string
<charT,traits, Allocator>& lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
```

```
operator+
(const charT* lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
(charT lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
(const basic_string
<charT,traits,Allocator>& lhs,
const charT* rhs);
```

```
template
<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
operator+
```

```
(const basic_string
    <charT,traits,Allocator>& lhs, charT rhs);
```

Return The combined strings.

21.3.7.2 operator==

Test for lexicographical equality.

Prototype

```
template
<class charT, class traits, class Allocator>
bool operator==
(const basic_string
    <charT,traits,Allocator>& lhs,
const basic_string
    <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator==
(const charT* lhs,
const basic_string
    <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator==
(const basic_string
    <charT,traits,Allocator>& lhs,
const charT* rhs);
```

Return True if the strings match otherwise false.

21.3.7.3 operator!=

Test for lexicographical inequality.

Prototype

```
template
<class charT, class traits, class Allocator>
bool operator!=
(const basic_string
    <charT,traits,Allocator>& lhs,
const basic_string
    <charT,traits,Allocator>& rhs);
```

```
template
<class charT, class traits, class Allocator>
bool operator!=
(const charT* lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```



```
template
<class charT, class traits, class Allocator>
bool operator!=
(const basic_string
<charT,traits,Allocator>& lhs,
const charT* rhs);
```

Return True if the strings do not match otherwise false.

21.3.7.4 operator<

Tests for a lexicographically less than condition.

Prototype

```
template
<class charT, class traits, class Allocator>
bool operator<
(const basic_string
<charT,traits,Allocator>& lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```



```
template
<class charT, class traits, class Allocator>
bool operator<
(const charT* lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```



```
template
<class charT, class traits, class Allocator>
bool operator<
(const basic_string
<charT,traits,Allocator>& lhs,
const charT* rhs);
```

Return True if the first argument is lexicographically less than the second argument otherwise false.

21.3.7.5 operator>

Tests for a lexicographically greater than condition.

Prototype

```
template
<class charT, class traits, class Allocator>
bool operator>
(const basic_string
<charT,traits,Allocator>& lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```



```
template
<class charT, class traits, class Allocator>
bool operator>
(const charT* lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```



```
template
<class charT, class traits, class Allocator>
bool operator>
(const basic_string
<charT,traits,Allocator>& lhs,
const charT* rhs);
```

Return True if the first argument is lexicographically greater than the second argument otherwise false.

21.3.7.6 operator<=

Tests for a lexicographically less than or equal to condition.

Prototype

```
template
<class charT, class traits, class Allocator>
bool operator<=
(const basic_string
<charT,traits,Allocator>& lhs,
const basic_string
<charT,traits,Allocator>& rhs);
```



```
template
<class charT, class traits, class Allocator>
bool operator<=
```

```
(const charT* lhs,
 const basic_string
 <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator<=
(const basic_string
 <charT,traits,Allocator>& lhs,
 const charT* rhs);
```

Return True if the first argument is lexicographically less than or equal to the second argument otherwise false.

21.3.7.7 operator>=

Tests for a lexicographically greater than or equal to condition.

Prototype

```
template
<class charT, class traits, class Allocator>
bool operator>=
(const basic_string
 <charT,traits,Allocator>& lhs,
 const basic_string
 <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator>=
(const charT* lhs,
 const basic_string
 <charT,traits,Allocator>& rhs);

template
<class charT, class traits, class Allocator>
bool operator>=
(const basic_string
 <charT,traits,Allocator>& lhs,
 const charT* rhs);
```

Return True if the first argument is lexicographically greater than or equal to the second argument otherwise false.

21.3.7.8 swap

This non member `swap` exchanges the first and second arguments.

Prototype `template <class charT, class traits, class Allocator> void swap (basic_string<charT,traits,Allocator>& lhs, basic_string <charT,traits,Allocator>& rhs);`

21.3.7.9 Inserters and extractors

Overloaded inserters and extractors for `basic_string` types.

operator>>

Overloaded extractor for stream input operations.

Prototype `template <class charT, class traits, class Allocator> basic_istream<charT,traits>& operator>> (basic_istream<charT,traits>& is, basic_string<charT,traits,Allocator>& str);`

Remarks Characters are extracted and appended until `n` characters are stored or `end-of-file` occurs on the input sequence;

operator<<

Inserts characters from a string object from into a output stream.

Prototype `template <class charT, class traits, class Allocator> basic_ostream<charT, traits>& operator<< (basic_ostream<charT, traits>& os, const basic_string <charT,traits,Allocator>& str);`

getline

Extracts characters from a `stream` and appends them to a `string`.

Prototype

```
template
<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
getline
(basic_istream<charT,traits>& is,
basic_string
<charT,traits,Allocator>& str,
charT delim);
template
<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
getline
(basic_istream<charT,traits>& is,
basic_string<charT,traits,Allocator>& str)
```

Remarks

Extracts characters from a `stream` and appends them to the `string` until the `end-of-file` occurs on the input sequence (in which case, the `getline` function calls `setstate(eofbit)` or the delimiter is encountered in which case, the delimiter is extracted but not appended).

If the function extracts no characters, it calls `setstate(failbit)` in which case it may throw an exception.

23.4 Null Terminated Sequence Utilities

The standard requires C++ versions of the standard libraries for use with characters and Null Terminated Character Type Sequences.

Character Support

The standard provides for namespace and character type support.

Table 6.2 Headers <cctype.h> and <cwctype.h>

| <cctype.h> | <cwctype.h> | <cctype.h> | <cwctype.h> |
|-------------------------|--------------------------|-------------------------|--------------------------|
| isalnum | iswalnum | isprint | iswprint |
| isalpha | iswalpha | ispunct | iswpunct |
| iscntrl | iswcntrl | isspace | iswspace |
| isdigit | iswdigit | isupper | iswupper |
| isgraph | iswgraph | isxdigit | iswxdigit |
| islower | iswlower | | |
| tolower | towlower | toupper | towupper |
| | wctype | | iswctype |
| | wctrans | | towctrans |
| Macros | | EOF | WEOF |

String Support

The standard provides for namespace and wide character type for Null Terminated Character Type Sequence functionality.

Table 6.3 Headers <cstring.h> and <wchar.h>

| <cstring.h> | <wchar.h> | <cstring.h> | <wchar.h> |
|--------------------------|------------------------|--------------------------|------------------------|
| memchr | wmemchr | strerror | |
| memcmp | wmemcmp | strlen | wcslen |
| memcpy | wmemcpy | strncat | wcsncat |
| memmove | wmemmove | strcmp | wcsncmp |
| memset | wmemset | strncpy | wcsncpy |
| strcat | wscat | strpbrk | wcspbrk |
| strchr | wcschr | strrchr | wcsrchr |
| strcmp | wcsncmp | strspn | wcsspn |
| strcoll | wcscoll | strstr | wcsstr |

| <cstring.h> | <wchar.h> | <cstring.h> | <wchar.h> |
|--------------------------|------------------------|--------------------------|------------------------|
| memchr | wmemchr | strerror | |
| memcmp | wmemcmp | strlen | wcslen |
| memcpy | wmemcpy | strncat | wcsncat |
| memmove | wmemmove | strncmp | wcsncmp |
| memset | wmemset | strncpy | wcsncpy |
| strcpy | wcsncpy | strtok | wcstok |
| strcspn | wcsncspn | strxfrm | wcsxfrm |
| Type | mbstate_t | size_t | wint_t |
| Macro | NULL | NULL | WCHAR_MAX |
| Macro | | | WCHAR_MIN |

Input and Output Manipulations

The standard provides for namespace and wide character support for manipulation and conversions of input and output and character and character sequences.

Table 6.4 Additional <wchar.h> and <stdlib.h> support

| wchar.h | wchar.h | wchar.h | <cstdlib.h> |
|----------------|----------------|----------------|--------------------------|
| btowc | mbrtowc | wcrtomb | atol |
| fgetwc | mbsinit | wcscoll | atof |
| fgetws | mbsrtowcs | wcsftime | atoi |
| fputwc | putwc | wcstod | mblen |
| fputws | putwchar | wcstol | mbstowcs |
| fwide | swscanf | wcsrtombs | mbtowc |
| fwprintf | swprintf | wcstoul | strtod |
| fwscanf | ungetwc | wctob | strtol |
| getwc | vfwprintf | wprintf | strtoul |

Strings Library

23.4 Null Terminated Sequence Utilities

| wchar.h | wchar.h | wchar.h | <cstdlib.h> |
|----------|-----------|---------|-------------|
| btowc | mbrtowc | wcrtomb | atol |
| getwchar | vwprintf | wscanf | wctomb |
| mbrlen | vswprintf | | wcstombs |

Localization Library

This chapter describes components that C++ library that may use for porting to different cultures.

The Localization library (clause 22)

Much of named locales is implementation defined behavior and is not portable between vendors. This document specifies the behavior of MSL C++. Other vendors may not provide this functionality, or may provide it in a different manner.

The chapter is constructed in the following sub sections and mirrors clause 22 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Strings and Characters in Locale Data Files” on page 153](#)
- [“22.1 Locales” on page 155](#)
- [“22.2 Standard Locale Categories” on page 166](#)
- [“22.3 C Library Locales” on page 269](#)

Supported Locale Names

MSL C++ predefines only two names: “C” and “”. However, other names sent to the locale constructor are interpreted as file names containing data to create a named locale. So localizing your program is as easy as creating a data file specifying the desired behavior. The format for this data file is outlined below for each different facet.

A **locale** is a collection of **facets**. And a **facet** is a class that provides a certain behavior. The “C” locale contains the following facets:

- `ctype<char> & ctype<wchar_t>`

Localization Library

The Localization library (clause 22)

- `codecvt<char, char, mbstate_t>` &
`codecvt<wchar_t, char, mbstate_t>`
- `num_get<char>` & `num_get<wchar_t>`
- `num_put<char>` & `num_put<wchar_t>`
- `numpunct<char>` & `numpunct<wchar_t>`
- `collate<char>` & `collate<wchar_t>`
- `time_get<char>` & `time_get<wchar_t>`
- `time_put<char>` & `time_put<wchar_t>`
- `money_get<char>` & `money_get<wchar_t>`
- `money_put<char>` & `money_put<wchar_t>`
- `moneypunct<char, bool>` & `moneypunct<wchar_t, bool>`
- `messages<char>` & `messages<wchar_t>`

A named locale replaces many of these facets with “`_byname`” versions, whose behavior can vary based on the name passed.

- `ctype_byname<char>` & `ctype_byname<wchar_t>`
- `codecvt_byname<char, char, mbstate_t>` &
`codecvt_byname<wchar_t, char, mbstate_t>`
- `numpunct_byname<char>` &
`numpunct_byname<wchar_t>`
- `collate_byname<char>` & `collate_byname<wchar_t>`
- `time_get_byname<char>` &
`time_get_byname<wchar_t>`
- `time_put_byname<char>` &
`time_put_byname<wchar_t>`
- `moneypunct_byname<char, bool>` &
`moneypunct_byname<wchar_t, bool>`
- `messages_byname<char>` &
`messages_byname<wchar_t>`

The behavior of each of these “`_byname`” facets can be specified with a data file. A single data file can contain data for all of the byname facets. That way, when you code:

```
locale myloc( "MyLocale" );
```

then the file "MyLocale" will be used for each "_byname" facet in myloc.

Strings and Characters in Locale Data Files

The named locale facility involves reading strings and characters from files. This document gives the details of the syntax used to enter strings and characters.

Character Syntax

Characters in a locale data file can in general appear quoted () or not. For example:

```
thousands_sep = ,  
thousands_sep = ','
```

Both of the above statements set thousands_sep to a comma. Quotes might be necessary to disambiguate the intended character from ordinary whitespace. For example, to set the thousands_sep to a space character, quotes must be used:

```
thousands_sep = ' '
```

The whitespace appearing before and after the equal sign is not necessary and insignificant.

Escape sequences

The usual C escape sequences are recognized. For example, to set the thousands_sep to the single quote character, an escape sequence must be used:

```
thousands_sep = '\''
```

The recognized escape sequences are:

- '\n' - newline
- '\t' - horizontal tab
- '\v' - vertical tab
- '\b' - backspace
- '\r' - carriage return

- \f - form feed
- \a - alert
- \\ - \
- \? - ?
- \" - "
- \' - '
- \u \U - universal character
- \x - hexadecimal character
- \ooo - octal character

The octal character may have from 1 to 3 octal digits (digits must be in the range [0, 7]. The parser will read as many digits as it can to interpret a valid octal number. For example:

\18

This is the character '\1' followed by the character '8'.

\17

But this is the single character '\17'.

The hexadecimal and universal character formats are all identical with each other, and have slightly relaxed syntax compared to the formats specified in the standard. The x (or u or U) is followed by zero to `sizeof(charT) *CHAR_BIT/4` hexadecimal digits. `charT` is `char` when reading narrow data, and `wchar_t` when reading wide data (even when reading wide data from a narrow file). On Macintosh and Windows this translates to 0 to 2 digits when reading a `char`, and from 0 to 4 digits when reading a `wchar_t`. Parsing the character is terminated when either the digit limit has been reached, or a non-hexadecimal digit has been reached. If there are 0 valid digits, then the character is read as '\0'. Example (assume a 8 bit `char` and 16 bit `wchar_t`):

\x01234

When reading narrow data this is the following sequence of 4 char's:
'\1' '2' '3' '4'

The '\x01' is read as one character, but the following '2' is not included because a 8 bit `char` can only hold 2 hex digits.

When reading wide data the above example parses to the following two wchar_t's: L'\\x123' L'4'

The '\\x0123' is read as one wchar_t, but the following '4' is not included because a 16 bit wchar_t can only hold 4 hex digits.

Errors

If a character is expected, but an end of file occurs, then failbit is set. If a character is started with a single quote, and end of file occurs before the character within the quotes can be read, or if a closing quote is not found directly after the character, then failbit will be set. Depending on the context of when the character is being read, setting failbit may or may not cause a runtime error to be thrown.

String Syntax

Strings can be quoted or not (using ""). If the string contains white space, then it must be quoted. For example:

Hi there!

This would be parsed as two strings: "Hi" and "there!". But the following is one string:

"Hi there!"

If a string begins with quotes, but does not end with a quote (before end of file), then failbit will be set. This may nor may not cause a runtime error to be thrown (depending on the context).

Any of the escape sequences described under character syntax are allowed within strings. But within strings, single quotes do not delimit characters. Instead single quotes are just another character in the string. Note that you can use \\\" to place the string quote character within a string.

22.1 Locales

The header <locale> defines classes used to contain and manipulate information for a locale.

- ["22.1.1 Class locale" on page 157](#)
- ["22.1.2 Locale Globals" on page 164](#)

- [“22.1.3 Convenience Interfaces” on page 165](#)
- [“22.1.3.1 Character Classification” on page 165](#)

Listing 7.1 Header <locale> Synopsis

```
namespace std {  
  
class locale;  
template <class Facet> const Facet& use_facet(const locale&);  
template <class Facet> bool has_facet(const locale&) throw();  
  
template <class charT> bool isspace (charT c, const locale& loc);  
template <class charT> bool isprint (charT c, const locale& loc);  
template <class charT> bool iscntrl (charT c, const locale& loc);  
template <class charT> bool isupper (charT c, const locale& loc);  
template <class charT> bool islower (charT c, const locale& loc);  
template <class charT> bool isalpha (charT c, const locale& loc);  
template <class charT> bool isdigit (charT c, const locale& loc);  
template <class charT> bool ispunct (charT c, const locale& loc);  
template <class charT> bool isxdigit(charT c, const locale& loc);  
template <class charT> bool isalnum (charT c, const locale& loc);  
template <class charT> bool isgraph (charT c, const locale& loc);  
template <class charT> charT toupper(charT c, const locale& loc);  
template <class charT> charT tolower(charT c, const locale& loc);  
22- 2 Localization library DRAFT: 25 November 1997 22.1 Locales  
  
class ctype_base;  
template <class charT> class ctype;  
template <> class ctype<char>; // specialization  
template <class charT> class ctype_byname;  
template <> class ctype_byname<char>; // specialization  
class codecvt_base;  
template <class internT, class externT, class stateT>  
class codecvt;  
template <class internT, class externT, class stateT>  
class codecvt_byname;  
  
template <class charT, class InputIterator> class num_get;  
template <class charT, class OutputIterator> class num_put;  
template <class charT> class numpunct;  
template <class charT> class numpunct_byname;
```

```
template <class charT> class collate;
template <class charT> class collate_byname;

class time_base;
template <class charT, class InputIterator> class time_get;
template <class charT, class InputIterator> class
time_get_byname;
template <class charT, class OutputIterator> class time_put;
template <class charT, class OutputIterator> class
time_put_byname;

class money_base;
template <class charT, class InputIterator> class money_get;
template <class charT, class OutputIterator> class money_put;
template <class charT, bool Intl> class moneypunct;
template <class charT, bool Intl> class moneypunct_byname;

class messages_base;
template <class charT> class messages;
template <class charT> class messages_byname;
}
```

22.1.1 Class locale

The class locale contains a set of facets for locale implementation. These facets are as if they were an index and an interface at the same time.

Combined Locale Names

Two locale constructors can result in a new locale whose name is a combination of the names of two other locales:

```
locale(const locale& other, const char* std_name, category);
locale(const locale& other, const locale& one, category);
```

If other has a name (and if one has a name in the case of the second constructor), then the resulting locale's name is composed from the two locales' names. A combined name locale has the format:

```
collate_name/ctype_name/monetary_name/numeric_name/time_name/
messages_name
```

Each name is the name of a locale from which that category of facets was copied.

Listing 7.2 Locale example usage:

```
#include <locale>
#include <iostream>

int main()
{
    using std::locale;
    locale loc(locale("other"), locale("one"),
               locale::collate | locale::numeric);
    std::cout << loc.name() << '\n';
    locale loc2(locale(), loc, locale::monetary |
                locale::collate);
    std::cout << loc2.name() << '\n';
}
```

The locale loc is created from two locales: other and one. The facets in the categories collate and numeric are taken from one. The rest of the facets are taken from other. The name of the resulting locale is:

one/other/other/one/other/other

The locale loc2 is created from the “C” locale and from loc (which already has a combined name). It takes only the monetary and collate facets from loc, and the rest from “C”:

one/C/other/C/C/C

Using this format, two locales can be compared by name, and if their names are equal, then they have the same facets.

Listing 7.3 Class Locale

```
namespace std {
class locale {
public:

    class facet;
    class id;
    typedef int category;
```

```
static const category
none = 0,
collate = 0x010, ctype = 0x020,
monetary = 0x040, numeric = 0x080,
time = 0x100, messages = 0x200,
all = collate | ctype | monetary | numeric | time | messages;

locale() throw()
locale(const locale& other) throw()
explicit locale(const char* std_name);
locale(const locale& other, const char* std_name, category);
template <class Facet> locale(const locale& other, Facet* f);
locale(const locale& other, const locale& one, category);
~locale() throw();

const locale& operator=(const locale& other) throw();
template <class Facet> locale combine(const locale& other);

basic_string<char> name() const;
bool operator==(const locale& other) const;
bool operator!=(const locale& other) const;
template <class charT, class Traits, class Allocator>
bool operator()(const basic_string<charT,Traits,Allocator>& s1,
const basic_string<charT,Traits,Allocator>& s2) const;

static locale global(const locale&);
static const locale& classic();
};

}
```

22.1.1.1 Locale Types

This library contains various types specific for locale implementation.

22.1.1.1 Type `locale::Category`

An integral type used as a mask for all types.

Prototype `typedef int category;`

Each `locale` member function takes a `locale::category` argument based on a corresponding `facet`.

Table 7.1 **Locale Category Facets**

| Category | Includes Facets |
|----------|---|
| collate | <code>collate<char></code> , <code>collate<wchar_t></code> |
| ctype | <code>ctype<char></code> , <code>ctype<wchar_t></code> , <code>codecvt<char,char,mbstate_t></code> , <code>codecvt<wchar_t,char,mbstate_t></code> |
| messages | <code>messages<char></code> , <code>messages<wchar_t></code> |
| monetary | <code>moneypunct<char></code> , <code>moneypunct<wchar_t></code> <code>moneypunct<char,true></code> , <code>moneypunct<wchar_t,true></code> , <code>money_get<char></code> , <code>money_get<wchar_t></code> <code>money_put<char></code> , <code>money_put<wchar_t></code> |
| numeric | <code>numpunct<char></code> , <code>numpunct<wchar_t></code> , <code>num_get<char></code> , <code>num_get<wchar_t></code> <code>num_put<char></code> , <code>num_put<wchar_t></code> |
| time | <code>time_get<char></code> , <code>time_get<wchar_t></code> , <code>time_put<char></code> , <code>time_put<wchar_t></code> |

An implementation is included for each `facet` template member of a `category`.

Table 7.2 **Required Instantiations**

| Category | Includes Facets |
|----------|---|
| collate | <code>collate_byname<char></code> , <code>collate_byname<wchar_t></code> |
| ctype | <code>ctype_byname<char></code> , <code>ctype_byname<wchar_t></code> |
| messages | <code>messages_byname<char></code> , <code>messages_byname<wchar_t></code> |

| Category | Includes Facets |
|----------|--|
| monetary | moneypunct_byname<char,International>, moneypunct_byname<wchar_t,International>, money_get<C,InputIterator>, money_put<C,OutputIterator> |
| numeric | numpunct_byname<char>, numpunct_byname<wchar_t> num_get<C,InputIterator>, num_put<C,OutputIterator> |
| time | time_get<char,InputIterator>, time_get_byname<char,InputIterator>, time_get<wchar_t,OutputIterator>, time_get_byname<wchar_t,OutputIterator>, time_put<char,OutputIterator>, time_put_byname<char,OutputIterator>, time_put<wchar_t,OutputIterator> time_put_byname<wchar_t,OutputIterator> |

22.1.1.2 Class `Locale::facet`

The class `facet` is the base class for `locale` feature sets.

Listing 7.4 class `locale:: facet` synopsis

```
namespace std {
class locale::facet {
protected:
explicit facet(size_t refs = 0);
virtual ~facet();
private:
facet(const facet&); // not defined
void operator=(const facet&); // not defined };
}
```

22.1.1.3 Class `locale::id`

The class `locale::id` is used for an index for locale facet identification.

Listing 7.5 class locale::id synopsis

```
namespace std {
class locale::id {
public:
id();
private:
void operator=(const id&); // not defined
id(const id&); // not defined };
}
```

22.1.1.2 Locale Constructors**Constructors**

Constructs an object of `locale`.

Prototype `locale() throw();`
 `locale(const locale& other) throw();`
 `explicit locale(const char* std_name);`
 `locale(const locale& other,`
 `const char* std_name, category);`
 `template <class Facet> locale`
 `(const locale& other, Facet* f);`
 `locale(const locale& other,`
 `const locale& one, category cats);`

Prototype `locale("")`

The `" "` `locale` will attempt to read the environment variable `LANG` and create a `locale` with the associated string. If `getenv("LANG")` returns `null`, then `"C"` is used. There is no data file associated with the `"C"` `locale`. The `"C"` `locale` is coded directly into MSL C++.

destructor

Removes a `locale` object.

Prototype `~locale() throw();`

22.1.1.3 Locale Members

Member functions of the class `locale`.

combine

Creates a copy of the `locale` except for the type `Facet` of the argument.

Prototype `template <class Facet> locale combine(const locale& other);`

Return The newly created locale is returned.

name

Returns the name of the `locale`.

Prototype `basic_string<char> name() const;`

Return Returns the name of the locale or "*" if there is none.

22.1.1.4 Locale Operators

The class `locale` has overloaded operators.

operator ==

The locale equality operator.

Prototype `bool operator==(const locale& other) const;`

Return The equality operator returns true if both arguments are the same locale.

operator !=

The locale non-equality operator

Prototype `bool operator!=(const locale& other) const;`

Return The non-equality operator returns true if the locales are not the same.

operator ()

Compares two strings using `use_facet<collate<>>`.

Prototype `template <class charT,
 class Traits, class Allocator>
bool operator()(`

```
const basic_string<charT,Traits,Allocator>& s1,
const basic_string<charT,Traits,Allocator>& s2)
const;
```

Return Returns true if the first argument is less than the second argument for ordering.

22.1.1.5 Locale Static Members

global

Installs a new global locale.

Prototype static locale global(const locale& loc);

Return Global returns the previous locale.

classic

Sets the locale to “C” locale equivalent to locale(“C”).

Prototype static const locale& classic();

Return This function returns the “C” locale.

22.1.2 Locale Globals

Locale has two global functions.

use_facet

Retrieves a reference to a facet of a locale.

Prototype template <class Facet> const Facet& use_facet
(const locale& loc);

Throws a bad_cast exception if has_facet is false.

Return The function returns a facet reference to corresponding to its argument.

has_facet

Tests a locale to see if a facet is present

Prototype template <class Facet> bool has_facet

(const locale& loc) throw();

Return If a facet requested is present `has_facet` returns true.

22.1.3 Convenience Interfaces

Character classification functionality is provided for in the `locale` class.

22.1.3.1 Character Classification

Listing 7.6 Character Classification

```
template <class charT> bool isspace (charT c, const locale& loc);
template <class charT> bool isprint (charT c, const locale& loc);
template <class charT> bool iscntrl (charT c, const locale& loc);
template <class charT> bool isupper (charT c, const locale& loc);
template <class charT> bool islower (charT c, const locale& loc);
template <class charT> bool isalpha (charT c, const locale& loc);
template <class charT> bool isdigit (charT c, const locale& loc);
template <class charT> bool ispunct (charT c, const locale& loc);
template <class charT> bool isxdigit(charT c, const locale& loc);
template <class charT> bool isalnum (charT c, const locale& loc);
template <class charT> bool isgraph (charT c, const locale& loc);
```

In the character classification functions true is returned if the function evaluates to true.

22.1.3.2 Character Conversions

Character conversion functionality is provided for in the `locale` class.

toupper

Converts to upper case character using the locale specified.

Prototype template <class charT> charT toupper
 (charT c, const locale& loc) const;

Return Returns the upper case character.

tolower

Converts to a lower case character using the locale specified.

Prototype template <class charT> charT tolower
(charT c, const locale& loc) const;

Return Returns the lower case character.

22.2 Standard Locale Categories

The standard provides for various locale categories for providing formatting and manipulation of data and streams.

- [“22.2.1 The Ctype Category” on page 166](#)
- [“22.2.2 The Numeric Category” on page 190](#)
- [“22.2.3 The Numeric Punctuation Facet” on page 195](#)
- [“22.2.4 The Collate Category” on page 202](#)
- [“22.2.5 The Time Category” on page 213](#)
- [“22.2.6 The Monetary Category” on page 241](#)
- [“22.2.7 The Message Retrieval Category” on page 260](#)
- [“22.2.8 Program-defined Facets” on page 269](#)

22.2.1 The Ctype Category

The type `ctype_base` provides for const enumerations.

Listing 7.7 Ctype Category

```
namespace std {  
class ctype_base  
{  
public:  
    enum mask  
    {  
        alpha   = 0x0001,  
        blank   = 0x0002,  
        cntrl   = 0x0004,  
        digit   = 0x0008,  
        graph   = 0x0010,
```

```
    lower   = 0x0020,
    print   = 0x0040,
    punct   = 0x0080,
    space   = 0x0100,
    upper   = 0x0200,
    xdigit = 0x0400,
    alnum   = alpha | digit
  };
};

}
```

22.2.1.1 Template Class Ctype

The class `ctype` provides for character classifications.

Listing 7.8 Template Class Ctype Synopsis

```
template <class charT>
class ctype : public locale::facet, public ctype_base {
public:
  typedef charT char_type;
  explicit ctype(size_t refs = 0);
  bool is(mask m, charT c) const;
  const charT* is
    (const charT* low, const charT* high, mask* vec) const;
  const charT* scan_is
    (mask m,const charT* low, const charT* high) const;
  const charT* scan_not
    (mask m,const charT* low, const charT* high) const;
  charT toupper(charT c) const;
  const charT* toupper(charT* low, const charT* high) const;
  charT tolower(charT c) const;
  const charT* tolower(charT* low, const charT* high) const;
  charT widen(char c) const;
  const char* widen
    (const char* low, const char* high, charT* to) const;
  char narrow(charT c, char default) const;
  const charT* narrow
    (const charT* low, const charT*, char default, char* to) const;
  static locale::id id;
protected:
  ~ctype(); //virtual
```

```
virtual bool do_is(mask m, charT c) const;
virtual const charT* do_is
    (const charT* low, const charT* high, mask* vec) const;
virtual const charT* do_scan_is
    (mask m, const charT* low, const charT* high) const;
virtual const charT* do_scan_not
    (mask m, const charT* low, const charT* high) const;
virtual charT do_toupper(charT) const;
virtual const charT* do_toupper
    (charT* low, const charT* high) const;
virtual charT do_tolower(charT) const;
virtual const charT* do_tolower
    (charT* low, const charT* high) const;
virtual charT do_widen(char) const;
virtual const char* do_widen
    (const char* low, const char* high, charT* dest) const;
virtual char do_narrow(charT, char default) const;
virtual const charT* do_narrow
    (const charT* low, const charT* high,
     char default, char* dest) const;
};
```

22.2.1.1 Ctype Members

is

An overloaded function that tests for or places a mask.

Prototype `bool is(mask m, charT c) const;`

Test if `c` matches the mask `m`.

Return Returns true if the char `c` matches mask.

Prototype `const charT* is
 (const charT* low, const charT* high,
 mask* vec) const;`

Fills between the low and high with the mask argument.

Return Returns the second argument.

scan_is

Scans the range for a mask value.

Prototype `const charT* scan_is
 (mask m, const charT* low,
 const charT* high) const;`

Return Returns a pointer to the first character in the range that matches the mask, or the `high` argument if there is no match.

scan_not

Scans the range for exclusion of the mask value.

Prototype `const charT* scan_not(mask m,
 const charT* low, const charT* high) const;`

Return Returns a pointer to the first character in the range that does not match the mask, or the `high` argument if all characters match

toupper

Converts to a character or a range of characters to uppercase.

Prototype `charT toupper(charT) const;
const charT* toupper
 (charT* low, const charT* high) const;`

Return Returns the converted char if it exists.

tolower

Converts to a character or a range of characters to lowercase.

Prototype `charT tolower(charT c) const;
const charT* tolower(charT* low, const charT*
 high) const;`

Return Returns the converted char if it exists.

widen

Converts a `char` or range of `char` type to the `charT` type.

Prototype `charT widen(char c) const;`

```
const char* widen(const char* low, const char*
high, charT* to) const;
```

Return The converted `charT` is returned.

narrow

Converts a `charT` or range of `charT` type to the `char` type.

Prototype `char narrow(charT c, char default) const;`
`const charT* narrow(const charT* low, const`
`charT*, char default,`
`char* to) const;`

Return The converted `char` is returned.

22.2.1.1.2 Ctype Virtual Functions

Virtual functions must be overloaded in the locale.

do_is

Implements `is`.

Prototype `bool do_is`
 `(mask m, charT c) const;`
`const charT* do_is`
 `(const charT* low, const charT* high,`
 `mask* vec) const;`

do_scan_is

Implements `scan_is`.

Prototype `const charT* do_scan_is(mask m,`
 `const charT* low, const charT* high) const;`

do_scan_not

Implements `scan_not`.

Prototype `const charT* do_scan_not(mask m,`
 `const charT* low, const charT* high) const;`

do_toupper

Implements `toupper`.

Prototype `chart do_toupper(chart c) const;`
 `const chartT* do_toupper(chartT* low, const chartT*`
 `high) const;`

do_tolower

Implements `tolower`.

Prototype `chart do_tolower(chart c) const;`
 `const chartT* do_tolower(chartT* low, const chartT*`
 `high) const;`

do_widen

Implements `widen`.

Prototype `chart do_widen(char c) const;`
 `const char* do_widen(const char* low, const char*`
 `high,`
 `chartT* dest) const;`

do_narrow

Implements `narrow`.

Prototype `char do_narrow(chartT c, char default) const;`
 `const chartT* do_narrow(const chartT* low, const`
 `chartT* high,`
 `char default, char* dest) const;`

22.2.1.2 Template class ctype_byname

The template class `ctype_byname` has several responsibilities:

- character classification
- conversion to upper/lower case
- conversion to/from `char`

Listing 7.9 Class ctype_byname synopsis

```
Template class ctype_byname
namespace std { template <class charT>
class ctype_byname : public ctype<charT> {
public:
typedef ctype<charT>::mask mask;
explicit ctype_byname(const char*, size_t refs = 0);
protected:
~ctype_byname();
// virtual
virtual bool do_is(mask m, charT c) const;
virtual const charT* do_is
    (const charT* low, const charT* high, mask* vec) const;
virtual const char* do_scan_is
    (mask m, const charT* low, const charT* high) const;
virtual const char* do_scan_not
    (mask m, const charT* low, const charT* high) const;
virtual charT do_toupper(charT) const;
virtual const charT* do_toupper
    (charT* low, const charT* high) const;
virtual charT do_tolower(charT) const;
virtual const charT* do_tolower
    (charT* low, const charT* high) const;
virtual charT do_widen(char) const;
virtual const char* do_widen
    (const char* low, const char* high, charT* dest) const;
virtual char do_narrow(charT, char default) const;
virtual const charT* do_narrow
    (const charT* low, const charT* high,
     char default, char* dest) const;
};

}
```

Ctype_byname Constructor

Prototype `explicit ctype_byname(const char*,
size_t refs = 0);`

The facet ctype has several responsibilities:

- character classification
- conversion to upper/lower case

- conversion to/from char

The first two of these items can be customized with `ctype_byname`. If you construct `ctype_byname` with a `const char*` that refers to a file, then that file is scanned by `ctype_byname`'s constructor for information to customize character classification, and case transformation tables.

```
ctype_byname<char> ct( "en_US" );
// looks for the file "en_US"
```

If the file "en_US" exists, has ctype data in it, and there are no syntax errors in the data, then `ct` will behave as dictated by that data. If the file exists, but does not have ctype data in it, then the facet will behave as if it were constructed with "C". If the file has ctype data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

For `ctype_byname<char>`, the ctype data section begins with:

```
$ctype_narrow
```

For `ctype_byname<wchar_t>`, the ctype data section begins with:

```
$ctype_wide
```

Classification

The classification table is created with one or more entries of the form:

```
ctype[character1 - character2] =
  ctype_classification |
  ctype_classification | ...
ctype[character] = ctype_classification |
  ctype_classification | ...
```

where character, character1 and character2 are characters represented according to the rules for "[Strings and Characters in Locale Data Files](#)". The characters may appear as normal characters:

```
ctype[a - z]
ctype['a' - 'z']
```

or as octal, hexadecimal or universal:

```
ctype['\101']
ctype['\x41']
ctype['\u41']
```

The usual escape sequences are also recognized: \n, \t, \a, \\, \' and so on.

On the right hand side of the equal sign, ctype_classification is one of:

- alpha
- blank
- cntrl
- digit
- graph
- lower
- print
- punct
- space
- upper
- xdigit

An | can be used to assign a character, or range of characters, more than one classification. These keywords correspond to the names of the enum `ctype_base::mask`, except that alnum is not present. To get `alnum` simply specify "alpha | digit". The keyword blank is introduced, motivated by C99's `isblank` function.

Each of these keywords represent one bit in the `ctype_base::mask`. Thus for each entry into the ctype table, one must specify all attributes that apply. For example, in the "C" locale a-z are represented as:

```
ctype['a' - 'z'] =
xdigit | lower | alpha | graph | print
```

Case Transformation

Case transformation is usually handled by a table that maps each character to itself, except for those characters being transformed -

which are mapped to their transformed counterpart. For example, a lower case map might look like:

```
lower['a'] == 'a'  
lower['A'] == 'a'
```

This is represented in the ctype data as two tables: lower and upper. You can start a map by first specifying that all characters map to themselves:

```
lower['\0' - '\xFF'] = '\0' - '\xFF'
```

You can then override a subrange in this table to specify that 'A' - 'Z' maps to 'a' - 'z':

```
lower['A' - 'Z'] = 'a' - 'z'
```

These two statements have completely specified the lower case mapping for an 8 bit char. The upper case table is similar. For example, here is the specification for upper case mapping of a 16 bit wchar_t in the "C" locale:

```
upper['\0' - '\xFFFF'] = '\0' - '\xFFFF'  
upper['a' - 'z'] = 'A' - 'Z'
```

Below is the complete "C" locale specification for both ctype_byname<char> and ctype_byname<wchar_t>. Note that a "C" data file does not actually exist. But if you provided a locale data file with this information in it, then the behavior would be the same as the "C" locale.

Listing 7.10 Example of "C" Locale

```
$ctype_narrow
ctype['\x00' - '\x08'] = cntrl
ctype['\x09']           = cntrl | space | blank
ctype['\x0A' - '\x0D'] = cntrl | space
ctype['\x0E' - '\x1F'] = cntrl
ctype['\x20']           = space | blank | print
ctype['\x21' - '\x2F'] = punct | graph | print
ctype['\x30' - '\x39'] = digit | xdigit | graph | print
ctype['\x3A' - '\x40'] = punct | graph | print
ctype['\x41' - '\x46'] = xdigit | upper | alpha | graph | print
ctype['\x47' - '\x5A'] = upper | alpha | graph | print
ctype['\x5B' - '\x60'] = punct | graph | print
```

```
ctype['\x61' - '\x66'] = xdigit | lower | alpha | graph | print
ctype['\x67' - '\x7A'] = lower | alpha | graph | print
ctype['\x7B' - '\x7E'] = punct | graph | print
ctype['\x7F']          = cntrl

lower['\0' - '\xFF'] = '\0' - '\xFF'
lower['A' - 'Z']     = 'a' - 'z'

upper['\0' - '\xFF'] = '\0' - '\xFF'
upper['a' - 'z']     = 'A' - 'Z'

$ctype_wide
ctype['\x00' - '\x08'] = cntrl
ctype['\x09']          = cntrl | space | blank
ctype['\x0A' - '\x0D'] = cntrl | space
ctype['\x0E' - '\x1F'] = cntrl
ctype['\x20']           = space | blank | print
ctype['\x21' - '\x2F'] = punct | graph | print
ctype['\x30' - '\x39'] = digit | xdigit | graph | print
ctype['\x3A' - '\x40'] = punct | graph | print
ctype['\x41' - '\x46'] = xdigit | upper | alpha | graph | print
ctype['\x47' - '\x5A'] = upper | alpha | graph | print
ctype['\x5B' - '\x60'] = punct | graph | print
ctype['\x61' - '\x66'] = xdigit | lower | alpha | graph | print
ctype['\x67' - '\x7A'] = lower | alpha | graph | print
ctype['\x7B' - '\x7E'] = punct | graph | print
ctype['\x7F']          = cntrl

lower['\0' - '\xFFFF'] = '\0' - '\xFFFF'
lower['A' - 'Z']       = 'a' - 'z'

upper['\0' - '\xFFFF'] = '\0' - '\xFFFF'
upper['a' - 'z']       = 'A' - 'Z'
```

22.2.1.3 Ctype Specializations

The category `ctype` has various specializations to help localization.

Listing 7.11 Ctype Specializations synopsis

```
namespace std {
template <> class ctype<char>
```

```
: public locale::facet, public ctype_base {
public:
typedef char char_type;
explicit
ctype(const mask* tab = 0, bool del = false,
size_t refs = 0);
bool is(mask m, char c) const;
const char* is(const char* low, const char* high, mask* vec)
const;
const char* scan_is (mask m,
const char* low, const char* high) const;
const char* scan_not(mask m,
const char* low, const char* high) const;
char toupper(char c) const;
const char* toupper(char* low, const char* high) const;
char tolower(char c) const;
const char* tolower(char* low, const char* high) const;
char widen(char c) const;
const char* widen
    (const char* low, const char* high, char* to) const;
char narrow(char c, char dfault) const;
const char* narrow
    (const char* low, const char* high,
     char dfault, char* to) const;
static locale::id id;
static const size_t table_size;
protected:
const mask* table() const throw();
static const mask* classic_table() throw();
~ctype(); //virtual
virtual char do_toupper(char c) const;
virtual const char* do_toupper
    (char* low, const char* high) const;
virtual char do_tolower(char c) const;
virtual const char* do_tolower
    (char* low, const char* high) const;
virtual char do_widen(char c) const;
virtual const char* do_widen
    (const char* low, const char* high, char* to) const;
virtual char do_narrow(char c, char dfault) const;
virtual const char* do_narrow
    (const char* low, const char* high,
```

```
    char dfault, char* to) const; };
```

The class `ctype<char>` has four protected data members:

- `const mask* __table_;`
- `const unsigned char* __lower_map_;`
- `const unsigned char* __upper_map_;`
- `bool __owns_;`

Each of the pointers refers to an array of length

`ctype<char>::table_size`. The destructor `~ctype<char>()` will delete `__table_` if `__owns_` is true, but it will not delete `__lower_map_` and `__upper_map_`. The derived class destructor must take care of deleting these pointers if they are allocated on the heap (`ctype<char>` will not allocate these pointers). A derived class can set these pointers however it sees fit, and have `ctype<char>` implement all of the rest of the functionality.

The class `ctype<wchar_t>` has three protected data members:

```
Metrowerks::range_map<charT, ctype_base::mask> __table_;  
Metrowerks::range_map<charT, charT> __lower_map_;  
Metrowerks::range_map<charT, charT> __upper_map_;
```

The class `range_map` works much like the tables in `ctype<char>` except that they are sparse tables. This avoids having tables of length `0xFFFF`. These tables map the first template parameter into the second.

Listing 7.12 The range_map interface

```
template <class T, class U>  
class range_map  
{  
public:  
    U operator[](const T& x) const;  
    void insert(const T& x1, const T& x2, const U& y1, const U& y2);  
    void insert(const T& x1, const T& x2, const U& y1);  
    void insert(const T& x1, const U& y1);
```

```
    void clear();  
};
```

When constructed, the `range_map` implicitly holds a map of all `T` that map to `U()`. Use of the insert methods allows exceptions to that default mapping. For example, the first insert method maps the range `[x1 - x2]` into `[y1 - y2]`. The second insert method maps the `x`-range into a constant: `y1`. And the third insert method maps the single `T(x1)` into `U(y1)`. The method `clear()` brings the `range_map` back to the default setting: all `T` map into `U()`.

A class derived from `ctype<wchar_t>` can fill `__table_`, `__lower_map_` and `__upper_map_` as it sees fit, and allow the base class to query these tables. For an example see `ctype_byname<wchar_t>`.

22.2.1.3.1 Specialized Ctype Constructor and Destructor

Specialized `ctype<char>` and `ctype<wchar_t>` constructors and destructors.

Constructor

Constructs a `ctype` object.

Prototype `explicit ctype
(const mask* tbl = 0, bool del = false,
size_t refs = 0);`

destructor

Removes a `ctype` object.

Prototype `~ctype();`

22.2.1.3.2 Specialized Ctype Members

Specialized `ctype<char>` and `ctype<wchar_t>` member functions.

Prototype `bool is(mask m, char c) const;
const char* is(const char* low, const char* high,
mask* vec) const;`

```
Prototype const char* scan_is(mask m,
                           const char* low, const char* high) const;

Prototype const char* scan_not(mask m,
                           const char* low, const char* high) const;

Prototype char toupper(char c) const;
const char* toupper(char* low, const char* high)
const;

Prototype char tolower(char c) const;
const char* tolower(char* low, const char* high)
const;

Prototype char widen(char c) const;
const char* widen(const char* low, const char*
high,
char* to) const;

Prototype char narrow(char c, char /*default*/) const;
const char* narrow(const char* low, const char*
high,
char /*default*/, char* to) const;

Prototype const mask* table() const throw();
```

22.2.1.3.3 Ctype<Char> Static Members

Specialized ctype<char> static members. are provided.

classic_table

Determines the classification of characters in the "C" locale.

```
Prototype static const mask* classic_table() throw();

Return Returns to a table that represents the classification in a "C" locale.
```

22.2.1.3.4 Ctype<Char> Virtual Functions

Specialize ctype<char> virtual member functions are identical functionality to "[22.2.1.1.2 Ctype Virtual Functions](#)" on page 170.

```
Prototype char do_toupper(char) const;
const char* do_toupper
```

```
(char* low, const char* high) const;  
  
Prototype char do_tolower(char) const;  
const char* do_tolower  
(char* low, const char* high) const;  
  
Prototype virtual char do_widen(char c) const;  
virtual const char* do_widen  
(const char* low, const char* high,  
char* to) const;  
  
Prototype virtual char do_narrow(char c, char default) const;  
virtual const char* do_narrow  
(const char* low, const char* high,  
char default, char* to) const;
```

22.2.1.4 Class ctype_byname<char>

A specialization of ctype_byname of type char.

Listing 7.13 Ctype_byname<char> Synopsis

```
namespace std {  
template <> class ctype_byname<char> : public ctype<char> {  
public:  
    explicit ctype_byname(const char*, size_t refs = 0);  
protected:  
    ~ctype_byname(); // virtual  
    virtual char do_toupper(char c) const;  
    virtual const char* do_toupper  
(char* low, const char* high) const;  
    virtual char do_tolower(char c) const;  
    virtual const char* do_tolower  
(char* low, const char* high) const;  
    virtual char do_widen(char c) const;  
    virtual const char* do_widen  
(char* low, const char* high, char* to) const;  
    virtual char do_widen(char c) const;  
    virtual const char* do_widen(char* low, const char* high) const;  
};  
}
```

Ctype_byname<char> Constructor

Prototype `explicit ctype_byname(const char*, size_t refs = 0);`

The facet ctype has several responsibilities:

- character classification
- conversion to upper/lower case
- conversion to/from char

For a full and complete description of this facet specialization see [“Ctype_byname Constructor” on page 172](#) which list the process in greater detail.

22.2.1.5 Template Class Codecvt

A class used for converting one character encoded types to another. For example, from wide character to multibyte character sets.

Listing 7.14 Template Class Codevect Synopsis

```
namespace std {
class codecvt_base {
public:
enum result
    { ok, partial, error, noconv };
};

template <class internT, class externT, class stateT>
class codecvt : public locale::facet, public codecvt_base {
public:
typedef internT intern_type;
typedef externT extern_type;
typedef stateT state_type;
explicit codecvt(size_t refs = 0)
result out
    (stateT& state, const internT* from, const internT* from_end,
     const internT*& from_next, externT* to, externT* to_limit,
     externT*& to_next) const;
result unshift
    (stateT& state, externT* to, externT* to_limit,
```

```
externT*& to_next) const;
result in
    (stateT& state, const externT* from, const externT* from_end,
     const externT*& from_next, internT* to, internT* to_limit,
     internT*& to_next) const;
int encoding() const throw();
bool always_noconv() const throw();
int length
    (const stateT&, const externT* from, const externT* end,
     size_t max) const;
int max_length() const throw();
static locale::id id;
protected:
~codecvt(); //virtual
virtual result do_out
    (stateT& state, const internT* from, const internT* from_end,
     const internT*& from_next, externT* to, externT* to_limit,
     externT*& to_next) const;
virtual result do_in
    (stateT& state, const externT* from, const externT* from_end,
     const externT*& from_next, internT* to, internT* to_limit,
     internT*& to_next) const;
virtual result do_unshift
    (stateT& state, externT* to, externT* to_limit,
     externT*& to_next) const;
virtual int do_encoding() const throw();
virtual bool do_always_noconv() const throw();
virtual int do_length
    (const stateT&, const externT* from, const externT* end,
     size_t max) const;
virtual int do_max_length() const throw();
};
```

22.2.1.5.1 Codecvt Members

Member functions of the codecvt class.

out

Convert internal representation to external.

Prototype `result out(stateT& state, const internT* from, const internT* from_end, const internT*& from_next, externT* to, externT* to_limit, externT*& to_next) const;`

unshift

Converts the shift state.

Prototype `result unshift(stateT& state, externT* to, externT* to_limit, externT*& to_next) const;`

in

Converts external representation to internal.

Prototype `result in(stateT& state, const externT* from, const externT* from_end, const externT*& from_next, internT* to, internT* to_limit, internT*& to_next) const;`

always_noconv

Determines if no conversion is ever done.

Prototype `bool always_noconv() const throw();`

Return Returns true if no conversion will be done.

length

Determines the length between two points.

Prototype `int length(stateT& state, const externT* from, const externT* from_end, size_t max) const;`

Return The distance between two points is returned.

max_length

Determines the length necessary for conversion.

Prototype `int max_length() const throw();`

Return The number of elements to convert from `externT` to `internT` is returned.

22.2.1.5.2 `Codecvt` Virtual Functions

Virtual functions for `codecvt` implementation.

Prototype `result do_out(stateT& state, const internT* from,
 const internT* from_end,
 const internT*& from_next, externT* to,
 externT* to_limit, externT*& to_next) const;`

Implements `out`.

Return The result is returned as a value as in [“Convert Result Values” on page 186](#).

Prototype `result do_in(stateT& state, const externT* from,
 const externT* from_end,
 const externT*& from_next, internT* to,
 internT* to_limit, internT*& to_next) const;`

Implements `in`.

Return The result is returned as a value as in [“Convert Result Values” on page 186](#).

Prototype `result do_unshift(stateT& state,
 externT* to, externT* to_limit, externT*& to_next)
 const;`

Implements `unshift`.

Return The result is returned as a value as in [“Convert Result Values” on page 186](#).

Prototype `int do_encoding() const throw();`

Implements `encoding`.

Prototype `bool do_always_noconv() const throw();`

Implements `always_noconv`.

Prototype `int do_length(stateT& state, const externT* from,
 const externT* from_end, size_t max) const;`

Implements `length`.

Prototype int do_max_length() const throw();

Implements max_length.

Table 7.3 Convert Result Values

| Value | Meaning |
|---------|--|
| error | Encountered a from_type character it could not convert |
| noconv | No conversion was needed |
| ok | Completed the conversion |
| partial | Not all source characters converted |

22.2.1.6 Template Class `Codecvt_byname`

The facet `codecvt` is responsible for translating internal characters (`wchar_t`) to/from external char's in a file.

Listing 7.15 Template Class `Codecvt_byname` Synopsis

```
namespace std {
template <class internT, class externT, class stateT>
class codecvt_byname : public codecvt<internT, externT, stateT> {
public:
    explicit codecvt_byname(const char*, size_t refs = 0);
protected:
    ~codecvt_byname(); // virtual
    virtual result do_out
        (stateT& state, const internT* from, const internT* from_end,
         const internT*& from_next, externT* to, externT* to_limit,
         externT*& to_next) const;
    virtual result do_in
        (stateT& state, const externT* from, const externT* from_end,
         const externT*& from_next, internT* to, internT* to_limit,
         internT*& to_next) const;
    virtual result do_unshift
        (stateT& state, externT* to, externT* to_limit,
         externT*& to_next) const;
    virtual int do_encoding() const throw();
    virtual bool do_always_noconv() const throw();
}
```

```
virtual int do_length
    (const stateT&, const externT* from, const externT* end,
     size_t max) const;
virtual result do_unshift
    (stateT& state, externT* to, externT* to_limit,
     externT*& to_next) const;
virtual int do_max_length() const throw();
};

}
```

There are several techniques for representing a series of `wchar_t`'s with a series of `char`'s. The `codecvt_byname` facet can be used to select among several of the encodings. If you construct `codecvt_byname` with a `const char*` that refers to a file, then that file is scanned by `codecvt_byname`'s constructor for information to customize the encoding.

```
codecvt_byname<wchar_t, char, std::mbstate_t>
cvt( "en_US" );
```

If the file "en_US" exists, has `codecvt` data in it, and there are no syntax errors in the data, then `cvt` will behave as dictated by that data. If the file exists, but does not have `codecvt` data in it, then the facet will behave as if it were constructed with "C". If the file has `codecvt` data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

For `codecvt_byname<char, char, mbstate_t>`, the `codecvt` data section begins with:

```
$codecvt_narrow
```

For `codecvt_byname<wchar_t, char, mbstate_t>`, the `codecvt` data section begins with:

```
$codecvt_wide
```

Although `$codecvt_narrow` is a valid data section, it really does not do anything. The `codecvt_byname<char, char, mbstate_t>` facet does not add any functionality beyond `codecvt<char, char, mbstate_t>`. This facet is a degenerate case of `noconv` (no conversion). This can be represented in the locale data file as:

```
$codecvt_narrow  
noconv
```

The facet `codecvt_byname<wchar_t, char, mbstate_t>` is much more interesting. After the data section introduction (`$codecvt_wide`), one of these keywords can appear:

- noconv
- UCS-2
- JIS
- Shift-JIS
- EUC
- UTF-8

These keywords will be parsed as strings according to the rules for [“Strings and Characters in Locale Data Files” on page 153](#).

Codecvt_byname Keywords

These Codecvt_byname keywords will be parsed as strings according to the rules for entering strings in locale data files.

noconv This conversion specifies that the base class should handle the conversion. The MSL C++ implementation of `codecvt<wchar_t, char, mbstate_t>` will I/O all bytes of the `wchar_t` in native byte order.

UCS-2 This encoding input and outputs the two lowest order bytes of the `wchar_t`, high byte first. For a big-endian, 16 bit `wchar_t` platform, this encoding is equivalent to noconv.

JIS This is an early encoding used by the Japanese to represent a mixture of ASCII and a subset of Kanji.

Shift-JIS Another early encoding used by the Japanese to represent a mixture of ASCII and a subset of Kanji.

EUC Extended Unix Code.

UTF-8 A popular Unicode multibyte encoding. For example

```
$codecvt_wide  
UTF-8
```

specifies that `codecvt_byname<wchar_t, char, mbstate_t>` will implement the `UTF-8` encoding scheme. If this data is in a file

called “en_US”, then the following program can be used to output a `wchar_t` string in UTF-8 to a file:

Listing 7.16 Example of Writing a `wchar_t` String in utf-8 to a File:

```
#include <locale>
#include <fstream>

int main()
{
    std::locale loc("en_US");
    std::wofstream out;
    out.imbue(loc);
    out.open("test.dat");
    out << L"This is a test \x00DF";
}
```

The binary contents of the file is (in hex):

54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 C3 9F

Without the UTF-8 encoding, the default encoding will take over (all `wchar_t` bytes in native byte order):

```
#include <fstream>

int main()
{
    std::wofstream out("test.dat");
    out << L"This is a test \x00DF";
}
```

On a big-endian machine with a 2 byte `wchar_t`
the resulting file in hex is:

00 54 00 68 00 69 00 73 00 20 00 69 00 73 00 20
00 61 00 20 00 74 00 65 00 73 00 74 00 20 00 DF

Extending `codecvt` by derivation

The facet `codecvt` can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons. Naturally, you can derive from `codecvt` and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the MSL C++ specific classes used to implement

`codecvt_byname`. There are five implementation specific facets that you can use in place of `codecvt` or `codecvt_byname` to get the behavior of one of the five encodings:

- `__ucs_2`
- `__jis`
- `__shift_jis`
- `__euc`
- `__utf_8`

These classes are templated simply on the internal character type (and should be instantiated with `wchar_t`). The external character type is implicitly `char`, and the state type is implicitly `mbstate_t`.

Listing 7.17 An example use of `__utf_8` is:

```
#include <locale>
#include <fstream>

int main()
{
    std::locale loc(std::locale(), new std::__utf_8<wchar_t>());
    std::wofstream out;
    out.imbue(loc);
    out.open("test.dat");
    out << L"This is a test \x00DF";
}
```

Result

54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 C3 9F

This locale (and wofstream) will have all of the facets of the current global locale except that its `codecvt<wchar_t, char, mbstate_t>` will use the UTF-8 encoding scheme. Thus the binary contents of the file is (in hex):

22.2.2 The Numeric Category

A class for numeric formatting and manipulation for locales.

22.2.2.1 Template Class `Num_get`

A class for formatted numeric input.

Listing 7.18 Template Class num_get Synopsis

```
namespace std {
template <class charT, class InputIterator =
istreambuf_iterator<charT> >
class num_get : public locale::facet {
public:
typedef charT char_type;
typedef InputIterator iter_type;
explicit num_get(size_t refs = 0);
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, bool& v) const;
iter_type get(iter_type in, iter_type end, ios_base& ,
    ios_base::iostate& err, long& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned short& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned int& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, unsigned long& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, float& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, double& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, long double& v) const;
iter_type get(iter_type in, iter_type end, ios_base&,
    ios_base::iostate& err, void*& v) const;
static locale::id id;
protected:
~num_get(); //virtual
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, bool& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, long& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, unsigned short& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, unsigned int& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, unsigned long& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
```

```
    ios_base::iostate& err, float& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, double& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, long double& v) const;
virtual iter_type do_get(iter_type, iter_type, ios_base&,
    ios_base::iostate& err, void*& v) const;
};

}
```

22.2.2.1 Num_get Members

The class num_get includes specific functions for parsing and formatting of numbers.

get

The function `get` is overloaded for un-formatted input.

```
Prototype iter_type get(iter_type in, iter_type end,
    ios_base& str,ios_base::iostate& err,
    long& val) const;
iter_type get(iter_type in, iter_type end,
    ios_base& str,ios_base::iostate& err,
    unsigned short& val) const;
iter_type get(iter_type in, iter_type end,
    ios_base& str,ios_base::iostate& err,
    unsigned int& val) const;
iter_type get(iter_type in, iter_type end,
    ios_base& str,ios_base::iostate& err,
    unsigned long& val) const;
iter_type get(iter_type in, iter_type end,
    ios_base& str,ios_base::iostate& err,
    short& val) const;
iter_type get(iter_type in, iter_type end,
    ios_base& str,ios_base::iostate& err,
    double& val) const;
iter_type get(iter_type in, iter_type end,
    ios_base& str,ios_base::iostate& err,
    long double& val) const;
iter_type get(iter_type in, iter_type end,
    ios_base& str,ios_base::iostate& err,
```

void*& val) const;

Return returns and iterator type.

22.2.2.1.2 Num_get Virtual Functions

Prototype iter_type do_get(iter_type in, iter_type end,
 ios_base& str,ios_base::iostate& err,
 long& val) const;
iter_type do_get(iter_type in, iter_type end,
 ios_base& str, ios_base::iostate& err,
 unsigned short& val) const;
iter_type do_get(iter_type in, iter_type end,
 ios_base& str,ios_base::iostate& err,
 nsined int& val) const;
iter_type do_get(iter_type in, iter_type end,
 ios_base& str,ios_base::iostate& err,
 unsigned long& val) const;
iter_type do_get(iter_type in, iter_type end,
 ios_base& str,ios_base::iostate& err,
 float& val) const;
iter_type do_get(iter_type in, iter_type end,
 ios_base& str,ios_base::iostate& err,
 double& val) const;
iter_type do_get(iter_type in, iter_type end,
 ios_base& str,ios_base::iostate& err,
 long double& val) const;
iter_type do_get(iter_type in, iter_type end,
 ios_base& str,ios_base::iostate& err,
 void*& val) const;

Prototype iiter_type do_get(iter_type in, iter_type end,
 ios_base& str,
 ios_base::iostate& err, bool& val) const;

Implements the relative versions of get.

22.2.2 Template Class Num_put

A class for formatted numeric output.

Listing 7.19 Template Class num_put Synopsis

```
namespace std {
template <class charT, class OutputIterator =
ostreambuf_iterator<charT> >
class num_put : public locale::facet {
public:
typedef charT char_type;
typedef OutputIterator iter_type;
explicit num_put(size_t refs = 0);
iter_type put(iter_type s, ios_base& f,
    char_type fill, bool v) const;
iter_type put(iter_type s, ios_base& f,
    char_type fill, long v) const;
iter_type put(iter_type s, ios_base& f,
    char_type fill, unsigned long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill,
    double v) const;
iter_type put(iter_type s, ios_base& f, char_type fill,
    long double v) const;
iter_type put(iter_type s, ios_base& f, char_type fill,
    const void* v) const;
static locale::id id;
protected:
~num_put(); //virtual
virtual iter_type do_put(iter_type, ios_base&, char_type fill,
bool v) const;
virtual iter_type do_put
    (iter_type, ios_base&, char_type fill, long v) const;
virtual iter_type do_put
    (iter_type, ios_base&, char_type fill, unsigned long) const;
virtual iter_type do_put
    (iter_type, ios_base&, char_type fill, double v) const;
virtual iter_type do_put
    (iter_type, ios_base&, char_type fill, long double v) const;
virtual iter_type do_put
    (iter_type, ios_base&, char_type fill, const void* v) const;
};
```

22.2.2.1 Num_put Members

The class `num_put` includes specific functions for parsing and formatting of numbers.

put

The function `put` is overloaded for un-formatted output.

Prototype `iter_type put(iter_type out, ios_base& str,
 char_type fill, bool val) const;`
`iter_type put(iter_type out, ios_base& str,
 char_type fill, long val) const;`
`iter_type put(iter_type out, ios_base& str,
 char_type fill, unsigned long val) const;`
`iter_type put(iter_type out, ios_base& str,
 char_type fill, double val) const;`
`iter_type put(iter_type out, ios_base& str,
 char_type fill, long double val) const;`
`iter_type put(iter_type out, ios_base& str,
 char_type fill const void* val) const;`

22.2.2.2 Num_put Virtual Functions

Implementation functions for `put`.

Prototype `iter_type do_put(iter_type out, ios_base& str,
 char_type fill, bool val) const;`
`iter_type do_put(iter_type out, ios_base& str,
 char_type fill, long val) const;`
`iter_type do_put(iter_type out, ios_base& str,
 char_type fill, unsigned long val) const;`
`iter_type do_put(iter_type out, ios_base& str,
 char_type fill, double val) const;`
`iter_type do_put(iter_type out, ios_base& str,
 char_type fill, long double val) const;`
`iter_type do_put(iter_type out, ios_base& str,
 char_type fill const void* val) const;`

22.2.3 The Numeric Punctuation Facet

A facet for numeric punctuation in formatting and parsing.

22.2.3.1 Template Class `Numpunct`

A class for numeric punctuation conversion.

Listing 7.20 Template Class `numpunct` Synopsis

```
namespace std {
template <class charT>
class numpunct : public locale::facet {
public:
typedef charT char_type;
typedef basic_string<charT> string_type;
explicit numpunct(size_t refs = 0);
char_type decimal_point() const;
char_type thousands_sep() const;
string grouping() const;
string_type truename() const;
string_type falsename() const;
static locale::id id;
protected:
~numpunct(); //virtual
virtual char_type do_decimal_point() const;
virtual char_type do_thousands_sep() const;
virtual string do_grouping() const;
virtual string_type do_truename() const;
virtual string_type do_falsename() const;
};
```

22.2.3.1.1 `Numpunct` Members

The template class `numpunct` provides various functions for punctuation localizations.

decimal_point

Determines the character used for a decimal point.

Prototype `char_type decimal_point() const;`

Return Returns the character used for a decimal point.

thousands_sep

Determines the character used for a thousand separator.

Prototype `char_type thousands_sep() const;`

Return Returns the character used for the thousand separator.

grouping

Describes the thousand separators.

Prototype `string grouping() const;`

Return Returns a string describing the thousand separators.

truename

Determines the localization for “true”.

Prototype `string_type truename() const;`

Return Returns a string describing the localization of the word “true”.

falseename

Determines the localization for “false”.

Prototype `string_type falseename() const;`

Return Returns a string describing the localization of the word “false”.

22.2.3.1.2 numpunct virtual functions

Implementation of the public functions.

Prototype `char_type do_decimal_point() const;`

Implements `decimal_point`.

Prototype `string_type do_thousands_sep() const;`

Implements `thousands_sep`.

Prototype `string do_grouping() const;`

Implements `grouping`.

Prototype `string_type do_truename() const;`

Implements `truename`.

Prototype `string_type do_falsename() const;`

Implements `falsename`.

22.2.3.2 Template Class `Numpunct_byname`

The facet `numpunct` is responsible for specifying the punctuation used for parsing and formatting numeric quantities. You can specify the decimal point character, the thousands separator, the grouping, and the spelling of true and false. If you construct `numpunct_byname` with a `const char*` that refers to a file, then that file is scanned by `numpunct_byname`'s constructor for information to customize the encoding.

```
numpunct_byname<char> np( "en_US" );
```

If the file "en_US" exists, has `numpunct` data in it, and there are no syntax errors in the data, then `np` will behave as dictated by that data. If the file exists, but does not have `numpunct` data in it, then the facet will behave as if it were constructed with "C". If the file has `numpunct` data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

For `numpunct_byname<char>`, the `numpunct` data section begins with:

```
$numeric_narrow
```

For `numpunct_byname<wchar_t>`, the `numpunct` data section begins with:

```
$numeric_wide
```

The syntax for both the narrow and wide data sections is the same. There are five keywords that allow you to specify the different parts of the `numpunct` data:

1. [decimal_point](#)
2. [thousands_sep](#)
3. [grouping](#)
4. [false_name and true_name](#)

You enter data with one of these keywords, followed by an equal sign '=' , and then the data. You can specify any or all of the 5 keywords. Data not specified will default to that of the "C" locale. The first two keywords (decimal_point and thousands_sep) have character data associated with them. See the rules for "[Character Syntax](#)" on page 153 for details. The last three keywords have string data associated with them. See the rules for "[String Syntax](#)" on page 155.

Listing 7.21 Class Numpunct_byname Synopsis

```
namespace std {
template <class charT>
class numpunct_byname : public numpunct<charT> {
// this class is specialized for char and wchar_t.
public:
typedef charT char_type;
typedef basic_string<charT> string_type;
explicit numpunct_byname(const char*, size_t refs = 0);
protected:
~numpunct_byname(); // virtual
virtual char_type do_decimal_point() const;
virtual char_type do_thousands_sep() const;
virtual string do_grouping() const;
virtual string_type do_truename() const; // for bool
virtual string_type do_falsename() const; // for bool
};
```

decimal_point The decimal point data is a single character, as in:

```
decimal_point = '.'
```

thousands_sep The character to be used for the thousands separator is specified with thousands_sep, as in:

```
thousands_sep = ','
```

grouping The grouping string specifies the number of digits to group, going from right to left. For example, the grouping: 321 means that the number 12345789 would be printed as in:

```
1,2,3,4,56,789
```

The above grouping string can be specified as:

```
grouping = 321
```

A grouping string of “0” or “” means: do not group.

false_name and true_name The names of false and true can be specified with false_name and true_name. For example:

```
false_name = "no way"  
true_name = sure
```

Numeric_wide

For \$numeric_wide, wide characters can be represented with the hex or universal format (e.g. “\u64D0”).

Listing 7.22 Example of Numeric_wide use

Given the data file:

```
$numeric_narrow  
decimal_point = ','  
thousands_sep = '.'  
grouping = 32  
false_name = nope  
true_name = sure  
  
#include <sstream>  
#include <locale>  
#include <iostream>  
  
int main()  
{  
    std::locale loc("my_loc");  
    std::cout.imbue(loc);  
    std::istringstream in("1.23.456 nope 1.23.456,789");  
    in.imbue(loc);  
    in >> std::boolalpha;  
    long i;  
    bool b;  
    double d;  
    in >> i >> b >> d;  
    std::cout << i << '\n'  
        << std::boolalpha << !b << '\n'  
        << std::fixed << d;  
}
```

The output is:
1.23.456
sure
1.23.456,789000

Extending numpunct by derivation

It is easy enough to derive from numpunct and override the virtual functions in a portable manner. But numpunct also has a non-standard protected interface that you can take advantage of if you wish. There are five protected data members:

Prototype `char_type __decimal_point_;`
 `char_type __thousands_sep_;`
 `string __grouping_;`
 `string_type __truename_;`
 `string_type __falsename_;`

A derived class could set these data members in its constructor to whatever is appropriate, and thus not need to override the virtual methods.

Listing 7.23 Example of numpunct<char>

```
struct mypunct: public std::numpunct<char>
{
    mypunct();
};

mypunct::mypunct()
{
    __decimal_point_ = ',';
    __thousands_sep_ = '.';
    __grouping_ = "\3\2";
    __falsename_ = "nope";
    __truename_ = "sure";
}

int main()
{
    std::locale loc(std::locale(), new mypunct);
```

```
    std::cout.imbue(loc);
    // ...
}
```

22.2.4 The Collate Category

A class used for the comparison and manipulation of strings.

22.2.4.1 Template Class Collate

```
namespace std {
template <class charT>
class collate : public locale::facet {
public:
typedef charT char_type;
typedef basic_string<charT> string_type;
explicit collate(size_t refs = 0);
int compare(const charT* low1, const charT* high1,
const charT* low2, const charT* high2) const;
string_type transform(const charT* low, const charT* high) const;
long hash(const charT* low, const charT* high) const;
static locale::id id;
protected:
~collate(); //virtual
virtual int do_compare(const charT* low1, const charT* high1,
const charT* low2, const charT* high2) const;
virtual string_type do_transform
(const charT* low, const charT* high) const;
virtual long do_hash (const charT* low, const charT* high) const;
};

}
```

22.2.4.1.1 Collate Members

Member functions used for comparison and hashing of strings.

compare

Lexicographical comparison of strings.

Prototype int compare(const charT* low1, const charT* high1,
const charT* low2, const charT* high2) const;

Return A value of 1 is returned if the first is lexicographically greater than the second. A value of negative 1 is returned if the second is greater than the first. A value of zero is returned if the strings are the same.

transform

Provides a string object to be compared to other transformed strings.

Prototype `string_type transform
(const charT* low, const charT* high) const;`

Remarks The `transform` member function is used for comparison of a series of strings.

Return Returns a string for comparison.

hash

Determines the hash value for the string.

Prototype `long hash(const charT* low, const charT* high)
const;`

Return Returns the hash value of the string

22.2.4.1.2 Collate Virtual Functions

Localized implementation functions for public collate member functions.

Prototype `int do_compare
(const charT* low1, const charT* high1,
 const charT* low2, const charT* high2) const;`

Implements `compare`.

Prototype `string_type do_transform(const charT* low, const
charT* high) const;`

Implements `transform`

Prototype `long do_hash(const charT* low, const charT* high) const;`

Implements `hash`.

22.2.4.2 Template Class Collate_byname

The facet collate is responsible for specifying the sorting rules used for sorting strings. The base class collate does a simple lexical comparison on the binary values in the string. `collate_byname` can perform much more complex comparisons that are based on the Unicode sorting algorithm. If you construct `collate_byname` with a `const char*` that refers to a file, then that file is scanned by `collate_byname`'s constructor for information to customize the collation rules.

```
collate_byname<char> col( "en_US" );
```

If the file "en_US" exists, has collate data in it, and there are no syntax errors in the data, then `col` will behave as dictated by that data. If the file exists, but does not have collate data in it, then the facet will behave as if it were constructed with "C". If the file has collate data in it, but there is a syntax error in the data, or if the file does not exist, then a `std::runtime_error` is thrown.

Listing 7.24 Class Collate_byname Synopsis

```
namespace std {
template <class charT>
class collate_byname : public collate<charT> {
public:
typedef basic_string<charT> string_type;
explicit collate_byname(const char*, size_t refs = 0);
protected:
~collate_byname(); // virtual
virtual int do_compare(const charT* low1, const charT* high1,
const charT* low2, const charT* high2) const;
virtual string_type do_transform
(const charT* low, const charT* high) const;
virtual long do_hash (const charT* low, const charT* high) const;
};
```

Collate Data Section

For `collate_byname<char>`, the collate data section begins with:

```
$collate_narrow
```

For `collate_byname<wchar_t>`, the collate data section begins with:

```
$collate_wide
```

The syntax for both the narrow and wide data sections is the same. The data consists of a single string that has a syntax very similar to Java's RuleBasedCollator class. This syntax is designed to provide a level three sorting key consistent with the sorting algorithm specified by the Unicode collation algorithm.

Rule Format

The collation string rule is composed of a list of collation rules, where each rule is of three forms:

```
< modifier >
< relation > < text-argument >
< reset >      < text-argument >
```

Text-Argument: A text-argument is any sequence of characters, excluding special characters (that is, common whitespace characters and rule syntax characters. If those characters are desired, you can put them in single quotes (e.g. ampersand => '&').

Modifier: There is a single modifier which is used to specify that all accents (secondary differences) are backwards.

'@' : Indicates that accents are sorted backwards, as in French.

Relation: The relations are the following:

```
'<' : Greater, as a letter difference (primary)
';' : Greater, as an accent difference (secondary)
',' : Greater, as a case difference (tertiary)
'=' : Equal
```

Reset: There is a single reset which is used primarily for expansions, but which can also be used to add a modification at the end of a set of rules.

'&': Indicates that the next rule follows the position to where the reset text-argument would be sorted.

Relational

The relational allow you to specify the relative ordering of characters. For example, the following string expresses that 'a' is less than 'b' which is less than 'c':

```
"< a < b < c"
```

For the time being, just accept that a string should start with '<'. That rule will be both relaxed and explained later.

Many languages (including English) consider 'a' < 'A', but only as a tertiary difference. And such minor differences are not considered significant unless more important differences are found to be equal. For example consider the strings:

aa
Aa
ab

Since 'a' < 'A', then "aa" < "Aa". But "Aa" < "ab" because the difference between the second characters 'a' and 'b' is more important than the difference between the first characters 'A' and 'a'. This type of relationship can be expressed in the collation rule with:

```
"< a, A < b, B < c, C"
```

This says that 'a' is less 'A' by a tertiary difference, and then 'b' and 'B' are greater than 'a' and 'A' by a primary difference (and similarly for 'c' and 'C').

Accents are usually considered secondary differences. For example, lower case e with an acute accent might be considered to be greater than lower case e, but only by a secondary difference. This can be represented with a semicolon like:

```
"... < e, E ; é, É < ..."
```

Note that characters can be entered in hexadecimal or universal format. They can also be quoted with single quotes (for example 'a'). If it is ambiguous whether a character is a command or a text argument, adding quotes specifies that it is a text argument.

Characters not present in a rule are implicitly ordered after all characters that do appear in a rule.

French collation

Normally primary, secondary and tertiary differences are considered left to right. But in French, secondary differences are considered right to left. This can be specified in the rule string by starting it with '@':

```
"@ ... < e, E ; é, É < ..."
```

Contraction

Some languages sort groups of letters as a single character. Consider the two strings: "acha" and "acia". In English they are sorted as just shown. But Spanish requires "ch" to be considered a single character that is sorted after 'c' and before 'd'. Thus the order in Spanish is reversed relative to English (that is "acia" < "acha"). This can be specified like:

```
"... < c < ch < d ..."
```

Taking case into account, you can expand this idea to:

```
"... < c, C < ch, CH, Ch, CH < d, D ..."
```

Expansion

Some languages expand a single character into multiple characters for sorting purposes. For example in English the ligature 'æ' might be sorted as 'a' followed by 'e'. To represent this in a rule, the reset character (&) is used. The idea is to reset the current sorting key to an already entered value, and create multiple entries for the ligature. For example:

```
"... < a < b < c < d < e ... < z & a = æ & e = æ ..."
```

This rule resets the sort key to that of 'a', and then enters 'æ'. Then resets the sort key to that of 'e' and enters 'æ' again. This rule says that 'æ' is exactly equivalent to 'a' followed by 'e'. Alternatively ';' could have been used instead of '='. This would have made "æe" less than "æ" but only by a secondary difference.

Ignorable Characters

Characters in the rule before the first '`<`' are ignorable. That is they are not considered during the primary sorting at all (it is as if they aren't even there). Accents and punctuation are often marked as ignorable, but given a non-ignorable secondary or tertiary weight. For example, the default Java rule starts out with:

```
"=\u200B=\u200C=\u200D=\u200E=\u200F ...
;"\u0020;"\u00A0..."
```

This completely ignores the first five characters (formatting control), and ignores except for secondary differences the next two characters (spacing characters).

This is why all example rules up till now started with '`<`' (so that none of the characters would be ignorable).

Listing 7.25 Example of locale sorting

Assume the file "my_loc" has the following data in it:

```
$collate_narrow
"; - = ' '
< a, A < b, B < c, C
< ch, cH, Ch, CH
< d, D < e, E < f, F
< g, G < h, H < i, I
< j, J < k, K < l, L
< ll, lL, Ll, LL
< m, M < n, N < o, O
< p, P < q, Q < r, R
< s, S < t, T < u, U
< v, V < w, W < x, X
< y, Y < z, Z"
```

Note how the space character was entered using quotes to disambiguate it from insignificant white space. The program below creates a vector of strings and sorts them both by "binary order" (just using string's operator `<`), and by the custom rule above using a locale as the sorting key.

```
#include <iostream>
#include <algorithm>
```

```
#include <vector>
#include <string>
#include <iostream>

int main()
{
    std::vector<std::string> v;
    v.push_back( "aaaaaaAB" );
    v.push_back( "aaaaaaA" );
    v.push_back( "AaaaaaB" );
    v.push_back( "AaaaaaA" );
    v.push_back( "blackbird" );
    v.push_back( "black-bird" );
    v.push_back( "black bird" );
    v.push_back( "blackbirds" );
    v.push_back( "acia" );
    v.push_back( "acha" );
    std::ostream_iterator<std::string> out(std::cout, "\n");
    std::cout << "Binary order:\n\n";
    std::sort(v.begin(), v.end());
    std::copy(v.begin(), v.end(), out);
    std::cout << '\n';
    std::locale loc("my_loc");
    std::sort(v.begin(), v.end(), loc);
    std::cout << "Customized order:\n\n";
    std::copy(v.begin(), v.end(), out);
    std::cout << '\n';
}
```

The output is:

Binary order:

```
AaaaaaA
AaaaaaB
aaaaaaA
aaaaaaB
acha
acia
black bird
black-bird
blackbird
blackbirds
```

Customized order:

```
aaaaaaA
AaaaaaA
aaaaaaB
AaaaaaB
acia
acha
blackbird
black-bird
black bird
blackbirds
```

Extending collate by derivation

The behavior of collate can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons. Naturally, you can derive from collate and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the MSL C++ specific protected interface of collate_byname if you wish (to make your job easier if portability is not a concern).

The class collate_byname has one protected data member:

```
__collation_rule<charT> rule_;
```

Listing 7.26 The class std::__collation_rule interface:

```
template <class charT>
class __collation_rule
{
    struct value
    {
        charT primary;
        charT secondary;
        charT tertiary;
        ;
    };

public:
```

```

struct entry
    : value
{
    unsigned char length;
};

__collation_rule();
explicit __collation_rule(const basic_string<charT>& rule);
void set_rule(const basic_string<charT>& rule);
entry operator()(const charT* low,
                  const charT* high, int& state) const;
bool is_french() const;
bool empty() const;
};

```

Most of this interface is to support `collate_byname`. If you simply derive from `collate_byname`, set the rule with a string, and let `collate_byname` do all the work, then there is really very little you have to know about `__collation_rule`.

A `__collation_rule` can be empty (contain no rule). In that case `collate_byname` will use `collate`'s sorting rule. This is also the case if `collate_byname` is constructed with "C". And once constructed, `__collation_rule`'s rule can be set or changed with `set_rule`. That is all you need to know to take advantage of all this horsepower!

Listing 7.27 Example of a `__collation_rule`:

```

#include <iostream>
#include <locale>
#include <string>

struct my_collate
    : public std::collate_byname<char>
{
    my_collate();
    ;

my_collate::my_collate()
    : std::collate_byname<char>("C")
{

```

```
rule_.set_rule( "< a = A < b = B < c = C
                 "< d = D < e = E < f = F"
                 "< g = G < h = H < i = I"
                 "< j = J < k = K < l = L"
                 "< m = M < n = N < o = O"
                 "< p = P < q = Q < r = R"
                 "< s = S < t = T < u = U"
                 "< v = V < w = W < x = X"
                 "< y = Y < z = Z");
}

int main()
{
    std::locale loc(std::locale(), new my_collate);
    std::string s1("Arnold");
    std::string s2("arnold");
    if (loc(s1, s2))
        std::cout << s1 << " < " << s2 << '\n';
    else if (loc(s2, s1))
        std::cout << s1 << " > " << s2 << '\n';
    else
        std::cout << s1 << " == " << s2 << '\n';
}
```

The custom facet `my_collate` derives from `std::collate_byname<char>` and sets the rule in its constructor. That's all it has to do. For this example, a case-insensitive rule has been constructed. The output of this program is:

Arnold == arnold

Alternatively, you could use `my_collate` directly (this is exactly what MSL C++'s locale does):

Listing 7.28 Example of custom facet `my_collate`:

```
int main()
{
    my_collate col;
    std::string s1("Arnold");
    std::string s2("arnold");
    switch (col.compare(s1.data(), s1.data() + s1.size(),
```

```
        s2.data(), s2.data() + s2.size())
    )
{
case -1:
    std::cout << s1 << " < " << s2 << '\n';
    break;
case 0:
    std::cout << s1 << " == " << s2 << '\n';
    break;
case 1:
    std::cout << s1 << " > " << s2 << '\n';
    break;
}
}
```

The output of this program is also:
Arnold == arnold

22.2.5 The Time Category

The facets `time_get` and `time_put` are conceptually simple: they are used to parse and format dates and times in a culturally sensitive manner. But as is not uncommon, there can be a lot of details. And for the most part, the standard is quiet about the details, leaving much of the behavior of these facets in the “implementation defined” category. Therefore this document not only discusses how to extend and customize the time facets, but it also explains much of the default behavior as well.

22.2.5.1 Template Class `Time_get` Synopsis

```
namespace std {
class time_base {
public:
enum dateorder { no_order, dmy, mdy, ymd, ydm };
};

template <class charT, class InputIterator =
istreambuf_iterator<charT> >
class time_get : public locale::facet, public time_base {
public:
typedef charT char_type;
typedef InputIterator iter_type;
```

```
explicit time_get(size_t refs = 0);
dateorder date_order() const { return do_date_order(); }
iter_type get_time(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const;
iter_type get_date(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const;
iter_type get_weekday(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const;
iter_type get_monthname(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const;
iter_type get_year(iter_type s, iter_type end, ios_base& f,
ios_base::iostate& err, tm* t) const;
static locale::id id;
protected:
~time_get(); //virtual
virtual dateorder do_date_order() const;
virtual iter_type do_get_time(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_date(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_weekday(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_monthname(iter_type s, ios_base&,
ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_year(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
};
```

22.2.5.1.1 Time_get Members

The facet `time_get` has 6 member functions:

- `date_order`
- `get_time`
- `get_date`
- `get_weekday`

- `get_monthname`
- `get_year`

Prototype `dateorder date_order() const;`

Determines how the date, month and year are ordered.

Returns an enumeration representing the date, month, year order.
Returns zero if it is un-ordered.

Prototype `iter_type get_time
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;`

Determines the localized time.

Returns an iterator immediately beyond the last character
recognized as a valid time.

Prototype `iter_type get_date
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;`

Determines the localized date.

Returns an iterator immediately beyond the last character
recognized as a valid date.

Prototype `iter_type get_weekday
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;`

Determines the localized weekday.

Returns an iterator immediately beyond the last character
recognized as a valid weekday.

Prototype `iter_type get_monthname
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;`

Determines the localized month name.

Returns an iterator immediately beyond the last character
recognized as a valid month name.

Prototype `iter_type get_year(iter_type s, iter_type end,
ios_base& str, ios_base::iostate& err,
tm* t) const;`

Determines the localized year.

Returns an iterator immediately beyond the last character
recognized as a valid year.

22.2.5.1.2 Time_get Virtual Functions

The facet time_get has 6 protected virtual members:

- `do_date_order`
- `do_get_time`
- `do_get_date`
- `do_get_weekday`
- `do_get_monthname`
- `do_get_year`

Prototype `dateorder do_date_order() const;`

The method `do_date_order` returns `no_order`. This result can
be changed via derivation.

Prototype `iter_type do_get_time(iter_type s, iter_type end,
ios_base& str, ios_base::iostate& err,
tm* t) const;`

The method `do_get_time` parses time with the format:

`"%H:%M:%S"`

Prototype `iter_type do_get_date
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;`

The method `do_get_date` parses a date with the format:

`"%A %B %d %T %Y"`

This format string can be changed via the named locale facility, or
by derivation.

Prototype `iter_type do_get_weekday
(iter_type s, iter_type end, ios_base& str,`

```
ios_base::iostate& err, tm* t) const;
```

The method `do_get_weekday` parses with the format:

"%A"

Although the format string can only be changed by derivation, the names of the weekdays themselves can be changed via the named locale facility or by derivation.

Prototype `iter_type do_get_monthname
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;`

The method `do_get_monthname` parses with the format:

"%B"

Although the format string can only be changed by derivation, the names of the months themselves can be changed via the named locale facility or by derivation.

Prototype `iter_type do_get_year
(iter_type s, iter_type end, ios_base& str,
ios_base::iostate& err, tm* t) const;`

The method `do_get_year` parses a year with the format:

"%Y"

This behavior can only be changed by derivation.

The details of what these formats mean can be found in the ["Format/Parsing Table" on page 219](#).

In addition to the above mentioned protected methods, MSL C++ provides a non-standard, non-virtual protected method:

```
iter_type __do_parse(iter_type in, iter_type end,  
ios_base& str, ios_base::iostate& err,  
const basic_string<charT>& pattern, tm* t)  
const;
```

This method takes the parameters typical of the standard methods, but adds the pattern parameter of type `basic_string`. The pattern is a general string governed by the rules outlined in the section ["Format Parsing" on page 218](#). Derived classes can make use of this method to parse patterns not offered by `time_get`.

Listing 7.29 Derived classes example:

```
template <class charT, class InputIterator>
typename my_time_get<charT, InputIterator>::iter_type
my_time_get<charT, InputIterator>::do_get_date_time(
    iter_type in, iter_type end, std::ios_base& str,
    std::ios_base::iostate& err, std::tm* t) const
{
    const std::ctype<charT>& ct = std::use_facet<std::ctype<charT>>
        (str.getloc());
    return __do_parse(in, end, str, err, ct.widen("%c"), t);
}
```

Format Parsing

These commands follow largely from the C90 and C99 standards. However a major difference here is that most of the commands have meaning for parsing as well as formatting, whereas the C standard only uses these commands for formatting. The pattern string consists of zero or more conversion specifiers and ordinary characters (char or wchar_t). A conversion specifier consists of a % character, possibly followed by an E or O modifier character (described below), followed by a character that determines the behavior of the conversion specifier. Ordinary characters (non-conversion specifiers) must appear in the source string during parsing in the appropriate place or failbit gets set. On formatting, ordinary characters are sent to the output stream unmodified.

The E modifier can appear on any conversion specifier. But it is ignored for both parsing and formatting.

The O modifier can appear on any conversion specifier. It is ignored for parsing, but effects the following conversion specifiers on output by not inserting leading zeroes: %C, %d, %D, %F, %g, %H, %I, %j, %m, %M, %S, %U, %V, %W, %Y

Table 7.4 Format/Parsing Table

| Modifier | Parse | Format |
|----------|---|---|
| %a | Reads one of the locale's weekday names. The name can either be the full name, or the abbreviated name. Case is significant. On successful parsing of one of the weekday names, sets tm_wday, otherwise sets failbit. For parsing, this format is identical to %A. | Outputs the locale's abbreviated weekday name as specified by tm_wday. The "C" locale's abbreviated weekday names are: Sun, Mon, Tue, Wed, Thu, Fri, Sat. |
| %A | For parsing, this format is identical to %a. | Outputs the locale's full weekday name as specified by tm_wday. The "C" locale's full weekday names are: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday. |
| %b | Reads one of the locale's month names. The name can either be the full name, or the abbreviated name. Case is significant. On successful parsing of one of the month names, sets tm_mon, otherwise sets failbit. For parsing, this format is identical to %B. | Outputs the locale's abbreviated month name as specified by tm_mon. The "C" locale's abbreviated month names are: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec. |
| %B | For parsing, this format is identical to %b. | Outputs the locale's full month name as specified by tm_mon. The "C" locale's full month names are: January, February, March, April, May, June, July, August, September, October, November, December. |
| %c | Reads the date-and-time as specified by the current locale. The "C" locale specification is "%A %B %d %T %Y". On successful parsing this sets tm_wday, tm_mon, tm_mday, tm_sec, tm_min, tm_hour and tm_year. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set. | Outputs the locale's date-and-time. The "C" locale's date-and-time format is "%A %B %d %T %Y". This information is specified by tm_wday, tm_mon, tm_mday, tm_sec, tm_min, tm_hour and tm_year. |

Localization Library

22.2 Standard Locale Categories

| Modifier | Parse | Format |
|----------|--|--|
| %C | This is not a valid parse format. If %C is used in a parse pattern, a runtime_error is thrown. | Outputs the current year divided by 100. Single digit results will be pre-appended with '0' unless the O modifier is used. |
| %d | Reads the day of the month. The result must be in the range [1, 31] else failbit will be set. Upon successful parsing tm_mday is set. For parsing, this format is identical to %e. | Outputs the day of the month as specified by tm_mday. Single digit results will be pre-appended with '0' unless the O modifier is used. |
| %D | Is equivalent to "%m/%d/%y". | Is equivalent to "%m/%d/%y". If the O modifier is used, is equivalent to "%Om/%Od/%y". |
| %e | Reads the day of the month. The result must be in the range [1, 31] else failbit will be set. Upon successful parsing tm_mday is set. For parsing, this format is identical to %d. | Outputs the day of the month as specified by tm_mday. Single digit results will be pre-appended with a space. |
| %F | Is equivalent to "%Y-%m-%d" (the ISO 8601 date format). | Is equivalent to "%Y-%m-%d". If the O modifier is used, is equivalent to "%Y-%Om-%Od". |
| %g | This is not a valid parse format. If %g is used in a parse pattern, a runtime_error is thrown. | Outputs the last 2 digits of the ISO 8601 week-based year . Single digit results will be pre-appended with '0' unless the O modifier is used. Specified by tm_year, tm_wday and tm_yday. |
| %G | This is not a valid parse format. If %G is used in a parse pattern, a runtime_error is thrown. | Outputs the ISO 8601 week-based year . Specified by tm_year, tm_wday and tm_yday. |
| %h | Is equivalent to %b. | Is equivalent to %b. |
| %H | Reads the hour (24-hour clock) as a decimal number. The result must be in the range [0, 23] else failbit will be set. Upon successful parsing tm_hour is set. | Outputs the hour (24-hour clock) as specified by tm_hour. Single digit results will be pre-appended with '0' unless the O modifier is used. |

| Modifier | Parse | Format |
|----------|---|---|
| %I | Reads the hour (12-hour clock) as a decimal number. The result must be in the range [1, 12] else failbit will be set. Upon successful parsing tm_hour is set. This format is usually used with %p to specify am/pm. If a %p is not parsed with the %I, am is assumed. | Outputs the hour (12-hour clock) as specified by tm_hour. Single digit results will be pre-appended with '0' unless the O modifier is used. |
| %j | This is not a valid parse format. If %j is used in a parse pattern, a runtime_error is thrown. | Outputs the day of the year as specified by tm_yday in the range [001, 366]. If the O modifier is used, leading zeroes are suppressed. |
| %m | Reads the month as a decimal number. The result must be in the range [1, 12] else failbit will be set. Upon successful parsing tm_mon is set. | Outputs the month as specified by tm_mon as a decimal number in the range [1, 12]. Single digit results will be pre-appended with '0' unless the O modifier is used. |
| %M | Reads the minute as a decimal number. The result must be in the range [0, 59] else failbit will be set. Upon successful parsing tm_min is set. | Outputs the minute as specified by tm_min as a decimal number in the range [0, 59]. Single digit results will be pre-appended with '0' unless the O modifier is used. |
| %n | Is equivalent to '\n'. A newline must appear in the source string at this position else failbit will be set. | Is equivalent to '\n'. A newline is output. |
| %p | Reads the locale's designation for am or pm. If neither of these strings are parsed then failbit will be set. A successful read will modify tm_hour, but only if %I is successfully parsed in the same parse pattern. | Outputs the locale's designation for am or pm, depending upon the value of tm_hour. The "C" locale's designations are am and pm. |

Localization Library

22.2 Standard Locale Categories

| Modifier | Parse | Format |
|----------|---|--|
| %r | Reads the 12-hour time as specified by the current locale. The “C” locale specification is “%I:%M:%S %p”. On successful parsing this sets tm_hour, tm_min, and tm_sec. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set. | Outputs the locale's 12-hour time. The “C” locale's date-and-time format is “%I:%M:%S %p”. This information is specified by tm_hour, tm_min, and tm_sec. |
| %R | Is equivalent to “%H:%M”. | Is equivalent to “%H:%M”. If the O modifier is used, is equivalent to “%OH:%M”. |
| %S | : Reads the second as a decimal number. The result must be in the range [0, 60] else failbit will be set. Upon successful parsing tm_sec is set. | Outputs the second as specified by tm_sec as a decimal number in the range [0, 60]. Single digit results will be pre-appended with '0' unless the O modifier is used. |
| %t | Is equivalent to '\t'. A tab must appear in the source string at this position else failbit will be set. | Is equivalent to '\t'. A tab is output. |
| %T | Is equivalent to “%H:%M:%S”. | Is equivalent to “%H:%M:%S”. If the O modifier is used, is equivalent to “%OH:%M:%S”. |
| %u | Reads the ISO 8601 weekday as a decimal number [1, 7], where Monday is 1. If the result is outside the range [1, 7] failbit will be set. Upon successful parsing tm_wday is set. | Outputs tm_wday as the ISO 8601 weekday in the range [1, 7] where Monday is 1. |
| %U | This is not a valid parse format. If %U is used in a parse pattern, a runtime_error is thrown. | Outputs the week number of the year (the first Sunday as the first day of week 1) as a decimal number in the range [00, 53] using tm_year, tm_wday and tm_yday. If the O modifier is used, any leading zero is suppressed. |

| Modifier | Parse | Format |
|----------|---|---|
| %V | This is not a valid parse format. If %V is used in a parse pattern, a runtime_error is thrown. | Outputs the “ISO 8601 week-based year” week number in the range [01, 53]. Specified by tm_year, tm_wday and tm_yday. If the O modifier is used, any leading zero is suppressed. |
| %w | Reads the weekday as a decimal number [0, 6], where Sunday is 0. If the result is outside the range [0, 6] failbit will be set. Upon successful parsing tm_wday is set. | Outputs tm_wday as the weekday in the range [0, 6] where Sunday is 0. |
| %W | This is not a valid parse format. If %W is used in a parse pattern, a runtime_error is thrown. | Outputs the week number in the range [00, 53]. Specified by tm_year, tm_wday and tm_yday. The first Monday as the first day of week 1. If the O modifier is used, any leading zero is suppressed. |
| %x | Reads the date as specified by the current locale. The “C” locale specification is “%A %B %d %Y”. On successful parsing this sets tm_wday, tm_mon, tm_mday, and tm_year. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set. | Outputs the locale's date. The “C” locale's date format is “%A %B %d %Y”. This information is specified by tm_wday, tm_mon, tm_mday, and tm_year. |
| %X | Reads the time as specified by the current locale. The “C” locale specification is “%H:%M:%S”. On successful parsing this sets tm_hour, tm_min, and tm_sec. If the entire pattern is not successfully parsed, then no tm members are set and failbit is set. | Outputs the locale's time. The “C” locale's time format is “%H:%M:%S”. This information is specified by tm_hour, tm_min, and tm_sec. |

Localization Library

22.2 Standard Locale Categories

| Modifier | Parse | Format |
|----------|--|---|
| %y | Reads the year as a 2 digit number. The century is specified by the locale. The "C" locale specification is 20 (the 21st century). On successful parsing this sets tm_year. If the year is not successfully parsed, then tm_year is not set and failbit is set. | Outputs the last two digits of tm_year. Single digit results will be pre-appended with '0' unless the O modifier is used. |
| %Y | Reads the year. On successful parsing this sets tm_year. If the year is not successfully parsed, then tm_year is not set and failbit is set. | Outputs the year as specified by tm_year. (e.g. 2001) |
| %z | Reads the offset from UTC in the ISO 8601 format "-0430" (meaning 4 hours 30 minutes behind UTC, west of Greenwich). Two strings are accepted according to the current locale, one indicating Daylight Savings Time is not in effect, the other indicating it is in effect. Depending upon which string is read, tm_isdst will be set to 0 or 1. If the locale's designations for these strings are zero length, then no parsing is done and tm_isdst is set to -1. If the locale has non-empty strings for the UTC offset and neither is successfully parsed, failbit is set. | Outputs the UTC offset according to the current locale and the setting of tm_isdst (if non-negative). The "C" locale's designation for these strings is "" (an empty string). |
| %Z | : Reads the time zone name. Two strings are accepted according to the current locale, one indicating Daylight Savings Time is not in effect, the other indicating it is in effect. Depending upon which string is read, tm_isdst will be set to 0 or 1. If the locale's designations for these strings are zero length, then no parsing is done and tm_isdst is set to -1. If the locale has non-empty strings for the time zone names and neither is successfully parsed, failbit is set. | Outputs the time zone according to the current locale and the setting of tm_isdst (if non-negative). The "C" locale's designation for these strings is "" (an empty string). |

| Modifier | Parse | Format |
|-----------------------|--|---------------------------|
| %% | A % must appear in the source string at this position else failbit will be set | A % is output. |
| % followed by a space | One or more white space characters are parsed in this position. White space is determined by the locale's ctype facet. If at least one white space character does not exist in this position, then failbit is set. | A space (' ') for output. |

ISO 8601 week-based year

The %g, %G, and %v give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %G is replaced by 1998 and %v is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %G is replaced by 1998 and %v is replaced by 1.

22.2.5.2 Template Class Time_get_byname

Listing 7.30 Template class Time_get_byname Synopsis

```
Template class time_get_byname
namespace std {
template <class charT, class InputIterator =
istreambuf_iterator<charT> >
class time_get_byname : public time_get<charT, InputIterator> {
public:
typedef time_base::dateorder dateorder;
typedef InputIterator iter_type
explicit time_get_byname(const char*, size_t refs = 0);
protected:
~time_get_byname(); // virtual
virtual dateorder do_date_order() const;
```

```
virtual iter_type do_get_time(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_date(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_weekday(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_monthname(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
virtual iter_type do_get_year(iter_type s, iter_type end,
ios_base&,
ios_base::iostate& err, tm* t) const;
};

}
```

22.2.5.3 Template Class Time_put

The time_put facet format details are described in the listing
[“Format/Parsing Table” on page 219.](#)

Listing 7.31 Template Class Time_put Synopsis

```
namespace std {
template <class charT, class OutputIterator =
ostreambuf_iterator<charT> >
class time_put : public locale::facet {
public:
typedef charT char_type;
typedef OutputIterator iter_type;
explicit time_put(size_t refs = 0);
// the following is implemented in terms of other member
functions.
iter_type put(iter_type s, ios_base& f, char_type fill, const tm*
tmb,
const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& f, char_type fill,
const tm* tmb, char format, char modifier = 0) const;
static locale::id id;
protected:
```

```
~time_put(); //virtual
virtual iter_type do_put(iter_type s, ios_base&, char_type, const
tm* t,
char format, char modifier) const;
};
```

22.2.5.3.1 Time_put Members

The class time_put has one member function.

Prototype iter_type put(iter_type s, ios_base& str,
 char_type fill, const tm* t, const charT*
 pattern, const charT* pat_end) const;
 iter_type put(iter_type s, ios_base& str,
 char_type fill, const tm* t, char format,
 char modifier = 0) const;

Formats a localized time.

Returns an iterator immediately beyond the last character.

22.2.5.3.2 Time_put Virtual Functions

The class time_put has one virtual member function.

Prototype iter_type do_put(iter_type s, ios_base&,
 char_type fill, const tm* t, char format,
 char modifier) const;

Implements the public member function `put`.

22.2.5.4 Template Class Time_put_byname Synopsis

```
namespace std {
template <class charT, class OutputIterator =
ostreambuf_iterator<charT> >
class time_put_byname : public time_put<charT, OutputIterator>
{
public:
typedef charT char_type;
typedef OutputIterator iter_type;
explicit time_put_byname(const char*, size_t refs = 0);
protected:
```

```
~time_put_byname() // virtual
virtual iter_type do_put(iter_type s, ios_base&, char_type, const
tm* t,
char format, char modifier) const;
};

}
```

Extending The Behavior Of The Time Facets

The time facets can easily be extended and customized for many different cultures. To stay portable one can derive from `time_get` and `time_put` and re-implement the behavior described above. Or one could take advantage of the MSL C++ implementation of these classes and build upon the existing functionality quite easily. Specifically you can easily alter the following data in the MSL time facets:

- The abbreviations of the weekday names
- The full weekday names
- The abbreviations of the month names
- The full month names
- The date-and-time format pattern (what %c will expand to)
- The date format pattern (what %x will expand to)
- The time format pattern (what %X will expand to)
- The 12 hour time format pattern (what %r will expand to)
- The strings used for AM/PM
- The strings used for the UTC offset
- The strings used for time zone names
- The default century to be used when parsing %y

Extending locale by using named locale facilities

The easiest way to specify the locale specific data is to use the named locale facilities. When you create a named locale with a string that refers to a locale data file, the time facets parse that data file for time facet data.

```
locale loc("my_locale");
```

The narrow file “my_locale” can hold time data for both narrow and wide time facets. Wide characters and strings can be represented in the narrow file using hexadecimal or universal format (e.g. '\u06BD'). Narrow time data starts with the keyword:

```
$time_narrow
```

And wide time data starts with the keyword:

```
$time_wide
```

Otherwise, the format for the time data is identical for the narrow and wide data.

There are twelve keywords that allow you to enter the time facet data:

1. abbrev_weekday
2. weekday
3. abbrev_monthname
4. monthname
5. date_time
6. am_pm
7. time_12hour
8. date
9. time
10. time_zone
11. utc_offset
12. default_century

You enter data with one of these keywords, followed by an equal sign '=' , and then the data. You can specify any or all of the 12 keywords in any order. Data not specified will default to that of the “C” locale.

NOTE See [“String Syntax” on page 155](#) for syntax details.

abrev_weekday This keyword allows you to enter the abbreviations for the weekday names. There must be seven strings that follow this keyword, corresponding to Sun through Sat. The “C” designation is:

```
abrev_weekday = Sun Mon Tue Wed Thu Fri Sat
```

weekday This keyword allows you to enter the full weekday names. There must be seven strings that follow this keyword, corresponding to Sunday through Saturday. The “C” designation is:

```
weekday = Sunday Monday Tuesday Wednesday  
Thursday Friday Saturday
```

abrev_monthname This keyword allows you to enter the abbreviations for the month names. There must be twelve strings that follow this keyword, corresponding to Jan through Dec. The “C” designation is:

```
abrev_monthname = Jan Feb Mar Apr May Jun Jul  
Aug Sep Oct Nov Dec
```

monthname This keyword allows you to enter the full month names. There must be twelve strings that follow this keyword, corresponding to January through December. The “C” designation is:

```
monthname =  
January February March April May June July  
August September October November December
```

date_time This keyword allows you to enter the parsing/formatting string to be used when %c is encountered. The “C” locale has:

```
date_time = "%A %B %d %T %Y"
```

The date_time string must not contain %c, else an infinite recursion will occur.

am_pm This keyword allows you to enter the two strings that designate AM and PM. The “C” locale specifies:

```
am_pm = am pm
```

time_12hour This keyword allows you to enter the parsing/formatting string to be used when %r is encountered. The “C” locale has:

```
time_12hour = "%I:%M:%S %p"
```

The time_12hour string must not contain %r, else an infinite recursion will occur.

date This keyword allows you to enter the parsing/formatting string to be used when %x is encountered. The “C” locale has:

```
date = "%A %B %d %Y"
```

The date string must not contain %x, else an infinite recursion will occur.

time This keyword allows you to enter the parsing/formatting string to be used when %X is encountered. The “C” locale has:

```
time = "%H:%M:%S"
```

The time string must not contain %X, else an infinite recursion will occur.

time_zone This keyword allows you to enter two strings that designate the names of the locale's time zones: the first being the name for the time zone when Daylight Savings Time is not in effect, and the second name for when it is. The “C” locale has:

```
time_zone = " " "
```

This means that time zone information is not available in the “C” locale.

utc_offset This keyword allows you to enter two strings that designate the UTC offsets of the locale's time zones: the first being the offset for the time zone when Daylight Savings Time is not in effect, and the second string for when it is. The “C” locale has:

```
utc_offset = " " "
```

This means that UTC offset information is not available in the “C” locale.

default_century This keyword allows you to enter the default century which is used to create the correct year when parsing the %y format. This format parses a number and then computes the year by adding it to 100*default_century. The “C” locale has:

```
default_century = 20
```

Assume a Date class. The I/O for the Date class can be written using time_get and time_put in a portable manner. The input operator might look like:

Listing 7.32 Date Class Example Use

```
template<class charT, class traits>
std::basic_istream<charT, traits>&
operator >>(std::basic_istream<charT, traits>& is, Date& item)
{
    typename std::basic_istream<charT, traits>::sentry ok(is);
    if (ok)
    {
        std::ios_base::iostate err = std::ios_base::goodbit;
        try
        {
            const std::time_get<charT>& tg =
                std::use_facet<std::time_get<charT> >
                (is.getloc());
            std::tm t;
            tg.get_date(is, 0, is, err, &t);
            if (!(err & std::ios_base::failbit))
                item = Date(t.tm_mon+1, t.tm_mday, t.tm_year+1900);
        }
        catch (...)
        {
            err |= std::ios_base::badbit | std::ios_base::failbit;
        }
        is.setstate(err);
    }
    return is;
}
```

The code extracts the time_get facet from the istream's locale and uses its get_date method to fill a tm. If the extraction was successful, then the data is transferred from the tm into the Date class.

Listing 7.33 The output method

```

template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator <<(std::basic_ostream<charT, traits>& os, const Date&
item)
{
    std::basic_ostream<charT, traits>::sentry ok(os);
    if (ok)
    {
        bool failed;
        try
        {
            const std::time_put<charT>& tp =
                std::use_facet<std::time_put<charT>>
            (os.getloc());
            std::tm t;
            t.tm_mday = item.day();
            t.tm_mon = item.month() - 1;
            t.tm_year = item.year() - 1900;
            t.tm_wday = item.dayOfWeek();
            charT pattern[2] = {'%', 'x'};
            failed = tp.put(os, os, os.fill(), &t, pattern,
                            pattern+2).failed();
        }
        catch (...)
        {
            failed = true;
        }
        if (failed)
            os.setstate(std::ios_base::failbit |
                        std::ios_base::badbit);
    }
    return os;
}

```

After extracting the time_put facet from the ostream's locale, you transfer data from your Date class into the tm (or the Date class could simply export a tm). Then the put method is called with the tm and using the pattern "%x". There are several good things about the Date's I/O methods:

- They are written in portable standard C++.
- They are culturally sensitive since they use the locale's time facets.
- They can handle narrow or wide streams.
- The streams can be in memory (e.g. stringstream) or file based streams (fstream)
- For wide file streams, routing is automatically going through a codecvt that could (for example) be using something like UTF-8 to convert to/from the external file.
- They are relatively simple considering the tremendous flexibility involved.

With the Date's I/O done, the rest of the example is very easy. A French locale can be created with the following data in a file named "French":

```
$time_narrow
date = "%A, le %d %B %Y"
weekday =
    dimanche lundi mardi mercredi jeudi vendredi samedi
abrev_weekday =
    dim lun mar mer jeu ven sam
monthname = j
    janvier février mars avril mai juin juillet août
    septembre octobre novembre décembre
abrev_monthname =
    jan fév mar avr mai juin juil aoû sep oct nov déc
```

Now a program can read and write Date's in both English and French (and the Date class is completely ignorant of both languages).

Listing 7.34 Example of dates in English and French

```
#include <locale>
#include <iostream>
#include <sstream>
#include "Date.h"

int
main()
```

```
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale("French"));
    std::cout << "En Paris, c'est " << today << '\n';
    std::cout.imbue(std::locale("US"));
    std::cout << "But in New York it is " << today << '\n';
}
```

This program reads in a Date using the “C” locale from an `istringstream`. Then `cout` is imbued with “French” and the same Date is written out. And finally the same stream is imbued again with a “US” locale and the same Date is written out again. The output is:

```
En Paris, c'est samedi, le 24 février 2001
But in New York it is Saturday February 24 2001
```

For this example the “US” locale was implemented with an empty file. This was possible since the relevant parts of the “US” locale coincide with the “C” locale.

Extending by derivation

The behavior of the time facets can still be customized if you are on a platform that does not support a file system, or if you do not wish to use data files for other reasons. Naturally, you can derive from `time_get` and `time_put` and override each of the virtual methods in a portable manner as specified by the C++ standard. Additionally you can take advantage of the MSL C++ implementation if you wish (to make your job easier if portability is not a concern).

The central theme of the MSL time facets design is a non-standard facet class called `std::timepunct`:

Listing 7.35 Template Class Timepunct Synopsis

```
template <class charT>
class timepunct
    : public locale::facet
{
```

```
public:
    typedef charT           char_type;
    typedef basic_string<charT> string_type;

    explicit timepunct(size_t refs = 0);

    const string_type& abrev_weekday(int wday) const
        {return __weekday_names_[7+wday];}
    const string_type& weekday(int wday) const
        {return __weekday_names_[wday];}
    const string_type& abrev_monthname(int mon) const
        {return __month_names_[12+mon];}
    const string_type& monthname(int mon) const {
        return __month_names_[mon];}
    const string_type& date_time() const
        {return __date_time_;}
    const string_type& am_pm(int hour) const
        {return __am_pm_[hour/12];}
    const string_type& time_12hour() const
    {
        return __12hr_time_;
    }
    const string_type& date() const
        {return __date_;}
    const string_type& time() const
        {return __time_;}
    const string_type& time_zone(int isdst) const
        {return __time_zone_[isdst];}
    const string_type& utc_offset(int isdst) const
        {return __utc_offset_[bool(isdst)];}
    int             default_century() const
        {return __default_century_;}

    static locale::id id;

protected:
    virtual ~timepunct() {}

    string_type __weekday_names_[14];
    string_type __month_names_[24];
    string_type __am_pm_[2];
    string_type __date_time_;
    string_type __date_;
    string_type __time_;
```

```
string_type __12hr_time_;
string_type __time_zone_[2];
string_type __utc_offset_[2];
int         __default_century_;
};
```

This class is analogous to `numpunct` and `moneypunct`. It holds all of the configurable data. The facets `time_get` and `time_put` refer to `timepunct` for the data and then behave accordingly. All of the data in `timepunct` is protected so that the constructor of a derived facet can set this data however it sees fit. The `timepunct` facet will set this data according to the “C” locale.

Both the full weekday names and the abbreviated weekday names are stored in `__weekday_names_`. The full names occupy the first seven elements of the array, and the abbreviated names get the last seven slots. Similarly for `__month_names_`.

The `__am_pm_` member holds the strings that represent AM and PM, in that order.

The `__date_time_` member holds the formatting/parsing string for the date-and-time. This is the member that gets queried when `%c` comes up. Do not put `%c` in this string or an infinite recursion will occur. The default for this string is “`%A %B %d %T %Y`”.

The `__date_` member holds the formatting/parsing string for the date. This is the member that gets queried when `%x` comes up. Do not put `%x` in this string or an infinite recursion will occur. The default for this string is “`%A %B %d %Y`”.

The `__time_` member holds the formatting/parsing string for the time. This is the member that gets queried when `%x` comes up. Do not put `%x` in this string or an infinite recursion will occur. The default for this string is “`%H:%M:%S`”.

The `__12hr_time_` member holds the formatting/parsing string for the 12-hour-time. This is the member that gets queried when `%r` comes up. Do not put `%r` in this string or an infinite recursion will occur. The default for this string is “`%I:%M:%S %p`”.

The `__time_zone_` member contains two strings. The first is the name of the time zone when Daylight Savings Time is not in effect.

The second string is the name of the time zone when Daylight Savings Time is in effect. These can be used to parse or format the `tm_isdst` member of a `tm`. These strings may be empty (as they are in the “C” locale) which means that time zone information is not available.

The `__utc_offset_` member contains two strings. The first represents the UTC offset when Daylight Savings Time is not in effect. The second string is the offset when Daylight Savings Time is in effect. These can be used to parse or format the `tm_isdst` member of a `tm`. These strings may be empty (as they are in the “C” locale) which means that UTC offset information is not available.

The final member, `__default_century_` is an `int` representing the default century to assume when parsing a two digit year with `%y`. The value 19 represents the 1900's, 20 represents the 2000's, etc. The default is 20.

It is a simple matter to derive from `timepunct` and set these data members to whatever you see fit.

Timepunct_byname

You can use `timepunct_byname` to get the effects of a named locale for time facets instead of using a named locale:

The `time_get_byname` and `time_put_byname` facets do not add any functionality over `time_get` and `time_put`.

Listing 7.36 Using Timepunct_byname

```
#include <locale>
#include <iostream>
#include <sstream>
#include "Date.h"

int
main()
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale(std::locale(),

```

```

        new std::timepunct_byname<char>("French")) );
std::cout << "En Paris, c'est " << today << '\n';
std::cout.imbue(std::locale(std::locale(),
    new std::timepunct_byname<char>("US"))));
std::cout << "But in New York it is " << today << '\n';
}

```

This has the exact same effect as the named locale example.

But the timepunct_byname example still uses the files "French" and "US". Below is an example timepunct derived class that avoids files but still captures the functionality of the above examples.

Listing 7.37 Example Timepunct Facet Use

```

// The first job is to create a facet derived from timepunct
// that stores the desired data in the timepunct:

class FrenchTimepunct
    : public std::timepunct<char>
{
public:
    FrenchTimepunct();
};

FrenchTimepunct::FrenchTimepunct()
{
    __date_ = "%A, le %d %B %Y";
    __weekday_names_[0] = "dimanche";
    __weekday_names_[1] = "lundi";
    __weekday_names_[2] = "mardi";
    __weekday_names_[3] = "mercredi";
    __weekday_names_[4] = "jeudi";
    __weekday_names_[5] = "vendredi";
    __weekday_names_[6] = "samedi";
    __weekday_names_[7] = "dim";
    __weekday_names_[8] = "lun";
    __weekday_names_[9] = "mar";
    __weekday_names_[10] = "mer";
    __weekday_names_[11] = "jeu";
    __weekday_names_[12] = "ven";
    __weekday_names_[13] = "sam";
}

```

```
__month_names_[0] = "janvier";
__month_names_[1] = "février";
__month_names_[2] = "mars";
__month_names_[3] = "avril";
__month_names_[4] = "mai";
__month_names_[5] = "juin";
__month_names_[6] = "juillet";
__month_names_[7] = "août";
__month_names_[8] = "septembre";
__month_names_[9] = "octobre";
__month_names_[10] = "novembre";
__month_names_[11] = "décembre";
__month_names_[12] = "jan";
__month_names_[13] = "fév";
__month_names_[14] = "mar";
__month_names_[15] = "avr";
__month_names_[16] = "mai";
__month_names_[17] = "juin";
__month_names_[18] = "juil";
__month_names_[19] = "aoû";
__month_names_[20] = "sep";
__month_names_[21] = "oct";
__month_names_[22] = "nov";
__month_names_[23] = "déc";
}

//Though tedious, the job is quite simple.
//Next simply use your facet:

int main()
{
    std::istringstream in("Saturday February 24 2001");
    Date today;
    in >> today;
    std::cout.imbue(std::locale(std::locale(),
        new FrenchTimepunct));
    std::cout << "En Paris, c'est " << today << '\n';
    std::cout.imbue(std::locale::classic());
    std::cout << "But in New York it is " << today << '\n';
}
```

Here we have explicitly asked for the classic locale, instead of the “US” locale since the two are the same (but executing `classic()` does not involve file I/O). Using the global

locale (locale()) instead of classic() would have been equally fine in this example.

22.2.6 The Monetary Category

There are five standard money classes:

- class `money_base`;
- template <class `charT`, class `InputIterator` = `istreambuf_iterator<charT>` > class `money_get`;
- template <class `charT`, class `OutputIterator` = `ostreambuf_iterator<charT>` > class `money_put`;
- template <class `charT`, bool `International` = false> class `moneypunct`;
- template <class `charT`, bool `International` = false> class `moneypunct_byname`;

The first of these (`money_base`) is not a facet, but the remaining four are. The `money_base` class is responsible only for specifying pattern components that will be used to specify how monetary values are parsed and formatted (currency symbol first or last, etc.).

The facets `money_get` and `money_put` are responsible for parsing and formatting respectively. Though their behavior is made up of virtual methods, and thus can be overridden via derivation, it will be exceedingly rare for you to feel the need to do so. Like the numeric facets, the real customization capability comes with the “punct” classes: `moneypunct` and `moneypunct_byname`.

A user-defined Money class (there will be an example later on) can use `money_get` and `money_put` in defining its I/O, and remain completely ignorant of whether it is dealing with francs or pounds. Instead clients of Money will imbue a stream with a locale that specifies this information. On I/O the facets `money_get` and `money_put` query `moneypunct` (or `moneypunct_byname`) for the appropriate locale-specific data. The Money class can remain blissfully ignorant of cultural specifics, and at the same time, serve all cultures!

A sample Money class

The very reason that we can design a Money class before we know the details of `moneypunct` customization is because the Money class

can remain completely ignorant of this customization. This Money class is meant only to demonstrate I/O. Therefore it is as simple as possible. We begin with a simple struct:

Listing 7.38 A example demonstration of input and output

```
struct Money
{
    long double amount_;
};
```

The I/O methods for this class follow a fairly standard formula, but reference the money facets to do the real work:

```
template<class charT, class traits>
std::basic_istream<charT,traits>&
operator >>(std::basic_istream<charT,traits>& is, Money& item)
{
    typename std::basic_istream<charT,traits>::sentry ok(is);
    if (ok)
    {
        std::ios_base::iostate err = std::ios_base::goodbit;
        try
        {
            const std::money_get<charT>& mg =
                std::use_facet<std::money_get<charT> > (is.getloc());
            mg.get(is, 0, false, is, err, item.amount_);
        }
        catch (...)
        {
            err |= std::ios_base::badbit | std::ios_base::failbit;
        }
        is.setstate(err);
    }
    return is;
}

template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator <<(std::basic_ostream<charT, traits>& os,
            const Money& item)
{
```

```
    std::basic_ostream<charT, traits>::sentry ok(os);
    if (ok)
    {
        bool failed;
        try
        {
            const std::money_put<charT>& mp =
                std::use_facet<std::money_put<charT>>(os.getloc());
            failed = mp.put(os, false, os, os.fill(),
                            item.amount_).failed();
        }
        catch (...)
        {
            failed = true;
        }
        if (failed)
            os.setstate(std::ios_base::failbit |
                        std::ios_base::badbit);
    }
    return os;
}
```

The extraction operator (`>>`) obtains a reference to `money_get` from the stream's locale, and then simply uses its `get` method to parse directly into Money's `amount_`. The insertion operator (`<<`) does the same thing with `money_put` and its `put` method. These methods are extremely flexible, as all of the formatting details (save one) are saved in the stream's locale. That one detail is whether we are dealing a local currency format, or an international currency format. The above methods hard wire this decision to "local" by specifying `false` in the `get` and `put` calls. The `moneypunct` facet can store data for both of these formats. An example difference between an international format and a local format is the currency symbol. The US local currency symbol is "\$", but the international US currency symbol is "USD".

For completeness, we extend this example to allow client code to choose between local and international formats via a stream manipulator. See Matt Austern's excellent C/C++ Users Journal article: The Standard Librarian: User-Defined Format Flags for a complete discussion of the technique used here.

To support the manipulators, our simplistic Money struct is expanded in the following code example.

Listing 7.39 Example of manipulator support.

```
struct Money
{
    enum format {local, international};

    static void set_format(std::ios_base& s, format f)
        {flag(s) = f;}
    static format get_format(std::ios_base& s)
        {return static_cast<format>(flag(s));}
    static long& flag(std::ios_base& s);

    long double amount_;
};
```

An enum has been added to specify local or international format. But this enum is only defined within the Money class. There is no format data member within Money. That information will be stored in a stream by clients of Money. To aid in this effort, three static methods have been added: set_format, get_format and flag. The first two methods simply call flag which has the job of reading and writing the format information to the stream. Although flag is where the real work is going on, its definition is surprisingly simple.

Listing 7.40 Money class flag

```
long&
Money::flag(std::ios_base& s)
{
    static int n = std::ios_base::xalloc();
    return s.iword(n);
}
```

As described in Austern's C/C++ User Journal article, flag uses the stream's xalloc facility to reserve an area of storage which will be the same location in all streams. And then it uses iword to obtain a reference to that storage for a particular stream. Now it is

easier to see how `set_format` and `get_format` are simply writing and reading a long associated with the stream `s`.

To round out this manipulator facility we need the manipulators themselves to allow client code to write statements like:

```
in >> international >> money;
out << local << money << '\n';
```

These are easily accomplished with a pair of namespace scope methods:

Listing 7.41 Money class manipulators

```
template<class charT, class traits>
std::basic_ios<charT, traits>&
local(std::basic_ios<charT, traits>& s)
{
    Money::set_format(s, Money::local);
    return s;
}

template<class charT, class traits>
std::basic_ios<charT, traits>&
international(std::basic_ios<charT, traits>& s)
{
    Money::set_format(s, Money::international);
    return s;
}
```

And finally, we need to modify the `Money` inserter and extractor methods to read this information out of the stream, instead of just blindly specifying `false` (`local`) in the `get` and `put` methods.

Listing 7.42 Money class inserters and extractors

```
template<class charT, class traits>
std::basic_istream<charT, traits>&
operator >>(std::basic_istream<charT, traits>& is, Money& item)
{
    typename std::basic_istream<charT, traits>::sentry ok(is);
```

```
if (ok)
{
    std::ios_base::iostate err = std::ios_base::goodbit;
    try
    {
        const std::money_get<charT>& mg =
            std::use_facet<std::money_get<charT>>(is.getloc());
        mg.get(is, 0, Money::get_format(is) ==
            Money::international, is, err, item.amount_);
    } catch (...)
    {
        err |= std::ios_base::badbit |
            std::ios_base::failbit;
    }
    is.setstate(err);
}
return is;
}

template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator <<(std::basic_ostream<charT, traits>& os,
    const Money& item)
{
    std::basic_ostream<charT, traits>::sentry ok(os);
    if (ok)
    {
        bool failed;
        try
        {
            const std::money_put<charT>& mp =
                std::use_facet<std::money_put<charT>>(os.getloc());
            failed = mp.put(os, Money::get_format(os) ==
                Money::international, os, os.fill(),
                item.amount_).failed();
        }
        catch (...)
        {
            failed = true;
        }
        if (failed)
            os.setstate(std::ios_base::failbit |
```

```
        std::ios_base::badbit);
    }
    return os;
}
```

Because we gave the enum `Money::local` the value 0, this has the effect of making local the default format for a stream.

We now have a simple Money class that is capable of culturally sensitive input and output, complete with local and international manipulators! To motivate the following sections on how to customize moneypunct data. Below is sample code that uses our Money class, along with the named locale facility:

Listing 7.43 Example of using a money class

```
int main()
{
    std::istringstream in("USD (1,234,567.89)");
    Money money;
    in >> international >> money;
    std::cout << std::showbase << local << money << '\n';
    std::cout << international << money << '\n';
    std::cout.imbue(std::locale("Norwegian"));
    std::cout << local << money << '\n';
    std::cout << international << money << '\n';
}
```

And the output is:
\$-1,234,567.89
USD (1,234,567.89)
-1 234 567,89 kr
NOK (1 234 567,89)

22.2.6.1 Template Class Money_get

The template class `Money_get` is used for `locale` monetary input routines.

Listing 7.44 Template Class Money_get Synopsis

```
namespace std {
template <class charT,
class InputIterator = istreambuf_iterator<charT> >
class money_get : public locale::facet {
public:
typedef charT char_type;
typedef InputIterator iter_type;
typedef basic_string<charT> string_type;
explicit money_get(size_t refs = 0);
iter_type get(iter_type s, iter_type end, bool intl,
ios_base& f, ios_base::iostate& err,
long double& units) const;
iter_type get(iter_type s, iter_type end, bool intl,
ios_base& f, ios_base::iostate& err,
string_type& digits) const;
static locale::id id;
protected:
~money_get(); //virtual
virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
ios_base::iostate& err, long double& units) const;
virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
ios_base::iostate& err, string_type& digits) const;
};
}
```

22.2.6.1.1 Money_get Members

Localized member functions for inputting monetary values.

get

Inputs a localized monetary value.

Prototype iter_type get(iter_type s, iter_type end,
 bool intl, ios_base& f, ios_base::iostate& err,
 long double& quant) const;
 iter_type get(s, iter_type end, bool intl,
 ios_base& f, ios_base::iostate& err,
 string_type& quant) const;

Return Returns an iterator immediately beyond the last character recognized as a valid monetary quantity.

22.2.6.1.2 Money_get Virtual Functions

Implementation functions for localization of the `money_get` public member functions.

Prototype `iter_type do_get(iter_type s, iter_type end,
 bool intl, ios_base& str, ios_base::iostate&
 err,
 long double& units) const;`
`iter_type do_get(iter_type s, iter_type end,
 bool intl, ios_base& str, ios_base::iostate& err
 string_type& digits) const;`

Implements a localized monetary `get` function.

22.2.6.2 Template Class Money_put

The template class `money_put` is used for locale monetary output routines.

Listing 7.45 Template Class Money_put Synopsis

```
namespace std {
template <class charT,
class OutputIterator = ostreambuf_iterator<charT> >
class money_put : public locale::facet {
public:
typedef charT char_type;
typedef OutputIterator iter_type;
typedef basic_string<charT> string_type;
explicit money_put(size_t refs = 0);
iter_type put(iter_type s, bool intl, ios_base& f,
char_type fill, long double units) const;
iter_type put(iter_type s, bool intl, ios_base& f,
char_type fill, const string_type& digits) const;
static locale::id id;
protected:
~money_put(); //virtual
virtual iter_type
do_put(iter_type, bool, ios_base&, char_type fill,
long double units) const;
```

```
virtual iter_type  
do_put(iter_type, bool, ios_base&, char_type fill,  
const string_type& digits) const;  
};  
}
```

22.2.6.2.1 Money_put Members

Localized member functions for outputting monetary values.

put

Outputs a localized monetary value.

Prototype iter_type put(iter_type s, bool intl, ios_base& f,
 char_type fill, long double quant) const;
 iter_type put(iter_type s, bool intl, ios_base& f,
 char_type fill, const string_type& quant) const;

Return Returns an iterator immediately beyond the last character
 recognized as a valid monetary quantity.

22.2.6.2.2 Money_put Virtual Functions

Implementation functions for localization of the `money_put`
public member functions.

Prototype iter_type do_put(iter_type s, bool intl,
 ios_base& str, char_type fill,
 long double units) const;
 iter_type do_put(iter_type s, bool intl,
 ios_base& str, char_type fill,
 const string_type& digits) const;

Implements a localized put function.

Class Moneypunct

An object used for localization of monetary punctuation.

22.2.6.3 Template Class Moneypunct Synopsis

```
namespace std {  
class money_base {  
public:
```

```
enum part { none, space, symbol, sign, value };
struct pattern { char field[4]; };
};

template <class charT, bool International = false>
class moneypunct : public locale::facet, public money_base {
public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;
    explicit moneypunct(size_t refs = 0);
    charT decimal_point() const;
    charT thousands_sep() const;
    string grouping() const;
    string_type curr_symbol() const;
    string_type positive_sign() const;
    string_type negative_sign() const;
    int frac_digits() const;
    pattern pos_format() const;
    pattern neg_format() const;
    static locale::id id;
    static const bool intl = International;
protected:
    ~moneypunct(); //virtual
    virtual charT do_decimal_point() const;
    virtual charT do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_curr_symbol() const;
    virtual string_type do_positive_sign() const;
    virtual string_type do_negative_sign() const;
    virtual int do_frac_digits() const;
    virtual pattern do_pos_format() const;
    virtual pattern do_neg_format() const;
};

}
```

22.2.6.3.1 Moneypunct Members

Member functions to determine the punctuation used for monetary formatting.

decimal_point

Determines what character to use as a decimal point.

Prototype `charT decimal_point() const;`

Return Returns a char to be used as a decimal point.

thousands_sep

Determines which character to use for a thousandths separator.

Prototype `charT thousands_sep() const;`

The character to be used for the thousands separator is specified with `thousands_sep`.

Return Returns the character to use for a thousandths separator.

grouping

Determines a string that determines the grouping of thousands.

Prototype `string grouping() const;`

The grouping string specifies the number of digits to group, going from right to left.

Return Returns the string that determines the grouping of thousands.

curr_symbol

Determines a string of the localized currency symbol.

Prototype `string_type curr_symbol() const;`

Return Returns the string of the localized currency symbol.

positive_sign

Determines a string of the localized positive sign.

Prototype `string_type positive_sign() const;`

Return Returns the string of the localized positive sign.

negative_sign

Determines a string of the localized negative sign.

Prototype `string_type negative_sign() const;`

Return Returns the string of the localized negative sign.

frac_digits

Determines a string of the localized fractional digits.

Prototype `int frac_digits() const;`

Return Returns the string of the localized fractional digits.

pos_format

Determines the format of the localized non-negative values.

Prototype `pattern pos_format() const;`

These keywords allow you to enter the format for both positive and negative values. There are 5 keywords to specify a format:

1. none
2. space
3. symbol
4. sign
5. value

A monetary format is a sequence of four of these keywords. Each value: symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last. The behavior of breaking any of these rules is undefined. The default pattern for positive values, and for local and international formats is:

`pos_format = symbol sign none value`

Return Returns the format of the localized non-negative values.

neg_format

Determines the format of the localized non-negative values.

Prototype `pattern neg_format() const;`

These keywords allow you to enter the format for both positive and negative values. There are 5 keywords to specify a format:

1. none

2. space
3. symbol
4. sign
5. value

A monetary format is a sequence of four of these keywords. Each value: symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last. The behavior of breaking any of these rules is undefined. The default pattern for negative values, and for local and international formats is:

```
neg_format = symbol sign none value
```

Return Returns the format of the localized non-negative values.

22.2.6.3.2 **Moneypunct** Virtual Functions

Virtual functions that implement the localized public member functions.

Prototype `charT do_decimal_point() const;`

Implements decimal_point.

Prototype `charT do_thousands_sep() const;`

Implements thousands_sep.

Prototype `string do_grouping() const;`

Implements grouping.

Prototype `string_type do_curr_symbol() const;`

Implements cur_symbol.

Prototype `string_type do_positive_sign() const;`

Implements positive_sign.

Prototype `string_type do_negative_sign() const;`

`do_negative_sign()`

Implements negative_sign.

Prototype `int do_frac_digits() const;`

Implements frac_digits.

Prototype pattern do_pos_format() const;

Implements pos_format.

Prototype pattern do_neg_format() const;

Implements neg_format.

Extending moneypunct by derivation

It is easy enough to derive from moneypunct and override the virtual functions in a portable manner. But moneypunct also has a non-standard protected interface that you can take advantage of if you wish. There are nine protected data members:

```
charT      __decimal_point_;
charT      __thousands_sep_;
string     __grouping_;
string_type __cur_symbol_;
string_type __positive_sign_;
string_type __negative_sign_;
int        __frac_digits_;
pattern    __pos_format_;
pattern    __neg_format_;
```

A derived class could set these data members in its constructor to whatever is appropriate, and thus not need to override the virtual methods.

Listing 7.46 Extending Moneypunct by derivation

```
struct mypunct
  : public std::moneypunct<char, false>
{
  mypunct();
};

mypunct::mypunct()
{
  __decimal_point_ = ',';
  __thousands_sep_ = ' ';
  __cur_symbol_ = "kr";
  __pos_format_.field[0] = __neg_format_.field[0] = char(sign);
  __pos_format_.field[1] = __neg_format_.field[1] = char(value);
  __pos_format_.field[2] = __neg_format_.field[2] = char(space);
```

```
__pos_format_.field[3] = __neg_format_.field[3] =
char(symbol);
}

int
main()
{
    std::locale loc(std::locale(), new mypunct);
    std::cout.imbue(loc);
    // ...
}
```

Indeed, this is just what `moneypunct_byname` does after reading the appropriate data from a locale data file.

2.2.6.4 Template Class `Moneypunct_byname`

A template class for implementation of the `moneypunct` template class.

22.2.6.4 Template Class `Moneypunct_byname` Synopsis

```
namespace std {
template <class charT, bool Intl = false>
class moneypunct_byname : public moneypunct<charT, Intl> {
public:
    typedef money_base::pattern pattern;
    typedef basic_string<charT> string_type;
    explicit moneypunct_byname(const char*, size_t refs = 0);
protected:
    ~moneypunct_byname(); // virtual
    virtual charT do_decimal_point() const;
    virtual charT do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_curr_symbol() const;
    virtual string_type do_positive_sign() const;
    virtual string_type do_negative_sign() const;
    virtual int do_frac_digits() const;
    virtual pattern do_pos_format() const;
    virtual pattern do_neg_format() const;
};
```

When a named locale is created:

```
std::locale my_loc( "MyLocale" );
```

this places the facet `moneypunct_byname("MyLocale")` in the locale. The `moneypunct_byname` constructor considers the name it is constructed with as the name of a data file which may or may not contain `moneypunct` data. There are 4 keywords that mark the beginning of `moneypunct` data in a locale data file.

1. `$money_local_narrow`
2. `$money_international_narrow`
3. `$money_local_wide`
4. `$money_international_wide`

These data sections can appear in any order in the locale data file. And they are all optional. Any data not specified defaults to that of the “C” locale. Wide characters and strings can be represented in the narrow locale data file using hexadecimal or universal format (for example, '\u06BD'). See the rules for [“Strings and Characters in Locale Data Files” on page 153](#) for more syntax details.

Data file syntax

The syntax for entering `moneypunct` data is the same under all four keywords. There are 9 keywords that can be used within a `$money_XXX` data section to specify `moneypunct` data. The keywords can appear in any order and they are all optional.

1. `decimal_point`
2. `thousands_sep`
3. `grouping`
4. `curr_symbol`
5. `positive_sign`
6. `negative_sign`
7. `frac_digits`
8. `pos_format`
9. `neg_format`

Each of these keywords is followed by an equal sign (=) and then the appropriate data (described below).

decimal_point The decimal point data is a single character, as in:

```
decimal_point = '.'
```

The default decimal point is '.'.

thousands_sep The character to be used for the thousands separator is specified with `thousands_sep`, as in:

```
thousands_sep = ','
```

The default thousands separator is ','.

grouping The grouping string specifies the number of digits to group, going from right to left. For example, the grouping: 321 means that the number 12345789 would be printed as in:

```
1,2,3,4,56,789
```

The above grouping string can be specified as:

```
grouping = 321
```

A grouping string of "0" or "" means: don't group. The default grouping string is "3".

curr_symbol The currency symbol is specified as a string by `curr_symbol`, as in:

```
curr_symbol = $
```

It is customary for international currency symbols to be four characters long, but this is not enforced by the `locale` facility. The default local currency symbols is "\$". The default international currency symbol is "USD".

positive_sign The string to be used for the positive sign is specified by `positive_sign`. Many locales set this as the empty string, as in:

```
positive_sign = ""
```

The default positive sign is the empty string.

negative_sign The negative sign data is a string specified by `negative_sign`, as in:

```
negative_sign = ()
```

The precise rules for how to treat signs that are longer than one character are laid out in the standard. Suffice it to say that this will typically enclose a negative value in parentheses.

The default negative sign for local formats is " - ", and for international formats is " () ".

frac_digits The number of digits to appear after the decimal point is specified by `frac_digits`, as in:

```
frac_digits = 2
```

The default value is 2.

pos_format / neg_format These keywords allow you to enter the format for both positive and negative values. There are 5 keywords to specify a format:

1. none
2. space
3. symbol
4. sign
5. value

A monetary format is a sequence of four of these keywords. Each value: symbol, sign, value, and either space or none appears exactly once. The value none, if present, is not first; the value space, if present, is neither first nor last. The behavior of breaking any of these rules is undefined.

The default pattern for positive and negative values, and for local and international formats is:

```
pos_format = symbol sign none value
neg_format = symbol sign none value
```

Listing 7.47 Example Data file

To have the example code run correctly, we need a file named "Norwegian" containing the following data:

```
$money_local_narrow
decimal_point = ','
```

```
thousands_sep = ' '
curr_symbol = kr
pos_format = sign value space symbol
neg_format = sign value space symbol

$money_international_narrow
decimal_point = ','
thousands_sep = ' '
curr_symbol = "NOK "
```

NOTE Not all of the fields have been specified because the default values for these fields were already correct. On the other hand, it does not hurt to specify default data to improve (human) readability in the data file.

22.2.7 The Message Retrieval Category

The messages facet is the least specified facet in the C++ standard. Just about everything having to do with messages is implementation defined.

Listing 7.48 22.2.7.1 Template Class Messages Synopsis

```
namespace std {
class messages_base
{
public:
    typedef int catalog;
};

template <class charT>
class messages
    : public locale::facet,
      public messages_base
{
public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;

    explicit messages(size_t refs = 0);
```

```
catalog open(const basic_string<char>& fn,
            const locale& loc) const;
string_type get(catalog c, int set, int msgid,
                const string_type& dfault) const;
void close(catalog c) const;

static locale::id id;

protected:
    virtual ~messages();
    virtual catalog do_open(const basic_string<char>& fn,
                           const locale& loc) const;
    virtual string_type do_get(catalog c, int set, int msgid,
                               const string_type& dfault) const;
    virtual void do_close(catalog c) const;
};

}
```

The intent is that you can use this class to read messages from a catalog. There may be multiple sets of messages in a catalog. And each message set can have any number of `int/string` pairs. But beyond that, the standard is quiet.

Does the string `fn` in `open` refer to a file? If so, what is the format of the `set/msgid/string` data to be read in from the file? There is also a `messages_byname` class that derives from `messages`. What functionality does `messages_byname` add over `messages`?

Unfortunately the answers to all of these questions are implementation defined. This document seeks to answer those questions. Please remember that applications depending on these answers will probably not be portable to other implementations of the standard C++ library.

22.2.7.1.1 Messages Members

Public member functions for catalog message retrieval.

open

Opens a message catalog for reading

Prototype `catalog open(const basic_string<char>& name,
 const locale& loc) const;`

Return Returns a value that may be passed to get to retrieve a message from
 a message catalog.

get

Retrieves a message from a message catalog.

Prototype `string_type get(catalog cat, int set, int msgid,
 const string_type& dfault) const;`

Return Returns the message in the form of a string.

close

Closes a message catalog.

Prototype `void close(catalog cat) const;`

22.2.7.1.2 Messages Virtual Functions

Virtual functions used to localize the public member functions.

Prototype `catalog do_open(const basic_string<char>& name,
 const locale& loc) const;`

Implements open.

Prototype `string_type do_get(catalog cat, int set, int
 msgid,
 const string_type& dfault) const;`

Implements get.

Prototype `void do_close(catalog cat) const;`

Implements close.

MSL C++ implementation of messages

The Metrowerks standard C++ library has a custom implementation
of messages.

Listing 7.49 Example code to open a catalog:

```
typedef std::messages<char> Msg;
const Msg& ct = std::use_facet<Msg>(std::locale::classic());
Msg::catalog cat = ct.open("my_messages",
    std::locale::classic());
if (cat < 0)
{
    std::cout << "Can't open message file\n";
    std::exit(1);
}
```

The first line simply type defines `messages<char>` for easier reading or typing. The second line extracts the messages facet from the “C” locale. The third line instructs the messages facet to look for a file named “`my_messages`” and read message set data out of it using the classic (“C”) locale (one could specify a locale with a specialized codecvt facet for reading the data file). If the file is not found, the open method returns -1. The facet `messages<char>` reads data from a narrow file (`ifstream`). The facet `messages<wchar_t>` reads data from a wide file (`wifstream`).

The messages data file can contain zero or more message data sets of the format:

```
$set setid
msgid message
msgid message
msgid message
...
...
```

The keyword `$set` begins a message data set. The `setid` is the set number. It can be any int. Set id's do not need to be contiguous. But the set id must be unique among the sets in this catalog.

The `msgid` is the message id number. It can be any int. Message id's do not need to be contiguous. But the message id must be unique among the messages in this set.

The message is an optionally quoted (“) string that is the message for this `setid` and `msgid`. If the message contains white space, it must be quoted. The message can have characters represented

escape sequences using the hexadecimal or universal format. For example (see also [“String Syntax” on page 155](#)):

```
"\u0048\u0069\u0020\u0054\u0068\u0065\u0072\u0065\u0021"
```

The message data set terminates when the data is not of the form

```
msgid message
```

Thus, there are no syntax errors in this data. Instead, a syntax error is simply interpreted as the end of the data set. The catalog file can contain data other than message data sets. The messages facet will scan the file until it encounters \$set setid.

Listing 7.50 Example of message facet

An example message data file might contain:

```
$set 1
1 "First Message"
2 "Error in foo"
3 Baboo
4 "\u0048\u0069\u0020\u0054\u0068\u0065\u0072\u0065\u0021"
```

```
$set 2
1 Ok
2 Cancel
```

A program that uses messages to read and output this file follows:

```
#include <locale>
#include <iostream>

int main()
{
    typedef std::messages<char> Msg;
    const Msg& ct = std::use_facet<Msg>(std::locale::classic());
    Msg::catalog cat = ct.open( "my_messages",
        std::locale::classic());
    if (cat < 0)
    {
        std::cout << "Can't open message file\n";
        return 1;
    }
    std::string eof( "no more messages" );
    for (int set = 1; set <= 2; ++set)
```

```

{
    std::cout << "set " << set << "\n\n";
    for (int msgid = 1; msgid < 10; ++msgid)
    {
        std::string msg = ct.get(cat, set, msgid, eof);
        if (msg == eof)
            break;
        std::cout << msgid << "\t" << msg << '\n';
    }
    std::cout << '\n';
}
ct.close(cat);
}

```

The output of this program is:

set 1

```

1 First Message
2 Error in foo
3 Baboo
4 Hi There!

```

set 2

```

1 Ok
2 Cancel

```

22.2.7.2 Template Class `Messages_byname` Synopsis

The class `messages_byname` adds no functionality over `messages`. The `const char*` that it is constructed with is ignored. To localize messages for a specific culture, either open a different catalog (file), or have different sets in a catalog represent messages for different cultures.

Listing 7.51 Template Class `Messages_byname` Synopsis

```

namespace std {
template <class charT>
class messages_byname : public messages<charT> {
public:
typedef messages_base::catalog catalog;

```

```
typedef basic_string<charT> string_type;
explicit messages_byname(const char*, size_t refs = 0);
protected:
~messages_byname(); // virtual
virtual catalog do_open(const basic_string<char>&, const locale&) const;
virtual string_type do_get(catalog, int set, int msgid,
const string_type& dfault) const;
virtual void do_close(catalog) const;
};
```

Extending messages by derivation

If you are on a platform without file support, or you do not want to use files for messages for other reasons, you may derive from `messages` and override the virtual methods as described by the standard. Additionally you can take advantage of the MSL C++ specific protected interface of `messages` if you wish (to make your job easier if portability is not a concern).

The `messages` facet has the non-virtual protected member:

```
string_type& __set(catalog c, int set, int msgid);
```

You can use this to place the quadruple (`c`, `set`, `msgid`, `string`) into `messages`' database. The constructor of the derived facet can fill the database using multiple calls to `__set`. Below is an example of such a class. This example also overrides `do_open` to double check that the catalog name is a valid name, and then return the proper catalog number. And `do_close` is also overridden to do nothing. The `messages` destructor will reclaim all of the memory used by its database:

Listing 7.52 Example of extending message by derivation

```
#include <iostream>
#include <string>
#include <map>

class MyMessages
: public std::messages<char>
```

```
{  
public:  
    MyMessages();  
protected:  
    virtual catalog do_open(const std::string& fn,  
                           const std::locale&) const;  
    virtual void    do_close(catalog) const {}  
private:  
    std::map<std::string, catalog> catalogs_;  
};  
  
MyMessages::MyMessages()  
{  
    catalogs_[ "my_messages" ] = 1;  
    __set(1, 1, 1) = "set 1: first message";  
    __set(1, 1, 2) = "set 1: second message";  
    __set(1, 1, 3) = "set 1: third message";  
    __set(1, 2, 1) = "set 2: first message";  
    __set(1, 2, 2) = "set 2: second message";  
    __set(1, 2, 3) = "set 2: third message";  
}  
  
MyMessages::catalog  
MyMessages::do_open(const std::string& fn, const std::locale&)  
const  
{  
    std::map<std::string, catalog>::const_iterator i =  
        catalogs_.find(fn);  
    if (i == catalogs_.end())  
        return -1;  
    return i->second;  
}  
  
int main()  
{  
    typedef MyMessages Msg;  
    Msg ct;  
    Msg::catalog cat = ct.open("my_messages",  
                               std::locale::classic());  
    if (cat < 0)  
    {  
        std::cout << "Can't open message file\n";  
    }  
}
```

```
    return 1;
}
std::string eof("no more messages");
for (int set = 1; set <= 2; ++set)
{
    std::cout << "set " << set << "\n\n";
    for (int msgid = 1; msgid < 10; ++msgid)
    {
        std::string msg = ct.get(cat, set, msgid, eof);
        if (msg == eof)
            break;
        std::cout << msgid << "\t" << msg << '\n';
    }
    std::cout << '\n';
}
ct.close(cat);
}
```

The main program (client code) is nearly identical to the previous example. Here we simply create and use the customized messages facet. Alternatively we could have created a locale and installed this facet into it. And then extracted the facet back out of the locale using `use_facet` as in the first example.

The output of this program is:

```
set 1

1  set 1: first message
2  set 1: second message
3  set 1: third message

set 2

1  set 2: first message
2  set 2: second message
3  set 2: third message
```

22.2.8 Program-defined Facets

A C++ program may add its own locales to be added to and used the same as the built in facets. To do this derive a class from `locale::facet` with the static member `static locale::id id`.

22.3 C Library Locales

The C++ header `<clocale>` are the same as the C header `locale` but in standard namespace.

Table 7.5 Header `<clocale>` Synopsis

| Type | Name(s) | Name(s) |
|----------|------------|-------------|
| Macro | LC_ALL | LC_COLLATE |
| Macro | LC_CTYPE | LC_MONETARY |
| Macro | LC_NUMERIC | LC_TIME |
| Macro | NULL | |
| Struct | lconv | |
| Function | localeconv | setlocale |

Containers Library

Containers are used to store and manipulate collections of information.

The Containers library (clause 23)

The chapter is constructed in the following sub sections and mirrors clause 23 of the ISO (the International Organization for Standardization) C++ Standard :

- [“23.1 Container Requirements” on page 271](#)
- [“23.2 Sequences” on page 274](#)
- [“23.3 Associative Containers” on page 304](#)
- [“23.3.5 Template Class Bitset” on page 325](#)

23.1 Container Requirements

Container objects store other objects and control the allocation and de-allocation of those objects. There are five classes implementing these requirements.

- [“23.2.1 Template Class Deque” on page 280](#)
- [“23.2.2 Template Class List” on page 284](#)
- [“23.2.3 Container adaptors” on page 291](#)
- [“23.2.4 Template Class Vector” on page 296](#)
- [“23.2.5 Class Vector<bool>” on page 301](#)

All containers must meet basic requirements.

The swap(), equal() and lexicographical_compare() algorithms are defined in the algorithm library for more information see [“The Algorithms Library \(clause 25\)” on page 361](#).

The member function `size()` returns the number of elements in a container.

The member function `begin()` returns an iterator to the first element and `end` returns an iterator to the last element.

If `begin()` equals `end()` the container is empty.

Copy constructors for container types copy and allocator argument from their first parameter. All other constructors take an Allocator reference argument.

The member function `get_allocator()` returns a copy of the Allocator object used in construction of the container.

If an iterator type of the container is bi-directional or a random access iterator the container is reversible.

Unless specified containers meet these requirements.

If an exception is thrown by an `insert()` function while inserting a single element, that function has no effects.

If an exception is thrown by a `push_back()` or `push_front()` function, that function has no effects.

The member functions `erase()`, `pop_back()` or `pop_front()` do not throw an exception.

None of the copy constructors or assignment operators of a returned iterator throw an exception.

The member function `swap()` does not throw an exception, Except if an exception is thrown by the copy constructor or assignment operator of the container's `compare` object.

The member function `swap()` does not invalidate any references, pointers, or iterators referring to the elements of the containers being swapped.

23.1.1 Sequences Requirements

A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement.

The Library includes three kinds of sequence containers `vector`, `lists`, `deque` and `adaptors` classes

Additional Requirements

The iterator and const_iterator types for sequences must be at least of the forward iterator category.

The iterator returned from `a.insert(p,t)` points to the copy of `t` inserted into `a`.

The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased.

If no prior element exists for `a.erase` then `a.end()` is returned.

The previous conditions are true for `a.erase(q1,q2)` as well.

For every sequence defined in this clause the constructor

```
template <class InputIterator>
X(InputIterator f, InputIterator l,
    const Allocator& a = Allocator())
```

shall have the same effect as:

```
X(static_cast<typename X::size_type>(f),
static_cast<typename X::value_type>(l),a)
```

if `InputIterator` is an integral type.

Member functions in the forms:

```
template <class InputIterator>
rt fx1(iterator p, InputIterator f,
InputIterator l);
```

```
template <class InputIterator>
rt fx2(InputIterator f, InputIterator l);
```

```
template <class InputIterator>
rt fx3(iterator i1, iterator i2, InputIterator
f, InputIterator l);
```

shall have the same effect, respectively, as:

```
fx1(p, static_cast<typename X::size_type>(f),
static_cast<typename X::value_type>(l));
```

```
fx2(static_cast<typename X::size_type>(f),
static_cast<typename X::value_type>(l));
```

```
fx3(i1, i2,
    static_cast<typenameX::size_type>(f),
    static_cast<typename X::value_type>(l));
```

if InputIterator is an integral type.

The member function `at()` provides bounds-checked access to container elements.

The member function `at()` throws `out_of_range` if `n >= a.size()`.

23.1.2 Associative Containers Requirements

Associative containers provide an ability for optimized retrieval of data based on keys.

Associative container are parameterized on Key and an ordering relation. Furthermore, map and multimap associate an arbitrary type T with the key.

The phrase “equivalence of keys” means the equivalence relation imposed by the comparison and not the `operator ==` on keys.

An associative container supports both unique keys as well as support for equivalent keys.

The classes `set` and `map` support unique keys.

The classes `multiset` and `multimap` support equivalent keys.

An iterator of an associative container must be of the bidirectional iterator category.

The insert members shall not affect the validity of iterators.

Iterators of associative containers iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them.

23.2 Sequences

The sequence libraries consist of several headers.

- [“23.2.1 Template Class Deque” on page 280](#)
- [“23.2.2 Template Class List” on page 284](#)
- [“23.2.3 Container adaptors” on page 291](#)
- [“23.2.4 Template Class Vector” on page 296](#)
- [“23.2.5 Class Vector<bool>” on page 301](#)

Listing 8.1 Class <deque> Synopsis

```
namespace std {  
    template <class T, class Allocator = allocator<T>> class deque;  
  
    template <class T, class Allocator>  
        bool operator==  
            (const deque<T,Allocator>& x, const deque<T,Allocator>& y);  
  
    template <class T, class Allocator>  
        bool operator<  
            (const deque<T,Allocator>& x, const deque<T,Allocator>& y);  
  
    template <class T, class Allocator>  
        bool operator!=  
            (const deque<T,Allocator>& x, const deque<T,Allocator>& y);  
  
    template <class T, class Allocator>  
        bool operator>  
            (const deque<T,Allocator>& x, const deque<T,Allocator>& y);  
  
    template <class T, class Allocator>  
        bool operator>=  
            (const deque<T,Allocator>& x, const deque<T,Allocator>& y);  
  
    template <class T, class Allocator>  
        bool operator<=  
            (const deque<T,Allocator>& x, const deque<T,Allocator>& y);  
  
    template <class T, class Allocator>  
        void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);  
}
```

Listing 8.2 Class <list> Synopsis

```
namespace std {  
template <class T, class Allocator = allocator<T> > class list;  
  
template <class T, class Allocator>  
bool operator==(  
    const list<T,Allocator>& x, const list<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator<  
    (const list<T,Allocator>& x, const list<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator!=  
    (const list<T,Allocator>& x, const list<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator>  
    (const list<T,Allocator>& x, const list<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator>=(const list<T,Allocator>& x, const  
list<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator<=  
    (const list<T,Allocator>& x, const list<T,Allocator>& y);  
  
template <class T, class Allocator>  
void swap(list<T,Allocator>& x, list<T,Allocator>& y);  
}
```

Listing 8.3 Class <queue> Synopsis

```
namespace std {  
template <class T, class Container = deque<T> > class queue;  
  
template <class T, class Container>  
bool operator==(  
    const queue<T, Container>& x, const queue<T, Container>& y);
```

```
template <class T, class Container>
bool operator<
    (const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator!=
    (const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator>
    (const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator>=
    (const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container>
bool operator<=
    (const queue<T, Container>& x, const queue<T, Container>& y);

template <class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
          class priority_queue;
}
```

Listing 8.4 Class <stack> Synopsis

```
namespace std {
template <class T, class Container = deque<T> > class stack;

template <class T, class Container>
bool operator==
    (const stack<T, Container>& x, const stack<T, Container>& y);

template <class T, class Container>
bool operator<
    (const stack<T, Container>& x, const stack<T, Container>& y);

template <class T, class Container>
```

```
bool operator!=  
    (const stack<T, Container>& x, const stack<T, Container>& y);  
  
template <class T, class Container>  
bool operator>  
    (const stack<T, Container>& x, const stack<T, Container>& y);  
  
template <class T, class Container>  
bool operator>=  
    (const stack<T, Container>& x, const stack<T, Container>& y);  
  
template <class T, class Container>  
bool operator<=  
    (const stack<T, Container>& x, const stack<T, Container>& y);  
}
```

Listing 8.5 Class <vector> Synopsis

```
namespace std {  
template <class T, class Allocator = allocator<T> > class vector;  
  
template <class T, class Allocator>  
bool operator==  
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator<  
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator!=  
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator>  
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);  
  
template <class T, class Allocator>  
bool operator>=  
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
```

```
template <class T, class Allocator>
bool operator<=
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);

template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);

template <class Allocator>
class vector<bool,Allocator>;

template <class Allocator>
bool operator==( 
    const vector<bool,Allocator>& x,
    const vector<bool,Allocator>& y);

template <class Allocator>
bool operator<
    (const vector<bool,Allocator>& x,
    const vector<bool,Allocator>& y);

template <class Allocator>
bool operator!=
    (const vector<bool,Allocator>& x,
    const vector<bool,Allocator>& y);

template <class Allocator>
bool operator>
    (const vector<bool,Allocator>& x,
    const vector<bool,Allocator>& y);

template <class Allocator>
bool operator>=
    (const vector<bool,Allocator>& x,
    const vector<bool,Allocator>& y);

template <class Allocator>
bool operator<=
    (const vector<bool,Allocator>& x,
    const vector<bool,Allocator>& y);

template <class Allocator>
```

```
void swap(vector<bool,Allocator>& x, vector<bool,Allocator>& y);  
}
```

23.2.1 Template Class Deque

A deque is a kind of sequence that supports random access iterators. The deque class also supports insert and erase operations at the beginning middle or the end. However, deque is especially optimized for pushing and popping elements at the beginning and end.

A deque satisfies all of the requirements of a container and of a reversible container as well as of a sequence.

Listing 8.6 Template Class Deque Synopsis

```
namespace std {  
template <class T, class Allocator = allocator<T> >  
class deque {  
public:  
    typedef typename Allocator::reference      reference  
    typedef typename Allocator::const_reference const_reference  
    // Implementation defined types  
    typedef      iterator;  
    typedef      const_iterator;  
    typedef      size_type;  
    typedef      difference_type;  
    typedef      T      value_type;  
    typedef Allocator allocator_type;  
    typedef typename Allocator::pointer pointer;  
    typedef typename Allocator::const_pointer const_pointer;  
    typedef std::reverse_iterator<iterator> reverse_iterator;  
    typedef std::reverse_iterator<const_iterator>  
    const_reverse_iterator;  
  
    explicit deque  
        (const Allocator& = Allocator());  
    explicit deque  
        (size_type n, const T& value = T(),  
         const Allocator& = Allocator());  
    template <class InputIterator>
```

```
deque
    (InputIterator first, InputIterator last,
     const Allocator& = Allocator());
deque(const deque<T,Allocator>& x);

~deque();

deque<T,Allocator>& operator=
    (const deque<T,Allocator>& x);
template <class InputIterator> void assign
    (InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
allocator_type get_allocator() const;
// iterators:
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
// capacity
size_type size() const;
size_type max_size() const;
void resize(size_type sz, T c = T());
bool empty() const;
// element access
reference operator[](size_type n);
const_reference operator[](size_type n) const;
reference at(size_type n);
const_reference at(size_type n) const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;
// Modifiers
void push_front(const T& x);
void push_back(const T& x);
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
template <class InputIterator> void insert
    (iterator position, InputIterator first, InputIterator last);
```

```
void pop_front();
void pop_back();
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void swap(deque<T,Allocator>& );
void clear();
};

template <class T, class Allocator>
bool operator==
  (const deque<T,Allocator>& x, const deque<T,Allocator>& y);

template <class T, class Allocator>
bool operator<
  (const deque<T,Allocator>& x, const deque<T,Allocator>& y);

template <class T, class Allocator>
bool operator!=
  (const deque<T,Allocator>& x, const deque<T,Allocator>& y);

template <class T, class Allocator>
bool operator>
  (const deque<T,Allocator>& x, const deque<T,Allocator>& y);

template <class T, class Allocator>
bool operator>=
  (const deque<T,Allocator>& x, const deque<T,Allocator>& y);

template <class T, class Allocator>
bool operator<=
  (const deque<T,Allocator>& x, const deque<T,Allocator>& y);

template <class T, class Allocator>
void swap
  (deque<T,Allocator>& x, deque<T,Allocator>& y);
}
```

23.2.1.1 Constructors

The deque constructor creates an object of the class deque.

Prototype `explicit deque(const Allocator& = Allocator());`

```
explicit deque(size_type n, const T& value = T(),
               const Allocator& = Allocator());
template <class InputIterator>
deque(InputIterator first, InputIterator last,
      const Allocator& = Allocator());
```

assign

The **assign** function is overloaded to allow various types to be assigned to a deque.

Prototype `template <class InputIterator>`
 `void assign`
 `(InputIterator first, InputIterator last);`
 `void assign(size_type n, const T& t);`

23.2.1.2 Deque Capacity

The class `deque` has one member function to resize the deque.

resize

This function resizes the deque.

Prototype `void resize(size_type sz, T c = T());`

23.2.1.3 Deque Modifiers

The `deque` class has member functions to modify the deque.

insert

The **insert** function is overloaded to insert a value into deque.

Prototype `iterator insert(iterator position, const T& x);`
 `void insert`
 `(iterator position, size_type n, const T& x);`
template <class InputIterator>
void insert
(iterator position, InputIterator first,
InputIterator last);

erase

An overloaded function that allows the removal of a value at a position.

Prototype iterator erase(iterator position);
 iterator erase(iterator first, iterator last);

Return An iterator to the position erased.

23.2.1.4 Deque Specialized Algorithms

Deque has one specialize swap function.

swap

Swaps the element at one position with another.

Prototype template <class T, class Allocator>
 void swap
 (deque<T,Allocator>& x,deque<T,Allocator>& y);

23.2.2 Template Class List

A list is a sequence that supports bidirectional iterators and allows insert and erase operations anywhere within the sequence.

In a list fast random access to list elements is not supported.

A list satisfies all of the requirements of a container as well as those of a reversible container and of a sequence except for operator[] and the member function at which are not included.

Listing 8.7 Template Class List Synopsis

```
namespace std {  
template <class T, class Allocator = allocator<T> >  
class list {  
public:  
//Defined types:  
typedef typename Allocator::reference reference;  
typedef typename Allocator::const_reference const_reference;  
//implementation defined  
typedef iterator;
```

```
typedef      const_iterator;
typedef      size_type;
typedef      difference_type;

typedef T    value_type;
typedef Allocator allocator_type;
typedef typename Allocator::pointer      pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef std::reverse_iterator<iterator>    reverse_iterator;
typedef std::reverse_iterator<const_iterator>
    const_reverse_iterator;

explicit list(const Allocator& = Allocator());
explicit list(size_type n, const T& value = T(),
    const Allocator& = Allocator());
template <class InputIterator>
list(InputIterator first, InputIterator last,
    const Allocator& = Allocator());
list(const list<T,Allocator>& x);
~list();
list<T,Allocator>& operator=(const list<T,Allocator>& x);
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
allocator_type get_allocator() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

bool empty() const;
size_type size() const;
size_type max_size() const;
void resize(size_type sz, T c = T());

reference front();
const_reference front() const;
```

```
reference back();
const_reference back() const;

void push_front(const T& x);
void pop_front();
void push_back(const T& x);
void pop_back();
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
void insert
    (iterator position, InputIterator first,InputIterator last);
iterator erase(iterator position);
iterator erase(iterator position, iterator last);
void swap(list<T,Allocator>& );
void clear();

void splice(iterator position, list<T,Allocator>& x);
void splice(iterator position, list<T,Allocator>& x, iterator i);
void splice(iterator position, list<T,Allocator>& x,
            iterator first, iterator last);
void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
void unique();
template <class BinaryPredicate>
void unique(BinaryPredicate binary_pred);
void merge(list<T,Allocator>& x);
template <class Compare>
void merge(list<T,Allocator>& x, Compare comp);
void sort();
template <class Compare> void sort(Compare comp);
void reverse();
};

template <class T, class Allocator>
bool operator==
    (const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
bool operator<
    (const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class T, class Allocator>
bool operator!=
```

```
(const list<T,Allocator>& x, const list<T,Allocator>& y);  
template <class T, class Allocator>  
bool operator>  
(const list<T,Allocator>& x, const list<T,Allocator>& y);  
template <class T, class Allocator>  
bool operator>=(  
    const list<T,Allocator>& x, const list<T,Allocator>& y);  
template <class T, class Allocator>  
bool operator<=  
(const list<T,Allocator>& x, const list<T,Allocator>& y);  
  
template <class T, class Allocator>  
void swap(list<T,Allocator>& x, list<T,Allocator>& y);  
}
```

23.2.2.1 Constructors

The overloaded list constructors create objects of type list.

Prototype `explicit list(const Allocator& = Allocator());
explicit list(size_type n, const T& value = T(),
 const Allocator& = Allocator());
template <class InputIterator>
list(InputIterator first, InputIterator last,
 const Allocator& = Allocator());`

assign

The overloaded assign function allows values to be assigned to a list after construction.

Prototype `template <class InputIterator>
void assign(InputIterator first,
 InputIterator last);
void assign(size_type n, const T& t);`

23.2.2.2 List Capacity

The list class provides for one member function to resize the list.

resize

Resizes the list.

Prototype `void resize(size_type sz, T c = T());`

23.2.2.3 List Modifiers

The list class has several overloaded functions to allow modification of the list object.

insert

The insert member function insert a value at a position.

Prototype `iterator insert(iterator position, const T& x);`
 `void insert(`
 `iterator position, size_type n, const T& x);`
 `template <class InputIterator>`
 `void insert(`
 `iterator position, InputIterator first,`
 `InputIterator last);`

push_front

The push_front member function pushes a value at the front of the list.

Prototype `void push_front(const T& x);`

push_back

The push_back member function pushes a value onto the end of the list.

Prototype `void push_back(const T& x);`

erase

The erase member function removes a value at a position or range.

Prototype `iterator erase(iterator position);`
 `iterator erase(iterator first, iterator last);`

Return Returns an iterator to the last position.

pop_front

The pop_front member function removes a value from the top of the list.

Prototype `void pop_front();`

pop_back

The `pop_back` member function removes a value from the end of the list.

Prototype `void pop_back();`

clear

Clears a list by removing all elements.

Prototype `void clear();`

23.2.2.4 List Operations

The `list` class provides for operations to manipulate the list.

splice

Moves an element or a range of elements in front of a position specified.

Prototype `void splice
 (iterator position, list<T,Allocator>& x);`
Prototype `void splice
 (iterator position, list<T,Allocator>& x,
 iterator i);`
Prototype `void splice
 (iterator position, list<T,Allocator>& x,
 iterator first, iterator last);`

remove

Removes all element with a value.

Prototype `void remove(const T& value);`

remove_if

Removes all element for which the predicate is true.

Prototype `template <class Predicate>
 void remove_if(Predicate pred);`

unique

Removes duplicates of consecutive elements.

Prototype `void unique();`
 `template <class BinaryPredicate>`
 `void unique(BinaryPredicate binary_pred);`

merge

Moves sorted elements into a list according to the compare argument.

Prototype `void merge(list<T,Allocator>& x);`
 `template <class Compare>`
 `void merge(list<T,Allocator>& x, Compare comp);`

reverse

Reverses the order of the list.

Prototype `void reverse();`

sort

Sorts a list according to the Compare function or by less than value for the parameterless version.

Prototype `void sort();`
 `template <class Compare> void sort(Compare comp);`

23.2.2.5 List Specialized Algorithms

The list class provides a swapping function.

swap

Changes the position of the first argument with the second argument.

Prototype `template <class T, class Allocator>`
 `void swap`
 `(list<T,Allocator>& x, list<T,Allocator>& y);`

23.2.3 Container adaptors

Container adaptors take a Container template parameter so that the container is copied into the Container member of each adaptor.

23.2.3.1 Template Class Queue

Any of the sequence types supporting operations `front()`, `back()`, `push_back()` and `pop_front()` can be used to instantiate queue.

Listing 8.8 Template Class Queue Synopsis

```
namespace std {
template <class T, class Container = deque<T> >
class queue {

public:
typedef typename Container::value_type      value_type;
typedef typename Container::size_type        size_type;
typedef Container container_type;

protected:
Container c;

public:
explicit queue(const Container& = Container());
bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }
value_type& front() { return c.front(); }
const value_type& front() const { return c.front(); }
value_type& back() { return c.back(); }
const value_type& back() const { return c.back(); }
void push(const value_type& x) { c.push_back(x); }
void pop() { c.pop_front(); }
};

template <class T, class Container>
bool operator==
  (const queue<T, Container>& x, const queue<T, Container>& y);
template <class T, class Container>
bool operator<
  (const queue<T, Container>& x, const queue<T, Container>& y);
template <class T, class Container>
```

```
bool operator!=  
    (const queue<T, Container>& x, const queue<T, Container>& y);  
template <class T, class Container>  
bool operator>  
    (const queue<T, Container>& x, const queue<T, Container>& y);  
template <class T, class Container>  
bool operator>=  
    (const queue<T, Container>& x, const queue<T, Container>& y);  
template <class T, class Container>  
bool operator<=  
    (const queue<T, Container>& x, const queue<T, Container>& y);  
}
```

operator ==

A user supplied operator for the queue class that compares the queue's data member.

Prototype `bool operator ==`

Return Returns true if the data members are equal.

operator <

A user supplied operator for the queue class that compares the queue's data member.

Prototype `bool operator <`

Return Returns true if the data member is less than the compared queue.

23.2.3.2 Template Class Priority_queue

You can instantiate any `priority_queue` with any sequence that has random access iterator and supporting operations `front()`, `push_back()` and `pop_back()`.

Instantiation of a `priority_queue` requires supplying a function or function object for making the priority comparisons.

Listing 8.9 Template Class Priority_queue Synopsis

```
namespace std {
    template <class T, class Container = vector<T>,
              class Compare = less<typename Container::value_type> >
    class priority_queue {

        public:
            typedef typename Container::value_type      value_type;
            typedef typename Container::size_type       size_type;
            typedef Container      container_type;

        protected:
            Container c;
            Compare comp;

        public:
            explicit priority_queue(const Compare& x = Compare(),
                                    const Container& = Container());
            template <class InputIterator>
            priority_queue
                (InputIterator first, InputIterator last,
                 const Compare& x = Compare(), const Container& = Container());
            bool empty() const { return c.empty(); }
            size_type size() const { return c.size(); }
            const value_type& top() const { return c.front(); }
            void push(const value_type& x);
            void pop();
        };
    };
}
```

23.2.3.2.1 Constructors

Creates an object of type `priority_queue`.

Prototype `priority_queue(const Compare& x = Compare(),
 const Container& y = Container());
 template <class InputIterator>
 priority_queue
 (InputIterator first, InputIterator last,
 const Compare& x = Compare(),
 const Container& y = Container());`

23.2.3.2.2 priority_queue members

The class `priority_queue` provides public member functions for manipulation the `priority_queue`.

push

Inserts an element into the `priority_queue`.

Prototype `void push(const value_type& x);`

pop

Removes an element from a `priority_queue`.

Prototype `void pop();`

23.2.3.3 Template Class Stack

A stack class may be instantiated by any sequence supporting operations `back()`, `push_back()` and `pop_back()`.

Listing 8.10 Template Class Stack Synopsis

```
namespace std {
template <class T, class Container = deque<T> >
class stack {

public:
typedef typename Container::value_type      value_type;
typedef typename Container::size_type        size_type;
typedef Container    container_type;

protected:
Container c;

public:
explicit stack(const Container& = Container());
bool empty() const;
size_type size() const;
value_type& top();
const value_type& top() ;
```

```
void push(const value_type& x) ;}
void pop();
};

template <class T, class Container>
bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator<(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator!= (const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator>(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
template <class T, class Container>
bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y); }
```

Public Member Functions

Constructors

Creates an object of type stack with a container object.

Prototype `explicit stack(const Container& = Container());`

empty

Signifies when the stack is empty

Prototype `bool empty() const;`

Return Returns true if there are no elements in the stack.

size

Gives the number of elements in a stack.

Prototype `size_type size() const;`

Return Returns the number of elements in a stack.

top

Gives the top element in the stack.

Prototype `value_type& top()`
`const value_type& top() const`

Return Returns the value at the top of the stack.

push

Puts a value onto a stack.

Prototype `void push(const value_type& x) { c.push_back(x); }`

pop

Removes an element from a stack.

Prototype `void pop()`

23.2.4 Template Class Vector

A vector is a kind of sequence container that supports random access iterators. You can use insert and erase operations at the end and in the middle but at the end is faster.

A vector satisfies all of the requirements of a container and of a reversible container and of a sequence. It also satisfies most of the optional sequence requirements with the exceptions being `push_front` and `pop_front` member functions.

Listing 8.11 Template Class Vector Synopsis

```
namespace std {
template <class T, class Allocator = allocator<T> >
class vector {

public:
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
// Implementation Defined Types
```

```
typedef      iterator;
typedef      const_iterator;
typedef      size_type;
typedef      difference_type;

typedef T      value_type;
typedef Allocator      allocator_type;
typedef typename Allocator::pointer      pointer;
typedef typename Allocator::const_pointer      const_pointer;
typedef std::reverse_iterator<iterator>      reverse_iterator;
typedef std::reverse_iterator<const_iterator>
      const_reverse_iterator;

explicit vector(const Allocator& = Allocator());
explicit vector
  (size_type n, const T& value = T(),
   const Allocator& = Allocator());
template <class InputIterator> vector
  (InputIterator first, InputIterator
   last, const Allocator& = Allocator());
vector(const vector<T,Allocator>& x);

~vector();

vector<T,Allocator>&
operator=
  (const vector<T,Allocator>& x);
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& u);
allocator_type get_allocator() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

size_type size() const;
```

```
size_type max_size() const;
void resize(size_type sz, T c = T());
size_type capacity() const;
bool empty() const;
void reserve(size_type n);

reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

void push_back(const T& x);
void pop_back();
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
void insert
    (iterator position, InputIterator first, InputIterator last);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void swap(vector<T,Allocator>&);
void clear();
};

template <class T, class Allocator>
bool operator==
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
template <class T, class Allocator>
bool operator<
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
template <class T, class Allocator>
bool operator!=
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
template <class T, class Allocator>
bool operator>
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
template <class T, class Allocator>
bool operator>=
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
```

```
template <class T, class Allocator>
bool operator<=
    (const vector<T,Allocator>& x, const vector<T,Allocator>& y);

template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
}
```

23.2.4.1 Constructors

The vector class provides overloaded constructors for creation of a vector object.

Prototype `vector(const Allocator& = Allocator());`
 `explicit vector`
 `(size_type n, const T& value = T(),`
 `const Allocator& = Allocator());`
 `template <class InputIterator>`
 `vector`
 `(InputIterator first, InputIterator last,`
 `const Allocator& = Allocator());`
 `vector(const vector<T,Allocator>& x);`

assign

The member function assign allows you to assign values to an already created object.

Prototype `template <class InputIterator>`
 `void assign`
 `(InputIterator first, InputIterator last);`
 `void assign(size_type n, const T& t);`

capacity

Tells the maximum number of elements the vector can hold.

Prototype `size_type capacity() const;`
Return Returns the maximum number of elements the vector can hold.

resize

Resizes a vector if a second argument is given the elements are filled with that value.

Prototype `void resize(size_type sz, T c = T());`

23.2.4.3 Vector Modifiers

The vector class provides various member functions for vector data manipulation.

insert

The member function insert inserts a value or a range of values at a set position.

Prototype `iterator insert(iterator position, const T& x);`
 `void insert`
 `(iterator position, size_type n, const T& x);`
template <class InputIterator>
void insert
 `(iterator position, InputIterator first,`
 `InputIterator last);`

erase

Removes elements at a position or for a range.

Prototype `iterator erase(iterator position);`
 `iterator erase(iterator first, iterator last);`

23.2.4.4 Vector Specialized Algorithms

The vector class provides for a specialized swap function.

swap

Swaps the data of one argument with the other argument.

Prototype `template <class T, class Allocator>`
 `void swap`
 `(vector<T,Allocator>& x,`
 `vector<T,Allocator>& y);`

23.2.5 Class Vector<bool>

A specialized vector for `bool` elements is provided to optimize allocated space.

Listing 8.12 Class Vector <Bool> Synopsis

```
namespace std {
template <class Allocator> class vector<bool, Allocator> {
public:

typedef bool const_reference;
// Implementation Defined Types
typedef iterator;
typedef const_iterator;
typedef size_type;
typedef difference_type;
typedef pointer;
typedef const_pointer;

typedef bool value_type;
typedef Allocator allocator_type;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator>
const_reverse_iterator;

class reference {
friend class vector;
reference();
public:
~reference();
operator bool() const;
reference& operator=(const bool x);
reference& operator=(const reference& x);
void flip();
};

explicit vector(const Allocator& = Allocator());
explicit vector(
    size_type n, const bool& value = bool(),
    const Allocator& = Allocator());
template <class InputIterator>
```

Containers Library

23.2 Sequences

```
vector
    (InputIterator first, InputIterator last,
     const Allocator& = Allocator());
vector(const vector<bool,Allocator>& x);

~vector();

vector<bool,Allocator>& operator=
    (const vector<bool,Allocator>& x);

template <class InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);

allocator_type get_allocator() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

size_type size() const;
size_type max_size() const;
void resize(size_type sz, bool c = false);
size_type capacity() const;
bool empty() const;
void reserve(size_type n);

reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

void push_back(const bool& x);
```

```
void pop_back();

iterator insert(iterator position, const bool& x);
void insert (iterator position, size_type n, const bool& x);
template <class InputIterator>
void insert
    (iterator position, InputIterator first, InputIterator last);

iterator erase(iterator position);
iterator erase(iterator first, iterator last);

void swap(vector<bool,Allocator>& );
static void swap(reference x, reference y);

void flip();
void clear();
};

template <class Allocator>
bool operator==
    (const vector<bool,Allocator>& x,
     const vector<bool,Allocator>& y);
template <class Allocator>
bool operator<
    (const vector<bool,Allocator>& x,
     const vector<bool,Allocator>& y);
template <class Allocator>
bool operator!=
    (const vector<bool,Allocator>& x,
     const vector<bool,Allocator>& y);
template <class Allocator>
bool operator>
    (const vector<bool,Allocator>& x,
     const vector<bool,Allocator>& y);
template <class Allocator>
bool operator>=
    (const vector<bool,Allocator>& x,
     const vector<bool,Allocator>& y);
template <class Allocator>
bool operator<=
    (const vector<bool,Allocator>& x,
     const vector<bool,Allocator>& y);
```

```
template <class Allocator>
void swap(vector<bool,Allocator>& x, vector<bool,Allocator>& y);
}
```

23.3 Associative Containers

The associative container library consists of four template container classes.

- [“23.3.1 Template Class Map” on page 307](#)
- [“23.3.2 Template Class Multimap” on page 313](#)
- [“23.3.3 Template Class Set” on page 318](#)
- [“23.3.4 Template Class Multiset” on page 321](#)

Listing 8.13 Header <Map> Synopsis

```
namespace std {
template <class Key, class T, class Compare = less<Key>,
class Allocator = allocator<pair<const Key, T> > > class map;
template <class Key, class T, class Compare, class Allocator>
bool operator==
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator<
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator!=(
    const map<Key,T,Compare,Allocator>& x,
    const map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator>=
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator<=
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);
```

```
(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
void swap(map<Key,T,Compare,Allocator>& x
          map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T> > >
class multimap;
template <class Key, class T, class Compare, class Allocator>
bool operator==
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator<
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator!=
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator>
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator>=
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator<=
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
void swap(
    multimap<Key,T,Compare,Allocator>& x,
    multimap<Key,T,Compare,Allocator>& y); }
```

Listing 8.14 Header <Set> Synopsis

```
namespace std {
template <class Key, class Compare = less<Key>,
```

Containers Library

23.3 Associative Containers

```
class Allocator = allocator<Key> > class set;
template <class Key, class Compare, class Allocator>
bool operator==
    (const set<Key,Compare,Allocator>& x,
     const set<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator<
    (const set<Key,Compare,Allocator>& x,
     const set<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator!=
    (const set<Key,Compare,Allocator>& x,
     const set<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator>
    (const set<Key,Compare,Allocator>& x,
     const set<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator>=
    (const set<Key,Compare,Allocator>& x,
     const set<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator<=
    (const set<Key,Compare,Allocator>& x,
     const set<Key,Compare,Allocator>& y);

template <class Key, class Compare, class Allocator>
void swap
    (set<Key,Compare,Allocator>& x,
     set<Key,Compare,Allocator>& y);

template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
class multiset;
template <class Key, class Compare, class Allocator>
bool operator==
    (const multiset<Key,Compare,Allocator>& x,
     const multiset<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator<
    (const multiset<Key,Compare,Allocator>& x,
     const multiset<Key,Compare,Allocator>& y);
```

```
template <class Key, class Compare, class Allocator>
bool operator!=
    (const multiset<Key,Compare,Allocator>& x,
     const multiset<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator>
    (const multiset<Key,Compare,Allocator>& x,
     const multiset<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator>=
    (const multiset<Key,Compare,Allocator>& x,
     const multiset<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator<=
    (const multiset<Key,Compare,Allocator>& x,
     const multiset<Key,Compare,Allocator>& y);
template <class Key, class Compare, class Allocator>
void swap
    (multiset<Key,Compare,Allocator>& x,
     multiset<Key,Compare,Allocator>& y);
}
```

23.3.1 Template Class Map

The map class is an associative container that supports unique keys and provides for retrieval of values of another type T based on the keys. The map template class supports bidirectional iterators.

The template class map satisfies all of the requirements of a normal container and those of a reversible container, as well as an associative container.

A map also provides operations for unique keys.

Listing 8.15 Template Class Map Synopsis

```
namespace std {
template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T> > > class map {
public:
```

Containers Library

23.3 Associative Containers

```
typedef Key key_type;
typedef T mapped_type;
typedef pair<const Key, T> value_type;
typedef Compare key_compare;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
// Implementation Defined Types
typedef iterator;
typedef const_iterator;
typedef size_type;
typedef difference_type;

typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator>
const_reverse_iterator;

class value_compare
    :public binary_function<value_type,value_type,bool> {
friend class map;

protected:
Compare comp;
value_compare(Compare c) : comp(c) {}

public:
bool operator()
    (const value_type& x, const value_type& y) const;
};

explicit map(
    const Compare& comp = Compare(),
    const Allocator& = Allocator());
template <class InputIterator>
map
    (InputIterator first, InputIterator last,
    const Compare& comp = Compare(),
    const Allocator& = Allocator());
map(const map<Key,T,Compare,Allocator>& x);
```

```
~map();

map<Key, T, Compare, Allocator>&
operator=(

    const map<Key, T, Compare, Allocator>& x);

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

bool empty() const;
size_type size() const;
size_type max_size() const;

T& operator[](const key_type& x);

pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);

void swap(map<Key, T, Compare, Allocator>&);

void clear();

key_compare key_comp() const;
value_compare value_comp() const;

iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
size_type count(const key_type& x) const;
```

Containers Library

23.3 Associative Containers

```
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

pair<iterator,iterator>
equal_range(const key_type& x);
pair<const_iterator,const_iterator>
equal_range(const key_type& x) const; }

template <class Key, class T, class Compare, class Allocator>
bool operator==
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator!=
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator>
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator>=
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<=
    (const map<Key,T,Compare,Allocator>& x,
     const map<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
void swap(map<Key,T,Compare,Allocator>& x,
          map<Key,T,Compare,Allocator>& y);
}
```

23.3.1.1 Constructors

The map class provides an overloaded constructor for creating an object of type map.

Prototype `explicit map
 (const Compare& comp = Compare(),
 const Allocator& = Allocator());
template <class InputIterator> map
 (InputIterator first, InputIterator last,
 const Compare& comp = Compare(),
 const Allocator& = Allocator());`

23.3.1.2 Map Element Access

The map class includes an element access operator.

operator []

Access an indexed element.

Prototype `T& operator[]
 (const key_type& x);`

Return Returns the value at the position indicated.

23.3.1.3 Map Operations

The map class includes member functions for map operations.

find

Finds an element based upon a key.

Prototype `iterator find
 (const key_type& x);
const_iterator find
 (const key_type& x) const;`

Return Returns the position where the element is found.

lower_bound

Finds the first position where an element based upon a key would be inserted.

Prototype iterator lower_bound
 (const key_type& x);
 const_iterator lower_bound
 (const key_type& x) const;

Return Returns the first position where an element would be inserted.

upper_bound

Finds the last position where an element based upon a key would be inserted.

Prototype iterator upper_bound
 (const key_type& x);
 const_iterator upper_bound
 (const key_type &x) const;

Return Returns the last position where an element would be inserted.

equal_range

Finds both the first and last position in a range where an element based upon a key would be inserted.

Prototype pair<iterator, iterator>
 equal_range
 (const key_type &x);
pair<const_iterator, const_iterator>
 equal_range
 (const key_type& x) const;

Return Returns a pair of elements representing a range for insertion.

23.3.1.4 Map Specialized Algorithms

The map class provides for a method to swap elements.

swap

Swaps the first argument with the second argument.

Prototype template <class Key, class T,
 class Compare, class Allocator>
void swap
 (map<Key,T,Compare,Allocator>& x,
 map<Key,T,Compare,Allocator>& y);

23.3.2 Template Class Multimap

A `mymap` container supports equivalent keys that may contain multiple copies of the same key value. Multimap provides for fast retrieval of values of another type based on the keys.

Multimap supports bidirectional iterators.

The `mymap` satisfies all of the requirements of a container, reversible container and associative containers.

Multimap supports the `a_eq` operations but not the `a_uniq` operations.

For a `mymap<Key, T>` the `key_type` is `Key` and the `value_type` is `pair<const Key, T>`

Listing 8.16 Template Class Multimap Synopsis

```
namespace std {
template <class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T>> > class multimap
{
public:

    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef Compare key_compare;
    typedef Allocator allocator_type;
    typedef typename Allocator::reference reference;
    typedef typename Allocator::const_reference const_reference;
//Implementation Defined Types
    typedef      iterator;
    typedef      const_iterator;
    typedef      size_type;
    typedef      difference_type;

    typedef typename Allocator::pointer pointer;
    typedef typename Allocator::const_pointer const_pointer;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator>
        const_reverse_iterator; class value_compare
```

Containers Library

23.3 Associative Containers

```
: public binary_function<value_type,value_type,bool> {
friend class multimap;

protected:
Compare comp;
value_compare(Compare c) : comp(c) {}

public:
bool operator()
(const value_type& x, const value_type& y) const {
    return comp(x.first, y.first); }
};

explicit multimap(const Compare& comp = Compare(),
    const Allocator& = Allocator());
template <class InputIterator>
multimap
(InputIterator first, InputIterator last,
const Compare& comp = Compare(),
const Allocator& = Allocator());
multimap(const multimap<Key,T,Compare,Allocator>& x);

~multimap();

multimap<Key,T,Compare,Allocator>&
operator=(
    const multimap<Key,T,Compare,Allocator>& x);

allocator_type get_allocator() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

bool empty() const;
size_type size() const;
size_type max_size() const;
```

```
iterator insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);
void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);
void swap(multimap<Key,T,Compare,Allocator>&);
void clear();

key_compare key_comp() const;
value_compare value_comp() const;

iterator find(const key_type& x);
const_iterator find(const key_type& x) const;

size_type count(const key_type& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
pair<iterator,iterator> equal_range(const key_type& x);
pair<const_iterator,const_iterator> equal_range
    (const key_type& x) const; }
template <class Key, class T, class Compare, class Allocator>
bool operator==
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator<
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator!=
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator>
    (const multimap<Key,T,Compare,Allocator>& x,
```

```
const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator>=
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
bool operator<=
    (const multimap<Key,T,Compare,Allocator>& x,
     const multimap<Key,T,Compare,Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
void swap
    (multimap<Key,T,Compare,Allocator>& x,
     multimap<Key,T,Compare,Allocator>& y); }
```

23.3.2.1 Constructors

The multimap constructor is overloaded for creation of a multimap object.

Prototype `explicit multimap
 (const Compare& comp = Compare(),
 const Allocator& = Allocator());
template <class InputIterator>
multimap
 (InputIterator first, InputIterator last,
 const Compare& comp = Compare(),
 const Allocator& = Allocator()0;`

23.3.2.2 Multimap Operations

The multimap class includes member functions for manipulation of multimap data.

find

Finds a value based upon a key argument.

Prototype `iterator find(const key_type &x);
const_iterator find(const key_type& x) const;`

Return Returns the position where the element is at.

lower_bound

Finds the first position where an element based upon a key would be inserted.

Prototype iterator lower_bound
 (const key_type& x);
const_iterator lower_bound
 (const key_type& x) const;

Return Returns the position where an element was found.

equal_range

Finds the first and last positions where a range of elements based upon a key would be inserted.

Prototype pair<iterator, iterator>
 equal_range
 (const key_type& x);
pair<const_iterator, const_iterator>
 equal_range
 (const key_type& x) const;

Return Returns a pair object that represents the first and last position where a range is found.

23.3.2.3 Multimap Specialized Algorithms

The multimap class provides a specialized function for swapping elements.

swap

Swaps the first argument for the last argument.

Prototype template <class Key, class T,
 class Compare, class Allocator>
void swap
 (multimap<Key,T,Compare,Allocator>& x,
 multimap<Key,T,Compare,Allocator>& y);

23.3.3 Template Class Set

The template class `set` is a container that supports unique keys and provides for fast retrieval of the keys themselves.

Set supports bidirectional iterators.

The class `set` satisfies all of the requirements of a container, a reversible container and an associative container.

A set supports the `a_uniq` operations but not the `a_eq` operations.

Listing 8.17 Template Class Set Synopsis

```
namespace std {
template <class Key, class Compare = less<Key>,
class Allocator = allocator<Key> > class set {
public:

typedef Key key_type;
typedef Key value_type;
typedef Compare key_compare;
typedef Compare value_compare;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
//implementation Defined Types
typedef      iterator;
typedef      const_iterator;
typedef      size_type;
typedef      difference_type;

typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator>
      const_reverse_iterator;

explicit set
  (const Compare& comp = Compare(),
  const Allocator& = Allocator());
template <class InputIterator>
```

```
set
(InputIterator first, InputIterator last,
 const Compare& comp = Compare(),
 const Allocator& = Allocator());
set(const set<Key,Compare,Allocator>& x);

~set();

set<Key,Compare,Allocator>&
operator=
 (const set<Key,Compare,Allocator>& x);
allocator_type get_allocator() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

bool empty() const;
size_type size() const;
size_type max_size() const;

pair<iterator,bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);

void swap(set<Key,Compare,Allocator>& );
void clear();

key_compare key_comp() const;
value_compare value_comp() const;

iterator find(const key_type& x) const;
```

```
size_type count(const key_type& x) const;
iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x) const;
pair<iterator,iterator> equal_range(const key_type& x) const;
};

template <class Key, class Compare, class Allocator>
bool operator==(const set<Key, Compare, Allocator>& x,
                  const set<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator<(const set<Key, Compare, Allocator>& x,
                  const set<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator!=!(const set<Key, Compare, Allocator>& x,
                  const set<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator>(const set<Key, Compare, Allocator>& x,
                  const set<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator>=(const set<Key, Compare, Allocator>& x,
                  const set<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator<=(const set<Key, Compare, Allocator>& x,
                  const set<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
void swap(set<Key, Compare, Allocator>& x,
          set<Key, Compare, Allocator>& y);
}
```

23.3.3.1 Constructors

The `set` class includes overloaded constructors for creation of a `set` object.

Prototype `explicit set`

```
(const Compare& comp = Compare(),
const Allocator& = Allocator());
template <class InputIterator> set
(InputIterator first, last,
const Compare& comp = Compare(),
const Allocator& = Allocator());
```

23.3.3.2 Set Specialized Algorithms

The set class specializes the swap function.

swap

Swaps the first argument with the second argument.

Prototype template <class Key, class Compare,
 class Allocator>
void swap
(set<Key, Compare, Allocator>& x,
set<Key, Compare, Allocator>& y);

23.3.4 Template Class Multiset

The template class multiset is an associative container that supports equivalent keys and retrieval of the keys themselves.

Multiset supports bidirectional iterators.

The multiset satisfies all of the requirements of a container, reversible container and an associative container.

A multiset supports the `a_eq` operations but not the `a_uniq` operations.

Listing 8.18 Template Class Multiset Synopsis

```
namespace std {
template <class Key, class Compare = less<Key>,
class Allocator = allocator<Key> > class multiset {
public:

typedef Key key_type;
typedef Key value_type;
```

Containers Library

23.3 Associative Containers

```
typedef Compare key_compare;
typedef Compare value_compare;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
// Implementation Defined Types
typedef      iterator;
typedef      const_iterator;
typedef      size_type;
typedef      difference_type;

typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator>
const_reverse_iterator;

explicit multiset
    (const Compare& comp = Compare(),
     const Allocator& = Allocator());
template <class InputIterator>
multiset
    (InputIterator first, InputIterator last,
     const Compare& comp = Compare(),
     const Allocator& = Allocator());
multiset(const multiset<Key,Compare,Allocator>& x);

~multiset();

multiset<Key,Compare,Allocator>&
operator=
    (const multiset<Key,Compare,Allocator>& x);
allocator_type get_allocator() const;

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
```

```
bool empty() const;
size_type size() const;
size_type max_size() const;

iterator insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);

void swap(multiset<Key, Compare, Allocator>&);

void clear();

key_compare key_comp() const;
value_compare value_comp() const;

iterator find(const key_type& x) const;

size_type count(const key_type& x) const;

iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x) const;

pair<iterator,iterator> equal_range(const key_type& x) const;
};

template <class Key, class Compare, class Allocator>
bool operator==(
    const multiset<Key, Compare, Allocator>& x,
    const multiset<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator< (
    const multiset<Key, Compare, Allocator>& x,
    const multiset<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator!=(
    const multiset<Key, Compare, Allocator>& x,
```

```
    const multiset<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator>
(  const multiset<Key, Compare, Allocator>& x,
   const multiset<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator>=
( const multiset<Key, Compare, Allocator>& x,
  const multiset<Key, Compare, Allocator>& y);
template <class Key, class Compare, class Allocator>
bool operator<=
( const multiset<Key, Compare, Allocator>& x,
  const multiset<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
void swap
( multiset<Key, Compare, Allocator>& x,
  multiset<Key, Compare, Allocator>& y);
}
```

23.3.4.1 Constructors

The multiset class includes overloaded constructors for creation of a multiset object.

Prototype `explicit multiset
 (const Compare& comp = Compare(),
 const Allocator& = Allocator());
template <class InputIterator>
multiset
 (InputIterator first, last,
 const Compare& comp = Compare(),
 const Allocator& = Allocator());`

23.3.4.2 Multiset Specialized Algorithms

The multiset class provides a specialized swap function.

swap

Swaps the first argument with the second argument.

Prototype `template <class Key, class`

```
Compare, class Allocator>
void swap
    (multiset<Key, Compare, Allocator>& x,
     multiset<Key, Compare, Allocator>& y);
```

23.3.5 Template Class Bitset

The `bitset` header defines a template class and related procedures for representing and manipulating fixed-size sequences of bits.

Listing 8.19 Header <Bitset> Synopsis

```
#include <cstddef>
#include <stdexcept>
#include <iostream>
namespace std {

template <size_t N> class bitset;

template <size_t N>
bitset<N> operator&
    (const bitset<N>&, const bitset<N>&);
template <size_t N> bitset<N>
operator|
    (const bitset<N>&, const bitset<N>&);
template <size_t N>
bitset<N> operator^(
    const bitset<N>&, const bitset<N>&);
template <class charT, class traits, size_t N>
basic_istream<charT, traits>&
operator>>
    (basic_istream<charT, traits>& is, bitset<N>& x);
template <class charT, class traits, size_t N>
basic_ostream<charT, traits>&
operator<<
    (basic_ostream<charT, traits>& os, const bitset<N>& x);
}
```

Listing 8.20 Template Class Bitset Synopsis

```
namespace std {
template<size_t N> class bitset {
public:

class reference {
friend class bitset;
reference();
public:
~reference();
reference& operator=(bool x);
reference& operator=(const reference&);
    bool operator~() const;
operator bool() const;
reference& flip(); /
};

bitset();
bitset(unsigned long val);
template<class charT, class traits, class Allocator>
explicit bitset(
const basic_string<charT,traits,Allocator>& str,
    typename basic_string<charT,traits,
    Allocator>::size_type pos = 0,
    typename basic_string<charT,traits, Allocator>::size_type n =
    basic_string<charT,traits,Allocator>::npos);

bitset<N>& operator&=(const bitset<N>& rhs);
bitset<N>& operator|=(const bitset<N>& rhs);
bitset<N>& operator^=(const bitset<N>& rhs);
bitset<N>& operator<=(size_t pos);
bitset<N>& operator>=(size_t pos);
bitset<N>& set();
bitset<N>& set(size_t pos, int val = true);
bitset<N>& reset();
bitset<N>& reset(size_t pos);
bitset<N> operator~() const;
bitset<N>& flip();
bitset<N>& flip(size_t pos);

reference operator[](size_t pos);
```

```
unsigned long to_ulong() const;
template <class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator> to_string() const;
size_t count() const;
size_t size() const;
bool operator==(const bitset<N>& rhs) const;
bool operator!=(const bitset<N>& rhs) const;
bool test(size_t pos) const;
bool any() const;
bool none() const;
bitset<N> operator<<(size_t pos) const;
bitset<N> operator>>(size_t pos) const;
};
```

Template Class Bitset

The template class bitset can store a sequence consisting of a fixed number of bits.

In the bitset class each bit represents either the value zero (reset) or one (set), there is no negative position. You can toggle a bit to change the value.

When converting between an object of class bitset and an integral value, the integral value corresponding to two or more bits is the sum of their bit values.

The `bitset` functions can report three kinds of errors as exceptions.

- An `invalid_argument` exception
- An `out_of_range` error exception
- An `overflow_error` exceptions

See “[19.1 Exception Classes](#)” on page [67](#), for more information on exception classes.

23.3.5.1 Constructors

The bitset class includes overloaded constructors for creation of a bitset object.

```
Prototype  bitset();
bitset(unsigned long val);
template <class charT,
          class traits, class Allocator>
explicit bitset
    (const basic_string<charT, traits,
     Allocator>& str, typename basic_string
      <charT, traits, Allocator>::size_type pos = 0,
     typename basic_string<charT, traits,
     Allocator>::size_type n = basic_string
      <charT, traits, Allocator>::npos);
```

23.3.5.2 Bitset Members

The bitset class provides various member operators.

operator &=

A bitwise “and equal” operator.

Prototype `bitset<N>& operator&=(const bitset<N>& rhs);`

Return Returns the result of the “and equals” operation.

operator |=

A bitwise “not equal” operator.

Prototype `bitset<N>& operator|=(const bitset<N>& rhs);`

Return Returns the result of the “not equals” operation.

operator ^=

A bitwise “exclusive or equals” operator.

Prototype `bitset<N>& operator^=(const bitset<N>& rhs);`

Return Returns the result of the “exclusive or equals” operation.

operator <<=

A bitwise “left shift equals” operator.

Prototype `bitset<N>& operator <<=(size_t pos);`

Return Returns the result of the “left shift equals” operation.

operator >>=

A bitwise “right shift equals” operator.

Prototype `bitset<N>& operator>>=(size_t pos);`

Return Returns the result of the “right shifts equals” operation.

Set

Sets all the bits or a single bit to a value.

Prototype `bitset<N>& set();`
`bitset<N>& set(size_t pos, int val = 1);`

For the function with no parameters sets all the bits to true. For the overloaded function with just a position argument sets that bit to true. For the function with both a position and a value sets the bit at that position to the value.

Return Returns the altered bitset.

reset

Sets the bits to false.

Prototype `bitset<N>& reset();`
`bitset<N>& reset(size_t pos);`

The reset function without any arguments sets all the bits to false. The reset function with an argument sets the bit at that position to false.

Return Returns the modified bitset.

operator ~

Toggles all bits in the bitset.

Prototype `bitset<N> operator~() const;`

Return Returns the modified bitset.

flip

Toggles all the bits in the bitset.

Prototype `bitset<N>& flip();
 bitset<N>& flip(size_t pos);`

Return Returns the modified bitset.

to_ulong

Gives the value as an unsigned long.

Prototype `unsigned long to_ulong() const;`

Return Returns the unsigned long value that the bitset represents.

to_string

Gives the string as zero and ones that the bitset represents.

Prototype `template <class charT,
 class traits, class Allocator>
basic_string<charT, traits, Allocator>
to_string() const;`

Return Returns a string that the bitset represents.

count

Tells the number of bits that are true.

Prototype `size_t count() const;`

Return Returns the number of set bits.

size

Tells the size of the bitset as the number of bits.

Prototype `size_t size() const;`

Return Returns the size of the bitset.

operator ==

The equality operator.

Prototype `bool operator==(const bitset<N>& rhs) const;`

Return Returns true if the argument is equal to the right side bitset.

operator !=

The inequality operator.

Prototype `bool operator!=(const bitset<N>& rhs) const;`

Return Returns true if the argument is not equal to the right side bitset.

test

Test if a bit at a position is set.

Prototype `bool test(size_t pos) const;`

Return Returns true if the bit at the position is true.

any

Tests if all bits are set to true.

Prototype `bool any() const;`

Return Returns true if any bits in the bitset are true.

none

Tests if all bits are set to false.

Prototype `bool none() const;`

Return Returns true if all bits are false.

operator <<

Shifts the bitset to the left a number of positions.

Prototype `bitset<N> operator<<(size_t pos) const;`

Return Returns the modified bitset.

operator >>

Shifts the bitset to the right a number of positions.

Prototype `bitset<N> operator>>(size_t pos) const;`

Return Returns the modified bitset.

23.3.5.3 Bitset Operators

Bitwise operators are included in the bitset class.

operator &

A bitwise and operator.

Prototype `bitset<N> operator&`
 `(const bitset<N>& lhs, const bitset<N>& rhs);`

Return Returns the modified bitset.

operator |

A bitwise or operator.

Prototype `bitset<N> operator|`
 `(const bitset<N>& lhs, const bitset<N>& rhs);`

Return Returns the modified bitset.

operator ^

A bitwise exclusive or operator.

Prototype `bitset<N> operator^`
 `(const bitset<N>& lhs, const bitset<N>& rhs);`

Return Returns the modified bitset.

operator >>

An extractor operator for a bitset input.

Prototype `template <class charT, class traits, size_t N>`
 `basic_istream<charT, traits>&`
 `operator>>`
 `(basic_istream<charT,`
 `traits>& is, bitset<N>& x);`

Return Returns the bitset.

operator <<

An inserter operator for a bitset output.

Prototype `template <class charT, class traits, size_t N>
 basic_ostream<charT, traits>&
operator<<
 (basic_ostream<charT, traits>& os,
 const bitset<N>& x);`

Return Returns the bitset.

Iterators Library

This chapter presents the concept of iterators in detail, defining and illustrating the five iterator categories of input iterators, output iterators, forward iterators, bidirectional iterators and random access iterators.

The Iterators library (clause 24)

This chapter describes the components used in C++ programs to perform iterations for container classes, streams and stream buffers.

The chapter is constructed in the following sub sections and mirrors clause 24 of the ISO (the International Organization for Standardization) C++ Standard ::

- [“24.1 Requirements” on page 336](#)
 - [“24.1.1 Input Iterators” on page 336](#)
 - [“24.1.2 Output Iterators” on page 336](#)
 - [“24.1.3 Forward Iterators” on page 336](#)
 - [“24.1.4 Bidirectional Iterators” on page 337](#)
 - [“24.1.5 Random Access Iterators” on page 337](#)
- [“24.2 Header <iterator>” on page 337](#)
- [“24.3 Iterator Primitives” on page 339](#)
 - [“24.3.1 Iterator Traits” on page 339](#)
 - [“24.3.2 Basic Iterator” on page 340](#)
 - [“24.3.3 Standard Iterator Tags” on page 340](#)
- [“24.4 Predefined Iterators” on page 342](#)
 - [“24.4.1 Reverse iterators” on page 342](#)
 - [“24.4.2 Insert Iterators” on page 347](#)
- [“24.5 Stream Iterators” on page 351](#)

- [“24.5.1 Template Class Istream iterator” on page 352](#)
- [“24.5.2 Template Class Ostream iterator” on page 354](#)
- [“24.5.3 Template Class Istreambuf iterator” on page 355](#)
- [“24.5.4 Template Class Ostreambuf iterator” on page 357](#)

24.1 Requirements

Iterators are a generalized pointer that allow the C++ program to work with various containers in a unified manner.

All iterators allow the dereference into a value type.

Since iterators are an abstraction of a pointer all functions that work with regular pointers work equally with regular pointers.

24.1.1 Input Iterators

There are requirements for input iterators, this manual, does not attempt to list them all.

Algorithms on input iterators should never attempt to pass through the same iterator more than once.

24.1.2 Output Iterators

There are requirements for output iterators, this manual, does not attempt to list them all.

An output iterator is assignable.

24.1.3 Forward Iterators

Forward iterators meet all the requirements of input and output iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.

24.1.4 Bidirectional Iterators

Bidirectional iterators meet the requirements of forward iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.

24.1.5 Random Access Iterators

Random access iterators meet the requirements of bidirectional iterators.

There are requirements for forward iterators, this manual, does not attempt to list them all.

24.2 Header <iterator>

The header iterator includes classes, types and functions used to allow the C++ program to work with various containers in a unified manner.

Listing 9.1 24.2 Header <iterator> Synopsis

```
namespace std {
    template<class Iterator> struct iterator_traits;
    template<class T> struct iterator_traits<T*>;
    template<class Category, class T, class Distance = ptrdiff_t,
             class Pointer = T*, class Reference = T&> struct iterator;
    struct input_iterator_tag {};
    struct output_iterator_tag {};
    struct forward_iterator_tag:
        public input_iterator_tag {};
    struct bidirectional_iterator_tag:
        public forward_iterator_tag {};
    struct random_access_iterator_tag:
        public bidirectional_iterator_tag {};

    template <class InputIterator, class Distance>
        void advance(InputIterator& i, Distance n);
    template <class InputIterator>
        typename iterator_traits<InputIterator>::difference_type
```

```
distance(InputIterator first, InputIterator last);

template <class Iterator> class reverse_iterator;
template <class Iterator> bool operator==(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator<(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator!=(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator>(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator>=(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator<=(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);
template <class Iterator>
typename reverse_iterator<Iterator>::difference_type operator-(const reverse_iterator<Iterator>& x, const reverse_iterator<Iterator>& y);
template <class Iterator>
reverse_iterator<Iterator> operator+(typename reverse_iterator<Iterator>::difference_type n, const reverse_iterator<Iterator>& x);

template <class Container> class back_insert_iterator;
template <class Container> back_insert_iterator<Container> back_inserter(Container& x);

template <class Container> class front_insert_iterator;
template <class Container> front_insert_iterator<Container> front_inserter(Container& x);

template <class Container> class insert_iterator;
template <class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i);
```

```
template <class T, class charT = char,
          class traits = char_traits<charT>,
          class Distance = ptrdiff_t> class istream_iterator;

template <class T, class charT, class traits, class Distance>
bool operator==
    (const istream_iterator<T,charT,traits,Distance>& x,
     const istream_iterator<T,charT,traits,Distance>& y);
template <class T, class charT, class traits, class Distance>
bool operator!=
    (const istream_iterator<T,charT,traits,Distance>& x,
     const istream_iterator<T,charT,traits,Distance>& y);

template <class T, class charT = char,
          class traits = char_traits<charT> > class ostream_iterator;

template<class charT, class traits = char_traits<charT> >
class istreambuf_iterator;

template <class charT, class traits> bool operator==
    (const istreambuf_iterator<charT,traits>& a,
     const istreambuf_iterator<charT,traits>& b);
template <class charT, class traits> bool operator!=
    (const istreambuf_iterator<charT,traits>& a,
     const istreambuf_iterator<charT,traits>& b);
template <class charT, class traits = char_traits<charT> >
class ostreambuf_iterator;
}
```

24.3 Iterator Primitives

The library provides several classes and functions to simplify the task of defining iterators::

24.3.1 Iterator Traits

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types for a particular iterator type. Therefore, it is required that if `iterator` is the type of an iterator, then the types

```
iterator_traits<Iterator>::difference_type  
iterator_traits<Iterator>::value_type  
iterator_traits<Iterator>::iterator_category
```

are defined as the iterator's difference type, value type and iterator category, respectively.

In the case of an output iterator, the types

```
iterator_traits<Iterator>::difference_type  
iterator_traits<Iterator>::value_type
```

defined as void.

The template `iterator_traits<Iterator>` is specialized for pointers and for pointers to const

24.3.2 Basic Iterator

The iterator template may be used as a base class for new iterators.

Listing 9.2 Basic Iterator Synopsis

```
namespace std {  
template<class Category, class T, class Distance = ptrdiff_t,  
class Pointer = T*, class Reference = T&>  
struct iterator {  
    typedef T value_type;  
    typedef Distance difference_type;  
    typedef Pointer pointer;  
    typedef Reference reference;  
    typedef Category iterator_category;  
};  
}
```

24.3.3 Standard Iterator Tags

The standard library includes category tag classes which are used as compile time tags for algorithm selection. These tags are used to determine the best iterator argument at compile time. These tags are:

```
input_iterator_tag  
output_iterator_tag  
forward_iterator_tag,  
bidirectional_iterator_tag  
random_access_iterator_tag
```

Listing 9.3 Iterator Tag Synopsis

```
namespace std {  
    struct input_iterator_tag {};  
    struct output_iterator_tag {};  
    struct forward_iterator_tag  
        : public input_iterator_tag {};  
    struct bidirectional_iterator_tag  
        : public forward_iterator_tag {};  
    struct random_access_iterator_tag  
        : public bidirectional_iterator_tag {};  
}
```

24.3.4 Iterator Operations

Since only random access iterators provide plus and minus operators, the library provides two template functions for this functionality.

advance

Increments or decrements iterators.

Prototype `template <class InputIterator, class Distance>
 void advance(InputIterator& i, Distance n);`

distance

Provides a means to determine the number of increments or decrements necessary to get from the beginning to the end.

Prototype `template<class InputIterator>
 typename iterator_traits<InputIterator>::
 difference_type distance
 (InputIterator first, InputIterator last);`

The distance from last must be reachable from first.

Return The the number of increments from first to last.

24.4 Predefined Iterators

The standard provides for two basic predefined iterators.

- [“24.4.1 Reverse iterators” on page 342](#)
- [“24.4.2 Insert Iterators” on page 347](#)

24.4.1 Reverse iterators

Both bidirectional and random access iterators have corresponding reverse iterator adaptors that they iterate through.

24.4.1.1 Template Class Reverse_iterator

A reverse_iterator must meet the requirements of a bidirectional iterator.

Listing 9.4 Template Class Reverse_iterator Synopsis

```
Template class reverse_iterator
namespace std {
template <class Iterator>
class reverse_iterator : public
    iterator<typename
iterator_traits<Iterator>::iterator_category,
    typename iterator_traits<Iterator>::value_type,
    typename iterator_traits<Iterator>::difference_type,
    typename iterator_traits<Iterator>::pointer,
    typename iterator_traits<Iterator>::reference> {

protected:
Iterator current;

public:
typedef Iterator
    iterator_type;
typedef typename iterator_traits<Iterator>::difference_type
    difference_type;
typedef typename iterator_traits<Iterator>::reference
    reference;
```

```
typedef typename iterator_traits<Iterator>::pointer
pointer;

reverse_iterator();
explicit reverse_iterator(Iterator x);
template <class U> reverse_iterator
(const reverse_iterator<U>& u);

Iterator base() const; // explicit
reference operator*() const;
pointer operator->() const;

reverse_iterator& operator++();
reverse_iterator operator++(int);
reverse_iterator& operator--();
reverse_iterator operator--(int);

reverse_iterator operator+ (difference_type n) const; *
reverse_iterator& operator+=(difference_type n);
reverse_iterator operator- (difference_type n) const;
reverse_iterator& operator-=(difference_type n);
reference operator[](difference_type n) const;
};

template <class Iterator> bool operator==(
    const reverse_iterator<Iterator>& x,
    const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator<
    (const reverse_iterator<Iterator>& x,
    const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator!=
    (const reverse_iterator<Iterator>& x,
    const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator>
    (const reverse_iterator<Iterator>& x,
    const reverse_iterator<Iterator>& y);

template <class Iterator> bool operator>=
    (const reverse_iterator<Iterator>& x,
    const reverse_iterator<Iterator>& y);
template <class Iterator> bool operator<=
    (const reverse_iterator<Iterator>& x,
```

```
const reverse_iterator<Iterator>& y);
template <class Iterator>
typename reverse_iterator<Iterator>::difference_type operator-
    (const reverse_iterator<Iterator>& x,
     const reverse_iterator<Iterator>& y);
template <class Iterator> reverse_iterator<Iterator> operator+
    (typename reverse_iterator<Iterator>::difference_type n,
     const reverse_iterator<Iterator>& x);
}
```

24.4.1.2 Reverse_iterator Requirements

Additional requirements may be necessary if random access operators are referenced in a way that requires instantiation.

Constructors

Creates an instance of a reverse_iterator object.

Prototype `explicit reverse_iterator(Iterator x);`
 `template <class U> reverse_iterator`
 `(const reverse_iterator<U> &u);`

base

The base operator is used for conversion.

Prototype `Iterator base() const; // explicit`
Return The current iterator is returned.

Reverse_iterator operators

Common operators are provided for reverse_iterators.

Prototype `reference operator*() const;`
Return A reference iterator is returned.
Prototype `pointer operator->() const;`
Return A pointer to the dereferenced iterator.
Prototype `reverse_iterator& operator++();`
 `reverse_iterator operator++(int);`

| | |
|-----------|--|
| Return | The <code>this</code> pointer is returned. |
| Prototype | <code>reverse_iterator& operator--();</code> <code>reverse_iterator operator--(int);</code> |
| Return | The <code>this</code> pointer is returned. |
| Prototype | <code>reverse_iterator operator+</code> <code>(typename reverse_iterator<Iterator></code> <code>::difference_type n) const;</code> |
| Return | The <code>reverse_iterator</code> representing the result of the operation is returned. |
| Prototype | <code>reverse_iterator& operator+=</code> <code>(typename reverse_iterator<Iterator></code> <code>::difference_type n);</code> |
| Return | The <code>reverse_iterator</code> representing the result of the operation is returned. |
| Prototype | <code>iterator operator-</code> <code>(typename reverse_iterator<Iterator></code> <code>::difference_type n) const;</code> |
| Return | The <code>reverse_iterator</code> representing the result of the operation is returned. |
| Prototype | <code>reverse_iterator& operator-=</code> <code>(typename reverse_iterator<Iterator></code> <code>::difference_type n);</code> |
| Return | The <code>reverse_iterator</code> representing the result of the operation is returned. |
| Prototype | <code>reference operator[]</code> <code>(typename reverse_iterator<Iterator></code> <code>::difference_type n) const;</code> |
| Return | An element access reference is returned. |
| Prototype | <code>template <class Iterator> bool operator==</code> <code>(const reverse_iterator<Iterator>& x,</code> <code>const reverse_iterator<Iterator>& y);</code> |
| Return | A bool true value is returned if the iterators are equal. |

| | |
|-----------|--|
| Prototype | <code>template <class Iterator> bool operator<</code> <code>(const reverse_iterator<Iterator>& x,</code> <code>const reverse_iterator<Iterator>& y);</code> |
| Return | A bool true value is returned if the first iterator is less than the second. |
| Prototype | <code>template <class Iterator> bool operator!=</code> <code>(const reverse_iterator<Iterator>& x,</code> <code>const reverse_iterator<Iterator>& y);</code> |
| Return | A bool true value is returned if the first iterator is not equal to the second. |
| Prototype | <code>template <class Iterator> bool operator></code> <code>(const reverse_iterator<Iterator>& x,</code> <code>const reverse_iterator<Iterator>& y);</code> |
| Return | A bool true value is returned if the first iterator is greater than the second. |
| Prototype | <code>template <class Iterator> bool operator>=</code> <code>(const reverse_iterator<Iterator>& x,</code> <code>const reverse_iterator<Iterator>& y);</code> |
| Return | The reverse_iterator representing the result of the operation is returned. |
| Prototype | <code>template <class Iterator> bool operator<=</code> <code>(const reverse_iterator<Iterator>& x,</code> <code>const reverse_iterator<Iterator>& y);</code> |
| Return | The reverse_iterator representing the result of the operation is returned. |
| Prototype | <code>template <class Iterator></code> <code>typename reverse_iterator<Iterator></code> <code>::difference_type operator-</code> <code>(const reverse_iterator<Iterator>& x,</code> <code>const reverse_iterator<Iterator>& y);</code> |
| Return | The reverse_iterator representing the result of the operation is returned. |
| Prototype | <code>template <class Iterator></code> <code>reverse_iterator<Iterator> operator+</code> |

```
(typename reverse_iterator<Iterator>
::difference_type n,
const reverse_iterator<Iterator>& x);
```

Return The reverse_iterator representing the result of the operation is returned.

24.4.2 Insert Iterators

Insert iterators, are provided to make it possible to deal with insertion in the same way as writing into an array.

24.4.2.1 Class Back_insert_iterator

A back_insert_iterator inserts at the back.

Listing 9.5 Template Class Back_insert_iterator Synopsis

```
namespace std {
template <class Container>
class back_insert_iterator :
    public iterator<output_iterator_tag,void,void,void,void> {

protected:
Container* container;

public:
typedef Container container_type;
explicit back_insert_iterator(Container& x);
back_insert_iterator<Container>& operator=
    (typename Container::const_reference value);
back_insert_iterator<Container>& operator*();
back_insert_iterator<Container>& operator++();
back_insert_iterator<Container> operator++(int);
};

template <class Container> back_insert_iterator<Container>
    back_inserter(Container& x);
}
```

Constructors

Constructs a back_insert_iterator object.

Prototype `explicit back_insert_iterator(Container& x);`

Operator =

An operator is provided for copying a `const_reference` value.

Prototype `back_insert_iterator<Container>& operator= (typename Container::const_reference value);`

Return A reference to the copied `back_insert_iterator` is returned.

Back_insert_iterator Operators

Several standard operators are provided for `Back_insert_iterator`.

Prototype `back_insert_iterator<Container>& operator*();`

Return The dereference iterator is returned.

Prototype `back_insert_iterator<Container>& operator++();`
`back_insert_iterator<Container> operator++(int);`

Return The incremented iterator is returned.

back_inserter

Provides a means to get the back iterator.

Prototype `template <class Container>`
`back_insert_iterator<Container> back_inserter (Container& x);`

Return The `back_insert_iterator` is returned.

24.4.2.3 Template Class Front_insert_iterator

A `front_insert_iterator` inserts at the front.

Listing 9.6 Template Class Front_insert_iterator Synopsis

```
namespace std {  
template <class Container>  
class front_insert_iterator :  
public iterator<output_iterator_tag,void,void,void,void> {  
protected:
```

```
Container* container;

public:
typedef Container container_type;
explicit front_insert_iterator(Container& x);
front_insert_iterator<Container>& operator=
    (typename Container::const_reference value);
front_insert_iterator<Container>& operator*();
front_insert_iterator<Container>& operator++();
front_insert_iterator<Container> operator++(int);
};

template <class Container>
front_insert_iterator<Container> front_inserter(Container& x);
}
```

Constructors

Creates a `front_insert_iterator` object.

Prototype `explicit front_insert_iterator(Container& x);`

Operator =

Assigns a value to an already create assignment operator.

Prototype `front_insert_iterator<Container>& operator=
 (typename Container::const_reference value);`

Return A `front_insert_iterator` copy of the `const_reference` value is returned.

Front_insert_iterator operators

Several common operators are provided for the `front_insert_iterator` class.

Prototype `front_insert_iterator<Container>& operator*();`

Return A `this` pointer is returned.

Prototype `front_insert_iterator<Container>& operator++();`
`front_insert_iterator<Container> operator++(int);`

A post or pre increment operator.

Return The this pointer.

front_inserter

Provides a means to get the front iterator.

Prototype `template <class Container>
 front_insert_iterator<Container>
front_inserter(Container& x);`

Returns The `front_insert_iterator` is returned.

24.4.2.5 Template Class `Insert_iterator`

A bidirectional insertion iterator.

Listing 9.7 Template Class `Insert_iterator` Synopsis

```
namespace std {  
template <class Container>  
class insert_iterator :  
public iterator<output_iterator_tag,void,void,void> {  
  
protected:  
Container* container;  
typename Container::iterator iter;  
  
public:  
typedef Container container_type;  
insert_iterator(Container& x, typename Container::iterator i);  
insert_iterator<Container>& operator=  
  (typename Container::const_reference value);  
insert_iterator<Container>& operator*();  
insert_iterator<Container>& operator++();  
insert_iterator<Container>& operator++(int);  
};  
template <class Container, class Iterator>  
  insert_iterator<Container>  
inserter(Container& x, Iterator i);  
}
```

Constructors

Creates an instance of an `insert_iterator` object.

Prototype `insert_iterator
(Container& x, typename Container::iterator i);`

operator =

An operator for assignment of a `const_reference` value.

Prototype `insert_iterator<Container>& operator=
(typename Container::const_reference value);`

Return A copy of the `insert_iterator`.

Insert_iterator Operators

Various operators are provided for an `insert_iterator`.

Prototype `insert_iterator<Container>& operator*();`

Return The dereferenced iterator is returned.

Prototype `insert_iterator<Container>& operator++();
insert_iterator<Container>& operator++(int);`

Return The `this` pointer is returned.

inserter

Provides a means to get the iterator.

Prototype `template <class Container, class Inserter>
insert_iterator<Container> inserter
(Container& x, Inserter i);`

Return The inserter iterator is returned.

24.5 Stream Iterators

Input and output iterators are provided to make it possible for algorithmic templates to work directly with input and output streams.

24.5.1 Template Class `Istream_iterator`

An `istream_iterator` reads (using `operator>>`) successive elements from the input stream. It reads after it is constructed, and every time the increment operator is used.

If an end of stream is reached the iterator returns false.

Since `istream` iterators are not assignable `istream` iterators can only be used for one pass algorithms.

Listing 9.8 Template Class `Istream_iterator` Synopsis

```
namespace std {
template <class T, class charT = char, class traits =
char_traits<charT>,
class Distance = ptrdiff_t>
class istream_iterator:
public iterator<input_iterator_tag,
T, Distance, const T*, const T&> {

public:
typedef charT char_type
typedef traits traits_type;
typedef basic_istream<charT,traits> istream_type;

istream_iterator();
istream_iterator(istream_type& s);
istream_iterator
  (const istream_iterator<T,charT,traits,Distance>& x);
~istream_iterator();

const T& operator*() const;
const T* operator->() const;
istream_iterator<T,charT,traits,Distance>& operator++();
istream_iterator<T,charT,traits,Distance> operator++(int);

private:
//basic_istream<charT,traits>* in_stream; exposition only
//T value; exposition only
};
template <class T, class charT, class traits, class Distance>
```

```
bool operator==  
    (const istream_iterator<T,charT,traits,Distance>& x,  
     const istream_iterator<T,charT,traits,Distance>& y);  
template <class T, class charT, class traits, class Distance>  
bool operator!=  
    (const istream_iterator<T,charT,traits,Distance>& x,  
     const istream_iterator<T,charT,traits,Distance>& y);  
}
```

Constructors

Creates and object of an `istream_iterator` object.

Prototype `istream_iterator();`
 `istream_iterator(istream_type& s);`
 `istream_iterator`
 (const `istream_iterator`
 <T, charT,traits,Distance>& x);

The parameterless iterator is the only legal constructor for an `end` condition.

destructor

Removes an instance of an `istream_iterator`.

Prototype `~istream_iterator();`

24.5.1.2 `Istream_iterator` Operations

Prototype `const T& operator*() const;`

Return A dereferenced iterator is returned.

Prototype `const T* operator->() const;`

Return The address of a dereferenced iterator is returned.

Prototype `istream_iterator<T,charT,traits,Distance>& operator++();`
 `istream_iterator<T,charT,traits,Distance>& operator++(int);`

Return The `this` pointer is returned.

Prototype `template <class T, class charT,`

```
class traits, class Distance> bool operator==  
    (const istream_iterator<T,charT, traits,  
     Distance> & x, const istream_iterator  
<T,charT,traits,Distance> & y);
```

Return A bool true value is retuned if the arguments ate the same.

24.5.2 Template Class Ostream_iterator

The ostream_iterator writes (using operator<<) successive elements onto the output stream.

Listing 9.9 Template Class Ostream_iterator Synopsis

```
namespace std {  
template <class T,  
          class charT = char, class traits = char_traits<charT> >  
class ostream_iterator:  
public iterator<output_iterator_tag, void, void, void, void> {  
  
public:  
    typedef charT char_type;  
    typedef traits traits_type;  
    typedef basic_ostream<charT,traits> ostream_type;  
  
    ostream_iterator(ostream_type& s);  
    ostream_iterator(ostream_type& s, const charT* delimiter);  
    ostream_iterator(const ostream_iterator<T,charT,traits>& x);  
    ~ostream_iterator();  
  
    ostream_iterator<T,charT,traits>& operator=(const T& value);  
    ostream_iterator<T,charT,traits>& operator*();  
    ostream_iterator<T,charT,traits>& operator++();  
    ostream_iterator<T,charT,traits>& operator++(int);  
  
private:  
    // basic_ostream<charT,traits>* out_stream; exposition only  
    // const char* delim; exposition only  
};  
}
```

Constructors

Creates and instance of an `ostream_iterator` object.

Prototype `ostream_iterator(ostream_type& s);`
`ostream_iterator(ostream_type& s, const charT* delimiter);`
`ostream_iterator(const ostream_iterator& x);`

destructor

Removes and instance of an `ostream_iterator` object.

Prototype `~ostream_iterator();`

24.5.2.2 Ostream_iterator Operations

Prototype `ostream_iterator& operator=(const T& value);`

Return Returns a value to an ostream iterator.

Prototype `ostream_iterator& operator*();`

Return The dereference iterator is returned.

Prototype `ostream_iterator& operator++();`
`ostream_iterator& operator++(int);`

Return The this pointer is returned.

24.5.3 Template Class `Istreambuf_iterator`

The `istreambuf_iterator` reads successive characters from the `istreambuf` object for which it was constructed.

An `istream_iterator` can only be used for a one pass algorithm.

Listing 9.10 Template Class `Istreambuf_iterator` Synopsis

```
namespace std {  
template<class charT, class traits = char_traits<charT> >  
class istreambuf_iterator  
: public iterator<input_iterator_tag, charT,  
typename traits::off_type, charT*, charT&> {
```

```
public:
typedef charT char_type;
typedef traits traits_type;
typedef typename traits::int_type int_type;
typedef basic_streambuf<charT,traits> streambuf_type;
typedef basic_istream<charT,traits> istream_type;

class proxy; // exposition only

public:
istreambuf_iterator() throw();
istreambuf_iterator(istream_type& s) throw();
istreambuf_iterator(streambuf_type* s) throw();
istreambuf_iterator(const proxy& p) throw();

charT operator*() const;
istreambuf_iterator<charT,traits>& operator++();
proxy operator++(int);
bool equal(istreambuf_iterator& b);
private:

streambuf_type* sbuf_; exposition only
};

template <class charT, class traits> bool operator==
  (const istreambuf_iterator<charT,traits>& a,
   const istreambuf_iterator<charT,traits>& b);
template <class charT, class traits> bool operator!=
  (const istreambuf_iterator<charT,traits>& a,
   const istreambuf_iterator<charT,traits>& b);
}
```

Constructors

An overloaded constructor is provided for creation of an `istreambuf_iterator` object.

Prototype `istreambuf_iterator() throw();`
 `istreambuf_iterator`
 `(basic_istream<charT,traits>& s) throw();`
 `istreambuf_iterator`
 `(basic_streambuf<charT,traits>* s) throw();`

```
istreambuf_iterator(const proxy& p) throw();
```

Istreambuf_iterator Operators

| | |
|-----------|---|
| Prototype | charT operator*() const |
| Return | A dereferenced character type is returned. |
| Prototype | istreambuf_iterator<charT,traits>& istreambuf_iterator<charT,traits>::operator++(); |
| Return | The this pointer is returned. |
| Prototype | template <class charT, class traits> bool operator== (const istreambuf_iterator<charT,traits>& a, const istreambuf_iterator<charT,traits>& b); |
| Return | True is returned if the arguments are equal. |
| Prototype | template <class charT, class traits> bool operator!= (const istreambuf_iterator<charT,traits>& a, const istreambuf_iterator<charT,traits>& b); |
| Return | True is returned if the arguments are not equal. |
| | equal |
| | An equality comparison. |
| Prototype | bool equal(istreambuf_iterator<charT,traits>& b); |
| Return | True is returned if the arguments are equal. |

24.5.4 Template Class Ostreambuf_iterator

The `ostreambuf_iterator` writes successive characters to the `ostreambuf` object for which it was constructed.

Listing 9.11 Template Class Ostreambuf_iterator Synopsis

```
namespace std {  
template <class charT, class traits = char_traits<charT> >  
class ostreambuf_iterator:
```

```
public iterator<output_iterator_tag, void, void, void, void> {  
  
public:  
typedef charT char_type;  
typedef traits traits_type;  
typedef basic_streambuf<charT,traits> streambuf_type;  
typedef basic_ostream<charT,traits> ostream_type;  
  
public:  
ostreambuf_iterator(ostream_type& s) throw();  
ostreambuf_iterator(streambuf_type* s) throw();  
ostreambuf_iterator& operator=(charT c);  
ostreambuf_iterator& operator*();  
ostreambuf_iterator& operator++();  
ostreambuf_iterator& operator++(int);  
  
bool failed() const throw();  
  
private:  
streambuf_type* sbuf_; exposition only  
};  
}
```

Constructors

The constructor is overloaded for creation of an `ostreambuf_iterator` object.

Prototype `ostreambuf_iterator(ostream_type& s) throw();`
 `ostreambuf_iterator(streambuf_type* s) throw();`

Ostreambuf_iterator Operators

Prototype `ostreambuf_iterator<charT,traits>&`
 `operator=(charT c);`

Return The result of the assignment is returned.

Prototype `ostreambuf_iterator<charT,traits>& operator*();`

Return The dereferenced `ostreambuf_iterator` is returned.

Prototype `ostreambuf_iterator<charT,traits>& operator++();`

```
ostreambuf_iterator<charT,traits>&
operator++(int);
```

Return The this pointer is returned.

failed

Reports a failure in writing.

Prototype `bool failed() const throw();`

Return The bool false value is returned if a write failure occurs.

Algorithms Library

This chapter discusses the algorithms library. These algorithms cover sequences, sorting, and numerics.

The Algorithms Library (clause 25)

The standard provides for various algorithms that a C++ program may use to perform algorithmic operations on containers and other sequences.

The chapter is constructed in the following sub sections and mirrors clause 25 of the ISO (the International Organization for Standardization) C++ Standard :

- [“25.1 Non-modifying Sequence Operations” on page 372](#)
- [“25.2 Mutating Sequence Operators” on page 377](#)
- [“25.3 Sorting And Related Operations” on page 385](#)
- [“25.4 C library algorithms” on page 396](#)

Header <algorithm>

The header algorithm provides classes, types and functions for use with the standard C++ libraries.

The standard algorithms can work with program defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

The names of the parameters used in this chapter reflect their usage.

A predicate parameter is used for a function object that returns a value testable as true. The binary predicate parameter takes two arguments.

Listing 10.1 Header <Algorithm> Synopsis

```
namespace std {
template<class InputIterator, class Function>
Function for_each(InputIterator first,
                  InputIterator last, Function f);
template<class InputIterator, class T>
InputIterator find(InputIterator first,
                   InputIterator last, const T& value);
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first,
                      InputIterator last, Predicate pred);
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end
  (ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_end
  (ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2,
   BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of
  (ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_first_of
  (ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2,
   BinaryPredicate pred);
template<class ForwardIterator>
ForwardIterator adjacent_find
  (ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find
  (ForwardIterator first, ForwardIterator last,
   BinaryPredicate pred);
template<class InputIterator, class T>
typename iterator_traits<InputIterator>
  ::difference_type count
```

```
(InputIterator first, InputIterator last, const T& value);
template<class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type count_if
    (InputIterator first, InputIterator last, Predicate pred);
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2);
template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, BinaryPredicate pred);
template<class InputIterator1, class InputIterator2>
bool equal
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2);
template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
bool equal
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search
    (ForwardIterator1 first1, ForwardIterator1 last1,
     ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate> ForwardIterator1 search
    (ForwardIterator1 first1, ForwardIterator1 last1,
     ForwardIterator2 first2, ForwardIterator2 last2,
     BinaryPredicate pred);
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n
    (ForwardIterator first, ForwardIterator last,
     Size count, const T& value);
template <class ForwardIterator, class Size,
          class T, class BinaryPredicate>
ForwardIterator1 search_n
    (ForwardIterator first, ForwardIterator last,
     Size count, const T& value, BinaryPredicate pred);

template<class InputIterator, class OutputIterator>
```

Algorithms Library

Header <algorithm>

```
OutputIterator copy
    (InputIterator first, InputIterator last,
     OutputIterator result);
template<class BidirectionalIterator1,
         class BidirectionalIterator2>
BidirectionalIterator2 copy_backward
    (BidirectionalIterator1 first, BidirectionalIterator1 last,
     BidirectionalIterator2 result);

template<class T> void swap(T& a, T& b);
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges
    (ForwardIterator1 first1, ForwardIterator1 last1,
     ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap
    (ForwardIterator1 a, ForwardIterator2 b);
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
OutputIterator transform
    (InputIterator first, InputIterator last,
     OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
OutputIterator transform
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, OutputIterator result,
     BinaryOperation binary_op);
template<class ForwardIterator, class T> void replace
    (ForwardIterator first, ForwardIterator last,
     const T& old_value, const T& new_value);
template<class ForwardIterator, class Predicate, class T>
void replace_if
    (ForwardIterator first, ForwardIterator last,
     Predicate pred, const T& new_value);
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy
    (InputIterator first, InputIterator last,
     OutputIterator result, const T& old_value, const T&
new_value);
template<class Iterator, class OutputIterator,
         class Predicate, class T>
```

```
OutputIterator replace_copy_if
    (Iterator first, Iterator last, OutputIterator result,
     Predicate pred, const T& new_value);
template<class ForwardIterator, class T>
void fill
    (ForwardIterator first, ForwardIterator last, const T& value);
template<class OutputIterator, class Size, class T>
void fill_n
    (OutputIterator first, Size n, const T& value);
template<class ForwardIterator, class Generator>
void generate
    (ForwardIterator first, ForwardIterator last,
     Generator gen);
template<class OutputIterator, class Size, class Generator>
void generate_n
    (OutputIterator first, Size n, Generator gen);
template<class ForwardIterator, class T>
ForwardIterator remove
    (ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if
    (ForwardIterator first, ForwardIterator last, Predicate pred);
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy
    (InputIterator first, InputIterator last,
     OutputIterator result, const T& value);
template<class InputIterator, class OutputIterator,
         class Predicate>
OutputIterator remove_copy_if
    (InputIterator first, InputIterator last,
     OutputIterator result, Predicate pred);
template<class ForwardIterator>
ForwardIterator unique
    (ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique
    (ForwardIterator first, ForwardIterator last,
     BinaryPredicate pred);
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy
    (InputIterator first, InputIterator last,
     OutputIterator result);
```

Algorithms Library

Header <algorithm>

```
template<class InputIterator, class OutputIterator,
         class BinaryPredicate>
OutputIterator unique_copy
    (InputIterator first, InputIterator last,
     OutputIterator result, BinaryPredicate pred);
template<class BidirectionalIterator>
void reverse
    (BidirectionalIterator first, BidirectionalIterator last);
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy
    (BidirectionalIterator first, BidirectionalIterator last,
     OutputIterator result);
template<class ForwardIterator>
void rotate
    (ForwardIterator first, ForwardIterator middle,
     ForwardIterator last);
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy
    (ForwardIterator first, ForwardIterator middle,
     ForwardIterator last, OutputIterator result);
template<class RandomAccessIterator>
void random_shuffle
    (RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle
    (RandomAccessIterator first, RandomAccessIterator last,
     RandomNumberGenerator& rand);

template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition
    (BidirectionalIterator first, BidirectionalIterator last,
     Predicate pred);
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition
    (BidirectionalIterator first, BidirectionalIterator last,
     Predicate pred);

template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void sort
    (RandomAccessIterator first, RandomAccessIterator last,
```

```
    Compare comp);
template<class RandomAccessIterator>
void stable_sort
    (RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void stable_sort
    (RandomAccessIterator first, RandomAccessIterator last,
 Compare comp);
template<class RandomAccessIterator>
void partial_sort
    (RandomAccessIterator first, RandomAccessIterator middle,
 RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void partial_sort
    (RandomAccessIterator first, RandomAccessIterator middle,
 RandomAccessIterator last, Compare comp);
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy
    (InputIterator first, InputIterator last,
 RandomAccessIterator result_first,
 RandomAccessIterator result_last);
template<class InputIterator, class RandomAccessIterator,
         class Compare>
RandomAccessIterator partial_sort_copy
    (InputIterator first, InputIterator last,
 RandomAccessIterator result_first,
 RandomAccessIterator result_last,      Compare comp);
template<class RandomAccessIterator>
void nth_element
    (RandomAccessIterator first, RandomAccessIterator nth,
 RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void nth_element
    (RandomAccessIterator first, RandomAccessIterator nth,
 RandomAccessIterator last, Compare comp);

template<class ForwardIterator, class T>
ForwardIterator lower_bound
    (ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound
    (ForwardIterator first, ForwardIterator last,
```

Algorithms Library

Header <algorithm>

```
    const T& value, Compare comp);
template<class ForwardIterator, class T>
ForwardIterator upper_bound
    (ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound
    (ForwardIterator first, ForwardIterator last,
     const T& value, Compare comp);
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
equal_range
    (ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator>
equal_range
    (ForwardIterator first, ForwardIterator last,
     const T& value, Compare comp);
template<class ForwardIterator, class T>
bool binary_search
    (ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
bool binary_search
    (ForwardIterator first, ForwardIterator last,
     const T& value, Compare comp);

template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator merge
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result);
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator merge
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result, Compare comp);
template<class BidirectionalIterator>
void inplace_merge
    (BidirectionalIterator first, BidirectionalIterator middle,
     BidirectionalIterator last);
template<class BidirectionalIterator,
```

```
    class Compare>
void inplace_merge
    (BidirectionalIterator first, BidirectionalIterator middle,
     BidirectionalIterator last, Compare comp);

template<class InputIterator1, class InputIterator2>
bool includes(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2,
         class Compare>
bool includes
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2, Compare comp);
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_union
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result);
template<class InputIterator1, class InputIterator2,
         class OutputIterator,      class Compare>
OutputIterator set_union
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result, Compare comp);
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_intersection
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result);
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_intersection
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result, Compare comp);
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_difference
    (InputIterator1 first1, InputIterator1 last1,
```

Algorithms Library

Header <algorithm>

```
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_difference
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result, Compare comp);
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_symmetric_difference
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result);
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_symmetric_difference
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result, Compare comp);

template<class RandomAccessIterator>
void push_heap
    (RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void push_heap
    (RandomAccessIterator first,
     RandomAccessIterator last, Compare comp);
template<class RandomAccessIterator>
void pop_heap
    (RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void pop_heap
    (RandomAccessIterator first, RandomAccessIterator last,
     Compare comp);
template<class RandomAccessIterator>
void make_heap
    (RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void make_heap
    (RandomAccessIterator first, RandomAccessIterator last,
     Compare comp);
```

```
template<class RandomAccessIterator>
void sort_heap
    (RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void sort_heap
    (RandomAccessIterator first, RandomAccessIterator last,
     Compare comp);

template<class T> const T& min(const T& a, const T& b);
template<class T, class Compare>
const T& min
    (const T& a, const T& b, Compare comp);
template<class T> const T& max(const T& a, const T& b);
template<class T, class Compare>
const T& max
    (const T& a, const T& b, Compare comp);
template<class ForwardIterator>
ForwardIterator min_element
    (ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
ForwardIterator min_element
    (ForwardIterator first, ForwardIterator last, Compare comp);
template<class ForwardIterator>
ForwardIterator max_element
    (ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
ForwardIterator max_element
    (ForwardIterator first, ForwardIterator last, Compare comp);
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2,
         class Compare>
bool lexicographical_compare
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2, Compare comp);

template<class BidirectionalIterator>
bool next_permutation
    (BidirectionalIterator first, BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
```

Algorithms Library

Header <algorithm>

```
bool next_permutation
    (BidirectionalIterator first, BidirectionalIterator last,
     Compare comp);
template<class BidirectionalIterator>
bool prev_permutation
    (BidirectionalIterator first, BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
bool prev_permutation
    (BidirectionalIterator first, BidirectionalIterator last,
     Compare comp);
}
```

25.1 Non-modifying Sequence Operations

Various algorithms are provided which do not modify the original object.

for_each

The function `for_each` is used to perform an operation for each element.

Prototype `template<class InputIterator, class Function>
Function for_each
 (InputIterator first, InputIterator last,
 Function f);`

Return The function `f` is returned.

find

The function `find` searches for the first element that contains the value passed.

Prototype `template<class InputIterator, class T>
InputIterator find
 (InputIterator first, InputIterator last,
 const T& value);`

Return Returns the type passed.

find_if

The function `find_if` searches for the first element that matches the criteria passed by the predicate.

Prototype `template<class InputIterator, class Predicate>
InputIterator find_if
 (InputIterator first, InputIterator last,
 Predicate pred);`

Return Returns the iterator of the matched value.

find_end

The function `find_end` searches for the last occurrence of a value.

Prototype `template<class ForwardIterator1,
 class ForwardIterator2>
ForwardIterator1 find_end
 (ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2);`

Prototype `template<class ForwardIterator1,
 class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_end
 (ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2, BinaryPredicate pred);`

Return Returns the iterator to the last value or the `last1` argument if none is found.

find_first_of

The function `find_first_of` searches for the first occurrence of a value.

Prototype `template<class ForwardIterator1,
 class ForwardIterator2>
ForwardIterator1 find_first_of
 (ForwardIterator1 first1,
 ForwardIterator1 last1,`

Algorithms Library

Header <algorithm>

```
    ForwardIterator2 first2,
    ForwardIterator2 last2);
```

Prototype `template<class ForwardIterator1,
 class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 find_first_of
 (ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2,
 ForwardIterator2 last2, BinaryPredicate pred);`

Return Returns the iterator to the last value or the `last1` argument if none is found.

adjacent_find

The function `adjacent_find` is used to search for two adjacent elements that are equal or equal according to the predicate argument.

Prototype `template<class ForwardIterator>
ForwardIterator adjacent_find
 (ForwardIterator first, ForwardIterator last);`

`template<class ForwardIterator,
 class BinaryPredicate>
ForwardIterator adjacent_find
 (ForwardIterator first, ForwardIterator last,
 BinaryPredicate pred);`

Return Returns the iterator to the first occurrence found or to `last` if no occurrence is found.

count

The function `count` is used to find the number of elements.

Prototype `template <class InputIterator, class T>
typename iterator_traits
 <InputIterator>::difference_type count
 (InputIterator first, InputIterator last,
 const T& value);`

Return Returns the number of elements (iterators) as an `iterator_traits difference_type`.

count_if

The function `count_if` is used to find the number of elements that match the criteria.

Prototype

```
template <class InputIterator, class Predicate>
typename iterator_traits
<InputIterator>::difference_type count_if
(InputIterator first, InputIterator last,
Predicate pred);
```

Return Returns the number of elements (iterators) as an `iterator_traits difference_type`.

mismatch

The function `mismatch` is used to find sequences that are not the same or differ according to the predicate criteria.

Prototype

```
template<class InputIterator1,
         class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch
(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2);
```

Prototype

```
template<class InputIterator1,
         class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch
(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, BinaryPredicate pred);
```

Return Returns a `pair<iterator>` that represent the beginning element and the range. If no mismatch is found the end and the corresponding range element is returned.

equal

The function `equal` is used to determine if a range two ranges are equal.

Prototype

```
template<class InputIterator1,
         class InputIterator2>
bool equal
(InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2);
```

Algorithms Library

Header <algorithm>

Prototype `template<class InputIterator1,`
 `class InputIterator2, class BinaryPredicate>`
 `bool equal`
 `(InputIterator1 first1, InputIterator1 last1,`
 `InputIterator2 first2, BinaryPredicate pred);`

Return A bool true is returned if the values are equal or meet the criteria of the predicate.

search

The function `search` is used to search for the first octanes of a sub-range that meet the criteria.

Prototype `template<class ForwardIterator1,`
 `class ForwardIterator2>`
 `ForwardIterator1 search`
 `(ForwardIterator1 first1,`
 `ForwardIterator1 last1,`
 `ForwardIterator2 first2,`
 `ForwardIterator2 last2);`

Prototype `template<class ForwardIterator1,`
 `class ForwardIterator2, class BinaryPredicate>`
 `ForwardIterator1 search`
 `(ForwardIterator1 first1,`
 `ForwardIterator1 last1,`
 `ForwardIterator2 first2,`
 `ForwardIterator2 last2, BinaryPredicate pred);`

Return An iterator to the first occurrence is returned or `last1` is returned if no criteria is met.

search_n

The function `search_n` is used to search for a number of consecutive elements with the same properties.

Prototype `template<class ForwardIterator,`
 `class Size, class T>`
 `ForwardIterator search_n`
 `(ForwardIterator first, ForwardIterator last,`
 `Size count, const T& value);`

Prototype `template<class ForwardIterator,`

```
    class Size, class T, class BinaryPredicate>
ForwardIterator search_n
    (ForwardIterator first,
     ForwardIterator last, Size count,
     const T& value, BinaryPredicate pred);
```

Return An iterator to the first occurrence is returned or `last1` is returned if no criteria is met.

25.2 Mutating Sequence Operators

Various algorithms are provided that are used to modify the original object.

copy

The function `copy` is used to copy a range.

Prototype

```
template<class InputIterator,
         class OutputIterator>
OutputIterator copy(
    InputIterator first, InputIterator last,
    OutputIterator result);
```

Return The position of the last copied element is returned.

copy_backward

The function `copy_backwards` is used to copy a range starting with the last element.

Prototype

```
template<class BidirectionalIterator1,
         class BidirectionalIterator2>
BidirectionalIterator2 copy_backward
    (BidirectionalIterator1 first,
     BidirectionalIterator1 last,
     BidirectionalIterator2 result);
```

Return The position of the last copied element is returned.

swap

The function `swap` is used to exchange values from two locations.

Prototype

```
template<class T> void swap(T& a, T& b);
```

Algorithms Library

Header <algorithm>

Return There is no return.

swap_ranges

The function `swap_ranges` is used swap elements of two ranges.

Prototype `template<class ForwardIterator1,
 class ForwardIterator2>
ForwardIterator2 swap_ranges
 (ForwardIterator1 first1,
 ForwardIterator1 last1,
 ForwardIterator2 first2);`

Return The position of the last swapped element is returned.

iter_swap

The function `iter_swap` is used to exchange two values pointed to by iterators.

Prototype `template<class ForwardIterator1,
 class ForwardIterator2>
void iter_swap
 (ForwardIterator1 a, ForwardIterator2 b);`

Return There is no return.

transform

The function `transform` is used to modify and copy elements of two ranges.

Prototype `template<class InputIterator,
 class OutputIterator, class UnaryOperation>
OutputIterator transform
 (InputIterator first, InputIterator last,
 OutputIterator result, UnaryOperation op);`

Prototype `template<class InputIterator1,
 class InputIterator2, class OutputIterator,
 class BinaryOperation>
OutputIterator transform
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, OutputIterator result,
 BinaryOperation binary_op);`

Return The position of the last transformed element is returned.

replace

The function `replace` is used to replace an element with another element of different value.

Prototype `template<class ForwardIterator, class T>
void replace
(ForwardIterator first, ForwardIterator last,
const T& old_value, const T& new_value);`

Prototype `template<class ForwardIterator,
class Predicate, class T>
void replace_if
(ForwardIterator first, ForwardIterator last,
Predicate pred, const T& new_value);`

Return There is no return.

replace_copy

The function `replace_copy` is used to replace specific elements while copying an entire range.

Prototype `template<class InputIterator,
class OutputIterator, class T>
OutputIterator replace_copy
(InputIterator first, InputIterator last,
OutputIterator result,
const T& old_value, const T& new_value);`

Return The position of the last copied element is returned.

replace_copy_if

The function `replace_copy_if` is used to replace specific elements that match certain criteria while copying the entire range.

Prototype `template<class Iterator,
class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if
(Iterator first, Iterator last,
OutputIterator result,
Predicate pred, const T& new_value);`

Algorithms Library

Header <algorithm>

Return The position of the last copied element is returned.

fill

The function `fill` is used to fill a range with values.

Prototype `template<class ForwardIterator, class T>
void fill
(ForwardIterator first, ForwardIterator last,
const T& value);`

Return There is no return value.

fill_n

The function `fill_n` is used to fill a number of elements with a specified value.

Prototype `template<class OutputIterator,
class Size, class T>
void fill_n
(OutputIterator first, Size n, const T& value);`

Return There is no return value.

generate

The function `generate` is used to replace elements with the result of an operation.

Prototype `template<class ForwardIterator, class Generator>
void generate
(ForwardIterator first, ForwardIterator last,
Generator gen);`

Return There is no return value.

generate_n

The function `generate_n` is used to replace a number of elements with the result of an operation.

Prototype `template<class OutputIterator,
class Size, class Generator>
void generate_n
(OutputIterator first, Size n, Generator gen);`

Return There is no return value.

remove

The function `remove` is used to remove elements with a specified value.

Prototype `template<class ForwardIterator, class T>
ForwardIterator remove
(ForwardIterator first, ForwardIterator last,
const T& value);`

Return The end of the resulting range is returned.

remove_if

The function `remove_if` is used to remove elements using a specified criteria.

Prototype `template<class ForwardIterator, class Predicate>
ForwardIterator remove_if
(ForwardIterator first, ForwardIterator last,
Predicate pred);`

Return The end of the resulting range is returned.

remove_copy

The function `remove_copy` is used to remove elements that do not match a value during a copy.

Prototype `template<class InputIterator,
class OutputIterator, class T>
OutputIterator remove_copy
(InputIterator first, InputIterator last,
OutputIterator result, const T& value);`

Return The end of the resulting range is returned.

remove_copy_if

The function `remove_copy_if` is used to remove elements that do not match a criteria while doing a copy.

Prototype `template<class InputIterator,
class OutputIterator, class Predicate>`

Algorithms Library

Header <algorithm>

```
OutputIterator remove_copy_if
    (InputIterator first, InputIterator last,
     OutputIterator result, Predicate pred);
```

Return The end of the resulting range is returned.

unique

The function `unique` is used to remove all adjacent duplicates.

Prototype `template<class ForwardIterator>
ForwardIterator unique
 (ForwardIterator first, ForwardIterator last);`

Prototype `template<class ForwardIterator,
 class BinaryPredicate>
ForwardIterator unique
 (ForwardIterator first, ForwardIterator last,
 BinaryPredicate pred);`

Return The end of the resulting range is returned.

unique_copy

The function `unique_copy` is used to remove adjacent duplicates while copying.

Prototype `template<class InputIterator,
 class OutputIterator>
OutputIterator unique_copy
 (InputIterator first, InputIterator last,
 OutputIterator result);`

Prototype `template<class InputIterator,
 class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy
 (InputIterator first, InputIterator last,
 OutputIterator result, BinaryPredicate pred);`

Return The end of the resulting range is returned.

reverse

The function `reverse` is used to reverse a sequence.

Prototype `template<class BidirectionalIterator>`

```
void reverse
    (BidirectionalIterator first,
     BidirectionalIterator last);
```

Return No value is returned.

reverse_copy

The function `reverse_copy` is used to copy the elements while reversing their order.

Prototype

```
template<class BidirectionalIterator,
          class OutputIterator>
OutputIterator reverse_copy
    (BidirectionalIterator first,
     BidirectionalIterator last,
     OutputIterator result);
```

Return The position of the last copied element is returned.

rotate

The function `rotate` is used to rotate the elements within a sequence.

Prototype

```
template<class ForwardIterator>
void rotate
    (ForwardIterator first, ForwardIterator middle,
     ForwardIterator last);
```

Return There is no return value.

rotate_copy

The function `rotate_copy` is used to copy a sequence with a rotated order.

Prototype

```
template<class ForwardIterator,
          class OutputIterator>
OutputIterator rotate_copy
    (ForwardIterator first, ForwardIterator middle,
     ForwardIterator last, OutputIterator result);
```

Return The position of the last copied element is returned.

random_shuffle

The function `random_shuffle` is used to exchange the order of the elements in a random fashion.

Prototype `template<class RandomAccessIterator>
void random_shuffle
 (RandomAccessIterator first,
 RandomAccessIterator last);`

Prototype `template<class RandomAccessIterator,
 class RandomNumberGenerator>
void random_shuffle
 (RandomAccessIterator first,
 RandomAccessIterator last,
 RandomNumberGenerator& rand);`

Return No value is returned.

partition

The function `partition` is used to change the order of the elements so that the elements that meet the criteria are first in order.

Prototype `template<class BidirectionalIterator,
 class Predicate>
BidirectionalIterator partition
 (BidirectionalIterator first,
 BidirectionalIterator last, Predicate pred);`

Return Returns an iterator to the first position where the predicate argument is false.

stable_partition

The function `stable_partition` is used to change the order of the elements so that the elements meet the criteria are first in order. The relative original order is preserved.

Prototype `template<class BidirectionalIterator,
 class Predicate>
BidirectionalIterator stable_partition
 (BidirectionalIterator first,
 BidirectionalIterator last, Predicate pred);`

Return Returns an iterator to the first position where the predicate argument is false.

25.3 Sorting And Related Operations

All of the sorting functions have two versions:, one that takes a function object for comparison and one that uses the less than operator.

sort

The function `sort` is used sorts the range according to the criteria.

Prototype `template<class RandomAccessIterator>`
`void sort`
 `(RandomAccessIterator first,`
 `RandomAccessIterator last);`

Prototype `template<class RandomAccessIterator,`
 `class Compare>`
`void sort(RandomAccessIterator first,`
 `RandomAccessIterator last, Compare comp);`

Return There is no return value.

stable_sort

The function `stable_sort` is used to sort the range but preserves the original order for equal elements.

Prototype `template<class RandomAccessIterator>`
`void stable_sort`
 `(RandomAccessIterator first,`
 `RandomAccessIterator last);`

Prototype `template<class RandomAccessIterator,`
 `class Compare>`
`void stable_sort`
 `(RandomAccessIterator first,`
 `RandomAccessIterator last, Compare comp);`

Return There is no return value.

partial_sort

The function `partial_sort` is used to sort a sub-range leaving the rest unsorted.

Prototype `template<class RandomAccessIterator>`
 `void partial_sort`
 (RandomAccessIterator first,
 RandomAccessIterator middle,
 RandomAccessIterator last);

`template<class RandomAccessIterator,`
 `class Compare>`
 `void partial_sort`
 (RandomAccessIterator first,
 RandomAccessIterator middle,
 RandomAccessIterator last, Compare comp);

Return There is no return value.

partial_sort_copy

The function `partial_sort_copy` is used to copy a partially sorted sequence.

Prototype `template<class InputIterator,`
 `class RandomAccessIterator>`
 `RandomAccessIterator partial_sort_copy`
 (InputIterator first, InputIterator last,
 RandomAccessIterator result_first,
 RandomAccessIterator result_last);

`template<class InputIterator,`
 `class RandomAccessIterator, class Compare>`
 `RandomAccessIterator partial_sort_copy`
 (InputIterator first, InputIterator last,
 RandomAccessIterator result_first,
 RandomAccessIterator result_last, Compare comp);

Return The position at the end of the copied elements is returned.

nth_element

The function `nth_element` is used to sort based upon a specified position.

Prototype `template<class RandomAccessIterator>
void nth_element
(RandomAccessIterator first,
RandomAccessIterator nth,
RandomAccessIterator last);
template<class RandomAccessIterator,
class Compare>
void nth_element
(RandomAccessIterator first,
RandomAccessIterator nth,
RandomAccessIterator last, Compare comp);`

Return There is no value returned.

lower_bound

The function `lower_bound` is used to find the first position that an element may be inserted without changing the order.

Prototype `template<class ForwardIterator, class T>
ForwardIterator lower_bound
(ForwardIterator first, ForwardIterator last,
const T& value);
template<class ForwardIterator,
class T, class Compare>
ForwardIterator lower_bound
(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);`

Return The position where the element can be inserted is returned.

upper_bound

The function `upper_bound` is used to find the last position that an element may be inserted without changing the order.

Prototype `template<class ForwardIterator, class T>
ForwardIterator upper_bound
(ForwardIterator first, ForwardIterator last,
const T& value);
template<class ForwardIterator,
class T, class Compare>
ForwardIterator upper_bound
(ForwardIterator first, ForwardIterator last,`

```
    const T& value, Compare comp);
```

Return The position where the element can be inserted is returned.

equal_range

The function `equal_range` is used to find the range as a pair where an element can be inserted without altering the order.

Prototype

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range
(ForwardIterator first,
 ForwardIterator last, const T& value);
```

Prototype

```
template<class ForwardIterator,
         class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range
(ForwardIterator first,
 ForwardIterator last, const T& value,
 Compare comp);
```

Return The range as a pair<> where the element can be inserted is returned.

binary_search

The function `binary_search` is used to see if a value is present in a range or that a value meets a criteria within that range.

Prototype

```
template<class ForwardIterator, class T>
bool binary_search
(ForwardIterator first, ForwardIterator last,
 const T& value);
```

Prototype

```
template<class ForwardIterator,
         class T, class Compare>
bool binary_search
(ForwardIterator first, ForwardIterator last,
 const T& value, Compare comp);
```

Return The bool value true is met if any element meets the criteria.

merge

The function `merge` is used to combine two sorted ranges.

Prototype

```
template<class InputIterator1,
```

```
        class InputIterator2, class OutputIterator>
OutputIterator merge
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result);
```

Prototype template<class InputIterator1,
 class InputIterator2,
 class OutputIterator, class Compare>
OutputIterator merge
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result, Compare comp);

Return The position of the first element not overwritten is returned.

inplace_merge

The function `replacement` is used to merge consecutive sequences to the first for a concatenation.

Prototype template<class BidirectionalIterator>
void inplace_merge
 (BidirectionalIterator first,
 BidirectionalIterator middle,
 BidirectionalIterator last);

Prototype template<class BidirectionalIterator,
 class Compare>
void inplace_merge
 (BidirectionalIterator first,
 BidirectionalIterator middle,
 BidirectionalIterator last, Compare comp);

Return There is no value returned.

includes

The function `includes` is used to determine if every element meets a specified criteria.

Prototype template<class InputIterator1,
 class InputIterator2>
bool includes
 (InputIterator1 first1, InputIterator1 last1,

Algorithms Library

Header <algorithm>

 InputIterator2 first2, InputIterator2 last2);

Prototype template<class InputIterator1,
 class InputIterator2, class Compare>
bool includes
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 Compare comp);

Return The bool value true is retuned if all values match or false if one or more does not meet the criteria.

set_union

The function `set_union` is used to process the sorted union of two ranges.

Prototype template<class InputIterator1,
 class InputIterator2, class OutputIterator>
OutputIterator set_union
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result);

Prototype template<class InputIterator1,
 class InputIterator2,
 class OutputIterator, class Compare>
OutputIterator set_union
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result, Compare comp);

Return The end of the constructed range is returned.

set_intersection

The function `set_intersection` is used to process the intersection of two ranges.

Prototype template<class InputIterator1,
 class InputIterator2, class OutputIterator>
OutputIterator set_intersection
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result);

Prototype `template<class InputIterator1,
 class InputIterator2, class OutputIterator,
 class Compare>
OutputIterator set_intersection
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result, Compare comp);`

Return The end of the constructed range is returned.

set_difference

The function `set_difference` is used to process all of the elements of one range that are not part of another range.

Prototype `template<class InputIterator1,
 class InputIterator2, class OutputIterator>
OutputIterator set_difference
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result);`

Prototype `template<class InputIterator1,
 class InputIterator2,
 class OutputIterator, class Compare>
OutputIterator set_difference
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result, Compare comp);`

Return The end of the constructed range is returned.

set_symmetric_difference

The function `set_symmetric_difference` is used to process all of the elements that are in only one of two ranges.

Prototype `template<class InputIterator1,
 class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference
 (InputIterator1 first1, InputIterator1 last1,
 InputIterator2 first2, InputIterator2 last2,
 OutputIterator result);`

Prototype `template<class InputIterator1,`

Algorithms Library

Header <algorithm>

```
class InputIterator2,
class OutputIterator, class Compare>
OutputIterator set_symmetric_difference
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     OutputIterator result, Compare comp);
```

Return The end of the constructed range is returned.

push_heap

The function `push_heap` is used to add an element to a heap.

Prototype `template<class RandomAccessIterator>
void push_heap
 (RandomAccessIterator first,
 RandomAccessIterator last);`

Prototype `template<class RandomAccessIterator,
 class Compare>
void push_heap
 (RandomAccessIterator first,
 RandomAccessIterator last, Compare comp);`

Return There is no value returned.

pop_heap

The function `pop_heap` is used to remove an element from a heap.

Prototype `template<class RandomAccessIterator>
void pop_heap
 (RandomAccessIterator first,
 RandomAccessIterator last);`

Prototype `template<class RandomAccessIterator,
 class Compare>
void pop_heap
 (RandomAccessIterator first,
 RandomAccessIterator last,
 Compare comp);`

Return There is no value returned.

make_heap

The function `make_heap` is used to convert a range into a heap.

Prototype `template<class RandomAccessIterator>
void make_heap
(RandomAccessIterator first,
RandomAccessIterator last);`

Prototype `template<class RandomAccessIterator,
class Compare>
void make_heap(
 RandomAccessIterator first,
 RandomAccessIterator last,
 Compare comp);`

Return There is no value returned.

sort_heap

The function `sort_heap` is used to sort a heap.

Prototype `template<class RandomAccessIterator>
void sort_heap
(RandomAccessIterator first,
RandomAccessIterator last);`

Prototype `template<class RandomAccessIterator,
class Compare>
void sort_heap
(RandomAccessIterator first,
RandomAccessIterator last,
Compare comp);`

NOTE This result is not stable

Return There is no value returned.

min

The function `min` is used to determine the lesser of two objects by value or based upon a comparison.

Prototype `template<class T>`

Algorithms Library

Header <algorithm>

```
const T& min
    (const T& a, const T& b);
```

Prototype template<class T, class Compare>
const T& min
 (const T& a, const T& b, Compare comp);

Return The lesser of the two objects is returned.

max

The function `max` is used to determine the greater of two objects by value or based upon a comparison.

Prototype template<class T>
const T& max
 (const T& a, const T& b);

Prototype template<class T, class Compare>
const T& max
 (const T& a, const T& b, Compare comp);

Return The greater of the two objects is returned.

min_element

The function `min_element` is used to determine the lesser element within a range based upon a value or a comparison.

Prototype template<class ForwardIterator>
ForwardIterator min_element
 (ForwardIterator first, ForwardIterator last);

Prototype template<class ForwardIterator, class Compare>
ForwardIterator min_element
 (ForwardIterator first, ForwardIterator last,
 Compare comp);

Return The position of the element is returned.

max_element

The function `max_element` is used to determine the greater element within a range based upon a value or a comparison.

Prototype template<class ForwardIterator>

```
ForwardIterator max_element
    (ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
ForwardIterator max_element
    (ForwardIterator first, ForwardIterator last,
     Compare comp);
```

Return The position of the element is returned.

lexicographical_compare

The function `lexicographical_compare` is used to determine if a range is lexicographically less than another.

Prototype

```
template<class InputIterator1,
         class InputIterator2>
bool lexicographical_compare
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2);
```

Prototype

```
template<class InputIterator1,
         class InputIterator2, class Compare>
bool lexicographical_compare
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     Compare comp);
```

Return Returns true if the first argument is less than the second and false for all other conditions.

next_permutation

The function `next_permutation` is used to sort in an ascending order based upon lexicographical criteria.

Prototype

```
template<class BidirectionalIterator>
bool next_permutation
    (BidirectionalIterator first,
     BidirectionalIterator last);
```

Prototype

```
template<class BidirectionalIterator,
         class Compare>
bool next_permutation
    (BidirectionalIterator first,
     BidirectionalIterator last, Compare comp);
```

Algorithms Library

Header <algorithm>

Return Returns true if all elements have been sorted.

prev_permutation

The function `prev_permutation` is used to sort in an descending order based upon lexicographical criteria.

Prototype `template<class BidirectionalIterator>
bool prev_permutation
 (BidirectionalIterator first,
 BidirectionalIterator last);`

Prototype `template<class BidirectionalIterator,
 class Compare>
bool prev_permutation
 (BidirectionalIterator first,
 BidirectionalIterator last, Compare comp);`

Return Returns true if all elements have been sorted.

25.4 C library algorithms

The C++ header <cstdlib> provides two variations from the standard C header stdlib.h for searching and sorting.

bsearch

The function signature of `bsearch`

```
bsearch(const void *, const void *, size_t,  
size_t, int (*)(const void *, const void *));
```

is replaced by

```
extern "C" void *bsearch  
(const void * key, const void * base,  
size_t nmemb, size_t size,  
int (* compar)(const void *, const void *));
```

and

```
extern "C++" void *bsearch  
(const void * key, const void * base,  
size_t nmemb, size_t size,  
int (* compar)(const void *, const void *));
```

qsort

The function signature of qsort

```
qsort(void *, size_t, size_t,  
      int (*)(const void *, const void *));
```

is replaced by

```
extern "C" void qsort  
void* base, size_t nmemb, size_t size,  
int (* compar)(const void*, const void*));
```

and

```
extern "C++" void qsort  
(void* base, size_t nmemb, size_t size,  
int (* compar)(const void*, const void*));
```

Algorithms Library

Header <algorithm>

Numerics Library

This chapter is a reference guide to the ANSI/ISO standard Numeric classes which are used to perform the semi-numerical operations.

The Numerics Library (clause 26)

The chapter is constructed in the following sub sections and mirrors clause 26 of the ISO (the International Organization for Standardization) C++ Standard :

- [“26.1 Numeric type requirements” on page 399](#)
- [“The Complex Class Library \(clause 26.2\)” on page 433](#) is a separate chapter
- [“26.3 Numeric arrays” on page 400](#)
- [“26.4 Generalized Numeric Operations” on page 427](#)
- [“26.5 C Library” on page 430](#)

26.1 Numeric type requirements

The complex and valarray components are parameterized by the type of information they contain and manipulate.

A C++ program shall instantiate these components only with a type `TYPE` that satisfies the following requirements:

`T` is not an abstract class (it has no pure virtual member functions);

- `TYPE` is not a reference type;
- `TYPE` is not cv-qualified;
- If `TYPE` is a class, it has a public default constructor;

- If `TYPE` is a class, it has a public copy constructor with the signature `TYPE::TYPE(const TYPE&)`
- If `TYPE` is a class, it has a public destructor;
- If `TYPE` is a class, it has a public assignment operator whose signature is either
 - `TYPE& TYPE::operator=(const TYPE&)`
 - or
 - `TYPE& TYPE::operator=(TYPE)`
- If `TYPE` is a class, the assignment operator, copy and default constructors, and destructor shall correspond to each other as far as an initialization of raw storage using the default constructor, followed by assignment, is the equivalent to initialization of raw storage using the copy constructor.

Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object.

- If `TYPE` is a class, it shall not overload unary `operator&`.

If an operation on `TYPE` throws an exception then the effects are undefined.

Specific classes member functions or general functions may have other restrictions.

26.3 Numeric arrays

The numeric array library consists of several classes and non member operators for the manipulation of array objects.

- [“26.3.2 Template Class Valarray” on page 405](#)
- [“26.3.3 Valarray Non-member Operations” on page 411](#)
- [“26.3.4 Class slice” on page 416](#)
- [“26.3.5 Template Class Slice_array” on page 418](#)
- [“26.3.6 Class Gslice” on page 419](#)
- [“26.3.7 Template Class Cslice_array” on page 421](#)
- [“26.3.8 Template Class Mask_array” on page 423](#)
- [“26.3.9 Template Class Indirect_array” on page 425](#)

26.3.1 Header <valarray>

A synopsis of the `valarray` header is listed in ["Header <valarray> synopsis"](#)

Listing 11.1 Header <valarray> synopsis

```
namespace std {
template<class T> class valarray;
class slice;
template<class T> class slice_array;
class gslice;
template<class T> class gslice_array;
template<class T> class mask_array;
template<class T> class indirect_array;
template<class T> valarray<T> operator*
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator*
    (const valarray<T>&, const T&);
template<class T> valarray<T> operator*
    (const T&, const valarray<T>&);
template<class T> valarray<T> operator/
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/
    (const valarray<T>&, const T&);
template<class T> valarray<T> operator/ (
    const T&, const valarray<T>&);
template<class T> valarray<T> operator%
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator%
    (const valarray<T>&, const T&);
template<class T> valarray<T> operator%
    (const T&, const valarray<T>&);
template<class T> valarray<T> operator+
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+
    (const valarray<T>&, const T&);
template<class T> valarray<T> operator+
    (const T&, const valarray<T>&);
template<class T> valarray<T> operator-
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator-
```

```
(const valarray<T>&, const T&);  
template<class T> valarray<T> operator-  
    (const T&, const valarray<T>&);  
template<class T> valarray<T> operator^  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<T> operator^  
    (const valarray<T>&, const T&);  
template<class T> valarray<T> operator^  
    (const T&, const valarray<T>&);  
template<class T> valarray<T> operator&  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<T> operator&  
    (const valarray<T>&, const T&);  
template<class T> valarray<T> operator&  
    (const T&, const valarray<T>&);  
template<class T> valarray<T> operator|  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<T> operator|  
    (const valarray<T>&, const T&);  
template<class T> valarray<T> operator|  
    (const T&, const valarray<T>&);  
template<class T> valarray<T> operator<<  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<T> operator<<  
    (const valarray<T>&, const T&);  
template<class T> valarray<T> operator<<  
    (const T&, const valarray<T>&);  
template<class T> valarray<T> operator>>  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<T> operator>>  
    (const valarray<T>&, const T&);  
template<class T> valarray<T> operator>>  
    (const T&, const valarray<T>&);  
template<class T> valarray<bool> operator&&  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator&&  
    (const valarray<T>&, const T&);  
template<class T> valarray<bool> operator&&  
    (const T&, const valarray<T>&);  
template<class T> valarray<bool> operator||  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator||
```

```
(const valarray<T>&, const T&);  
template<class T> valarray<bool> operator||  
    (const T&, const valarray<T>&);  
template<class T> valarray<bool> operator==  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator==  
    (const valarray<T>&, const T&);  
template<class T> valarray<bool> operator==  
    (const T&, const valarray<T>&);  
template<class T> valarray<bool> operator!=  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator!=  
    (const valarray<T>&, const T&);  
template<class T> valarray<bool> operator!=  
    (const T&, const valarray<T>&);  
template<class T> valarray<bool> operator<  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator<  
    (const valarray<T>&, const T&);  
template<class T> valarray<bool> operator<  
    (const T&, const valarray<T>&);  
template<class T> valarray<bool> operator>  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator>  
    (const valarray<T>&, const T&);  
template<class T> valarray<bool> operator>  
    (const T&, const valarray<T>&);  
template<class T> valarray<bool> operator<=>  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator<=>  
    (const valarray<T>&, const T&);  
template<class T> valarray<bool> operator<=>  
    (const T&, const valarray<T>&);  
template<class T> valarray<bool> operator>=  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<bool> operator>=  
    (const valarray<T>&, const T&);  
template<class T> valarray<bool> operator>=  
    (const T&, const valarray<T>&);  
template<class T> valarray<T> abs  
    (const valarray<T>&);  
template<class T> valarray<T> acos
```

```
(const valarray<T>&);  
template<class T> valarray<T> asin  
    (const valarray<T>&);  
template<class T> valarray<T> atan  
    (const valarray<T>&);  
template<class T> valarray<T> atan2  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<T> atan2(  
    const valarray<T>&, const T&);  
template<class T> valarray<T> atan2  
    (const T&, const valarray<T>&);  
template<class T> valarray<T> cos  
    (const valarray<T>&);  
template<class T> valarray<T> cosh  
    (const valarray<T>&);  
template<class T> valarray<T> exp  
    (const valarray<T>&);  
template<class T> valarray<T> log  
    (const valarray<T>&);  
template<class T> valarray<T> log10  
    (const valarray<T>&);  
template<class T> valarray<T> pow  
    (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<T> pow  
    (const valarray<T>&, const T&);  
template<class T> valarray<T> pow  
    (const T&, const valarray<T>&);  
template<class T> valarray<T> sin  
    (const valarray<T>&);  
template<class T> valarray<T> sinh  
    (const valarray<T>&);  
template<class T> valarray<T> sqrt  
    (const valarray<T>&);  
template<class T> valarray<T> tan  
    (const valarray<T>&);  
template<class T> valarray<T> tanh  
    (const valarray<T>&);  
}
```

26.3.2 Template Class Valarray

The template class valarray is a single direction smart array with element indexing beginning with the zero element.

Listing 11.2 Template Class Valarray Synopsis

```
namespace std {
template<class T> class valarray {
public:
typedef T value_type;

valarray();
explicit valarray(size_t);
valarray(const T&, size_t);
valarray(const T*, size_t);
valarray(const valarray&);

valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
~valarray();

valarray<T>& operator=(const valarray<T>&);
valarray<T>& operator=(const T&);
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);

T operator[](size_t) const;
T& operator[](size_t);

valarray<T> operator[](slice) const;
slice_array<T> operator[](slice);
valarray<T> operator[](const gslice&) const;
gslice_array<T> operator[](const gslice&);
valarray<T> operator[](const valarray<bool>&) const;
mask_array<T> operator[](const valarray<bool>&);
valarray<T> operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);
```

```
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<T> operator!() const;

valarray<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%=(const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator&= (const T&);
valarray<T>& operator|= (const T&);
valarray<T>& operator<<=(const T&);
valarray<T>& operator>=(const T&);
valarray<T>& operator*= (const valarray<T>&);
valarray<T>& operator/= (const valarray<T>&);
valarray<T>& operator%=(const valarray<T>&);
valarray<T>& operator+= (const valarray<T>&);
valarray<T>& operator-= (const valarray<T>&);
valarray<T>& operator^= (const valarray<T>&);
valarray<T>& operator|= (const valarray<T>&);
valarray<T>& operator&= (const valarray<T>&);
valarray<T>& operator<<=(const valarray<T>&);
valarray<T>& operator>=(const valarray<T>&);

size_t size() const;
T sum() const;
T min() const;
T max() const;
valarray<T> shift (int) const;
valarray<T> cshift(int) const;
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
void resize(size_t sz, T c = T());
};
```

Constructors

The class `valarray` provides overloaded constructors to create an object of `valarray` in several manners.

Prototype `valarray();
 explicit valarray(size_t);
 valarray(const T&, size_t);
 valarray(const T*, size_t);
 valarray(const valarray<T>&);
 valarray(const slice_array<T>&);
 valarray(const gslice_array<T>&);
 valarray(const mask_array<T>&);
 valarray(const indirect_array<T>&);`

Destructor

Removes a `valarray` object from memory.

Prototype `~valarray();`

Assignment Operator

The `valarray` class provides for various means of assignment to an already created object.

Prototype `valarray<T>& operator=(const valarray<T>&);
 valarray<T>& operator=(const T&);
 valarray<T>& operator=(const slice_array<T>&);
 valarray<T>& operator=(const gslice_array<T>&);
 valarray<T>& operator=(const mask_array<T>&);
 valarray<T>& operator=(const indirect_array<T>&);`

Return A `valarray` object is returned.

26.3.2.3 `valarray` element access

An index operator is provided for single element access of `valarray` objects.

`operator[]`

This operator provide element access for read and write operations.

Prototype `T operator[](size_t) const;
 T& operator[](size_t);`

Return A type is returned.

26.3.2.4 valarray subset operations

An index operator is provided for subset array access.

operator[]

The index operator is specialized for subset access to allow both read and write operations.

Prototype `valarray<T> operator[]
 (slice) const;
slice_array<T> operator[]
 (slice);
valarray<T> operator[]
 (const gslice&) const;
gslice_array<T> operator[]
 (const gslice&);
valarray<T> operator[]
 (const valarray<bool>&) const;
mask_array<T> operator[]
 (const valarray<bool>&);
valarray<T> operator[]
 (const valarray<size_t>&) const;
indirect_array<T> operator[]
 (const valarray<size_t>&);`

Return The return corresponds to the index type.

26.3.2.5 valarray unary operators

The `valarray` class provides operators for array manipulation.

Prototype `valarray<T> operator+() const;`
Returns a `valarray` sum of `x+y`;

Prototype `valarray<T> operator-() const;`
Returns a `valarray` result of `x-y`;

Prototype `valarray<T> operator~() const;`
Returns a `valarray` result of `x~y`;

Prototype `valarray<bool> operator!() const;`
Returns a `bool` `valarray` of `!x`;

26.3.2.6 Valarray Computed Assignment

The valarray class provides for a means of compound assignment and math operation. A valarray object is returned.

| | |
|-----------|--|
| Prototype | <code>valarray<T>& operator*= (const valarray<T>&); valarray<T>& operator*= (const T&);</code> Returns a valarray result of <code>x*=y;</code> |
| Prototype | <code>valarray<T>& operator/= (const valarray<T>&); valarray<T>& operator/= (const T&);</code> Returns a valarray result of <code>x/=y;</code> |
| Prototype | <code>valarray<T>& operator%= (const valarray<T>&); valarray<T>& operator%= (const T&);</code> Returns a valarray result of <code>x%=y;</code> |
| Prototype | <code>valarray<T>& operator+= (const valarray<T>&); valarray<T>& operator+= (const T&);</code> Returns a valarray result of <code>x+=y;</code> |
| Prototype | <code>valarray<T>& operator-= (const valarray<T>&); valarray<T>& operator-= (const T&);</code> Returns a valarray result of <code>x-=y;</code> |
| Prototype | <code>valarray<T>& operator^= (const valarray<T>&); valarray<T>& operator^= (const T&);</code> Returns a valarray result of <code>x^=y;</code> |
| Prototype | <code>valarray<T>& operator&= (const valarray<T>&); valarray<T>& operator&= (const T&);</code> Returns a valarray result of <code>x&=y;</code> |
| Prototype | <code>valarray<T>& operator = (const valarray<T>&); valarray<T>& operator = (const T&);</code> Returns a valarray result of <code>x!=y;</code> |
| Prototype | <code>valarray<T>& operator<<=(const valarray<T>&); valarray<T>& operator<<=(const T&);</code> Returns a valarray result of <code>x<<=y;</code> |
| Prototype | <code>valarray<T>& operator>>=(const valarray<T>&);</code> |

```
valarray<T>& operator>>=(const T&);
```

Returns a valarray result of `x>>=y;`

26.3.2.7 Valarray Member Functions

The `valarray` class provides member functions for array information.

size

Tells the size of the array.

Prototype `size_t size() const;`

Return Returns the size of the array.

sum

Tells the sum of the array elements.

Prototype `T sum() const;`

Return Returns the sum of the array elements.

min

Tells the smallest element of an array.

Prototype `T min() const;`

Return Returns the smallest element in an array.

max

Tells the largest element in an array.

Prototype `T max() const;`

Return Returns the largest element in an array.

shift

Returns a new array where the elements have been shifted a set amount.

Prototype `valarray<T> shift(int n) const;`

Return Returns the modified array.

cshift

A cyclical shift of an array.

Prototype `valarray<T> cshift(int n) const;`

Return Returns the modified array.

apply

Processes the elements of an array.

Prototype `valarray<T> apply(T func(T)) const;`
`valarray<T> apply(T func(const T&)) const;`

Remarks This function “applies” the function specified to all the elements of an array.

Return Return the modified array.

resize

Resizes an array and initializes the elements

Prototype `void resize(size_t sz, T c = T());`

Remarks If no object is provided the array is initialized with the default constructor.

26.3.3 Valarray Non-member Operations

Non-member operators are provided for manipulation or arrays.

26.3.3.1 Valarray Binary Operators

Non-member valarray operators are provided for the manipulation of arrays.

Prototype `template<class T> valarray<T> operator*
(const valarray<T>&, const valarray<T>&);`

Prototype `template<class T> valarray<T> operator/
(const valarray<T>&, const valarray<T>&);`

Prototype `template<class T> valarray<T> operator%`

```
(const valarray<T>&, const valarray<T>&);

Prototype template<class T> valarray<T> operator+
(const valarray<T>&, const valarray<T>&);

Prototype template<class T> valarray<T> operator-
(const valarray<T>&, const valarray<T>&);

Prototype template<class T> valarray<T> operator^
(const valarray<T>&, const valarray<T>&);

Prototype template<class T> valarray<T> operator&
(const valarray<T>&, const valarray<T>&);

Prototype template<class T> valarray<T> operator|
(const valarray<T>&, const valarray<T>&);

Prototype template<class T> valarray<T> operator<<
(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>
(const valarray<T>&, const valarray<T>&);

Prototype template<class T> valarray<T> operator*
(const valarray<T>&, const T&);

Prototype template<class T> valarray<T> operator*
(const T&, const valarray<T>&);

Prototype template<class T> valarray<T> operator/
(const valarray<T>&, const T&);
template<class T> valarray<T> operator/
(const T&, const valarray<T>&);

Prototype template<class T> valarray<T> operator%
(const valarray<T>&, const T&);
template<class T> valarray<T> operator%
(const T&, const valarray<T>&);

Prototype template<class T> valarray<T> operator+
(const valarray<T>&, const T&);
template<class T> valarray<T> operator+
(const T&, const valarray<T>&);

Prototype template<class T> valarray<T> operator-
(const valarray<T>&, const T&);
template<class T> valarray<T> operator-
```

| | |
|-----------|--|
| | (const T&, const valarray<T>&); |
| Prototype | template<class T> valarray<T> operator^ (const valarray<T>&, const T&); template<class T> valarray<T> operator^ (const T&, const valarray<T>&); |
| Prototype | template<class T> valarray<T> operator& (const valarray<T>&, const T&); template<class T> valarray<T> operator& (const T&, const valarray<T>&); |
| Prototype | template<class T> valarray<T> operator (const valarray<T>&, const T&); template<class T> valarray<T> operator (const T&, const valarray<T>&); |
| Prototype | template<class T> valarray<T> operator<< (const valarray<T>&, const T&); template<class T> valarray<T> operator<< (const T&, const valarray<T>&); |
| Prototype | template<class T> valarray<T> operator>> (const valarray<T>&, const T&); template<class T> valarray<T> operator>> (const T&, const valarray<T>&); |
| Return | Each operator returns an array whose length is equal to the lengths of the argument arrays and initialized with the result of applying the operator. |

26.3.3.2 Valarray Logical Operators

The `valarray` class provides logical operators for the comparison of like arrays.

| | |
|-----------|--|
| Prototype | template<class T> valarray<bool> operator== (const valarray<T>&, const valarray<T>&); |
| Prototype | template<class T> valarray<bool> operator!= (const valarray<T>&, const valarray<T>&); |
| Prototype | template<class T> valarray<bool> operator< (const valarray<T>&, const valarray<T>&); |
| Prototype | template<class T> valarray<bool> operator> |

```
(const valarray<T>&, const valarray<T>&);  
  
Prototype template<class T> valarray<bool> operator<=  
          (const valarray<T>&, const valarray<T>&);  
  
Prototype template<class T> valarray<bool> operator>=  
          (const valarray<T>&, const valarray<T>&);  
  
Prototype template<class T> valarray<bool> operator&&  
          (const valarray<T>&, const valarray<T>&);  
  
Prototype template<class T> valarray<bool> operator||  
          (const valarray<T>&, const valarray<T>&);  
  
Return All of the logical operators returns a bool array whose length is  
       equal to the length of the array arguments. The elements of the  
       returned array are initialized with a boolean result of the match.
```

Non-member logical operations

Non-member logical operators are provided to allow for variations
of order of the operation.

```
Prototype template<class T> valarray<bool> operator==  
          (const valarray&, const T&);  
template<class T> valarray<bool> operator==  
          (const T&, const valarray&);  
  
Prototype template<class T> valarray<bool> operator!=  
          (const valarray&, const T&);  
template<class T> valarray<bool> operator!=  
          (const T&, const valarray&);  
  
Prototype template<class T> valarray<bool> operator<  
          (const valarray&, const T&);  
template<class T> valarray<bool> operator<  
          (const T&, const valarray&);  
  
Prototype template<class T> valarray<bool> operator>  
          (const valarray&, const T&);  
template<class T> valarray<bool> operator>  
          (const T&, const valarray&);  
  
Prototype template<class T> valarray<bool> operator<=  
          (const valarray&, const T&);  
template<class T> valarray<bool> operator<=
```

```
(const T&, const valarray&);
```

Prototype template<class T> valarray<bool> operator>=
 (const valarray&, const T&);
 template<class T> valarray<bool> operator>=
 (const T&, const valarray&);

Prototype template<class T> valarray<bool> operator&&
 (const valarray<T>&, const T&);
 template<class T> valarray<bool> operator&&
 (const T&, const valarray<T>&);

Prototype template<class T> valarray<bool> operator|||
 (const valarray<T>&, const T&);
 template<class T> valarray<bool> operator|||
 (const T&, const valarray<T>&);

Return The result of these operations is bool array whose length is equal to the length of the array argument. Each element of the returned array is the result of a logical match.

26.3.3.3 valarray transcendentals

Trigonometric and exponential functions are provided for the valarray classes.

Prototype template<class T> valarray<T> abs
 (const valarray<T>&);

Prototype template<class T> valarray<T> acos
 (const valarray<T>&);

Prototype template<class T> valarray<T> asin
 (const valarray<T>&);

Prototype template<class T> valarray<T> atan
 (const valarray<T>&);

Prototype template<class T> valarray<T> atan2
 (const valarray<T>&, const valarray<T>&);
 template<class T> valarray<T> atan2
 (const valarray<T>&, const T&);
 template<class T> valarray<T> atan2
 (const T&, const valarray<T>&);

Prototype template<class T> valarray<T> cos

```
(const valarray<T>&);  
  
Prototype template<class T> valarray<T> cosh  
          (const valarray<T>&);  
  
Prototype template<class T> valarray<T> exp  
          (const valarray<T>&);  
  
Prototype template<class T> valarray<T> log  
          (const valarray<T>&);  
  
Prototype template<class T> valarray<T> log10  
          (const valarray<T>&);  
  
Prototype template<class T> valarray<T> pow  
          (const valarray<T>&, const valarray<T>&);  
template<class T> valarray<T> pow  
          (const valarray<T>&, const T&);  
template<class T> valarray<T> pow  
          (const T&, const valarray<T>&);  
  
Prototype template<class T> valarray<T> sin  
          (const valarray<T>&);  
  
Prototype template<class T> valarray<T> sinh  
          (const valarray<T>&);  
  
Prototype template<class T> valarray<T> sqrt  
          (const valarray<T>&);  
  
Prototype template<class T> valarray<T> tan  
          (const valarray<T>&);  
  
Prototype template<class T> valarray<T> tanh  
          (const valarray<T>&);  
  
Result A valarray object is returned with the individual elements  
       initialized with the result of the corresponding operation.
```

26.3.4 Class slice

A `slice` is a set of indices that have three properties, a starting index, the number of elements and the distance between the elements.

Listing 11.3 Class Slice Synopsis

```
namespace std {
class slice {
public:
slice();
slice(size_t, size_t, size_t);
size_t start() const;
size_t size() const;
size_t stride() const;
};
}
```

Constructors

A constructor is overloaded to initialize an object with values or without values.

Prototype `slice();`
 `slice(size_t start, size_t length, size_t stride);`
 `slice(const slice&);`

26.3.4.2 slice access functions

The slice class has three member functions.

start

`Start` tells `start` is the position where the slice starts.

Prototype `size_t start() const;`

Return The starting position is returned.

size

`Size` tells the size of the slice.

Prototype `size_t size() const;`

Return The size of the slice is returned by the `size` member function.

stride

The distance between elements is given by the `stride` function.

Prototype `size_t stride() const;`

Return The distance between each element is returned by `stride`.

26.3.5 Template Class Slice_array

The `slice_array` class is a helper class used by the slice subscript operator.

Listing 11.4 Template Class Slice_array Synopsis

```
namespace std {  
template <class T> class slice_array {  
public:  
    typedef T value_type;  
    void operator= (const valarray<T>&) const;  
    void operator*= (const valarray<T>&) const;  
    void operator/= (const valarray<T>&) const;  
    void operator%=(const valarray<T>&) const;  
    void operator+= (const valarray<T>&) const;  
    void operator-= (const valarray<T>&) const;  
    void operator^=(const valarray<T>&) const;  
    void operator&=(const valarray<T>&) const;  
    void operator|=(const valarray<T>&) const;  
    void operator<<=(const valarray<T>&) const;  
    void operator>>=(const valarray<T>&) const;  
    void operator=(const T&);  
    ~slice_array();  
private:  
    slice_array();  
    slice_array(const slice_array&);  
    slice_array& operator=(const slice_array&);  
};  
}
```

Constructors

Constructs a `slice_array` object.

Prototype `private:`
 `slice_array();`
 `slice_array(const slice_array&);`

Assignment Operator

The assignment operator allows for the initialization of a slice_array after construction.

Prototype `void operator=(const valarray<T>&) const;`
`slice_array& operator=(const slice_array&);`

26.3.5.3 slice_array computed assignment

Several compound assignment operators are provided.

Prototype `void operator*=(const valarray<T>&) const;`
Prototype `void operator/=(const valarray<T>&) const;`
Prototype `void operator%=(const valarray<T>&) const;`
Prototype `void operator+=(const valarray<T>&) const;`
Prototype `void operator-=(const valarray<T>&) const;`
Prototype `void operator^=(const valarray<T>&) const;`
Prototype `void operator&=(const valarray<T>&) const;`
Prototype `void operator|=(const valarray<T>&) const;`
Prototype `void operator<<=(const valarray<T>&) const;`
Prototype `void operator>>=(const valarray<T>&) const;`

There is no return for the compound operators.

26.3.5.4 Slice_array Fill Function

An assignment operation is provided to fill individual elements of the array.

Prototype `void operator=(const T&);`

No value is returned.

26.3.6 Class Gslice

A general slice class is provided for multidimensional arrays.

Listing 11.5 Gslice Class Synopsis

```
namespace std {
class gslice {
public:
gslice();
gslice
  (size_t s, const valarray<size_t>& l,
   const valarray<size_t>& d);
size_t start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
};
}
```

Constructors

An overloaded constructor is provided for the creation of a `gslice` object.

Prototype `gslice();`
`gslice(size_t start, const valarray<size_t>& lengths,`
`const valarray<size_t>& strides);`
`gslice(const gslice&);`

26.3.6.2 Gslice Access Functions

The `gslice` class provides for access to the `start`, `size` and `stride` of the slice class.

start

The `start` function give the starting position.

Prototype `size_t start() const;`

Return The starting position of the `gslice` is returned.

size

The `size` function returns the number of elements.

Prototype `valarray<size_t> size() const;`

Return The number of elements as a valarray is returned.

stride

The stride function tells the size of each element.

Prototype `valarray<size_t> stride() const;`

Return The size of the element as a valarray is returned.

26.3.7 Template Class Gslice_array

The `gslice_array` class is a helper class used by the gslice subscript operator.

Listing 11.6 Template Class Gslice_array Synopsis

```
namespace std {
template <class T> class gslice_array {
public:
typedef T value_type;
void operator= (const valarray<T>&) const;
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%=(const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^=(const valarray<T>&) const;
void operator&=(const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
void operator=(const T&);
~gslice_array();
private:
gslice_array();
gslice_array(const gslice_array&);
gslice_array& operator=(const gslice_array&);
};
```

Constructors

An overloaded constructor is provided for the creation of a `gslice_array` object.

Prototype `gslice_array();`
`gslice_array(const gslice_array&);`

Assignment Operators

An assignment operator is provided for initializing a `gslice_array` after it has been created.

Prototype `void operator=(const valarray<T>&) const;`
`gslice_array& operator=(const gslice_array&);`

Return A copy of the modified `gslice_array` is returned for the second assignment operator.

26.3.7.3 Gslice_array Computed Assignment

Several compound assignment operators are provided.

Prototype `void operator*=(const valarray<T>&) const;`
Prototype `void operator/=(const valarray<T>&) const;`
Prototype `void operator%=(const valarray<T>&) const;`
Prototype `void operator+=(const valarray<T>&) const;`
Prototype `void operator-=(const valarray<T>&) const;`
Prototype `void operator^=(const valarray<T>&) const;`
Prototype `void operator&=(const valarray<T>&) const;`
Prototype `void operator|=(const valarray<T>&) const;`
Prototype `void operator<<=(const valarray<T>&) const;`
Prototype `void operator>>=(const valarray<T>&) const;`

No return is given for the compound operators.

26.3.7.4 Fill Function

An assignment operation is provided to fill individual elements of the array.

Prototype `void operator=(const T&);`

There is no return for the fill function.

26.3.8 Template Class Mask_array

The `mask_array` class is a helper class used by the mask subscript operator.

Listing 11.7 Template Class Mask_array Synopsis

```
namespace std {
template <class T> class mask_array {
public:
typedef T value_type;
void operator= (const valarray<T>&) const;
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
void operator=(const T&);
~mask_array();
private:
mask_array();
mask_array(const mask_array&);
mask_array& operator=(const mask_array&);
};
```

Constructors

An overloaded constructor is provided for creating a `mask_array` object.

Prototype `private:`
 `mask_array();`
 `mask_array(const mask_array&);`

Assignment Operators

An overloaded assignment operator is provided for assigning values to a `mask_array` after construction.

Prototype `void operator=(const valarray<T>&) const;`
`mask_array& operator=(const mask_array&);`

Return The copy assignment operator returns a `mask_array` reference.

26.3.8.3 Mask_array Computed Assignment

Several compound assignment operators are provided.

Prototype `void operator*=(const valarray<T>&) const;`
Prototype `void operator/=(const valarray<T>&) const;`
Prototype `void operator%=(const valarray<T>&) const;`
Prototype `void operator+=(const valarray<T>&) const;`
Prototype `void operator-=(const valarray<T>&) const;`
Prototype `void operator^=(const valarray<T>&) const;`
Prototype `void operator&=(const valarray<T>&) const;`
Prototype `void operator|=(const valarray<T>&) const;`
Prototype `void operator<<=(const valarray<T>&) const;`
Prototype `void operator>>=(const valarray<T>&) const;`

There is no return value for the compound assignment operators.

26.3.8.4 Mask_array Fill Function

An assignment operation is provided to fill individual elements of the array.

Prototype void operator =(const T&);

There is no return for the fill function.

26.3.9 Template Class Indirect_array

The `indirect_array` class is a helper class used by the indirect subscript operator.

Listing 11.8 Template Class Indirect_array Synopsis

```
namespace std {
template <class T> class indirect_array {
public:
typedef T value_type;
void operator= (const valarray<T>&) const;
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
void operator=(const T&);
~indirect_array();
private:
indirect_array();
indirect_array(const indirect_array&);
indirect_array& operator=(const indirect_array&);
};
```

1 This template is a helper template used by the indirect subscript operator `indirect_array<T> valarray<T>::operator[](const valarray<size_t>&)`.

It has reference semantics to a subset of an array specified by an `indirect_array`.

Constructors

An overloaded constructor is provided for creating a `indirect_array` object.

Prototype `indirect_array();`
 `indirect_array(const indirect_array&);`

Assignment Operators

An overloaded assignment operator is provided for assigning values to a `indirect_array` after construction.

Prototype `void operator=(const valarray<T>&) const;`
 `indirect_array& operator=(const indirect_array&);`

Return The copy assignment operator returns a `indirect_array` reference.

26.3.9.3 Indirect_array Computed Assignment

Several compound assignment operators are provided.

Prototype `void operator*=(const valarray<T>&) const;`
Prototype `void operator/=(const valarray<T>&) const;`
Prototype `void operator%=(const valarray<T>&) const;`
Prototype `void operator+=(const valarray<T>&) const;`
Prototype `void operator-=(const valarray<T>&) const;`
Prototype `void operator^=(const valarray<T>&) const;`
Prototype `void operator&=(const valarray<T>&) const;`
Prototype `void operator|=(const valarray<T>&) const;`
Prototype `void operator<<=(const valarray<T>&) const;`

Prototype void operator>>=(const valarray<T>&) const;

There is no return value for the compound assignment operators.

26.3.9.4 indirect_array fill function

An assignment operation is provided to fill individual elements of the array.

Prototype void operator=(const T&);

There is no return for the fill function.

26.4 Generalized Numeric Operations

The standard library provides general algorithms for numeric processing.

Header <numeric>

The header <numeric> includes template functions for generalize numeric processing.

Listing 11.9 Header <numeric> synopsis

```
namespace std {
    template <class InputIterator, class T> T accumulate
        (InputIterator first, InputIterator last, T init);
    template <class InputIterator, class T, class BinaryOperation>
        T accumulate
        (InputIterator first, InputIterator last, T init,
         BinaryOperation binary_op);
    template <class InputIterator1, class InputIterator2, class T>
        T inner_product
        (InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, T init);
    template <class InputIterator1, class InputIterator2, class T,
              class BinaryOperation1, class BinaryOperation2> T inner_product
        (InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, T init, BinaryOperation1 binary_op1,
         BinaryOperation2 binary_op2);
    template <class InputIterator, class OutputIterator>
```

```
OutputIterator partial_sum
    (InputIterator first, InputIterator last,
     OutputIterator result);
template <class InputIterator, class OutputIterator,
          class BinaryOperation>
OutputIterator partial_sum
    (InputIterator first, InputIterator last,
     OutputIterator result, BinaryOperation binary_op);
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference
    (InputIterator first, InputIterator last,
     OutputIterator result);
template <class InputIterator, class OutputIterator,
          class BinaryOperation>
OutputIterator adjacent_difference
    (InputIterator first, InputIterator last,
     OutputIterator result, BinaryOperation binary_op);
}
```

accumulate

Accumulate the sum of a sequence.

Prototype template <class InputIterator, class T>
 T accumulate(
 InputIterator first, InputIterator last,
 T init);
 template <class InputIterator, class T,
 class BinaryOperation>
 T accumulate
 (InputIterator first, InputIterator last,
 T init, BinaryOperation binary_op);

Return The sum of the values in a range or the some of the values after being processed by an operation is returned.

inner_product

Computes and returns the value of a product of the values in a range.

Prototype template <class InputIterator1,
 class InputIterator2, class T>

```

T inner_product
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, T init);
template <class InputIterator1,
          class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
T inner_product
    (InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, T init,
     BinaryOperation1 binary_op1,
     BinaryOperation2 binary_op2);

```

Return The value of the product starting with an initial value in a range is returned. In the function with the operation argument it is the product after the operation is performed.

partial_sum

Computes the partial sum of a sequence of numbers.

Prototype

```

template <class InputIterator,
          class OutputIterator>
OutputIterator partial_sum
    (InputIterator first, InputIterator last,
     OutputIterator result);
template <class InputIterator,
          class OutputIterator, class BinaryOperation>
OutputIterator partial_sum
    (InputIterator first, InputIterator last,
     OutputIterator result,
     BinaryOperation binary_op);

```

The first computes the partial sum and sends it to the output iterator argument.

```

x, y, z
x, x+y, y+z.

```

The second form computes according to the operational argument and sends it to the output iterator argument. For example if the operational argument was a multiplication operation

```

x, y, z
x, x*y, y*z

```

Return The range as the result plus the last minus the first.

adjacent_difference

Computed the adjacent difference in a sequence of numbers.

Prototype

```
template <class InputIterator,
          class OutputIterator>
OutputIterator adjacent_difference
    (InputIterator first, InputIterator last,
     OutputIterator result);
template <class InputIterator,
          class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference
    (InputIterator first, InputIterator last,
     OutputIterator result,
     BinaryOperation binary_op);
```

The first computes the adjacent difference and sends it to the output iterator argument.

```
x, y, z
x, y-x, z-y.
```

The second form computes according to the operational argument and sends it to the output iterator argument. For example if the operational argument was a division operation

```
x, y, z
x, y/x, z/y
```

Return The range as the result plus the last minus the first.

26.5 C Library

The standard provides for the math functions included in the standard C library with some overloading for various types.

<cmath>

The contents of the `<cmath>` headers is the same as the Standard C library headers `<math.h>` with the addition to the double versions

of the math functions in `<cmath>`, C++ adds float and long double overloaded versions of some functions, with the same semantics.

<cstdlib>

The contents of the `<cstdlib>` headers is the same as the Standard C library headers `<stdlib.h>`. In addition to the int versions of certain math functions in `<cstdlib>`, C++ adds long overloaded versions of some functions, with the same semantics.

Listing 11.10

The Added C++ Signatures in Cstdlib and Cmath

```
long double abs (long double);
long double acos (long double);
long double asin (long double);
long double atan (long double);
long double atan2(long double, long double);
long double ceil (long double);
long double cos (long double);
long double cosh (long double);
long double exp (long double);
long double fabs (long double);
long double floor(long double);
long double fmod (long double, long double);
long double frexp(long double, int*);
long double ldexp(long double, int);
long double log (long double);
long double log10(long double);
long double modf (long double, long double* );
long double pow (long double, long double);
long double pow (long double, int);
long double sin (long double);
long double sinh (long double);
long double sqrt (long double);
long double tan (long double);
long double tanh (long double);

float abs (float);
float acos (float);
float asin (float);
float atan (float);
float atan2(float, float);
float ceil (float);
```

```
float cos (float);
float cosh (float);
float exp (float);
float fabs (float);
float floor(float);
float fmod (float, float);
float frexp(float, int*);
float ldexp(float, int);
float log (float);
float log10(float);
float modf (float, float*);
float pow (float, float);
float pow (float, int);
float sin (float);
float sinh (float);
float sqrt (float);
float tan (float);
float tanh (float);

double abs(double);
double pow(double, int);
```

Complex Class

The header <complex> defines a template class, and facilities for representing and manipulating complex numbers.

The Complex Class Library (clause 26.2)

The header <complex> defines classes, operators, and functions for representing and manipulating complex numbers

The chapter is constructed in the following sub sections and mirrors clause 26.2 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header <complex>” on page 434](#), shows the complex header class declarations
- [“26.2.3 Complex Specializations” on page 438](#), lists the float, double and long double specializations
- [“Complex Template Class” on page 440](#), is a template class for complex numbers.

_MSL_CX_LIMITED_RANGE

This flag effects the * and / operators of complex.

When defined, the “normal” formulas for multiplication and division are used. They may execute faster on some machines. However, infinities will not be properly calculated, and there is more roundoff error potential.

If the flag is undefined (default), then more complicated algorithms (from the C standard) are used which have better overflow and underflow characteristics and properly propagate infinity. Flipping this switch requires recompilation of the C++ library.

Complex Class

Header <complex>

NOTE It is recommend that the ansi_prefix.hpp is the place to define this flag if you want the simpler and faster multiplication and division algorithms.

Header <complex>

The header <complex> defines classes, operators, and functions for representing and manipulating complex numbers

Listing 12.1 Header <complex> forward declarations

```
namespace std {  
// forward declarations  
template<class T> class complex;  
template<> class complex<float>;  
template<> class complex<double>;  
template<> class complex<long double>;
```

```
26.2.6 operators:  
template<class T> complex<T> operator+  
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator+  
    (const complex<T>&, const T&);  
template<class T> complex<T> operator+  
    (const T&, const complex<T>&);  
template<class T> complex<T> operator-  
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator-  
    (const complex<T>&, const T&);  
template<class T> complex<T> operator-  
    (const T&, const complex<T>&);  
template<class T> complex<T> operator*  
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator*  
    (const complex<T>&, const T&);  
template<class T> complex<T> operator*  
    (const T&, const complex<T>&);  
template<class T> complex<T> operator/  
    (const complex<T>&, const complex<T>&);
```

```
template<class T> complex<T> operator/
    (const complex<T>&, const T&);
template<class T> complex<T> operator/
    (const T&, const complex<T>&);
template<class T> complex<T> operator+
    (const complex<T>&);
template<class T> complex<T> operator-
    (const complex<T>&);
template<class T> bool operator==
    (const complex<T>&, const complex<T>&);
template<class T> bool operator==
    (const complex<T>&, const T&);
template<class T> bool operator==
    (const T&, const complex<T>&);
template<class T> bool operator!=
    (const complex<T>&, const complex<T>&);
template<class T> bool operator!=
    (const complex<T>&, const T&);
template<class T> bool operator!=
    (const T&, const complex<T>&);
template<class T, class charT, class traits>
basic_istream<charT, traits>& operator>>
    (basic_istream<charT, traits>&, complex<T>&);
template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<
    (basic_ostream<charT, traits>&, const complex<T>&);
```

26.2.7 values:

```
template<class T> T real(const complex<T>&);
template<class T> T imag(const complex<T>&);
template<class T> T abs(const complex<T>&);
template<class T> T arg(const complex<T>&);
template<class T> T norm(const complex<T>&);
template<class T> complex<T> conj(const complex<T>&);
template<class T> complex<T> polar(const T&, const T&);
```

26.2.8 transcendentals:

```
template<class T> complex<T> cos (const complex<T>&);
template<class T> complex<T> cosh (const complex<T>&);
template<class T> complex<T> exp (const complex<T>&);
```

Complex Class

Header <complex>

```
template<class T> complex<T> log (const complex<T>&);  
template<class T> complex<T> log10(const complex<T>&);  
template<class T> complex<T> pow(const complex<T>&, int);  
template<class T> complex<T> pow(const complex<T>&, const T&);  
template<class T> complex<T> pow  
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> pow(const T&, const complex<T>&);  
template<class T> complex<T> sin (const complex<T>&);  
template<class T> complex<T> sinh (const complex<T>&);  
template<class T> complex<T> sqrt (const complex<T>&);  
template<class T> complex<T> tan (const complex<T>&);  
template<class T> complex<T> tanh (const complex<T>&);  
}
```

```
Template Class Complex  
namespace std {  
template<class T>  
class complex {  
public:  
typedef T value_type;  
complex(const T& re = T(), const T& im = T());  
complex(const complex&);  
template<class X> complex(const complex<X>&);  
T real() const;  
T imag() const;  
complex<T>& operator= (const T&);  
complex<T>& operator+=(const T&);  
complex<T>& operator-=(const T&);  
complex<T>& operator*=(const T&);  
complex<T>& operator/=(const T&);  
complex& operator=(const complex&);  
template<class X> complex<T>& operator= (const complex<X>&);  
template<class X> complex<T>& operator+=(const complex<X>&);  
template<class X> complex<T>& operator-=(const complex<X>&);  
template<class X> complex<T>& operator*=(const complex<X>&);  
template<class X> complex<T>& operator/=(const complex<X>&);  
};  
template<class T> complex<T> operator+  
    (const complex<T>&, const complex<T>&);  
template<class T> complex<T> operator+  
    (const complex<T>&, const T&);
```

```
template<class T> complex<T> operator+
    (const T&, const complex<T>&);
template<class T> complex<T> operator-
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator-
    (const complex<T>&, const T&);
template<class T> complex<T> operator-
    (const T&, const complex<T>&);
template<class T> complex<T> operator*
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator*
    (const complex<T>&, const T&);
template<class T> complex<T> operator*
    (const T&, const complex<T>&);
template<class T> complex<T> operator/
    (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/
    (const complex<T>&, const T&);
template<class T> complex<T> operator/
    (const T&, const complex<T>&);
template<class T> complex<T> operator+
    (const complex<T>&);
template<class T> complex<T> operator-
    (const complex<T>&);
template<class T> bool operator==
    (const complex<T>&, const complex<T>&);
template<class T> bool operator==
    (const complex<T>&, const T&);
template<class T> bool operator==
    (const T&, const complex<T>&);
template<class T> bool operator!=
    (const complex<T>&, const complex<T>&);
template<class T> bool operator!=
    (const complex<T>&, const T&);
template<class T> bool operator!=
    (const T&, const complex<T>&);
template<class T, class charT, class traits>
basic_istream<charT, traits>& operator>>
    (basic_istream<charT, traits>&, complex<T>&);
template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<
```

Complex Class

26.2.3 Complex Specializations

```
(basic_ostream<charT, traits>&, const complex<T>&);  
};
```

26.2.3 Complex Specializations

The standard specialize the template complex class for float, double and long double types.

Listing 12.2 Float specializations

```
template<> class complex<float> {  
public:  
    typedef float value_type;  
    complex(float re = 0.0f, float im = 0.0f);  
    explicit complex(const complex<double>&);  
    explicit complex(const complex<long double>&);  
    float real() const;  
    float imag() const;  
    complex<float>& operator=(float);  
    complex<float>& operator+=(float);  
    complex<float>& operator-=(float);  
    complex<float>& operator*=(float);  
    complex<float>& operator/=(float);  
    complex<float>& operator=(const complex<float>&);  
    template<class X> complex<float>& operator=(const complex<X>&);  
    template<class X> complex<float>& operator+=(const complex<X>&);  
    template<class X> complex<float>& operator-=(const complex<X>&);  
    template<class X> complex<float>& operator*=(const complex<X>&);  
    template<class X> complex<float>& operator/=(const complex<X>&);  
};
```

Listing 12.3 Double specializations

```
template<> class complex<double> {  
public:  
    typedef double value_type;  
    complex(double re = 0.0, double im = 0.0);  
    complex(const complex<float>&);  
    explicit complex(const complex<long double>&);  
    double real() const;
```

```
double imag() const;
complex<double>& operator= (double);
complex<double>& operator+=(double);
complex<double>& operator-=(double);
complex<double>& operator*=(double);
complex<double>& operator/=(double);
complex<double>& operator=(const complex<double>& );
template<class X> complex<double>& operator= (const complex<X>& );
template<class X> complex<double>& operator+=(const complex<X>& );
template<class X> complex<double>& operator-=(const complex<X>& );
template<class X> complex<double>& operator*=(const complex<X>& );
template<class X> complex<double>& operator/=(const complex<X>& );
};
```

Listing 12.4 Long Double specializations

```
template<> class complex<long double> {
public:
typedef long double value_type;
complex(long double re = 0.0L, long double im = 0.0L);
complex(const complex<float>& );
complex(const complex<double>& );
long double real() const;
long double imag() const;
complex<long double>& operator=(const complex<long double>& );
complex<long double>& operator= (long double);
complex<long double>& operator+=(long double);
complex<long double>& operator-=(long double);
complex<long double>& operator*=(long double);
complex<long double>& operator/=(long double);
template<class X> complex<long double>& operator=
    (const complex<X>& );
template<class X> complex<long double>& operator+ =
    (const complex<X>& );
template<class X> complex<long double>& operator- =
    (const complex<X>& );
template<class X> complex<long double>& operator* =
    (const complex<X>& );
template<class X> complex<long double>& operator/ =
    (const complex<X>& );
};
```

Complex Template Class

The template class complex contains Cartesian components real and imag for a complex number.

Remarks The effect of instantiating the template complex for any type other than float, double or long double is unspecified.

If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

The complex class consists of:

- [“Constructors and Assignments” on page 440.](#)
- [“Complex Member Functions” on page 441,](#)
- [“Operators” on page 441.](#)

Constructors and Assignments

Constructor, destructor and assignment operators and functions.

Constructors

Description Construct an object of a complex class.

Prototype `complex(const T& re = T(), const T& im = T());`
`complex(const complex&);`
`template<class X> complex(const complex<X>&);`

Remarks After construction real equal re and imag equals im.

Assignment Operator

Description An assignment operator for complex classes.

Prototype `complex<T>& operator= (const T&);`
`complex& operator= (const complex&);`
`template<class X> complex<T>& operator= (const complex<X>&);`

Remarks Assigns a floating point type to the Cartesian complex class.

Complex Member Functions

There are two public member functions

- [“real” on page 441](#)
- [“imag” on page 441](#)

real

Description Retrieves the real component.

Prototype `T real() const;`

imag

Description Retrieves the imag component.

Prototype `T imag() const;`

Operators

Several assignment operators are overloaded for the complex class manipulations.

- [“operator +=” on page 441](#)
- [“operator -=” on page 442](#)
- [“operator *=” on page 442](#)
- [“operator /=” on page 442](#)

operator +=

Description Adds and assigns to a complex class.

Prototype `complex<T>& operator+=(const T&);`
`template<class X> complex<T>& operator+=`
`(const complex<X>&);`

Remarks The first operator with a scalar argument adds the scalar value of the right hand side to the real component and stores the result in the object. The imaginary component is left alone.

The second operator with a complex type, adds the complex value of the right hand side to the object and stores the resultant in the object.

Complex Class

Complex Template Class

Returns The `this` pointer is returned.

operator -=

Description Subtracts and assigns from a complex class.

Prototype `complex<T>& operator-=(const T&);`
 `template<class X> complex<T>& operator-=`
 `(const complex<X>&);`

Remarks The first operator with a scalar argument subtracts the scalar value of the right hand side from the real component and stores the result in the object. The imaginary component is left alone.

The second operator with a complex type, subtracts the complex value of the right hand side from the object and stores the resultant in the object.

Returns The `this` pointer is returned.

operator *=

Description Multiplies by and assigns to a complex class.

Prototype `complex<T>& operator*=(const T&);`
 `template<class X> complex<T>& operator*=`
 `(const complex<X>&);`

Remarks The first operator with a scalar argument multiplies the scalar value of the right hand side to class object and stores result in the object.

The second operator with a complex type, multiplies the complex value of the right hand side to the object and stores the resultant in the object.

Returns The `this` pointer is returned.

operator /=

Description Divides by and assigns to a complex class.

Prototype `complex<T>& operator/=(const T&);`
 `template<class X> complex<T>& operator/=`
 `(const complex<X>&);`

Remarks The first operator with a scalar argument divides the scalar value of the right hand side to class object and stores result in the object.

The second operator with a complex type, divides the complex value of the right hand side into the object and stores the resultant in the object.

Returns The `this` pointer is returned.

Overloaded Operators and Functions

There are several non member functions and overloaded operators in the complex class library.

[“Complex Operators” on page 443](#)

[“Complex Value Operations” on page 446](#)

[“Complex Transcendentals” on page 447](#)

Complex Operators

The overloaded complex operators consists of:

- [“operator +” on page 443](#)
- [“operator -” on page 444](#)
- [“operator /” on page 444](#)
- [“operator !=” on page 445](#)
- [“operator >>” on page 445](#)
- [“operator <<” on page 445](#)

operator +

Description Adds to the complex class.

Prototype

```
template<class T> complex<T> operator+
    const complex<T>&, const complex<T>& );
template<class T> complex<T> operator+
    (const complex<T>&, const T& );
template<class T> complex<T> operator+
    (const T&, const complex<T>& );

template<class T> complex<T> operator+
    (const complex<T>&);
```

Complex Class

Complex Template Class

Remarks The addition performs an `+ =` operation.

Returns The complex class after the addition.

operator -

Description Subtracts from the complex class.

Prototype `template<class T> complex<T> operator-`
 `(const complex<T>&, const complex<T>&);`
`template<class T> complex<T> operator-`
 `(const complex<T>&, const T&);`
`template<class T> complex<T> operator-`
 `(const T&, const complex<T>&);`

`template<class T> complex<T> operator-`
 `(const complex<T>&);`

Remarks The subtraction performs a `- =` operation.

Returns The complex class after the Subtraction.

operator *

Description Multiplies the complex class.

Prototype `template<class T> complex<T> operator*`
 `(const complex<T>&, const complex<T>&);`
`template<class T> complex<T> operator*`
 `(const complex<T>&, const T&);`
`template<class T> complex<T> operator*`
 `(const T&, const complex<T>&);`

Remarks The multiplication performs a `* =` operation.

Returns The complex class after the multiplication.

operator /

Description Divides from the complex class.

Prototype `template<class T> complex<T> operator/`
 `(const complex<T>&, const complex<T>&);`
`template<class T> complex<T> operator/`
 `(const complex<T>&, const T&);`
`template<class T> complex<T> operator/`
 `(const T&, const complex<T>&);`

Remarks The division performs an /= operation.

Returns The complex class after the division.

operator ==

Description A boolean equality comparison.

Prototype template<class T> bool operator==
 (const complex<T>&, const complex<T>&);
 template<class T> bool operator==
 (const complex<T>&, const T&);
 template<class T> bool operator==
 (const T&, const complex<T>&);

Remarks Returns true if the real and the imaginary components are equal.

operator !=

Description A boolean non equality comparison.

Prototype template<class T> bool operator!=
 (const complex<T>&, const complex<T>&);
 template<class T> bool operator!=
 (const complex<T>&, const T&);
 template<class T> bool operator!=
 (const T&, const complex<T>&);

Remarks Returns true if the real or the imaginary components are not equal.

operator >>

Description Extracts a complex type from a stream.

Prototype template<class T, class charT, class traits>
 basic_istream<charT, traits>& operator>>
 (basic_istream<charT, traits>&, complex<T>&);

Remarks Extracts in the form of u, (u), or (u,v) where u is the real part and v is the imaginary part.

Any failure in extraction will set the failbit and result in undefined behavior.

operator <<

Description Inserts a complex number into a stream.

Complex Class

Complex Template Class

Prototype `template<class T, class charT, class traits>`
`basic_ostream<charT, traits>& operator<<`
`(basic_ostream<charT, traits>&,`
`const complex<T>&);`

Complex Value Operations

The complex value operations consists of:

- [“real” on page 446](#)
- [“imag” on page 446](#)
- [“abs” on page 446](#)
- [“arg” on page 447](#)
- [“norm” on page 447](#)
- [“conj” on page 447](#)
- [“polar” on page 447](#)

real

Description Retrieves the real component of a complex class.

Prototype `template<class T>`
`T real(const complex<T>&);`

Remarks Returns the real component of the argument.

imag

Description Retrieves the imaginary component of a complex class.

Prototype `template<class T>`
`T imag(const complex<T>&);`

Remarks Returns the imaginary component of the argument.

abs

Description Determines the absolute value of a complex class.

Prototype `template<class T>`
`T abs(const complex<T>&);`

Remarks Returns the absolute value of the complex class argument.

arg

| | |
|-------------|---|
| Description | Determines the phase angle. |
| Prototype | <code>template<class T> T arg(const complex<T>&);</code> |
| Remarks | Returns the phase angle of the complex class argument or <code>atan2(imag(x), real(x))</code> . |

norm

| | |
|-------------|---|
| Description | Determines the squared magnitude. |
| Prototype | <code>template<class T> T norm(const complex<T>&);</code> |
| Remarks | The squared magnitude of the complex class. |

conj

| | |
|-------------|--|
| Description | Determines the complex conjugate. |
| Prototype | <code>template<class T> complex<T> conj(const complex<T>&);</code> |
| Remarks | Returns the complex conjugate of the complex class argument. |

polar

| | |
|-------------|---|
| Description | Determines the polar coordinates. |
| Prototype | <code>template<class T> complex<T> polar(const T&, const T&);</code> |
| Remarks | Returns the complex value corresponding to a complex number whose magnitude is the first argument and whose phase angle is the second argument. |

Complex Transcendentals

The complex transcendentals consists of:

- [“cos” on page 448](#)
- [“cosh” on page 448](#)
- [“exp” on page 448](#)

- [“log” on page 448](#)
- [“log10” on page 449](#)
- [“pow” on page 449](#)
- [“sin” on page 449](#)
- [“sinh” on page 449](#)
- [“sqrt” on page 450](#)
- [“tan” on page 450](#)
- [“tanh” on page 450](#)

cos

Description Determines the cosine.

Prototype `template<class T>
 complex<T> cos (const complex<T>&);`

Remarks Returns the cosine of the complex class argument.

cosh

Description Determines the hyperbolic cosine.

Prototype `template<class T>
 complex<T> cosh (const complex<T>&);`

Remarks Returns the cosine of the complex class argument.

exp

Description Determines the exponential.

Prototype `template<class T>
 complex<T> exp (const complex<T>&);`

Remarks Returns the base exponential of the complex class argument.

log

Description Determines the natural base logarithm.

Prototype `template<class T>
 complex<T> log (const complex<T>&);`

Remarks Returns the natural base logarithm of the complex class argument, in the range of a strip mathematically unbounded along the real axis

and in the interval of $[i\pi, i\pi]$ along the imaginary axis. Where the argument is a negative real number, $\text{imag}(\log(\text{cpx}))$, is π .

log10

| | |
|-------------|--|
| Description | Determines the logarithm to base ten. |
| Prototype | <pre>template<class T> complex<T> log10(const complex<T>&);</pre> |
| Remarks | Returns the logarithm base(10) of the argument <code>cpx</code> defined as $\log(\text{cpx})/\log(10)$. |

pow

| | |
|-------------|---|
| Description | Raises the complex class to a set power. |
| Prototype | <pre>template<class T> complex<T> pow(const complex<T>&, int); template<class T> complex<T> pow(const complex<T>&, const T&); template<class T> complex<T> pow (const complex<T>&, const complex<T>&); template<class T> complex<T> pow(const T&, const complex<T>&);</pre> |
| Remarks | Returns the complex class raised to the power of second argument defined as the exponent of (the second argument times the log of the first argument). |

The value for `pow(0, 0)` will return (nan, nan).

sin

| | |
|-------------|--|
| Description | Determines the sine. |
| Prototype | <pre>template<class T> complex<T> sin (const complex<T>&);</pre> |

Remarks Returns the sine of the complex class argument.

sinh

| | |
|-------------|------------------------------------|
| Description | Determines the hyperbolic sine. |
| Prototype | <pre>template<class T></pre> |

Complex Class

Complex Template Class

```
complex<T> sinh (const complex<T>&);
```

Remarks Returns the hyperbolic sine of the complex class argument.

sqrt

Description Determines the square root.

Prototype template<class T>
 complex<T> sqrt (const complex<T>&);

Remarks Returns the square root of the complex class argument in the range of right half plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis.

tan

Description Determines the tangent.

Prototype template<class T>
 complex<T> tan (const complex<T>&);

Remarks Returns the tangent of the complex class argument.

tanh

Description Determines the hyperbolic tangent.

Prototype template<class T>
 complex<T> tanh (const complex<T>&);

Remarks Returns the hyperbolic tangent of the complex class argument.

Input and Output Library

A listing of the set of components that C++ programs may use to perform input/output operations.

The Input and Output Library (clause 27.1)

The chapter is constructed in the following sub sections and mirrors clause 27.1 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Input and Output Library Summary” on page 451](#)
- [“27.1 Iostreams requirements” on page 452](#)

Input and Output Library Summary

This library includes the headers.

Table 13.1 Input/Output Library Summary

| Include | Purpose |
|-------------|-----------------------------|
| <iostream> | Forward declarations |
| <iostream> | Standard iostream objects |
| <ios> | Iostream base classes |
| <streambuf> | Stream buffers |
| <iomanip> | Formatting and manipulators |
| <sstream> | String streams |

| Include | Purpose |
|------------------------------|--------------|
| <code><cstdlib></code> | |
| <code><fstream></code> | File Streams |
| <code><cstdio></code> | |
| <code><cwchar></code> | |

27.1 **iostreams requirements**

No requirements library has been defined.

Topics in this section are:

- [“27.1.1 Definitions” on page 452](#)
- [“27.1.2 Type requirements” on page 453](#)
- [“27.1.2.5 Type `SZ_T`” on page 453](#)

27.1.1 Definitions

Additional definitions are:

- **character** - A unit that can represent text
- **character container type** - A class or type used to represent a character.
- **iostream class templates** - Templates that take two arguments: `charT` and `traits`. The argument `charT` is a character container type. The argument `traits` is a structure which defines characteristics and functions of the `charT` type.
- **narrow-oriented iostream classes** - These classes are template instantiation classes. The traditional iostream classes are narrow-oriented iostream classes.
- **wide-oriented iostream classes** - These classes are template instantiation classes. They are used for the character container class `wchar_t`.
- **repositional streams and arbitrary-positional streams** - A repositional stream can seek to only a previously encountered position. An arbitrary-positional stream can integral position within the length of the stream.

27.1.2 Type requirements

Several types are required by the standards, they are consolidated in strings (chapter 21.)

27.1.2.5 Type **SZ_T**

A type that represents one of the signed basic integral types. It is used to represent the number of characters transferred in and input/output operation or for the size of the input/output buffers.

Forward Declarations

The header `<iostreamfwd>` is used for forward declarations of template classes.

The non-standard header `<stringfwd>` is used for forward declarations of string class objects.

The Streams and String Forward Declarations (clause 27.2)

The ANSI/ISO standard calls for forward declarations of input and output streams for basic input and output, basic input and basic output. This is for both normal and wide character formats.

The chapter mirrors clause 27.2 of the ISO (the International Organization for Standardization) C++ Standard :

Header `<iostfwd>`

The header `<iostfwd>` is used for forward declarations of template classes.

Listing 14.1 Header `<iostfwd>` Synopsis

```
namespace std {
    template<class charT> struct char_traits;

    template<class T> class allocator;
    // for MSL IO
    template <class charT, class traits = char_traits<charT> >
    class basic_ios;

    template <class charT, class traits = char_traits<charT> >
```

Forward Declarations

Header `<iostream>`

```
class basic_streambuf;

template <class charT, class traits = char_traits<charT> >
class basic_istream;

template <class charT, class traits = char_traits<charT> >
class basic_oiostream;

template <class charT, class traits = char_traits<charT> >
class basic_iostream;

template <class charT, class traits = char_traits<charT>, >
class Allocator = allocator<charT>
class basic_stringbuf;

template <class charT, class traits = char_traits<charT>, class
Allocator = allocator<charT> >
class basic_istringstream;

template <class charT, class traits = char_traits<charT>, class
Allocator = allocator<charT> >
class basic_ostringstream;

template <class charT, class traits = char_traits<charT>, class
Allocator = allocator<charT> >
class basic_stringstream;

template <class charT, class traits = char_traits<charT>
class basic_filebuf;

template <class charT, class traits = char_traits<charT> >
class basic_ifstream;

template <class charT, class traits = char_traits<charT> >
class basic_ofstream;

template <class charT, class traits = char_traits<charT> >
class basic_fstream;

template <class charT, class traits = char_traits<charT> >
class istreambuf_iterator;
```

```
template <class charT, class traits = char_traits<charT> >
class ostreambuf_iterator;

template <> struct char_traits<char>;

typedef basic_ios<char, char_traits<char> > ios;

typedef basic_streambuf<char, char_traits<char> > streambuf;
typedef basic_istream<char, char_traits<char> > istream;
typedef basic_ostream<char, char_traits<char> > ostream;
typedef basic_iostream<char, char_traits<char> > iostream;

typedef basic_stringbuf<char, char_traits<char>, allocator<char>
>
stringbuf;
typedef basic_istringstream<char, char_traits<char>,
allocator<char> > istringstream;
typedef basic_ostringstream<char, char_traits<char>,
allocator<char> > ostringstream;
typedef basic_stringstream<char, char_traits<char>,
allocator<char> > stringstream;

typedef basic_filebuf<char, char_traits<char> > filebuf;
typedef basic_ifstream<char, char_traits<char> > ifstream;
typedef basic_ofstream<char, char_traits<char> > ofstream;
typedef basic_fstream<char, char_traits<char> > fstream;

// for MSL wide character support

template <> struct char_traits<wchar_t>;

typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;

typedef basic_streambuf<wchar_t, char_traits<wchar_t> >
wstreambuf;
typedef basic_istream<wchar_t, char_traits<wchar_t> >
wistream;
typedef basic_ostream<wchar_t, char_traits<wchar_t> >
woostream;
typedef basic_iostream<wchar_t, char_traits<wchar_t>
> wiostream;
```

Forward Declarations

Header <stringfwd>

```
typedef basic_stringbuf<wchar_t, char_traits<wchar_t>,
    allocator<wchar_t> >           wstringbuf;
typedef basic_istream<wchar_t, char_traits<wchar_t>,
    allocator<wchar_t> >         wistreamstream;
typedef basic_ostringstream<wchar_t, char_traits<wchar_t>,
    allocator<wchar_t> >        wostreamstream;
typedef basic_stringstream<wchar_t, char_traits<wchar_t>,
    allocator<wchar_t> >       wstringstream;

typedef basic_filebuf<wchar_t, char_traits<wchar_t> >   wfilebuf;
typedef basic_ifstream<wchar_t, char_traits<wchar_t> >  wifstream;
typedef basic_ofstream<wchar_t, char_traits<wchar_t> >  wofstream;
typedef basic_fstream<wchar_t, char_traits<wchar_t> >   wfstream;

//end wide character support and end MSL IO

template <class state> class fpos;
typedef fpos<mbstate_t> streampos;
typedef fpos<mbstate_t> wstreampos;
typedef long streamoff;
} // namespace std
```

Remarks The template class `basic_ios<charT, traits>` serves as a base class for class `basic_istream` and `basic_ostream`.

The class `ios` is an instantiation of `basic_ios` specialized by the type `char`.

The class `wios` is an instantiation of `basic_ios` specialized by the type `wchar_t`.

Header <stringfwd>

This non-standard header can be used to forward declare `basic_string` (much like `<iostream>` forward declares streams). There is also a `<stringfwd.h>` that forward declares `basic_string` and places it into the global namespace.

NOTE The header <stringfwd> is a non standard header.

Listing 14.2 Header <stringfwd> Synopsis

```
namespace std { // Optional
template <class T> class allocator;
template<class chart> struct char_traits;
template <class chart, class traits, class Allocator>
class basic_string;

typedef basic_string <char, char_traits<char>, allocator<char> >
string;
typedef basic_string
<wchar_t, char_traits<wchar_t>, allocator<wchar_t> >
wstring;
}
```

Including <stringfwd> allows you to use a string object.

Listing 14.3 Example of <stringfwd> Inclusion of std::string

```
#include <stringfwd>
class MyClass
{
    ...
    std::string* my_string_ptr;
};
```

The headers <stringfwd.h> and <string> can be used in combination to place string into the global namespace, much like is done with other <name.h> headers. The header <string.h> does not work because that is a standard C header.

Listing 14.4 Example of Stringfwd usage

```
#include <stringfwd.h>
#include <string>

int main()
{
    string a("Hi");    // no std:: required
    return 0;
}
```

Forward Declarations

Header <stringfwd>

iostream Objects

The include header <iostream> declared input and output stream objects. The declared objects are associated with the standard C streams provided for by the functions in <cstdio>.

The Standard Input and Output Stream Library (clause 27.3)

The ANSI/ISO standard calls for predetermined objects for standard input, output, logging and error reporting. This is initialized for normal and wide character formats.

The chapter is constructed in the following sub sections and mirrors clause 27.3 of the ISO (the International Organization for Standardization) C++ Standard :

- [“27.3.1 Narrow stream objects” on page 462](#)
- [“27.3.2 Wide stream objects” on page 463](#)

Header <iostream>

Declaration of standard objects

Prototype

```
Header <iostream>
namespace std{
    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    extern ostream clog;

    extern wistream wcin;
    extern wostream wcout;
    extern wostream cerr;
```

iostream Objects

Header *<iostream>*

```
extern wostream wclog;
}
```

Stream Buffering

All streams are buffered (by default) except `cerr` and `wcerr`.

NOTE You can change the buffering characteristic of a stream with:

```
cout.setf(ios_base::unitbuf);
or
cerr.unsetf(ios_base::unitbuf);
```

27.3.1 Narrow stream objects

Narrow stream objects provide unbuffered input and output associated with standard input and output declared in `<cstdio>`.

istream cin

An unbuffered input stream.

Prototype `istream cin;`

Remarks The object `cin` controls input from an unbuffered stream buffer associated with `stdin` declared in `<cstdio>`. After `cin` is initialized `cin.tie()` returns `cout`.

Return An `istream` object;

ostream cout

An unbuffered output stream.

Prototype `ostream cout;`

Remarks The object `cout` controls output to an unbuffered stream buffer associated with `stdout` declared in `<cstdio>`.

Return An `ostream` object;

ostream cerr

Controls output to an unbuffered stream.

Prototype `ostream cerr;`

Remarks The object `cerr` controls output to an unbuffered stream buffer associated with `stderr` declared in `<cstdio>`. After `err` is initialized, `err.flags()` and `unitbuf` is nonzero.

Return An `ostream` object;

ostream clog

Controls output to a stream buffer.

Prototype `ostream clog;`

Remarks The object `clog` controls output to a stream buffer associated with `cerr` declared in `<cstdio>`.

Return An `ostream` object;

27.3.2 Wide stream objects

Wide stream objects provide unbuffered input and output associated with standard input and output declared in `<cstdio>`.

wistream wcin

An unbuffered wide input stream.

Prototype `wistream wcin;`

Remarks The object `wcin` controls input from an unbuffered wide stream buffer associated with `stdin` declared in `<cstdio>`. After `wcin` is initialized `win.tie()` returns `wout`.

Return A `wistream` object;

wostream wcout

An unbuffered wide output stream.

Prototype `wostream wcout;`

iostream Objects

Header *<iostream>*

Remarks The object `cout` controls output to an unbuffered wide stream buffer associated with `stdout` declared in `<cstdio>`.

Return A `wostream` object;

wostream wcerr

Controls output to an unbuffered wide stream.

Prototype `wostream wcerr;`

Remarks The object `werr` controls output to an unbuffered wide stream buffer associated with `stderr` declared in `<cstdio>`. After `werr` is initialized, `werr.flags()` and `unitbuf` is nonzero.

Return A `wostream` object;

wostream wlco

Controls output to a wide stream buffer.

Prototype `wostream wlco;`

Remarks The object `wlog` controls output to a wide stream buffer associated with `cerr` declared in `<cstdio>`.

Return A `wostream` object

iostreams Base Classes

The include header `<iostream>` contains the basic class definitions, types, and enumerations necessary for input and output stream reading writing and other manipulations.

Input and Output Stream Base Library (clause 27.4)

The chapter is constructed in the following sub sections and mirrors clause 27.4 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header <ios>” on page 465](#)
- [“27.4.1 Typedef Declarations” on page 467](#)
- [“27.4.2 Class ios_base” on page 468](#)
- [“27.4.4 Template class basic_ios” on page 484](#)
- [“27.4.5 ios_base manipulators” on page 500](#)

Header <ios>

The header file `<ios>` provides for implementation of stream objects for standard input and output.

Listing 16.1 <ios> Globals

```
typedef long streamoff;
typedef long streamsize;

class ios_base;
template <class charT, class traits = ios_traits<charT> >
class basic_ios
```

iostreams Base Classes

Header <iostream>

```
typedef basic_ios<char> ios;
typedef basic_ios<wchar_t> wios;

ios_base& boolalpha (ios_base& str)
ios_base& noboolalpha (ios_base& str)

ios_base& showbase (ios_base& str)
ios_base& noshowbase (ios_base& str)

ios_base& showpoint (ios_base& str)
ios_base& noshowpoint (ios_base& str)

ios_base& showpos (ios_base& str)
ios_base& noshowpos (ios_base& str)

ios_base& skipws (ios_base& str)
ios_base& noskipws (ios_base& str)

ios_base& uppercase (ios_base& str)
ios_base& nouppercase (ios_base& str)

ios_base& internal (ios_base& str)
ios_base& left (ios_base& str)
ios_base& right (ios_base& str)

ios_base& dec (ios_base& str)
ios_base& hex (ios_base& str)
ios_base& oct (ios_base& str)

ios_base& fixed (ios_base& str)
ios_base& scientific (ios_base& str)

ios_base& unitbuf(ios_base& str);
ios_base& nounitbuf(ios_base& str);
```

Listing 16.2 Template class fpos

```
template <class stateT>
class fpos
{
public:
```

```
fpos(streamoff o);
operator streamoff() const;
fpos& operator += (streamoff o);
fpos& operator -= (streamoff o);
fpos operator + (streamoff o) const;
fpos operator - (streamoff o) const;
// _lib.fpos.members_ Members
stateT state() const;
void state(stateT s);
};

template <class stateT>
streamoff operator -
(const fpos<stateT>& lhs, const fpos<stateT>& rhs);
```

Template Class **fpos**

The template class `fpos<stateT>` is a class used for specifying file position information. The template parameter corresponds to the type needed to hold state information in a multi-byte sequence (typically `mbstate_t` from `<cwchar>`). `fpos` is essentially a wrapper for whatever mechanisms are necessary to hold a stream position (and multi-byte state). In fact the standard stream position typedefs are defined in terms of `fpos`:

```
typedef fpos<mbstate_t> streampos;
typedef fpos<mbstate_t> wstreampos;
```

The template class `fpos` is typically used in the `istream` and `ostream` classes in calls involving file position such as `tellg`, `tellp`, `seekg` and `seekp`. Though in these classes the `fpos` is typedef'd to `pos_type`, and can be changed to a custom implementation by specifying a traits class in the stream's template parameters.

27.4.1 **Typedef Declarations**

The following typedef's are defined in the class `ios_base`.

```
typedef long streamoff;
typedef long streamsize;
```

27.4.2 Class *ios_base*

A base class for input and output stream mechanisms

The prototype is listed below. Additional topics in this section are:

- [“27.4.3.1 Typedef Declarations” on page 470](#)
- [“27.4.3.1.1 failure” on page 470](#)
- [“27.4.3.1.1.1 failure” on page 471](#)
- [“27.4.3.1.2 Type fmtflags” on page 471](#)
- [“27.4.3.1.3 Type iostate” on page 472](#)
- [“27.4.3.1.4 Type openmode” on page 473](#)
- [“27.4.3.1.5 Type seekdir” on page 473](#)
- [“27.4.3.1.6 Class Init” on page 474](#)
- [“Class Init Constructor” on page 474](#)
- [“27.4.3.2 ios_base fmtflags state functions” on page 475](#)
- [“27.4.3.3 ios_base locale functions” on page 482](#)
- [“27.4.3.4 ios_base storage function” on page 482](#)
- [“27.4.3.5 ios_base Constructor” on page 484](#)

Prototype

```
namespace std{
    class ios_base{
        public: class failure;

        typedef T1 fmtflags;
        static const fmtflags boolalpha;
        static const fmtflags dec;
        static const fmtflags fixed;
        static const fmtflags hex;
        static const fmtflags internal;
        static const fmtflags left;
        static const fmtflags oct;
        static const fmtflags right;
        static const fmtflags scientific;
        static const fmtflags showbase;
        static const fmtflags showpoint;
        static const fmtflags showpos;
```

```
static const fmtflags skipws;
static const fmtflags unitbuf;
static const fmtflags uppercase;
static const fmtflags adjustfield;
static const fmtflags basefield;
static const fmtflags floatfield;

typedef T2 iostate;
static const iostate badbit;
static const iostate eofbit;
static const iostate failbit;
static const iostate goodbit;

typedef T3 openmode;
static const openmode app;
static const openmode ate;
static const openmode binary;
static const openmode in;
static const openmode out;
static const openmode trunc;

typedef T4 seekdir;
static const seekdir beg;
static const seekdir cur;
static const seekdir end;

class Init;
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
void unsetf(fmtflags mask);
streamsize precision() const;
streamsize precision(streamsize prec);
streamsize width() const;
streamsize width(streamsize wide);
locale imbue(const locale& loc);
locale getloc() const;
static int xalloc();
long& iword(int index);
void*& pword(int index);
```

iostreams Base Classes

27.4.2 Class *ios_base*

```
~ios_base();

enum event { erase_event, imbue_event, copyfmt_event };
typedef void (*event_callback)(event, ios_base&, int
index);
void register_callback(event_callback fn, int index);
static bool sync_with_stdio(bool sync = true);

protected:
    ios_base();

private:
    // static int index;      exposition only
    // long* iarray;         exposition only
    // void** parray;        exposition only
};

}
```

Remarks The `ios_base` class is a base class and includes many enumerations and mechanisms necessary for input and output operations.

27.4.3.1 Typedef Declarations

No types are specified in the current standards.

27.4.3.1.1 `failure`

Define a base class for types of object thrown as exceptions.

Prototype

```
namespace std {
    class ios_base::failure : public exception {
        public:
            explicit failure(const string&)
            virtual ~failure();
            virtual const char* what() const;
    };
}
```

27.4.3.1.1.1 failure

Construct a class failure.

| | |
|-----------|---|
| Prototype | <code>explicit failure(const string& msg);</code> |
| Remarks | The function <code>failure()</code> construct a class failure initializing with <code>exception(msg)</code> . |

failure::what

To return the exception message.

| | |
|-----------|---|
| Prototype | <code>const char *what() const;</code> |
| Remarks | The function <code>what()</code> is use to deliver the <code>msg.str()</code> . |
| Return | Returns the message with which the exception was created. |

27.4.3.1.2 Type `fmtflags`

An enumeration used to set various formatting flags for reading and writing of streams.

Table 16.1 Format Flags Enumerations

| Flag | Effects when set |
|-------------------------|---|
| <code>boolalpha</code> | insert or extract <code>bool</code> type in alphabetic form |
| <code>dec</code> | decimal output |
| <code>fixed</code> | when set shows floating point numbers in normal manner, six decimal places is default |
| <code>hex</code> | hexadecimal output |
| <code>oct</code> | octal output |
| <code>left</code> | left justified |
| <code>right</code> | right justified |
| <code>internal</code> | pad a field between signs or base characters |
| <code>scientific</code> | show scientific notation for floating point numbers |

| Flag | Effects when set |
|-------------|---|
| showbase | shows the bases numeric values |
| showpoint | shows the decimal point and trailing zeros |
| showpos | shows the leading plus sign for positive numbers |
| skipws | skip leading white spaces with input |
| unitbuf | buffer the output and flush after insertion operation |
| uppercase | show the scientific notation, x or o in uppercase |

Table 16.2 Format flag field constants

| Constants | Allowable values |
|------------------|-------------------------|
| adjustfield | left right internal |
| basefield | dec oct hex |
| floatfield | scientific fixed |

Listing 16.3 Example of *ios* format flags usagesee `basic_ios::setf()` and `basic_ios::unsetf()`**27.4.3.1.3 Type iostate**

An enumeration that is used to define the various states of a stream.

Table 16.3 Enumeration iostate

| Flags | Usage |
|--------------|---|
| goodbit | True when all of badbit, eofbit and failbit are false. |
| badbit | True when the stream is in an irrecoverable error state (such as failure due to lack of memory) |

| Flags | Usage |
|---------------------------------------|--|
| failbit | True when a read or a write has failed for any reason (This can happen for example when the input read a character while attempting to read an integer.) |
| eofbit | True when the end of the stream has been detected. Note that eofbit can be set during a read, and yet the read may still succeed (failbit not set). (This can happen for example when an integer is the last character in a file.) |
| note: see variance from AT&T standard | |

Listing 16.4 Example of ios iostate flags usage:

See `basic_ios::setstate()` and `basic_ios::rdstate()`

27.4.3.1.4 Type **openmode**

An enumeration that is used to specify various file opening modes.

Table 16.4 Enumeration `openmode`

| Mode | Definition |
|-------------|--|
| app | Start the read or write at end of the file |
| ate | Start the read or write immediately at the end |
| binary | binary file |
| in | Start the read at end of the stream |
| out | Start the write at the beginning of the stream |
| trunc | Start the read or write at the beginning of the stream |

27.4.3.1.5 Type **seekdir**

An enumeration to position a pointer to a specific place in a file stream.

Table 16.5 Enumeration seekdir

| Enumeration | Position |
|--------------------|----------------------------|
| beg | Begging of stream |
| cur | Current position of stream |
| end | End of stream |

Listing 16.5 Example of ios seekdir usage:**See:** `streambuf::pubseekoff`**27.4.3.1.6 Class Init**

An object that associates `<iostream>` object buffers with standard stream declared in `<cstdio>`.

Prototype `namespace std { class ios_base::Init { public: Init(); ~Init(); private: // static int }; }`

Class Init Constructor**Default Constructor**

To construct an object of class `Init`:

Prototype `Init();`

Remarks The constructor `Init()` constructs an object of class `Init`. If `init_cnt` is zero the function stores the value one and constructs `cin`, `cout`, `cerr`, `clog`, `win`, `wout`, `werr` and `wlog`. In any case the constructor then adds one to `init_cnt`.

Destructor

Prototype `~Init();`

Remarks The destructor subtracts one from `init_cnt` and if the result is one calls `cout.flush()`, `cerr.flush()` and `clog.flush()`.

27.4.3.2 *ios_base* `fmtflags` state functions

To set the state of the *ios_base* format flags.

flags

To alter formatting flags using a mask.

Prototype `fmtflags flags() const`
`fmtflags flags(fmtflags)`

Remarks Use `flags()` when you would like to use a mask of several flags, or would like to save the current format configuration. The return value of `flags()` returns the current `fmtflags`. The overloaded `flags(fmtflags)` alters the format flags but will return the value prior to the flags being changed.

Return The `fmtflags` type before alterations.

NOTE See *ios* enumerators for a list of `fmtflags`.

See Also: `setiosflags()` and `resetiosflags()`

Listing 16.6 Example of `flags()` usage:

```
#include <iostream>

// showf() displays flag settings
void showf();

int main()
{
    using namespace std;
    showf(); // show format flags

    cout << "press enter to continue" << endl;
    cin.get();

    cout.setf(ios::right|ios::showpoint|ios::fixed);
```

iostreams Base Classes

27.4.2 Class *ios_base*

```
showf();
return 0;
}

// showf() displays flag settings
void showf()
{
using namespace std;

char fflags[][12] = {
    "boolalpha",
    "dec",
    "fixed",
    "hex",
    "internal",
    "left",
    "oct",
    "right",
    "scientific",
    "showbase",
    "showpoint",
    "showpos",
    "skipws",
    "unitbuf",
    "uppercase"
};

long f = cout.flags();      // get flag settings
cout.width(9); // for demonstration
    // check each flag
for(long i=1, j =0; i<=0x4000; i = i<<1, j++)
{
    cout.width(10); // for demonstration
    if(i & f)
        cout << fflags[j] << " is on \n";
    else
        cout << fflags[j] << " is off \n";
}

cout << "\n";
}
```

Result:

```
boolalpha    is off
dec          is on
fixed        is off
hex          is off
internal     is off
left         is off
oct           is off
right        is off
scientific   is off
showbase     is off
showpoint    is off
showpos      is off
skipws       is on
unitbuf      is off
uppercase    is off
```

press enter to continue

```
boolalpha is off
          dec is on
          fixed is on
          hex is off
internal is off
          left is off
          oct is off
          right is on
scientific is off
          showbase is off
showpoint is on
          showpos is off
          skipws is on
          unitbuf is off
uppercase is off
```

setf

Set the stream format flags.

Prototype `fmtflags setf(fmtflags)`
 `fmtflags setf(fmtflags, fmtflags)`

iostreams Base Classes

27.4.2 Class ios_base

Remarks You should use the function `setf()` to set the formatting flags for input/output. It is overloaded. The single argument form of `setf()` sets the flags in the mask. The two argument form of `setf()` clears the flags in the first argument before setting the flags with the second argument.

Return type `basic_ios::fmtflags`

Listing 16.7 Example of setf() usage:

```
#include <iostream>

int main()
{
using namespace std;

    double d = 10.01;

    cout.setf(ios::showpos | ios::showpoint);
    cout << d << endl;
    cout.setf(ios::showpoint, ios::showpos | ios::showpoint);
    cout << d << endl;

    return 0;
}
```

Result:

```
+10.01
10.01
```

unsetf

To un-set previously set formatting flags.

Prototype `void unsetf(fmtflags)`

Remarks Use the `unsetf()` function to reset any format flags to a previous condition. You would normally store the return value of `setf()` in order to achieve this task.

Return There is no return.

Listing 16.8 Example of unsetf() usage:

```
#include <iostream>

int main()
{
using namespace std;

double d = 10.01;

cout.setf(ios::showpos | ios::showpoint);
cout << d << endl;

cout.unsetf(ios::showpoint);
cout << d << endl;
return 0;
}

Result:
+10.01
+10.01
```

precision

Set and return the current format precision.

Prototype streamsize precision() const
 streamsize precision(streamsize prec)

Remarks Use the `precision()` function with floating point numbers to limit the number of digits in the output. You may use `precision()` with scientific or non-scientific floating point numbers. You may use the overloaded `precision()` to retrieve the current precision that is set.

With the flag `ios::floatfield` set the number in `precision` refers to the total number of significant digits generated. If the settings are for either `ios::scientific` or `ios::fixed` then the precision refers to the number of digits after the decimal place.

iostreams Base Classes

27.4.2 Class *ios_base*

NOTE This means that `ios::scientific` will have one more significant digit than `ios::floatfield`, and `ios::fixed` will have a varying number of digits.

Return The current value set.

See Also `setprecision()`

Listing 16.9 Example of precision() usage:

```
#include <iostream>
#include <cmath>

const double pi = 4 * std::atan(1.0);

int main()
{
using namespace std;

    double TenPi = 10*pi;

    cout.precision(5);
    cout.unsetf(ios::floatfield);
    cout << "floatfield:\t" << TenPi << endl;
    cout.setf(ios::scientific, ios::floatfield);
    cout << "scientific:\t" << TenPi << endl;
    cout.setf(ios::fixed, ios::floatfield);
    cout << "fixed:\t\t" << TenPi << endl;
    return 0;
}
```

Result:

```
floatfield: 31.416
scientific: 3.14159e+01
fixed:      31.41593
```

width

To set the width of the output field.

Prototype `streamsize width() const`
`streamsize width(streamsize wide)`

Remarks Use the `width()` function to set the field size for output. The function is overloaded to return just the current width setting if there is no parameter or to store and then return the previous setting before changing the fields width to the new parameter.

NOTE `Width` is the one and only modifier that is not sticky and needs to be reset with each use. `Width` is reset to `width(0)` after each use.

Return The previous width setting is returned.

Listing 16.10 Example of `width()` usage:

```
#include <iostream>

int main()
{
using namespace std;

    int width;

    cout.width(8);
    width = cout.width();
    cout.fill('*');
    cout << "Hi!" << '\n';

    // reset to left justified blank filler
    cout<< "Hi!" << '\n';

    cout.width(width);
    cout<< "Hi!" << endl;

    return 0;
}
```

Result:

```
Hi!*****
Hi!
Hi!*****
```

27.4.3.3 **ios_base** locale functions

Sets the locale for input output operations.

imbuf

Stores a value representing the locale.

Prototype `locale imbue(const locale loc);`

Remarks The precondition of the argument loc is equal to `getloc()`.

Return The previous value of `getloc()`.

getloc

Determined the imbued locale for input output operations.

Prototype `locale getloc() const;`

Return The global C++ locale if no locale has been imbued. Otherwise it returns the locale of the input and output operations.

27.4.3.4 **ios_base** storage function

To allocate storage pointers.

xalloc

Allocation function.

Prototype `static int xalloc()`

Return `index++`.

iword

Allocate an array of `int` and store a pointer.

Remark If `iarray` is a null pointer allocate an array and store a pointer to the first element. The function extends the array as necessary to include `iarray[idx]`. Each new allocated element is initialized to the return value may be invalid.

NOTE After a subsequent call to `iword()` for the same object the return value may be invalid.

Return `irarray[idx]`

pword

Allocate an array of pointers.

Prototype `void * &pword(int idx)`

Remarks If `parray` is a null pointer allocates an array of void pointers. Then extends `parray` as necessary to include the element `parray[idx]`.

NOTE After a subsequent call to `pword()` for the same object the return value may be invalid.

Return `parray[idx].`

register_callback

Registers functions when an event occurs.

Prototype `void register_callback
(event_callback fn,
int index);`

Remarks Registers the pair `(fn, index)` such that during calls to `imbuf()`, `copyfmt()` or `~ios_base()` the function `fn` is called with argument `index`. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

NOTE Identical pairs are not merged and a function registered twice will be called twice.

sync_with_stdio

Synchronizes stream input output with 'C' input and output functions.

Prototype static bool sync_with_stdio(bool sync = true);

Remarks Is not supported in the Metrowerks Standard Library.

Return Always returns `true` indicating that the MSLstreams are always synchronized with the C streams.

27.4.3.5 ios_base Constructor

Default Constructor

Construct and destruct an object of class `ios_base`

Prototype protected:
 `ios_base()`;

Remarks The `ios_base` constructor is protected so it may only be derived from. If the values of the `ios_base` members are undermined.

Destructor

Prototype `~ios_base()`;

Remarks Calls registered callbacks and destroys an object of class `ios_base`.

27.4.4 Template class *basic_ios*

A template class for input and output streams.

The prototype is listed below. Additional topics in this section are:

- [“27.4.4.1 basic_ios Constructor” on page 486](#)
- [“27.4.4.2 Member Functions” on page 487](#)
- [“27.4.4.3 basic_ios iostate flags functions” on page 491](#)

Prototype namespace std{
 template<class charT,
 class traits = ios_traits<charT>>
 class basic_ios : public ios_base {

```
public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    operator bool() const;
    bool operator!() const;
    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;

    explicit basic_ios
        (basic_streambuf<charT, traits>,
         traits *sb);

    virtual ~basic_ios();

    basic_ostream<charT, traits>* tie() const;
    basic_ostream<charT, traits>*
        tie(basic_streambuf<charT, traits*> sb);

    basic_streambuf<charT, traits>* rdbuf() const;
    basic_streambuf<charT, traits>*
        rdbuf(basic_streambuf<charT, traits*> sb);

    basic_ios& copyfmt(const basic_ios& rhs);

    char_type fill()const;
    char_type fill(char_type ch);

    locale imbue(const locale& loc);

protected:
    basic_ios();
    void init(basic_streambuf<charT, traits>* sb);
};

}
```

Remarks The `basic_ios` template class is a base class and includes many enumerations and mechanisms necessary for input and output operations.

27.4.4.1 `basic_ios` Constructor

Default and Overloaded Constructor

Construct an object of class `basic_ios` and assign values.

Prototype

```
public:  
    explicit basic_ios  
        (basic_streambuf<charT,traits>* sb);  
protected:  
    basic_ios();
```

Remarks The `basic_ios` constructor creates an object to class `basic_ios` and assigns values to its member functions by calling `init()`.

Destructor

Prototype

```
virtual ~basic_ios();
```

Remarks Destroys an object of type `basic_ios`.

Remarks The conditions of the member functions after `init()` are shown in the following table.

Table 16.6 Conditions after `init()`

| Member | Postcondition Value |
|---------------------------|--|
| <code>rdbuf()</code> | <code>sb</code> |
| <code>tie()</code> | <code>zero</code> |
| <code>rdstate()</code> | goodbit if stream buffer is not a null pointer otherwise badbit. |
| <code>exceptions()</code> | goodbit |
| <code>flags()</code> | <code>skipws dec</code> |
| <code>width()</code> | <code>zero</code> |

| Member | Postcondition Value |
|--------------------------|--------------------------------|
| <code>precision()</code> | <code>six</code> |
| <code>fill()</code> | the space character |
| <code>getloc()</code> | <code>locale::classic()</code> |
| <code>iarray</code> | a null pointer |
| <code>parray</code> | a null pointer |

27.4.4.2 Member Functions

tie

To tie an `ostream` to the calling stream.

| | |
|-----------|--|
| Prototype | <code>basic_ostream<charT, traits>* tie() const;</code> <code>basic_ostream<charT, traits>* tie</code> <code>(basic_ostream<charT, traits>* tiestr);</code> |
| Remarks | Any stream can have an <code>ostream</code> tied to it to ensure that the <code>ostream</code> is flushed before any operation. The standard input and output objects <code>cin</code> and <code>cout</code> are tied to ensure that <code>cout</code> is flushed before any <code>cin</code> operation. The function <code>tie()</code> is overloaded the parameterless version returns the current <code>ostream</code> that is tied if any. The <code>tie()</code> function with an argument ties the new object to the <code>ostream</code> and returns a pointer if any from the first. The post-condition of <code>tie()</code> function that takes the argument <code>tiestr</code> is that <code>tiestr</code> is equal to <code>tie();</code> |
| Return | A pointer to type <code>ostream</code> that is or previously was tied, or zero if there was none. |

Listing 16.11 Example of tie() usage:

The file MW Reference contains
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>
```

iostreams Base Classes

27.4.4 Template class *basic_ios*

```
char inFile[ ] = "MW Reference";

int main()
{
using namespace std;

    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        { cout << "file is not open"; exit(1);}
    ostream Out(inOut.rdbuf());

    if(inOut.tie())
        cout << "The streams are tied\n";
    else cout << "The streams are not tied\n";

    inOut.tie(&Out);
    inOut.rdbuf()->pubseekoff(0, ios::end);

    char str[ ] = "\nRegistered Trademark";
    Out << str;

    if(inOut.tie())
        cout << "The streams are tied\n";
    else cout << "The streams are not tied\n";

    inOut.close();
    return 0;
}
```

Result:

```
The streams are not tied
The streams are tied
```

```
The file MW Reference now contains
Metrowerks CodeWarrior "Software at Work"
Registered Trademark
```

rdbuf

To retrieve a pointer to the stream buffer.

Prototype `basic_streambuf<charT, traits>* rdbuf() const;`

```
basic_streambuf<charT, traits>* rdbuf  
(basic_streambuf<charT, traits>* sb);
```

Remarks To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer. The `rdbuf()` function that takes an argument has the post-condition of `sb` is equal to `rdbuf()`.

Return A pointer to `basic_streambuf` object.

Listing 16.12 Example of `rdbuf()` usage:

```
#include <iostream>  
  
struct address {  
    int number;  
    char street[40];  
} addbook;  
  
int main()  
{  
using namespace std;  
  
cout << "Enter your street number: ";  
cin >> addbook.number;  
  
cin.rdbuf()->pubsync(); // buffer flush  
  
cout << "Enter your street name: ";  
cin.get(addbook.street, 40);  
  
cout << "Your address is: "  
    << addbook.number << " " << addbook.street;  
  
return 0;  
}
```

Result:

```
Enter your street number: 2201  
Enter your street name: Donley Drive  
Your address is: 2201 Donley Drive
```

imbue

Stores a value representing the locale.

Prototype `locale imbue(const locale& rhs);`

Remarks The function `imbue()` calls `ios_base::imbue()` and `rdbuf->pubimbue()`.

Return The current locale.

fill

To insert characters into the stream's unused spaces.

Prototype `char_type fill() const`
`char_type fill(char_type)`

Remarks Use `fill(char_type)` in output to fill blank spaces with a character. The function `fill()` is overloaded to return the current filler without altering it.

Return The current character being used as a filler.

See Also `manipulator setfill()`

Listing 16.13 Example of fill() usage:

```
#include <iostream>

int main()
{
using namespace std;

    char fill;

    cout.width(8);
    cout.fill('*');
    fill = cout.fill();
    cout<< "Hi!" << "\n";
    cout << "The filler is a " << fill << endl;

    return 0;
}
```

```
Result:  
Hi!*****  
The filler is a *
```

copyfmt

Copies a `basic_ios` object.

Prototype `basic_ios& copyfmt(const basic_ios& rhs);`

Remarks Assigns members of `*this` object the corresponding objects of the `rhs` argument with certain exceptions. The exceptions are `rdstate()` is unchanged, `exceptions()` is altered last, and the contents or `pword` and `iword` arrays are copied not the pointers themselves.

Return The `this` pointer.

27.4.4.3 basic_ios iostate flags functions

To set flags pertaining to the state of the input and output streams.

operator bool

A `bool` operator.

Prototype `operator bool() const;`

Return `!fail()`

operator !

A `bool not` operator.

Prototype `bool operator !();`

Return `fail()`.

rdstate

To retrieve the state of the current formatting flags.

| | |
|-----------|--|
| Prototype | <code>iostate rdstate() const</code> |
| Remarks | This member function allows you to read and check the current status of the input and output formatting flags. The returned value may be stored for use in the function <code>ios::setstate()</code> to reset the flags at a later date. |
| Return | Type <code>iostate</code> used in <code>ios::setstate()</code> |
| See Also | <code>ios::setstate()</code> |

Listing 16.14 Example of rdstate() usage:

The file MW Reference contains:

ABCDEFGHIJKLMNPQRSTUVWXYZ

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char * inFile = "MW Reference";

using namespace std;

void status(ifstream &in);

int main()
{
    ifstream in(inFile);
    if(!in.is_open())
    {
        cout << "could not open file for input";
        exit(1);
    }

    int count = 0;
    int c;
    while((c = in.get()) != EOF)
    {
        // simulate a bad bit
        if(count++ == 12) in.setstate(ios::badbit);
        status(in);
    }
}
```

```
    status(in);
    in.close();
    return 0;
}

void status(ifstream &in)
{
    int i = in.rdstate();
    switch (i) {
        case ios::eofbit : cout << "EOF encountered \n";
                            break;
        case ios::failbit : cout << "Non-Fatal I/O Error n";
                            break;
        case ios::goodbit : cout << "GoodBit set \n";
                            break;
        case ios::badbit : cout << "Fatal I/O Error \n";
                            break;
    }
}
```

Result:

```
GoodBit set
Fatal I/O Error
```

clear

Clears `iostate` field.

Prototype `void clear
(iostate state = goodbit) throw failure;`

Remarks Use `clear()` to reset the `failbit`, `eofbit` or a `badbit` that may have been set inadvertently when you wish to override for continuation of your processing. Post-condition of `clear` is the argument is equal to `rdstate()`.

NOTE If `rdstate()` and `exceptions() != 0` an exception is thrown.

Return No value is returned.

Listing 16.15 Example of `clear()` usage:

The file MW Reference contains:

ABCDEFGH

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char * inFile = "MW Reference";

using namespace std;

void status(ifstream &in);

int main()
{
    ifstream in(inFile);
    if(!in.is_open())
    {
        cout << "could not open file for input";
        exit(1);
    }

    int count = 0;
    int c;
    while((c = in.get()) != EOF) {
        if(count++ == 4)
        {
            // simulate a failed state
            in.setstate(ios::failbit);
            in.clear();
        }
    }
}
```

```
status(in);
}

status(in);
in.close();
return 0;
}

void status(ifstream &in)
{
    // note: eof() is not needed in this example
    // if(in.eof()) cout << "EOF encountered \n"
    if(in.fail()) cout << "Non-Fatal I/O Error \n";
    if(in.good()) cout << "GoodBit set \n";
    if(in.bad()) cout << "Fatal I/O Error \n";
}
```

Result:

```
GoodBit set
Non-Fatal I/O Error
```

setstate

To set the state of the format flags.

Prototype `void setstate(iostate state) throw(failure);`

Remarks Calls `clear(rdstate() | state)` and may throw and exception.

Return No Return

Listing 16.16 Example of `setstate()` usage:

See `ios::rdstate()`

good

To test for the lack of error bits being set.

Prototype `bool good() const;`

Remarks Use the function `good()` to test for the lack of error bits being set.

Returns

True if `rdstate() == 0`.

Listing 16.17 Example of good() usage:

See `basic_ios::bad()`

eof

To test for the eofbit setting.

Prototype `bool eof() const;`

Remarks Use the `eof()` function to test for an eofbit setting in a stream being processed under some conditions. This end of file bit is not set by stream opening or closing, but only for operations that detect an end of file condition.

Return If eofbit is set in `rdstate()` true is returned.

Listing 16.18 Example of eof() usage:

MW Reference is simply a one line text document
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

```
#include <iostream>
#include <fstream>
#include <cstdlib>

const char* TheText = "MW Reference";

int main()
{
using namespace std;

    ifstream in(TheText);
```

```
if(!in.is_open())
{
    cout << "Couldn't open file for input";
    exit(1);
}

int i = 0;
char c;
cout.setf(ios::uppercase);

//eofbit is not set under normal file opening
while(!in.eof())
{
    c = in.get();
    cout << c << " " << hex << int(c) << "\n";

    // simulate an end of file state
    if(++i == 5) in.setstate(ios::eofbit);
}
return 0;
}
```

Result:

A 41
B 42
C 43
D 44
E 45

fail

To test for stream reading failure from any cause.

Prototype `bool fail() const`

Remarks The member function `fail()` will test for `failbit` and `badbit`.

Return True if `failbit` or `badbit` is set in `rdstate()`.

Listing 16.19 Example of fail() usage:

```
MW Reference file for input contains.  
float 33.33 double 3.16e+10 Integer 789 character C  
  
#include <iostream>  
#include <fstream>  
#include <cstdlib>  
  
int main()  
{  
using namespace std;  
  
char inFile[] = "MW Reference";  
ifstream in(inFile);  
if(!in.is_open())  
{cout << "Cannot open input file"; exit(1);}  
  
char ch = 0;  
  
while(!in.fail())  
{  
    if(ch)cout.put(ch);  
    in.get(ch);  
}  
  
return 0;  
}
```

Result:
float 33.33 double 3.16e+10 integer 789 character C

bad

To test for fatal I/O error.

Prototype `bool bad() const`

Remarks Use the member function `bad()` to test if a fatal input or output error occurred which sets the `badbit` flag in the stream.

Return True if `badbit` is set in `rdstate()`.

See Also `basic_ios::fail()`

Listing 16.20 Example of `bad()` usage:

The File MW Reference contains:

abcdefghijklmnoprstuvwxyz

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char * inFile = "MW Reference";

using namespace std;

void status(ifstream &in);

int main()
{
    ifstream in(inFile);
    if(!in.is_open())
    {
        cout << "could not open file for input";
        exit(1);
    }

    int count = 0;
    int c;
    while((c = in.get()) != EOF)
    {
        // simulate a failed state
        if(count++ == 4) in.setstate(ios::failbit);
        status(in);
    }

    status(in);
    in.close();
    return 0;
}

void status(ifstream &in)
{
    // note: eof() is not needed in this example
```

```
// if(in.eof()) cout << "EOF encountered \n";  
  
if(in.fail()) cout << "Non-Fatal I/O Error \n";  
if(in.good()) cout << "GoodBit set \n";  
if(in.bad()) cout << "Fatal I/O Error \n";  
}
```

Result:

```
GoodBit set  
GoodBit set  
GoodBit set  
GoodBit set  
Non-Fatal I/O Error  
Non-Fatal I/O Error
```

exceptions

To handle `basic_ios` exceptions.

Prototype `iostate exceptions() const;`
 `void exceptions(iostate except);`

Remarks The function `exceptions()` determines what elements in `rdstate()` cause exceptions to be thrown. The overloaded `exceptions(iostate)` calls `clear(rdstate())` and leaves the argument `except` equal to `exceptions()`.

Return A mask that determines what elements set in `rdstate()` cause undefined behavior.

27.4.5 *ios_base manipulators*

To provide an in line input and output formatting mechanism.

The topics in this section are:

- [“27.4.5.1 `fmtflags` manipulators” on page 501](#)
- [“27.4.5.2 `adjustfield` manipulators” on page 502](#)
- [“27.4.5.3 `basefield` manipulators” on page 502](#)
- [“27.4.5.4 `floatfield` manipulators” on page 503](#)

27.4.5.1 **fmtflags** manipulators

To provide an in line input and output numerical formatting mechanism.

Table 16.7 Prototype of *ios_base* manipulators

| Manipulator | Definition |
|--|--|
| <code>ios_base& boolalpha(ios_base&)</code> | insert and extract bool type in alphabetic format |
| <code>ios_base& noboolalpha (ios_base&)</code> | unsets insert and extract bool type in alphabetic format |
| <code>ios_base& showbase(ios_base& b)</code> | set the number base to parameter b |
| <code>ios_base& noshowbase (ios_base&)</code> | remove show base |
| <code>ios_base& showpoint (ios_base&)</code> | show decimal point |
| <code>ios_base& noshowpoint(ios_base&)</code> | do not show decimal point |
| <code>ios_base& showpos(ios_base&)</code> | show the positive sign |
| <code>ios_base& noshowpos(ios_base&)</code> | do not show positive sign |
| <code>ios_base& skipws(ios_base&)</code> | input only skip white spaces |
| <code>ios_base& noskipws(ios_base&)</code> | input only no skip white spaces |
| <code>ios_base& uppercase(ios_base&)</code> | show scientific in uppercase |

| | Manipulator | Definition |
|---------|--|-------------------------------------|
| | <code>ios_base& nouppercase (ios_base&)</code> | do not show scientific in uppercase |
| | <code>ios_base& unitbuf (ios_base::unitbuf)</code> | set the unitbuf flag |
| | <code>ios_base& nounitbuf (ios_base::unitbuf)</code> | unset the unitbuf flag |
| Remarks | Manipulators are used in the stream to alter the formatting of the stream. | |
| Return | A reference to an object of type <code>ios_base</code> is returned to the stream. (The <code>this</code> pointer.) | |

27.4.5.2 **adjustfield manipulators**

To provide an in line input and output orientation formatting mechanism.

Table 16.8 Adjustfield manipulators

| | Manipulator | Definition |
|---------|--|----------------------------------|
| | <code>ios_base& internal(ios_base&)</code> | fill between indicator and value |
| | <code>ios_base& left(ios_base&)</code> | left justify in a field |
| | <code>ios_base& right(ios_base&)</code> | right justify in a field |
| Remarks | Manipulators are used in the stream to alter the formatting of the stream. | |
| Return | A reference to an object of type <code>ios_base</code> is returned to the stream. (The <code>this</code> pointer.) | |

27.4.5.3 **basefield manipulators**

To provide an in line input and output numerical formatting mechanism.

Table 16.9 Basefield manipulators

| | Manipulator | Definition |
|---------|--|-----------------------------------|
| | <code>ios_base& dec(ios_base&)</code> | format output data as a decimal |
| | <code>ios_base& oct(ios_base&)</code> | format output data as octal |
| | <code>ios_base& hex(ios_base&)</code> | format output data as hexadecimal |
| Remarks | Manipulators are used in the stream to alter the formatting of the stream. | |
| Return | A reference to an object of type <code>ios_base</code> is returned to the stream. (The <code>this</code> pointer.) | |

27.4.5.4 floatfield manipulators

To provide an in line input and output numerical formatting mechanism.

Table 16.10 Floatfield manipulators

| | Manipulator | Definition |
|---------|--|--------------------------------|
| | <code>ios_base& fixed(ios_base&)</code> | format in fixed point notation |
| | <code>ios_base& scientific(ios_base&)</code> | use scientific notation |
| Remarks | Manipulators are used in the stream to alter the formatting of the stream. | |
| Return | A reference to an object of type <code>ios_base</code> is returned to the stream. (The <code>this</code> pointer.) | |

Listing 16.21 Example of manipulator usage:

```
#include <iostream>
#include <iomanip>

int main()
{
using namespace std;

long number = 64;
```

```
cout << "Original Number is "
    << number << "\n\n";
cout << showbase;
cout << setw(30) << "Hexadecimal :"
    << hex << setw(10) << right
    << number << '\n';
cout << setw(30) << "Octal :"
    << setw(10) << left
    << number << '\n';
cout << setw(30) << "Decimal :"
    << dec
    << setw(10) << right
    << number << endl;

return 0;
}
```

Result:

Original Number is 64

Hexadecimal : 0x40

Octal : 0100

Decimal : 64

Overloading Manipulators

To provide an in line formatting mechanism.

Prototype The basic template for parameterless manipulators,
 `ostream &manip-name(ostream &stream)`
 {
 // coding
 return stream;
 }

Remarks Use overloaded manipulators to provide specific and unique
 formatting methods relative to one class.

Return A reference to `ostream`. (Usually the `this` pointer.)

See Also `<iomanip>` for manipulators with parameters

Listing 16.22 Example of overloaded manipulator usage:

```
#include <iostream>

using namespace std;

ostream &rJus(ostream &stream);

int main()
{
    cout << "align right " << rJus << "for column";
    return 0;
}

ostream &rJus(ostream &stream)
{
    stream.width(30);
    stream.setf(ios::right);
    return stream;
}
```

Result:

for column

iostreams Base Classes

27.4.5 ios_base manipulators

Stream Buffers

The header `<streambuf>` defines types that control input and output to character sequences.

The Stream Buffers Library (clause 27.5)

The chapter is constructed in the following sub sections and mirrors clause 27.5 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header `<streambuf>`” on page 507](#)
- [“27.5.1 Stream buffer requirements” on page 507](#)
- [“27.5.2 Template class `basic_streambuf<charT, traits>`” on page 508](#)

Header `<streambuf>`

```
Prototype    namespace std {  
              template <class charT, class traits =  
                      char_traits<charT> >  
              class basic_streambuf;  
              typedef basic_streambuf<char> streambuf;  
              typedef basic_streambuf<wchar_t> wstreambuf;  
          }
```

27.5.1 Stream buffer requirements

Stream buffers can impose constraints. The constraints include:

- The input sequence can be not readable
- The output sequence can be not writable

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

- The sequences can be associated with other presentations such as external files
- The sequences can support operations to or from associated sequences.
- The sequences can impose limitations on how the program can read and write characters to and from a sequence or alter the stream position.

There are three pointers that control the operations performed on a sequence or associated sequences. These are used for read, writes and stream position alteration. If not `null` all pointers point to the same `charT` array object.

- The beginning pointer or lowest element in an array. - (`beg`)
- The next pointer of next element addressed for read or write. - (`next`)
- The end pointer of first element addressed beyond the end of the array. - (`end`)

27.5.2 Template class `basic_streambuf<charT, traits>`

The prototype is listed below. Additional topics in this section are:

- [“27.5.2.1 basic_streambuf Constructor” on page 511](#)
- [“27.5.2.2 basic_streambuf Public Member Functions” on page 511](#)
- [“27.5.2.2.1 Locales” on page 512](#)
- [“27.5.2.2.2 Buffer Management and Positioning” on page 512](#)
- [“27.5.2.2.3 Get Area” on page 517](#)
- [“27.5.2.2.4 Putback” on page 520](#)
- [“27.5.2.2.5 Put Area” on page 522](#)
- [“27.5.2.3 basic_streambuf Protected Member Functions” on page 523](#)
- [“27.5.2.3.1 Get Area Access” on page 523](#)
- [“27.5.2.3.2 Put Area Access” on page 525](#)
- [“27.5.2.4 basic_streambuf Virtual Functions” on page 526](#)

- [“27.5.2.4.1 Locales” on page 526](#)
- [“27.5.2.4.2 Buffer Management and Positioning” on page 526](#)
- [“27.5.2.4.3 Get Area” on page 527](#)
- [“27.5.2.4.4 Putback” on page 529](#)
- [“27.5.2.4.5 Put Area” on page 530](#)

Prototype

```
namespace std {
template< class charT, class traits = char_traits<charT> >
class basic_streambuf {
public:

    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    virtual ~basic_streambuf();

    locale pubimbue(const locale &loc);
    locale getloc() const;

    basic_streambuf<char_type, traits> * pubsetbuf
        (char_type* s, streamsize n);

    pos_type pubseekoff
        (off_type off,
         ios_base::seekdir way,
         ios_base::openmode which = ios_base::in | ios_base::out);
    pos_type pubseekpos
        (pos_type sp,
         ios_base::openmode which = ios::in | ios::out);

    int pubsync();

    streamsize in_avail();

    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
    streamsize sgetn(char_type *s, streamsize n);
```

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

```
int_type sputback(char_type C);
int_type sungetc();

int_type sputc(char_type c);
int_type sputn(char_type *s, streamsize n);

protected:
    basic_streambuf();

    char_type* eback() const;
    char_type* gptr() const;
    char_type* egptr() const;
    void gbump(int n);
    void setg
        (char_type *gbeg,
         char_type *gnext,
         char_type *gend);

    char_type* pbase() const;
    char_type* pptr() const;
    char_type* eptr() const;

    void pbump(int n);
    void setp(char_type *pbeg, char_type *pend);

    virtual void imbue(const locale &loc);

    virtual basic_streambuf<char_type, traits>* setbuf
        (char_type* s, streamsize n);

    virtual pos_type seekoff
        (off_type off,
         ios_base::seekdir way,
         ios_base::openmode which = ios::in | ios::out);
    virtual pos_type seekpos
        (pos_type sp,
         ios_base::openmode which = ios::in | ios::out);
    virtual int sync();

    virtual int showmany();
    virtual streamsize xsgetn(char_type *s,
```

```

        streamsize n);
virtual int_type underflow();
virtual int_type uflow();

virtual int_type
    pbackfail(int_type c = traits::eof());

virtual streamsize xsputn
    (const char_type *s,streamsize n);
virtual int_type overflow
    (int_type c = traits::eof());
};

}

```

Remarks The template class `basic_streambuf` is an abstract class for deriving various stream buffers whose objects control input and output sequences. The type `streambuf` is an instantiation of `char` type. the type `wstreambuf` is an instantiation of `wchar_t` type.

27.5.2.1 `basic_streambuf` Constructor

Default Constructor

Construct and destruct an object of type `basic_streambuf`.

Prototype `protected:`
`basic_streambuf();`

Remarks The constructor sets all pointer member objects to null pointers and calls `getloc()` to copy the global locale at the time of construction.

Destructor

Prototype `virtual ~basic_streambuf();`

Remarks Removes the object from memory.

27.5.2.2 `basic_streambuf` Public Member Functions

The public member functions allow access to member functions from derived classes.

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

27.5.2.2.1 Locales

Locales are used for encapsulation and manipulation of information to a particular locale.

`basic_streambuf::pubimbue`

To set the locale.

Prototype `locale pubimbue(const locale &loc);`

Remarks The function `pubimbue` calls `imbue(loc)`.

Return The previous value of `getloc()`.

`basic_streambuf::getloc`

To get the locale.

Prototype `locale getloc() const;`

Return If `pubimbue` has already been called one it returns the last value of `loc` supplied otherwise the current one. If `pubimbue` has been called but has not returned a value it from `imbue`, it then returns the previous value.

27.5.2.2 Buffer Management and Positioning

Functions used to manipulate the buffer and the input and output positioning pointers.

`basic_streambuf::pubsetbuf`

To set an allocation after construction.

Prototype `basic_streambuf<char_type, traits> *pubsetbuf
(char_type* s, streamsize n);`

Remarks The first argument is used in another function by a `filebuf` derived class. See `setbuf()`. The second argument is used to set the size of a dynamic allocated buffer.

Return A pointer to `basic_streambuf<char_type, traits>` via `setbuf(s, n)`.

Listing 17.1 Example of `basic_streambuf::pubsetbuf()` usage:

```
#include <iostream>
#include <sstream>

const int size = 100;
char temp[size] = "\0";

int main()
{
using namespace std;

    stringbuf strbuf;
    strbuf.pubsetbuf('\0', size);
    strbuf.sputn("Metrowerks CodeWarrior",50);
    strbuf.sgetn(temp, 50);
    cout << temp;

    return 0;
}
```

Result:

Metrowerks CodeWarrior

basic_streambuf::pubseekoff

Determines the position of the get pointer.

Prototype `pos_type pubseekoff
(off_type off,
ios_base::seekdir way, ios_base::openmode
which = ios_base::in | ios_base::out);`

Remarks The member function `pubseekoff()` is used to find the difference in bytes of the get pointer from a known position (such as the beginning or end of a stream). The function `pubseekoff()` returns a type `pos_type` which holds all the necessary information.

Return A `pos_type` via `seekoff(off, way, which)`

See Also `pubseekpos()`

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

Listing 17.2 Example of `basic_streambuf::pubseekoff()` usage:

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char inFile[] = "MW Reference";

int main()
{
using namespace std;

ifstream inOut(inFile, ios::in | ios::out);
if(!inOut.is_open())
    {cout << "Could not open file"; exit(1);}
ostream Out(inOut.rdbuf());

char str[] = "\nRegistered Trademark";

inOut.rdbuf()->pubseekoff(0, ios::end);

Out << str;

inOut.close();
return 0;
}
```

Result:

The File now reads:
Metrowerks CodeWarrior "Software at Work"
Registered Trademark

basic_streambuf::pubseekpos

Determine and move to a desired offset.

Prototype `pos_type pubseekpos
 (pos_type sp,`

```
ios_base::openmode which = ios::in | ios::out);
```

- Remarks** The function `pubseekpos()` is used to move to a desired offset using a type `pos_type`, which holds all necessary information.
- Return** A `pos_type` via `seekpos(sb, which)`
- See Also** `pubseekoff()`, `seekoff()`

Listing 17.3 Example of `streambuf::pubseekpos()` usage:

The file MW Reference contains:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
using namespace std;

ifstream in("MW Reference");
if(!in.is_open())
    {cout << "could not open file"; exit(1);}

streampos spEnd(0), spStart(0), aCheck(0);
spEnd = spStart = 5;

aCheck = in.rdbuf()->pubseekpos(spStart, ios::in);
cout << "The offset at the start of the reading"
    << " in bytes is "
    << static_cast<streamoff>(aCheck) << endl;

char ch;
while(spEnd != spStart+10)
{
    in.get(ch);
    cout << ch;
    spEnd = in.rdbuf()->pubseekoff(0, ios::cur);
}

aCheck = in.rdbuf()->pubseekoff(0,ios::cur);
```

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

```
cout << "\nThe final position's offset"
     << " in bytes now is "
     << static_cast<streamoff>(aCheck) << endl;

in.close();

return 0;
}
```

Result:

```
The offset for the start of the reading in bytes is 5
FGHIJKLMNOP
The final position's offset in bytes now is 15
```

basic_streambuf::pubsync

To synchronize the `streambuf` object with its input/output.

Prototype `int pubsync();`

Remarks The function `pubsync()` will attempt to synchronize the `streambuf` input and output.

Return Zero if successful or EOF if not via `sync()`.

Listing 17.4 Example of `streambuf::pubsync()` usage:

```
#include <iostream>

struct address {
    int number;
    char street[40];
}addbook;

int main()
{
using namespace std;

    cout << "Enter your street number: ";
    cin >> addbook.number;

    cin.rdbuf()->pubsync(); // buffer flush
```

```
cout << "Enter your street name: ";
cin.get(addbook.street, 40);

cout << "Your address is: "
    << addbook.number << " " << addbook.street;

return 0;
}
```

Result:

```
Enter your street number: 2201
Enter your street name: Donley Drive
Your address is: 2201 Donley Drive
```

27.5.2.2.3 Get Area

Public functions for retrieving input from a buffer.

basic_streambuf::in_avail

To test for availability of input stream.

Prototype `streamsize in_avail();`

Return If a read is permitted returns size of stream as a type `streamsize`.

basic_streambuf::snextc

To retrieve the next character in a stream.

Prototype `int_type snextc();`

Remarks The function `snextc()` calls `sbumpc()` to extract the next character in a stream. After the operation, the get pointer references the character following the last character extracted.

Return If `sbumpc` returns `traits::eof` returns that, otherwise returns `sgetc()`.

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

Listing 17.5 Example of `streambuf::snextc()` usage:

```
#include <iostream>
#include <sstream>

const int size = 100;

int main()
{
using namespace std;

    stringbuf strbuf;
    strbuf.pubsetbuf('\0', size);
    strbuf.sputn("ABCDE", 50);

    char ch;
        // look ahead at the next character
    ch = strbuf.snextc();
    cout << ch;
        // get pointer was not returned after peeking
    ch = strbuf.snextc();
    cout << ch;

    return 0;
}
```

Result:

BC

basic_streambuf::sbumpc

To move the get pointer.

Prototype `int_type sbumpc();`

Remarks The function `sbumpc()` moves the get pointer one element when called.

Return The value of the character at the `get` pointer. It returns `uflow()` if it fails to move the pointer.

See Also `sgetc()`

Listing 17.6 Example of `streambuf::sbumpc()` usage:

```
#include <iostream>
#include <sstream>

const int size = 100;
std::string buf = "Metrowerks CodeWarrior --Software at Work--";

int main()
{
using namespace std;

    stringbuf strbuf(buf);

    int ch;
    for (int i = 0; i < 23; i++)
    {
        ch = strbuf.sgetc();
        strbuf.sbumpc();
        cout.put(ch);
    }
    cout << endl;
    cout << strbuf.str() << endl;
    return 0;
}
```

Result:

```
Metrowerks CodeWarrior
Metrowerks CodeWarrior --Software at Work--
```

`basic_streambuf::sgetc`

To extract a character from the stream.

Prototype `int_type sgetc();`

Remarks The function `sgetc()` extracts a single character, without moving the get pointer.

Return A `int_type` type at the get pointer if available otherwise returns `underflow()`.

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

Listing 17.7 Example of `streambuf::sgetc()` usage:

See `streambuf::sbumpc()`

basic_streambuf::sgetn

To extract a series of characters from the stream.

Prototype `streamsize sgetn(char_type *s, streamsize n);`

Remarks The public member function `sgetn()` is used to extract a series of characters from the stream buffer. After the operation, the `get` pointer references the character following the last character extracted.

Return A `streamsize` type as returned from the function `xsgetn(s,n)`.

Listing 17.8 Example of `streambuf::sgetn()` usage:

See `pubsetbuf()`

27.5.2.2.4 Putback

Public functions to return a value to a stream.

basic_streambuf::sputback

To put a character back into the stream.

Prototype `int_type sputback(char_type c);`

Remarks The function `sputbackc()` will replace a character extracted from the stream with another character. The results are not assured if the putback is not immediately done or a different character is used.

Return If successful returns a pointer to the `get` pointer as an `int_type` otherwise returns `pbackfail(c)`.

Listing 17.9 Example of `streambuf::sputbackc()` usage:

```
#include <iostream>
#include <sstream>

std::string buffer = "ABCDEF";
```

```
int main()
{
using namespace std;

    stringbuf strbuf(buffer);
    char ch;

    ch = strbuf.sgetc(); // extract first character
    cout << ch;          // show it

        //get the next character
    ch = strbuf.snnextc();

    // if second char is B replace first char with x
    if(ch == 'B') strbuf.sputbackc('x');

        // read the first character now x
    cout << (char)strbuf.sgetc();

    strbuf.sbumpc();      // increment get pointer
        // read second character
    cout << (char)strbuf.sgetc();

    strbuf.sbumpc();      // increment get pointer
        // read third character
    cout << (char)strbuf.sgetc();

        // show the new stream after alteration
    strbuf.pubseekoff(0, ios::beg);
    cout << endl;

    cout << (char)strbuf.sgetc();

    while( (ch = strbuf.snnextc()) != EOF)
        cout << ch;

    return 0;
}
```

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

Result:

AxB
xCDEF

basic_streambuf::sungetc

To restore a character extracted.

Prototype `int_type sungetc();`

Remarks The function `sungetc()` restores the previously extracted character. After the operation, the get pointer references the last character extracted.

Return If successful returns a pointer to the `get` pointer as an `int_type` otherwise returns `pbackfail(c)`.

Listing 17.10 Example of `streambuf::sungetc()` usage:

See: `streambuf::sputbackc()`

27.5.2.2.5 Put Area

Public functions for inputting characters into a buffer.

basic_streambuf::sputc

To insert a character in the stream.

Prototype `int_type sputc(char_type c);`

Remarks The function `sputc()` inserts a character into the stream. After the operation, the get pointer references the character following the last character extracted.

Return If successful returns `c` as an `int_type` otherwise returns `overflow(c)`.

Listing 17.11 Example of `streambuf::sputc()` usage:

```
#include <iostream>
#include <sstream>
```

```
int main()
{
using namespace std;

    stringbuf strbuf;
    strbuf.sputc('A');

    char ch;
    ch = strbuf.sgetc();
    cout << ch;

    return 0;
}
```

Result:

A

basic_streambuf::sputn

To insert a series of characters into a stream.

Prototype `int_type sputn(char_type *s, streamsize n);`

Remarks The function `sputn()` inserts a series of characters into a stream. After the operation, the get pointer references the character following the last character extracted.

Return A `streamsize` type returned from a call to `xputn(s, n)`.

27.5.2.3 basic_streambuf Protected Member Functions

Protected member functions that are used for stream buffer manipulations by the `basic_streambuf` class and derived classes from it.

27.5.2.3.1 Get Area Access

Member functions for extracting information from a stream.

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

basic_streambuf::eback

Retrieve the beginning pointer for stream input.

Prototype `char_type* eback() const;`

Return The beginning pointer.

basic_streambuf::gptr

Retrieve the next pointer for stream input.

Prototype `char_type* gptr() const;`

Return The next pointer.

basic_streambuf::egptr

Retrieve the end pointer for stream input.

Prototype `char_type* egptr() const;`

Return The end pointer.

basic_streambuf::gbump

Advances the next pointer for stream input.

Prototype `void gbump(int n);`

Remarks The function `gbump()` advances the input pointer by the value of the `int n` argument.

basic_streambuf::setg

To set the beginning, next and end pointers.

Prototype `void setg
 (char_type *gbeg,
 char_type *gnext,
 char_type *gend);`

Remarks After the call to `setg()` the `gbeg` pointer equals `eback()`, the `gnext` pointer equals `gptr()`, and the `gend` pointer equals `egptr()`.

27.5.2.3.2 Put Area Access

Protected member functions for stream output sequences.

`basic_streambuf::pbase`

To retrieve the beginning pointer for stream output.

Prototype `char_type* pbase() const;`

Return The beginning pointer.

`basic_streambuf::pptr`

To retrieve the next pointer for stream output.

Prototype `char_type* pptr() const;`

Return The next pointer.

`basic_streambuf::epptr`

To retrieve the end pointer for stream output.

Prototype `char_type* epptr() const;`

Return The end pointer.

`basic_streambuf::pbump`

To advance the next pointer for stream output.

Prototype `void pbump(int n);`

Remarks The function `pbump()` advances the next pointer by the value of the `int` argument `n`.

`basic_streambuf::setp`

To set the values for the beginning, next and end pointers.

Prototype `void setp
 (char_type* pbeg,
 char_type* pend);`

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

Remarks After the call to `setp()`, `pbeg` equals `pbase()`, `pbeg` equals `pptr()` and `pend` equals `eptr()`.

27.5.2.4 `basic_streambuf` Virtual Functions

The virtual functions in `basic_streambuf` class are to be overloaded in any derived class.

27.5.2.4.1 Locales

To get and set the stream locale. These functions should be overridden in derived classes.

`basic_streambuf::imbue`

To change any translations base on locale.

Prototype `virtual void imbue(const locale &loc);`

Remarks The `imbue()` function allows the derived class to be informed in changes of locale and to cache results of calls to locale functions.

27.5.2.4.2 Buffer Management and Positioning

Virtual functions for positioning and manipulating the stream buffer. These functions should be overridden in derived classes.

`basic_streambuf::setbuf`

To set a buffer for stream input and output sequences.

Prototype `virtual basic_streambuf<char_type, traits> * setbuf(char_type* s, streamsize n);`

Remarks The function `setbuf()` is overridden in `basic_stringbuf` and `basic_filebuf` classes.

Return The `this` pointer.

basic_streambuf::seekoff

To return an offset of the current pointer in an input or output streams.

Prototype `virtual pos_type seekoff
 (off_type off,
 ios_base::seekdir way,
 ios_base::openmode which = ios::in | ios::out);`

Remarks The function `seekoff()` is overridden in `basic_stringbuf` and `basic_filebuf` classes.

Return A `pos_type` value, which is an invalid stream position.

basic_streambuf::seekpos

To alter an input or output stream position.

Prototype `virtual pos_type seekpos
 (pos_type sp,
 ios_base::openmode which = ios::in | ios::out);`

Remarks The function `seekpos()` is overridden in `basic_stringbuf` and `basic_filebuf` classes.

Return A `pos_type` value, which is an invalid stream position.

basic_streambuf::sync

To synchronize the controlled sequences in arrays.

Prototype `virtual int sync();`

Remarks If `pbase()` is non null the characters between `pbase()` and `pptr()` are written to the control sequence. The function `setbuf()` is overridden the `basic_filebuf` class.

Return Zero if successful and -1 if failure occurs.

27.5.2.4.3 Get Area

Virtual functions for extracting information from an input stream buffer. These functions should be overridden in derived classes.

Stream Buffers

27.5.2 Template class `basic_streambuf<charT, traits>`

`basic_streambuf::showmanc`

Shows how many characters in an input stream

Prototype `virtual int showmanc() ;`

Remarks If the function `showmanc()` returns a positive value then calls to `underflow()` will succeed. If `showmanc()` returns a negative number any calls to the functions `underflow()` and `uflow()` will fail.

Return Zero for normal behavior and negative or positive one.

`basic_streambuf::xsgetn`

To read a number of characters from and input stream buffer.

Prototype `virtual streamsize xsgetn
 (char_type *s, streamsize n);`

Remarks The characters are read by repeated calls to `sbumpc()` until either `n` characters have been assigned or `EOF` is encountered.

Return The number of characters read.

`basic_streambuf::underflow`

To show an underflow condition and not increment the get pointer.

Prototype `virtual int_type underflow();`

Remarks The function `underflow()` is called when a character is not available for `sgetc()`.

There are many constraints for `underflow()`.

- The pending sequence of characters is a concatenation of end pointer minus the get pointer plus some sequence of characters to be read from input.
- The result character if the sequence is not empty the first character in the sequence or the next character in the sequence.
- The backup sequence if the beginning pointer is null, the sequence is empty, otherwise the sequence is the get pointer minus the beginning pointer.

Return The first character of the pending sequence and does not increment the get pointer. If the position is `null` returns `traits::eof()` to indicate failure.

basic_streambuf::uflow

To show a underflow condition for a single character and increment the get pointer.

Prototype `virtual int_type uflow();`

Remarks The function `uflow()` is called when a character is not available for `sbumpc()`.

The constraints are the same as `underflow()`, with the exceptions that the resultant character is transferred from the pending sequence to the back up sequence and the pending sequence may not be empty.

Return Calls `underflow()` and if `traits::eof` is not returned returns the integer value of the get pointer and increments the next pointer for input.

27.5.2.4.4 Putback

Virtual functions for replacing data to a stream. These functions should be overridden in derived classes.

basic_streambuf::pbackfail

To show a failure in a put back operation.

Prototype `virtual int_type pbackfail
(int_type c = traits::eof());`

Remarks The resulting conditions are the same as the function `underflow()`.

Return The function `pbackfail()` is only called when a put back operation really has failed and returns `traits::eof`. If success occurs the return is undefined.

27.5.2.4.5 Put Area

Virtual function for inserting data into an output stream buffer.
These functions should be overridden in derived classes.

basic_streambuf::xsputn

Write a number of characters to an output buffer.

Prototype `virtual streamsize xsputn
 (const char_type *s, streamsize n);`

Remarks The function `xsputn()` writes to the output character by using repeated calls to `sputc(c)`. Write stops when `n` characters have been written or `EOF` is encountered.

Return The number of characters written in a type `streamsize`.

basic_streambuf::overflow

Consumes the pending characters of an output sequence.

Prototype `virtual int_type overflow
 (int_type c = traits::eof());`

Remarks The pending sequence is defined as the concatenation of the `put` pointer minus the `beginning` pointer plus either the sequence of characters or an empty sequence, unless the `beginning` pointer is null in which case the pending sequence is an empty sequence.

This function is called by `sputc()` and `sputn()` when the buffer is not large enough to hold the output sequence.

Overriding this function requires that:

- When overridden by a derived class how characters are consumed must be specified.
- After the overflow either the `beginning` pointer must be null or the `beginning` and `put` pointer must both be set to the same non-null value.
- The function may fail if appending characters to an output stream fails or failure to set the previous requirement occurs.

Return The function returns `traits::eof()` for failure or some unspecified result to indicate success.

Stream Buffers

27.5.2 *Template class basic_streambuf<charT, traits>*

Formatting and Manipulators

This chapter discusses formatting and manipulators in the input/output library.

The Formatting and Manipulators Library (clause 27.6)

There are three headers—`<iostream>`, `<ostream>`, and `<iomanip>`—that contain stream formatting and manipulator routines and implementations.

The chapter is constructed in the following sub sections and mirrors clause 27.6 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Headers” on page 533](#)
- [“27.6.1 Input Streams” on page 535](#)
- [“27.6.2 Output streams” on page 566](#)
- [“27.6.3 Standard manipulators” on page 588](#)

Headers

This section lists the header for `istream`, `ostream`, and `iomanip`.

Formatting and Manipulators

Headers

Header <iostream>

Prototype

```
#include <iostream>
namespace std{
template
<class charT, class traits = ios_traits<charT> >
class basic_istream;

typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

template
<class charT, class traits>
basic_istream<charT, traits> &ws
(basic_istream<charT, traits> (is));
}
```

Header <ostream>

```
#include <iostream>
namespace std{
template
<class charT, class traits = ios_traits<charT> >
class basic_ostream;

typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;

template
<class charT, class traits>
basic_ostream<charT, traits> &endl
(basic_ostream<charT, traits>& os);

template
<class charT, class traits>
basic_ostream<charT, traits> &ends
(basic_ostream<charT, traits>& os);

template
<class charT, class traits>
```

```
basic_ostream<charT, traits> &flush  
    (basic_ostream<charT, traits>& os);  
}
```

Header <iomanip>

```
#include <ios>  
namespace std {  
// return types are unspecified  
T1 resetiosflags(ios_base::fmtflags mask);  
T2 setiosflags (ios_base::fmtflag mask);  
T3 setbase(int base);  
T4 setfill(int c);  
T5 setprecision(int n);  
T6 setw(int n);  
}
```

27.6.1 Input Streams

The header <iostream> controls input from a stream buffer.

The topics in this section are:

- [“27.6.1.1 Template class basic_istream” on page 536](#)
- [“27.6.1.1.1 basic_istream Constructors” on page 538](#)
- [“27.6.1.1.2 Class basic_istream::sentry” on page 539](#)
- [“27.6.1.2 Formatted input functions” on page 540](#)
- [“27.6.1.2.1 Common requirements” on page 540](#)
- [“27.6.1.2.2 Arithmetic Extractors Operator >>” on page 541](#)
- [“27.6.1.2.3 basic_istream extractor operator >>” on page 542](#)
- [“27.6.1.3 Unformatted input functions” on page 547](#)
- [“27.6.1.4 Standard basic_istream manipulators” on page 564](#)
- [“27.6.1.4.1 basic_iostream Constructor” on page 566](#)

27.6.1.1 Template class `basic_istream`

A class that defines several functions for stream input mechanisms from a controlled stream buffer.

```
namespace std{
template <class charT, class traits = ios_traits<charT> >
class basic_istream : virtual public basic_ios<charT, traits> {
public:
    typedef charT
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    explicit basic_istream
    (basic_streambuf<charT, traits>* sb);

    virtual ~basic_istream();

    class sentry;

    basic_istream<charT, traits>& operator >>
        (basic_istream<charT, traits>& (*pf)
         (basic_istream<charT, traits>&))
    basic_istream<charT, traits>& operator >>
        (basic_ios<charT, traits>& (*pf)
         (basic_ios<charT, traits>&))
    basic_istream<charT, traits>& operator >>
        (char_type *s);
    basic_istream<charT, traits>& operator >>
        (char_type& c);
    basic_istream<charT, traits>& operator >>
        (bool& n);
    basic_istream<charT, traits>& operator >>
        (short& n);
    basic_istream<charT, traits>& operator >>
        (unsigned short& n);
    basic_istream<charT, traits>& operator >>
        (int& n);
    basic_istream<charT, traits>& operator >>
        (unsigned int& n);
```

```
basic_istream<charT, traits>& operator >>
    ( long& n);
basic_istream<charT, traits>& operator >>
    (unsigned long& n);
basic_istream<charT, traits>& operator >>
    (float& f);
basic_istream<charT, traits>& operator >>
    (double& f);
basic_istream<charT, traits>& operator >>
    (long double & f);
basic_istream<charT, traits>& operator >>
    (void*& p);
basic_istream<charT, traits>& operator >>
    (basic_streambuf<char_type, traits>* sb);

streamsize gcount() const;
int_type get();
basic_istream<charT, traits>& get
    (char_type& c);
basic_istream<charT, traits>& get
    (char_type* s,
     streamsize n,
     char_type delim = traits::newline());
basic_istream<charT, traits>& get
    (basic_streambuf<char_type,
     traits>& sb,
     char_type delim = traits::newline());

basic_istream<charT, traits>& getline
    (char_type* s,
     streamsize n,
     char_type delim = traits::newline());

basic_istream<charT, traits>& ignore
    (streamsize n = 1,
     int_type delim = traits::eof());

int_type peek();

basic_istream<charT, traits>& read
    (char_type* s, streamsize n);
streamsize readsome(charT_type* s, streamsize n);
```

Formatting and Manipulators

27.6.1 Input Streams

```
basic_istream<charT, traits>& putback(char_type c);
basic_istream<charT, traits>& unget();

int sync();

pos_type tellg();
basic_istream<charT, traits>& seekg
    (pos_type);
basic_istream<charT, traits>& seekg
    (off_type, ios_base::seekdir);
};

}
```

Remarks The `basic_istream` class is derived from the `basic_ios` class and provides many functions for input operations.

27.6.1.1 `basic_istream` Constructors

constructor

Creates an `basic_istream` object.

Prototype `explicit basic_istream
(basic_streambuf<charT, traits>* sb);`

Remarks The `basic_istream` constructor is overloaded. It can be created as a base class with no arguments. It may be a simple input class initialized to a previous object's stream buffer.

Destructor

Destroy the `basic_istream` object.

Prototype `virtual ~basic_istream()`

Remarks The `basic_istream` destructor removes from memory the `basic_istream` object.

Listing 18.1 Example of `basic_istream()` usage:

MW Reference file contains
Ask the teacher anything you want to know

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
using namespace std;

ofstream out("MW Reference", ios::out | ios::in);
if(!out.is_open())
{cout << "file did not open"; exit(1);}

istream inOut(out.rdbuf());

char c;
while(inOut.get(c)) cout.put(c);

return 0;
}
```

Result:

Ask the teacher anything you want to know

27.6.1.1.2 Class basic_istream::sentry

A class for exception safe prefix and suffix operations.

Prototype namespace std {
 template
 <class charT,
 class traits = char_traits<charT> >
 class basic_istream<charT, traits>::sentry {
 bool ok_;
 public:
 explicit sentry
 (basic_istream<charT,
 traits>& is,
 bool noskipws = false);
 ~sentry();
 operator bool() {return ok_;}
 };

{}

Class `basic_istream::sentry` Constructor

Constructor

Prepare for formatted or unformatted input

Prototype `explicit sentry
 (basic_istream<charT,
 traits>& is,
 bool noskipws = false);`

Remarks If after the operation `is.good()` is true `ok_` equals `true` otherwise `ok_` equals `false`. The constructor may call `setstate(failbit)` which may throw an exception.

Destructor

Prototype `~sentry();`

Remarks The destructor has no effects.

`sentry::operator bool`

To return the value of the data member `ok_`.

Prototype `operator bool();`

Return Operator `bool` returns the value of `ok_`

27.6.1.2 Formatted input functions

Formatted function provide mechanisms for input operations of specific types.

27.6.1.2.1 Common requirements

Each formatted input function begins by calling `ipfx()` and if the scan fails for any reason calls `setstate(failbit)`. The behavior of the scan functions are “as if” it was `fscanf()`.

27.6.1.2.2 Arithmetic Extractors Operator >>

Extractors that provide formatted arithmetic input operation.

Prototype

```
basic_istream<charT, traits>& operator >>
    (bool & n);
basic_istream<charT, traits>& operator >>
    (short &n);
```

Remarks: Extracts a short integer value and stores it in *n*.

```
basic_istream<charT, traits>& operator >>
    (unsigned short & n);
basic_istream<charT, traits>& operator >>
    (int & n);
basic_istream<charT, traits>& operator >>
    (unsigned int &n);
basic_istream<charT, traits>& operator >>
    (long & n);
basic_istream<charT, traits>& operator >>
    (unsigned long & n);
basic_istream<charT, traits>& operator >>
    (float & f);
basic_istream<charT, traits>& operator >>
    (double& f);
basic_istream<charT, traits>& operator >>(
    long double& f);
```

Remarks

The Arithmetic extractors extract a specific type from the input stream and store it in the address provided

Table 18.1 States and stdio equivalents

| state | stdio equivalent |
|------------------------------|------------------|
| (flags() & basefield) == oct | %o |
| (flags() & basefield) == hex | %x |
| (flags() & basefield) != 0 | %x |
| (flags() & basefield) == 0 | %i |
| Otherwise | |

Formatting and Manipulators

27.6.1 Input Streams

| state | stdio equivalent |
|------------------------|------------------|
| signed integral type | %d |
| unsigned integral type | %u |

27.6.1.2.3 basic_istream extractor operator >>

Extracts characters or sequences of characters and converts if necessary to numerical data.

Prototype `basic_istream<charT, traits>& operator >>
 basic_istream<charT, traits>& (*pf)
 (basic_istream<charT, traits>&))`

Remarks Returns `pf(*this)`.

```
basic_istream<charT, traits>& operator >>  
                    (basic_ios<charT, traits>& (*pf)  
                            (basic_ios<charT, traits>&))
```

Remarks Calls `pf(*this)` then returns `*this`.

```
basic_istream<charT, traits>& operator >>  
                    (char_type *s);
```

Remarks Extracts a char array and stores it in `s` if possible otherwise call `setstate(failbit)`. If `width()` is set greater than zero `width()-1` elements are extracted else up to size of `s-1` elements are extracted. Scan stops with a whitespace “as if” in `fscanf()`.

```
basic_istream<charT, traits>& operator >>  
                    (char_type& c);
```

Remarks Extracts a single character and stores it in `c` if possible otherwise call `setstate(failbit)`.

```
basic_istream<charT, traits>& operator >>  
                    (void*& p);
```

Remarks Converts a pointer to void and stores it in `p`.

```
basic_istream<charT, traits>& operator >>  
                    (basic_streambuf<char_type, traits>* sb);
```

Remarks Extracts a `basic_streambuf` type and stores it in `sb` if possible otherwise call `setstate(failbit)`.

Remarks The various overloaded extractors are used to obtain formatted input dependent upon the type of the argument. Since they return a reference to the calling stream they may be chained in a series of

extractions. The overloaded extractors work “as if” like `fscanf()` in standard C and read until a white space character or EOF is encountered.

NOTE The white space character is not extracted and is not discarded, but simply ignored. Be careful when mixing unformatted input operations with the formatted extractor operators. Such as when using console input.

Return The `this` pointer is returned.

See Also `basic_ostream::operator <<`

Listing 18.2 Example of `basic_istream::extractor` usage:

The MW Reference input file contains
`float 33.33 double 3.16e+10 Integer 789 character C`

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char ioFile[81] = "MW Reference";

int main()
{
using namespace std;

ifstream in(ioFile);
if(!in.is_open())
{cout << "cannot open file for input"; exit(1);}

char type[20];
double d;
int i;
char ch;

in    >> type >> d;
cout << type << " " << d << endl;
in    >> type >> d;
cout << type << " " << d << endl;
in    >> type >> i;
```

Formatting and Manipulators

27.6.1 Input Streams

```
cout << type << " " << i << endl;
in    >> type >> ch;
cout << type << " " << ch << endl;

cout << "\nEnter an integer: ";
cin >> i;
cout << "Enter a word: ";
cin >> type;
cout << "Enter a character \
      << "then a space then a double: ";
cin >> ch >> d;

cout << i << " " << type << " "
      << ch << " " << d << endl;

in.close();

return 0;
}
```

Result:

```
float 33.33
double 3.16e+10
Integer 789
character C
```

```
Enter an integer: 123 <enter>
Enter a word: Metrowerks <enter>
Enter a character then a space then a double: a 12.34 <enter>
123 Metrowerks a 12.34
```

Overloading Extractors:

To provide custom formatted data retrieval.

```
Prototype extractor prototype
Basic_istream &operator >>
(basic_istream &s,const imanip<T>&)
{
    // procedures
    return s;
}
```

Remarks You may overload the `extractor` operator to tailor the specific needs of a particular class.

Return The `this` pointer is returned.

Listing 18.3 Example of `basic_istream` overloaded extractor usage:

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

class phonebook {
    friend std::ostream &operator<<(std::ostream &stream,
        phonebook o);
    friend std::istream &operator>>(std::istream &stream,
        phonebook &o);

private:
    char name[80];
    int areacode;
    int exchange;
    int num;

public:
    void putname() {std::cout << num;}
    phonebook() {} // default constructor
    phonebook(char *n, int a, int p, int nm)
        {std::strcpy(name, n); areacode = a;
         exchange = p; num = nm;}
};

int main()
{
using namespace std;
    phonebook a;

    cin >> a;
    cout << a;

    return 0;
}
```

Formatting and Manipulators

27.6.1 Input Streams

```
std::ostream &operator<<(std::ostream &stream, phonebook o)
{
using namespace std;

    stream << o.name << " ";
    stream << "(" << o.areacode << " ) ";
    stream << o.exchange << "-";
    cout << setfill('0') << setw(4) << o.num << "\n";
    return stream;
}

std::istream &operator>>(std::istream &stream, phonebook &o)
{
using namespace std;

    char buf[5];
    cout << "Enter the name: ";
    stream >> o.name;
    cout << "Enter the area code: ";
    stream >> o.areacode;
    cout << "Enter exchange: ";
    stream >> o.exchange;
    cout << "Enter number: ";
    stream >> buf;
    o.num = atoi(buf);
    cout << "\n";
    return stream;
}
```

Result:

```
Enter the name: Metrowerks
Enter the area code: 512
Enter exchange: 873
Enter number: 4700
```

```
Metrowerks (512) 873-4700
```

27.6.1.3 Unformatted input functions

The various unformatted input functions all begin by construction an object of type `basic_istream::sentry` and ends by destroying the `sentry` object.

NOTE Older versions of the library may begin by calling `ipfx()` and end by calling `isfx()` and returning the value specified.

basic_istream::gcount

To obtain the number of bytes read.

Prototype `streamsize gcount() const;`

Remarks Use the function `gcount()` to obtain the number of bytes read by the last unformatted input function called by that object.

Return An `int` type count of the bytes read.

Listing 18.4 Example of basic_istream::gcount() usage:

```
#include <iostream>
#include <fstream>

const SIZE = 4;

struct stArray {
    int index;
    double dNum;
};

int main()
{
    using namespace std;

    ofstream fOut("test");
    if(!fOut.is_open())
        {cout << "can't open out file"; return 1;}

    stArray arr;
    short i;
```

Formatting and Manipulators

27.6.1 Input Streams

```
for(i = 1; i < SIZE+1; i++)
{
    arr.index = i;
    arr.dNum = i *3.14;
    fOut.write((char *) &arr, sizeof(stArray));
}
fOut.close();

stArray aIn[SIZE];

ifstream fIn("test");
if(!fIn.is_open())
    {cout << "can't open in file"; return 2;}

long count =0;
for(i = 0; i < SIZE; i++)
{    fIn.read((char *) &aIn[i], sizeof(stArray));

count+=fIn.gcount();
}

cout << count << " bytes read " << endl;
cout << "The size of the structure is "
    << sizeof(stArray) << endl;
for(i = 0; i < SIZE; i++)
cout << aIn[i].index << " " << aIn[i].dNum
    << endl;

fIn.close();

return 0;
}
```

Result:

```
48 bytes read
The size of the structure is 12
1 3.14
2 6.28
3 9.42
4 12.56
```

basic_istream::get

Overloaded functions to retrieve a `char` or a `char` sequence from an input stream.

Prototype

```
int_type get();
```

Remarks Extracts a character if available and returns that value. Else, calls `setstate(failbit)` and returns `eof()`.
`basic_istream<charT, traits>& get(char_type& c);`

Remarks Extracts a character and assigns it to `c` if possible else calls `setstate(failbit)`.

```
basic_istream<charT, traits>& get
    (char_type* s,
     streamsize n,
     char_type delim = traits::newline());
```

Remarks Extracts characters and stores them in a `char` array at an address pointed to by `s`, until

- A limit (the second argument minus one) or the number of characters to be stored is reached
- A delimiter (the default value is the newline character) is met. In which case, the delimiter is not extracted.
- If `end_of_file` is encountered in which case `setstate(eofbit)` is called.

If no characters are extracted calls `setstate(failbit)`. In any case it stores a `null` character in the next available location of array `s`.

```
basic_istream<charT, traits>& get
    (basic_streambuf<char_type,
     traits>& sb,
     char_type delim = traits::newline());
```

Remarks Extracts characters and assigns them to the `basic_streambuf` object `sb` if possible or else it calls `setstate(failbit)`. Extraction stops if...

- an insertion fails
- `end-of-file` is encountered.
- an exception is thrown
- the next available character `c == delim` (in which case `c` is not extracted.)

Formatting and Manipulators

27.6.1 Input Streams

Return An integer when used with no argument. When used with an argument if a character is extracted the `get()` function returns the `this` pointer. If no character is extracted `setstate(failbit)` is called. In any case a null char is appended to the array.

See Also `getline()`

Listing 18.5 Example of basic_istream::get() usage:

READ ONE CHARACTER:

MW Reference file for input
float 33.33 double 3.16e+10 Integer 789 character C

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
using namespace std;

char inFile[] = "MW Reference";
ifstream in(inFile);
if(!in.is_open())
{cout << "Cannot open input file"; exit(1);}

char ch;
while(in.get(ch)) cout << ch;

return 0;
}
```

Result:

float 33.33 double 3.16e+10 Integer 789 character C

READ ONE LINE:

```
#include <iostream>

const int size = 100;
char buf[size];
```

```
int main()
{
using namespace std;

    cout << " Enter your name: ";
    cin.get(buf, size);
    cout << buf;

    return 0;
}
```

Result:

```
Enter your name: Metrowerks CodeWarrior <enter>
Metrowerks CodeWarrior
```

basic_istream::getline

To obtain a delimiter terminated character sequence from an input stream.

Prototype `basic_istream<charT, traits>& getline
 (char_type* s,
 streamsiz n,
 char_type delim = traits::newline());`

Remarks The unformatted `getline()` function retrieves character input, and stores it in a character array buffer `s` if possible until the following conditions evaluated in this order occur. If no characters are extracted `setstate(failbit)` is called.

- `end-of-file` occurs in which case `setstate(eofbit)` is called.
- A delimiter (default value is the newline character) is encountered. In which case the delimiter is read and extracted but not stored.
- A limit (the second argument minus one) is read.
- if `n-1` chars are read that `failbit` gets set.

In any case it stores a null char into the next successive location of the array.

Return The `this` pointer is returned.

Formatting and Manipulators

27.6.1 Input Streams

See Also `basic_ostream::flush()`

Listing 18.6 Example of `basic_istream::getline()` usage:

```
#include <iostream>

const int size = 120;
int main()
{
using namespace std;

char compiler[size];

cout << "Enter your compiler: ";
cin.getline(compiler, size);

cout << "You use " << compiler;

return 0;
}
```

Result:

```
Enter your compiler:Metrowerks CodeWarrior <enter>
You use Metrowerks CodeWarrior
```

```
#include <iostream>

const int size = 120;
#define TAB '\t'

int main()
{
using namespace std;

cout << "What kind of Compiler do you use: ";
char compiler[size];

cin.getline(compiler, size,TAB);
cout << compiler;
cout << "\nsecond input not needed\n";
cin >> compiler;
cout << compiler;
```

```
    return 0;
}

Result:
What kind of Compiler do you use:
Metrowerks CodeWarrior<tab>Why?
Metrowerks CodeWarrior
second input not needed
Why?
```

basic_istream::ignore

To extract and discard a number of characters.

Prototype `basic_istream<charT, traits>& ignore
 (steamsize n = 1,
 int_type delim = traits::eof());`

Remarks The function `ignore()` will extract and discard characters until

- A limit is met (the first argument)
- end-of-file is encountered (in which case `setstate(eofbit)` is called.)
- The next character `c` is equal to the delimiter `delim`, in which case it is extracted except when `c` is equal to `traits::eof()`;

Return The `this` pointer is returned.

Listing 18.7 Example of basic_istream::ignore() usage:

The file MW Reference contains:

```
char ch; // to save char
/*This C comment will remain */
while((ch = in.get())!= EOF) cout.put(ch);
// read until failure
/* the C++ comments won't */

#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";
```

Formatting and Manipulators

27.6.1 Input Streams

```
char bslash = '/' ;

int main()
{
using namespace std;

    ifstream in(inFile);
    if(!in.is_open())
        {cout << "file not opened"; exit(1);}

    char ch;
    while((ch = in.get()) != EOF)
    {
        if(ch == bslash && in.peek() == bslash)
        {
            in.ignore(100, '\n');
            cout << '\n';
        }
        else        cout << ch;
    }

    return 0;
}
```

Result:

```
char ch;
    /*This C comment will remain */
while((ch = in.get())!= EOF) cout.put(ch);

/* the C++ comments won't */
```

basic_istream::peek

To view at the next character to be extracted.

Prototype `int_type peek();`

Remarks The function `peek()` allows you to look ahead at the next character in a stream to be extracted without extracting it.

Return If `good()` is false returns `traits::eof()` else returns the value of the next character in the stream.

Listing 18.8 Example of basic_istream::peek() usage:

See `basic_istream::ignore()`

basic_istream::read

To obtain a block of binary data from and input stream.

Prototype `basic_istream<charT, traits>& read
(char_type* s, streamsize n);`

Remarks The function `read()` will attempt to extract a block of binary data until the following conditions are met.

- A limit of `n` number of characters are stored.
- `end-of-file` is encountered on the input (in which case `setstate(failbit)` is called).

Return The `this` pointer is returned.

See Also `write()`

Listing 18.9 Example of basic_istream::read() usage:

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

struct stock {
    char name[80];
    double price;
    long trades;
};

char *Exchange = "BBSE";
char *Company = "Big Bucks Inc.';

int main()
{
using namespace std;

    stock Opening, Closing;
```

Formatting and Manipulators

27.6.1 Input Streams

```
strcpy(Opening.name, Company);
Opening.price = 180.25;
Opening.trades = 581300;

    // open file for output
ofstream Market(Exchange, ios::out | ios::trunc |
ios::binary);
if(!Market.is_open())
{cout << "can't open file for output"; exit(1);}

Market.write((char*) &Opening, sizeof(stock));
Market.close();

    // open file for input
ifstream Market2(Exchange, ios::in | ios::binary);
if(!Market2.is_open())
{cout << "can't open file for input"; exit(2);}

Market2.read((char*) &Closing, sizeof(stock));

cout << Closing.name << "\n"
<< "The number of trades was: " << Closing.trades << '\n';
cout << fixed << setprecision(2)
<< "The closing price is: $" << Closing.price << endl;

Market2.close();

return 0;
}
```

Result:

```
Big Bucks Inc.
The number of trades was: 581300
The closing price is: $180.25
```

basic_istream::readsome

Extracts characters and stores them in an array.

Prototype streamsize readsome
 (charT_type* s, streamsize n);

| | |
|---------|--|
| Remarks | The function <code>readsome</code> extracts and stores characters storing them in the buffer pointed to by <code>s</code> until the following conditions are met. |
| | <ul style="list-style-type: none"> • end-of-file is encountered (in which case <code>setstate(eofbit)</code> is called.) • No characters are extracted. • A limit of characters is extracted either <code>n</code> or the size of the buffer. |
| Return | The number of characters extracted. |

Listing 18.10 Example of basic_istream::readsome() usage.

The file MW Reference contains:

Metrowerks CodeWarrior
Software at Work
Registered Trademark

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>

const short SIZE = 81;

int main()
{
using namespace std;

ifstream in("MW Reference");
if(!in.is_open())
{cout << "can't open file for input"; exit(1);}

char Buffer[SIZE] = "\0";
ostringstream Paragraph;

while(in.good() && (in.peek() != EOF))
{
    in.readsome(Buffer, 5);
    Paragraph << Buffer;
}

cout << Paragraph.str();
}
```

```
    in.close();
    return 0;
}
```

Result:

Metrowerks CodeWarrior
Software at Work
Registered Trademark

basic_istream::putback

To replace a previously extracted character.

Prototype `basic_istream<charT, traits>& putback(char_type c);`

Remarks The function `putback()` allows you to replace the last character extracted by calling `rdbuf() ->sungetc()`. If the buffer is empty, or if `sungetc()` returns `eof`, `setstate(failbit)` may be called.

Return The `this` pointer is returned.

See Also sungetc()

Listing 18.11 Example of basic istream::putback usage:

The file MW Reference contains..

```
char ch; // to save char
        /* comment will remain */
while((ch = in.get())!= EOF) cout.put(ch);
// read until failure
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
```

```
char inFile[] = "MW Reference";
char bslash = '/';
```

```
int main()
{
```

```
using namespace std;

ifstream in(inFile);
if(!in.is_open())
{cout << "file not opened"; exit(1);}

char ch, tmp;
while((ch = in.get()) != EOF)
{
    if(ch == bslash)
    {
        in.get(tmp);
        if(tmp != bslash)
            in.putback(tmp);
        else continue;
    }
    cout << ch;
}

return 0;
}
```

Result:

```
char ch;                      to save char
/* comment will remain */
while((ch = in.get())!= EOF) cout.put(ch);
read until failure
```

basic_istream::unget

To replace a previously extracted character.

Prototype `basic_istream<charT, traits>::unget();`

Remarks Use the function `unget()` to return the previously extracted character. If `rdbuf()` is null or if end-of-file is encountered `setstate(badbit)` is called.

Return The this pointer is returned.

See Also `putback()`, `ignore()`

Formatting and Manipulators

27.6.1 Input Streams

Listing 18.12 Example of basic_istream::unget() usage:

The file MW Reference contains:

```
char ch;                                // to save char
    /* comment will remain */
    // read until failure
while((ch = in.get()) != EOF) cout.put(ch);

#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";
char bslash = '/';

int main()
{
using namespace std;

ifstream in(inFile);
if(!in.is_open())
{cout << "file not opened"; exit(1);}

char ch, tmp;
while((ch = in.get()) != EOF)
{
if(ch == bslash)
{
    in.get(tmp);
    if(tmp != bslash)
        in.unget();
    else continue;
}
cout << ch;
}

return 0;
}
```

Result:

```
char ch;                                to save char
    /* comment will remain */
```

```
    read until failure
while((ch = in.get()) != EOF) cout.put(ch);
```

basic_istream::sync

To synchronize input and output

Prototype int sync();

Remarks This functions attempts to make the input source consistent with the stream being extracted.

If rdbuf() ->pubsync() returns -1 setstate(badbit) is called and traits::eof is returned.

Return If rdbuf() is Null returns -1 otherwise returns zero.

Listing 18.13 Example of basic_istream::sync() usage:

The file MW Reference contains:
This functions attempts to make the input source
consistent with the stream being extracted.
--
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
using namespace std;

    ifstream in(inFile);
    if(!in.is_open())
        {cout << "could not open file"; exit(1);}

    char str[10];
    if(in.sync())      // return 0 if successful
        { cout << "cannot sync"; exit(1); }
    while (in.good())
```

Formatting and Manipulators

27.6.1 Input Streams

```
{  
    in.get(str, 10, EOF);  
    cout <<str;  
}  
return 0;  
}
```

Result:

This functions attempts to make the input source
consistent with the stream being extracted.

--

Metrowerks CodeWarrior "Software at Work"

basic_istream::tellg

To determine the offset of the get pointer in a stream

Prototype pos_type tellg();

Remarks The function tellg calls rdbuf()->pubseekoff(0,
 cur, in).

Return The current offset as a pos_type if successful else returns -1.

See Also basic_streambuf::pubseekoff()

Listing 18.14 Example of basic_istream::tellg() usage:

See basic_istream::seekg()

basic_istream::seekg

To move to a variable position in a stream.

Prototype basic_istream<charT, traits>& seekg(pos_type);
 basic_istream<charT, traits>& seekg
 (off_type, ios_base::seekdir dir);

Remarks The function seekg is overloaded to take a pos_type object, or
 an off_type object (defined in basic_ios class.) The function
 is used to set the position of the get pointer of a stream to a
 random location for character extraction.

Return The this pointer is returned.

See Also `basic_streambuf::pubseekoff()` and `pubseekpos()`.

Listing 18.15 Example of `basic_istream::seekg()` usage:

The file MW Reference contains:

ABCDEFGHIJKLMNPQRSTUVWXYZ

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
using namespace std;

ifstream in("MW Reference");
if(!in.is_open())
{cout << "could not open file"; exit(1);}

// note streampos is typedef in iosfwd
streampos spEnd(5), spStart(5);

in.seekg(spStart);
streampos aCheck = in.tellg();
cout << "The offset at the start of the reading in bytes is "
<< aCheck << endl;

char ch;
while(spEnd != spStart+10)
{
    in.get(ch);
    cout << ch;
    spEnd = in.tellg();
}

aCheck = in.tellg();
cout << "\nThe current position's offset in bytes now is "
<< aCheck << endl;
streamoff gSet = 0;
in.seekg(gSet, ios::beg);

aCheck = in.tellg();
cout << "The final position's offset in bytes now is "
```

Formatting and Manipulators

27.6.1 Input Streams

```
<< aCheck << endl;

in.close();
return 0;
}
```

Result:

```
The offset at the start of the reading in bytes is 5
FGHIJKLMNO
The current position's offset in bytes now is 15
The final position's offset in bytes now is 0
```

27.6.1.4 Standard basic_istream manipulators

basic_ifstream::ws

To provide inline style formatting.

Prototype template<class charT, class traits> basic_istream<charT, traits> &ws(basic_istream<charT, traits>& is);

Remarks The ws manipulator skips whitespace characters in input.

Return The this pointer.

Listing 18.16 Example of basic_istream:: manipulator ws usage:

The file MW Reference (where the number of blanks (and/or tabs) is unknown) contains:

a b c

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
    char * inFileNam = "MW Reference";
```

```
ifstream in(inFileName);
if (!in.is_open())
{cout << "Couldn't open for input\n"; exit(1);}

char ch;
in.unsetf(ios::skipws);

cout << "Does not skip whitespace\n| ";
while (1)
{
    in >> ch; // does not skip white spaces
    if (in.good())
        cout << ch;
    else break;
}
cout << "|\n\n";

//reset file position
in.clear();
in.seekg(0, ios::beg);

cout << "Does skip whitespace\n| ";
while (1)
{
    in >> ws >> ch; // ignore white spaces

    if (in.good())
        cout << ch;
    else break;
}
cout << "|\n" << endl;

in.close();
return(0);
}
```

Result:

Does not skip whitespace
| a b c |

Does skip whitespace
|abc|

27.6.1.4.1 basic_iostream Constructor

Constructor

Constructs an and destroy object of the class basic_iostream.

Prototype `explicit basic_iostream
 (basic_streambuf<charT, traits>* (sb);`

Remarks Calls basic_istream(<charT, traits> (sb) and basic_ostream(charT, traits>* (sb). After it is constructed rdbuf() equals sb and gcount() equals zero.

Destructor

Prototype `virtual ~basic_iostream();`

Remarks Destroys an object of type basic_iostream.

27.6.2 Output streams

The include file <ostream> includes classes and types that provide output stream mechanisms.

The topics in this section are:

- [“27.6.2.1 Template class basic_ostream” on page 567](#)
- [“27.6.2.2 basic_ostream Constructor” on page 568](#)
- [“Class basic_ostream::sentry Constructor” on page 570](#)
- [“27.6.2.3 Class basic_ostream::sentry” on page 570](#)
- [“27.6.2.4 Formatted output functions” on page 571](#)
- [“27.6.2.4.1 Common requirements” on page 571](#)
- [“27.6.2.4.2 Arithmetic Inserter Operator <<” on page 571](#)
- [“27.6.2.4.3 basic_ostream::operator<<” on page 573](#)
- [“27.6.2.5 Unformatted output functions” on page 577](#)
- [“27.6.2.6 Standard basic_ostream manipulators” on page 584](#)

27.6.2.1 Template class `basic_ostream`

A class for stream output mechanisms.

Prototype

```
namespace std{
template
<class charT, class traits = ios_traits<charT> >
class basic_ostream : virtual public basic_ios<charT, traits>{
public:
// Types:
typedef charT    char_type;
typedef typename traits::int_type int_type;
typedef typename traits::pos_type pos_type;
typedef typename traits::off_type off_type;

explicit basic_ostream (basic_streambuf<char_type,
traits*>*sb);
virtual ~basic_ostream();

class sentry;

basic_ostream<charT, traits>& operator<<
(basic_ostream<charT, traits>&(*pf)
(basic_ostream<charT, traits>&));
basic_ostream<charT, traits>& operator<<
(basic_ostream<charT, traits>&(*pf)
(basic_ios<charT, traits>&));
basic_ostream<charT, traits>& operator<<
(const char_type *s)
basic_ostream<charT, traits>& operator<<
(char_type c)
basic_ostream<charT, traits>& operator<<
(bool n)
basic_ostream<charT, traits>& operator<<
(short n)
basic_ostream<charT, traits>& operator<<
(unsigned short n)
basic_ostream<charT, traits>& operator<<
(int n)
basic_ostream<charT, traits>& operator<<
(unsigned int n)
```

Formatting and Manipulators

27.6.2 Output streams

```
basic_ostream<charT, traits>& operator<<
    (long n)
basic_ostream<charT, traits>& operator<<
    (unsigned long n)
basic_ostream<charT, traits>& operator<<
    (float f)
basic_ostream<charT, traits>& operator<<
    (double f)
basic_ostream<charT, traits>& operator<<
    (long double f)
basic_ostream<charT, traits>& operator<<
    (void p)
basic_ostream<charT, traits>& operator<<
    (basic_streambuf>char_type, traits>* sb);)

basic_ostream<charT, traits>& put(char_type c);

basic_ostream<charT, traits>& write
    (const char_type* s, streamsize n);

basic_ostream<charT, traits>& flush();

pos_type tellp();
basic_ostream<charT, traits>& seekp(pos_type);
basic_ostream<charT, traits>& seekp
    (off_type, ios_base::seekdir);
};

}
```

Remarks The `basic_ostream` class provides for output stream mechanisms for output stream classes. The `basic_ostream` class may be used as an independent class, as a base class for the `basic_ofstream` class or a user derived classes.

27.6.2.2 basic_ostream Constructor

To create and remove from memory `basic_ostream` object for stream output.

Prototype `explicit basic_ostream
 (basic_streambuf<char_type, traits>*sb);`

Remarks The `basic_ostream` constructor constructs and initializes the base class object.

Destructor

Prototype `virtual ~basic_ostream();`

Remarks Removes a `basic_ostream` object from memory.

Listing 18.17 Example of `basic_ostream()` usage:

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
using namespace std;

    ifstream inOut(inFile, ios::in | ios::out);
    if(!inOut.is_open())
        {cout << "Could not open file"; exit(1);}
    ostream Out(inOut.rdbuf());

    char str[] = "\nRegistered Trademark";

    inOut.rdbuf()->pubseekoff(0, ios::end);

    Out << str;

    inOut.close();

    return 0;
}
```

Result:

The File now reads:

Metrowerks CodeWarrior "Software at Work"
Registered Trademark

27.6.2.3 Class `basic_ostream::sentry`

A class for exception safe prefix and suffix operations.

Prototype namespace std {
 template
 < class charT,
 class traits = char_traits<charT> >
 class basic_ostream<charT, traits>::sentry
 {
 bool ok_;
 public:
 explicit sentry
 (basic_ostream<charT,
 traits>& os,
 bool noskipws = false);
 ~sentry();
 operator bool() {return ok_;}
 };
}

Class `basic_ostream::sentry` Constructor

Constructor

Prepare for formatted or unformatted output

Prototype explicit sentry
 (basic_ostream<charT, traits>& os);

Remarks If after the operation `os.good()` is true `ok_` equals true otherwise `ok_` equals false. The constructor may call `setstate(failbit)` which may throw an exception.

Destructor

Prototype ~sentry();

Remarks The destructor under normal circumstances will call `os.flush()`.

sentry::Operator bool

To return the value of the data member `ok_`.

Prototype `operator bool();`

Return Operator bool returns the value of `ok_`

27.6.2.4 Formatted output functions

Formatted output functions provide a manner of inserting for output specific data types.

27.6.2.4.1 Common requirements

Remarks The operations begins by calling `opfx()` and ends by calling `osfx()` then returning the value specified for the formatted output.

Some output maybe generated by converting the scalar data type to a NTBS (null terminated bit string) text.

If the function fails for any for any reason the function calls `setstate(failbit)`.

27.6.2.4.2 Arithmetic Inserter Operator <<

To provide formatted insertion of types into a stream.

Prototype `basic_ostream<charT, traits>& operator<<(short n)`
`basic_ostream<charT, traits>& operator<<(unsigned short n)`
`basic_ostream<charT, traits>& operator<<(int n)`
`basic_ostream<charT, traits>& operator<<(unsigned int n)`
`basic_ostream<charT, traits>& operator<<(long n)`
`basic_ostream<charT, traits>& operator<<(unsigned long n)`
`basic_ostream<charT, traits>& operator<<(float f)`
`basic_ostream<charT, traits>& operator<<`

Formatting and Manipulators

27.6.2 Output streams

```
(double f)
basic_ostream<charT, traits>& operator<<
    (long double f)
```

Remarks Converts an arithmetical value. The formatted values are converted "as if" they had the same behavior of the `fprintf()` function

Return The `this` pointer is returned

Table 18.2 Output states and stdio equivalents.

| Output State | stdio equivalent |
|---|--------------------|
| Integers | |
| <code>(flags() & basefield) == oct</code> | <code>%o</code> |
| <code>(flags() & basefield) == hex</code> | <code>%x</code> |
| <code>(flags() & basefield) != 0</code> | <code>%x</code> |
| Otherwise | |
| <code>signed integral type</code> | <code>%d</code> |
| <code>unsigned integral type</code> | <code>%u</code> |
| Floating Point Numbers | |
| <code>(flags() & floatfield) == fixed</code> | <code>%f</code> |
| <code>(flags() & floatfield) == scientific</code> | <code>%e</code> |
| <code>(flags() & uppercase) != 0</code> | <code>%E</code> |
| Otherwise | |
| <code>(flags() & uppercase) != 0</code> | <code>%g %G</code> |
| An integral type other than a char type | |
| <code>(flags() & showpos) != 0</code> | <code>+</code> |
| <code>(flags() & showbase) != 0</code> | <code>#</code> |
| A floating point type | |
| <code>(flags() & showpos) != 0</code> | <code>+</code> |
| <code>(flags() & showpoint) != 0</code> | <code>#</code> |

For any conversion if `width()` is non-zero then a field width a conversion specification has the value of `width()`.

For any conversion if `(flags() & fixed) != 0` or if `precision() > 0` the conversion specification is the value of `precision()`.

For any conversion padding behaves in the following manner.

Table 18.3 Conversion state and stdio equivalents.

| State | Justification | stdio equivalent |
|--|---------------|------------------|
| <code>(flags() & adjustfield) == left</code> | left | space padding |
| <code>(flags() & adjustfield) == internal</code> | Internal | zero padding |
| Otherwise | right | space padding |

Remarks The `ostream` insertion operators are overloaded to provide for insertion of most predefined types into and output stream. They return a reference to the `basic_stream` object so they may be used in a chain of statements to input various types to the same stream.

Return In most cases `*this` is returned unless failure in which case `setstate(failbit)` is called.

27.6.2.4.3 basic_ostream::operator<<

Prototype

```
basic_ostream<charT, traits>& operator<<
    (basic_ostream<charT, traits>&
        (*pf)(basic_ostream<charT, traits>&));
```

Remarks Returns `pf(*this)`.

```
basic_ostream<charT, traits>& operator<<
    (basic_ostream<charT, traits>&
        (*pf)(basic_ios<charT, traits>&));
```

Remarks Calls `pf(*this)` return `*this`.

```
basic_ostream<charT, traits>& operator<<
    (const char_type *s)
basic_ostream<charT, traits>& operator<<
    (char_type c)
basic_ostream<charT, traits>& operator<<
```

(bool n)

Remarks Behaves depending on how the `boolalpha` flag is set.

```
basic_ostream<charT, traits>& operator<<  
    (void p)
```

Remarks Converts the pointer to `void p` as if the specifier was `%p` and returns `*this`.

```
basic_ostream<charT, traits>& operator<<  
    (basic_streambuf>char_type, traits>* sb); )
```

Remarks If `sb` is null calls `setstate(failbit)` otherwise gets characters from `sb` and inserts them into `*this` until:

- end-of-file occurs.
- inserting into the stream fails.
- an exception is thrown.

If the operation fails calls `setstate(failbit)` or re-throws the exception, otherwise returns `*this`.

Remarks The formatted output functions insert the values into the appropriate argument type.

Return Most inserters (unless noted otherwise) return the `this` pointer.

Listing 18.18 Example of `basic_ostream` inserter usage:

```
#include <iostream>  
#include <fstream>  
#include <cstdlib>  
  
char oFile[81] = "MW Reference";  
  
int main()  
{  
using namespace std;  
  
    ofstream out(oFile);  
  
    out << "float " << 33.33;  
    out << " double " << 3.16e+10;  
    out << " Integer " << 789;  
    out << " character " << 'C' << endl;
```

```

    out.close();

    cout << "float " << 33.33;
    cout << "\ndouble " << 3.16e+10;
    cout << "\nInteger " << 789;
    cout << "\ncharacter " << 'C' << endl;

    return 0;
}

Result:
Output: to MWReference
float 33.33 double 3.16e+10 Integer 789 character C

Output to console
float 33.33
double 3.16e+10
Integer 789
character C

```

Overloading Inserters

To provide specialized output mechanisms for an object.

Prototype Overloading inserter prototype
`basic_ostream &operator<<`
`(basic_ostream &stream, const omanip<T>&)`
`{`
`// procedures;`
`return stream;`
`}`

Remarks You may overload the inserter operator to tailor it to the specific needs of a particular class.

Return The this pointer.

Listing 18.19 Example of overloaded inserter usage:

```
#include <iostream>
#include <string.h>
#include <iomanip>
```

Formatting and Manipulators

27.6.2 Output streams

```
class phonebook {
    friend ostream &operator<<
        (ostream &stream, phonebook o);
protected:
    char *name;
    int areacode;
    int exchange;
    int num;
public:
    phonebook(char *n, int a, int p, int nm) :
        areacode(a),
        exchange(p),
        num(nm),
        name(n) { }
};

int main()
{
using namespace std;

    phonebook a("Sales", 800, 377, 5416);
    phonebook b("Voice", 512, 873, 4700);
    phonebook c("Fax",      512, 873, 4900);

    cout << a << b << c;

    return 0;
}

std::ostream &operator<<(std::ostream &stream, phonebook o)
{
    stream << o.name << " ";
    stream << "(" << o.areacode << " ) ";
    stream << o.exchange << "-";
    stream << setfill('0') << setw(4)
        << o.num << "\n";
    return stream;
}
```

Result:

Sales (800) 377-5416

Voice (512) 873-4700
 Fax (512) 873-4900

27.6.2.5 Unformatted output functions

Each unformatted output function begins by creating an object of the class `sentry`. The unformatted output functions are ended by destroying the `sentry` object and may return a value specified.

`basic_ostream::tellp`

To return the offset of the put pointer in an output stream.

| | |
|-----------|--|
| Prototype | <code>pos_type tellp();</code> |
| Return | If <code>fail()</code> returns -1 else returns <code>rdbuf() ->pubseekoff(0, cur, out)</code> . |
| See Also | <code>basic_istream::tellg()</code> , <code>seekp()</code> . |

Listing 18.20 Example of `basic_ostream::tellp()` usage.
 see `basic_ostream::seekp()`.

`basic_ostream::seekp`

Randomly move to a position in an output stream.

| | |
|-----------|---|
| Prototype | <code>basic_ostream<charT, traits>& seekp(pos_type);</code> |
| Prototype | <code>basic_ostream<charT, traits>& seekp(off_type, iosbase::seekdir);</code> |
| Remarks | The function <code>seekp</code> is overloaded to take a single argument of a <code>pos_type</code> <code>pos</code> that calls <code>rdbuf() ->pubseekpos(pos)</code> . It is also overloaded to take two arguments an <code>off_type</code> <code>off</code> and <code>ios_base::seekdir</code> type <code>dir</code> that calls <code>rdbuf() ->pubseekoff(off, dir)</code> . |
| Return | The <code>this</code> pointer. |

See Also `basic_istream::seekg()`, `tellp()`

Listing 18.21 Example of basic_ostream::seekp() usage.

```
#include <iostream>
#include <sstream>
#include <string>

std::string motto = "Metrowerks CodeWarrior - Software at Work";

int main()
{
using namespace std;

ostringstream ostr(motto);
streampos cur_pos, start_pos;

cout << "The original array was :\n"
    << motto << "\n\n";
    // associate buffer
stringbuf *strbuf(ostr.rdbuf());

streamoff str_off = 10;
cur_pos = ostr.tellp();
cout << "The current position is "
    << cur_pos.offset()
    << " from the beginning\n";

ostr.seekp(str_off);

cur_pos = ostr.tellp();
cout << "The current position is "
    << cur_pos.offset()
    << " from the beginning\n";

strbuf->sputc('\0');

cout << "The stringbuf array is\n"
    << strbuf->str() << "\n\n";
cout << "The ostringstream array is still\n"
    << motto;

return 0;
}
```

Results:

The original array was :
Metrowerks CodeWarrior - Software at Work

The current position is 0 from the beginning
The current position is 10 from the beginning
The stringbuf array is
Metrowerks

The ostringstream array is still
Metrowerks CodeWarrior - Software at Work

basic_ostream::put

To place a single character in the output stream.

Prototype `basic_ostream<charT, traits>& put(char_type c);`

Remarks The unformatted function `put()` inserts one character in the output stream. If the operation fails calls `setstate(badbit)`.

Return The `this` pointer.

Listing 18.22 Example of basic_ostream::put() usage:

```
#include <iostream>

int main()
{
using namespace std;

    char *str = "Metrowerks CodeWarrior \"Software at Work\" ";
    while(*str)
    {
        cout.put(*str++);
    }
    return 0;
}
```

Result:

Metrowerks CodeWarrior "Software at Work"

basic_ostream::write

To insert a block of binary data into an output stream.

Prototype `basic_ostream<charT, traits>& write
 (const char_type* s, streamsize n);`

Remarks The overloaded function `write()` is used to insert a block of binary data into a stream. This function can be used to write an object by casting that object as a `unsigned char` pointer. If the operation fails calls `setstate(badbit)`.

Return A reference to `ostream`. (The `this` pointer.)

See Also `read()`

Listing 18.23 Example of basic_ostream::write() usage:

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <cstring>

struct stock {
    char name[80];
    double price;
    long trades;
};

char *Exchange = "BBSE";
char *Company = "Big Bucks Inc.';

int main()
{
using namespace std;

    stock Opening, Closing;

    strcpy(Opening.name, Company);
    Opening.price = 180.25;
    Opening.trades = 581300;

    // open file for output
```

```

ofstream Market(Exchange,
                 ios::out | ios::trunc | ios::binary);
if(!Market.is_open())
{cout << "can't open file for output"; exit(1);}

Market.write((char*) &Opening, sizeof(stock));
Market.close();

        // open file for input
ifstream Market2(Exchange, ios::in | ios::binary);
if(!Market2.is_open())
{cout << "can't open file for input"; exit(2);}

Market2.read((char*) &Closing, sizeof(stock));

cout << Closing.name << "\n"
    << "The number of trades was: "
    << Closing.trades << '\n';
cout << fixed << setprecision(2)
    << "The closing price is: $"
    << Closing.price << endl;

Market2.close();

return 0;
}

```

Result:

Big Bucks Inc.
 The number of trades was: 581300
 The closing price is: \$180.25

basic_ostream::flush

To force the output buffer to release its contents.

Prototype `basic_ostream<charT, traits>& flush();`

Remarks The function `flush()` is an output only function in C++. You may use it for an immediate expulsion of the output buffer. This is useful when you have critical data or you need to ensure that a sequence of

events occurs in a particular order. If the operation fails calls setstate(badbit).

Return The this pointer.

Listing 18.24 Example of basic_ostream::flush() usage:

```
#include <iostream>
#include <iomanip>
#include <ctime>

class stopwatch {
private:
    double begin, set, end;
public:
    stopwatch();
    ~stopwatch();
    void start();
    void stop();
};

stopwatch::stopwatch()
{
using namespace std;

begin = (double) clock() / CLOCKS_PER_SEC;
end    = 0.0;
start();
cout << "begin the timer: ";
}

stopwatch::~stopwatch()
{
using namespace std;

stop();      // set end
cout << "\nThe Object lasted: ";
cout << fixed << setprecision(2)
    << end - begin << " seconds \n";
}

// clock ticks divided by ticks per second
```

```

void stopwatch::start()
{
using namespace std;

    set = double(clock() / CLOCKS_PER_SEC);
}

void stopwatch::stop()
{
using namespace std;

    end = double(clock() / CLOCKS_PER_SEC);
}

void time_delay(unsigned short t);

int main()
{
using namespace std;

    stopwatch watch; // create object and initialize
    cout.flush(); // this flushes the buffer
    time_delay(5);
    return 0; // destructor called at return
}
//time delay function
void time_delay(unsigned short t)
{
using namespace std;

    time_t tStart, tEnd;
    time(&tStart);
    while(tStart + t > time(&tEnd)){};
}

```

Result:

Note: comment out the flush and both lines will display simultaneously at the end of the program.

begin the timer: < immediate display then pause >

begin the timer:

The Object lasted: 3.83 seconds

27.6.2.6 Standard `basic_ostream` manipulators

To provide an inline formatting mechanism.

`basic_ostream::endl`

To insert a newline and flush the output stream.

Prototype

```
template
< class charT, class traits >
basic_ostream<charT, traits> & endl
(basic_ostream<charT,traits>& os);
```

Remarks The manipulator `endl` takes no external arguments, but is placed in the stream. It inserts a newline character into the stream and flushes the output.

Return A reference to `basic_ostream`. (The `this` pointer.)

See Also `ostream:: operators`

`basic_ostream::ends`

To insert a NULL character.

Prototype

```
template
< class charT, class traits >
basic_ostream<charT, traits> & ends
(basic_ostream<charT,traits>& os);
```

Remarks The manipulator `ends`, takes no external arguments, but is placed in the stream. It inserts a NULL character into the stream, usually to terminate a string.

Return A reference to `ostream`. (The `this` pointer)

NOTE The `ostringstream` provides in-core character streams but must be null terminated by the user. The manipulator `ends` provides a null terminator.

Listing 18.25 Example of `basic_ostream::ends` usage:

```
#include <iostream>
#include <sstream>

int main()
{
using namespace std;

ostringstream out; // see note above
out << "Ask the teacher anything\n";
out << "OK, what is 2 + 2?\n";
out << 2 << " plus " << 2 << " equals "
    << 4 << ends;

cout << out.str();
return 0;
}
```

Result:

```
Ask the teacher anything
OK, what is 2 + 2?
2 plus 2 equals 4?
```

basic_ostream::flush

To flush the stream for output.

Prototype `template<class charT, class traits>`
 `basic_ostream<charT, traits> &`
 `flush(basic_ostream<charT,traits> (os);`

Remarks The manipulator `flush`, takes no external arguments, but is placed in the stream. The manipulator `flush` will attempt to release an output buffer for immediate use without waiting for an external input.

Return A reference to `ostream`. (The `this` pointer.)

See Also `ostream::flush()`

Formatting and Manipulators

27.6.2 Output streams

Listing 18.26 Example of basic_ostream:: flush usage:

```
#include <iostream>
#include <iomanip>
#include <ctime>

class stopwatch {
private:
    double begin, set, end;
public:
    stopwatch();
    ~stopwatch();
    void start();
    void stop();
};

stopwatch::stopwatch()
{
using namespace std;

begin = (double) clock() / CLOCKS_PER_SEC;
end    = 0.0;
start();
{
begin = (double) clock() / CLOCKS_PER_SEC;
end    = 0.0;
start();
cout << "begin time the timer: " << flush;
}
}

stopwatch::~stopwatch()
{
using namespace std;

stop(); // set end
cout << "\nThe Object lasted: ";
cout << fixed << setprecision(2)
    << end - begin << " seconds \n";
}

// clock ticks divided by ticks per second
```

```
void stopwatch::start()
{
using namespace std;

    set = double(clock() / CLOCKS_PER_SEC);
}

void stopwatch::stop()
{
using namespace std;

    end = double(clock() / CLOCKS_PER_SEC);
}

void time_delay(unsigned short t);

int main()
{
using namespace std;

    stopwatch watch; // create object and initialize
    time_delay(5);
    return 0; // destructor called at return
}
//time delay function
void time_delay(unsigned short t)
{
using namespace std;

    time_t tStart, tEnd;
    time(&tStart);
    while(tStart + t > time(&tEnd)){};
}
```

Results:

Note: comment out the flush and both lines display simultaneously at the end of the program.

```
begin time the timer:
< short pause >
The Object lasted: 3.78 seconds
```

27.6.3 Standard manipulators

The include file `iomanip` defines a template classes and related functions for input and output manipulation.

Standard Manipulator Instantiations

To create a specific use instance of a template by replacing the parameterized elements with pre-defined types.

resetiosflags

To unset previously set formatting flags.

Prototypes `smanip resetiosflags(ios_base::fmtflags mask)`

Remarks Use the manipulator `resetiosflags` directly in a stream to reset any format flags to a previous condition. You would normally store the return value of `setf()` in order to achieve this task.

Return A `smanip` type, that is an implementation defined type.

See Also `ios_base::setf()`, `ios_base::unsetf()`

Listing 18.27 Example of `resetiosflags()` usage:

```
#include <iostream>
#include <iomanip>

int main()
{
using namespace std;

double d = 2933.51;
long flags;
flags = ios::scientific | ios::showpos | ios::showpoint;

cout << "Original: " << d << endl;
cout << "Flags set: " << setiosflags(flags)
    << d << endl;
cout << "Flags reset to original: "
    << resetiosflags(flags) << d << endl;
```

```
    return 0;
}

Result:
Original: 2933.51
Flags set: +2.933510e+03
Flags reset to original: 2933.51
```

setiosflags

Set the stream format flags.

Prototypes smanip setiosflags(ios_base::fmtflags mask)

Remarks Use the manipulator `setiosflags()` to set the input and output formatting flags directly in the stream.

Return A smanip type, that is an implementation defined type.

See Also `ios_base::setf()`, `ios_base::unsetf()`

Listing 18.28 Example of `setiosflags()` usage:

See `resetiosflags()`

:setbase

To set the numeric base of an output.

Prototypes smanip setbase(int)

Remarks The manipulator `setbase()` directly sets the numeric base of integral output to the stream. The arguments are in the form of 8, 10, 16, or 0. 8 octal, 10 decimal and 16 hexadecimal. Zero represents `ios::basefield`, a combination of all three.

Return A smanip type, that is an implementation defined type.

See Also `ios_base::setf()`

Listing 18.29 Example of `setbase` usage:

```
#include <iostream>
#include <iomanip>
```

Formatting and Manipulators

27.6.3 Standard manipulators

```
int main()
{
using namespace std;

    cout << "Hexadecimal "
        << setbase(16) << 196 << '\n';
    cout << "Decimal " << setbase(10)           << 196 << '\n';
    cout << "Octal " << setbase(8) << 196 << '\n';

    cout.setf(ios::hex, ios::oct | ios::hex);
    cout << "Reset to Hex " << 196 << '\n';
    cout << "Reset basefield setting "
        << setbase(0) << 196 << endl;

    return 0;
}
```

Result:

```
Hexadecimal c4
Decimal 196
Octal 304
Reset to Hex c4
Reset basefield setting 196
```

setfill

To specify the characters to used to insert in unused spaces in the output.

Prototypes smanip setfill(int c)

Remarks Use the manipulator `setfill()` directly in the output to fill blank spaces with character `c`.

Return A smanip type, that is an implementation defined type.

See Also `basic_ios::fill`

Listing 18.30 Example of `basic_ios::setfill()` usage:

```
#include <iostream>
#include <iomanip>
```

```
int main()
{
using namespace std;

    cout.width(8);
    cout << setfill('*') << "Hi!" << "\n";
    char fill = cout.fill();
    cout << "The filler is a " << fill << endl;

    return 0;
}
```

Result:
Hi!*****
The filler is a *

setprecision

Set and return the current format precision.

Prototypes `smanip<int> setprecision(int)`

Remarks Use the manipulator `setprecision()` directly in the output stream with floating point numbers to limit the number of digits. You may use `setprecision()` with scientific or non-scientific floating point numbers.

With the flag `ios::floatfield` set the number in `precision` refers to the total number of significant digits generated. If the settings are for either `ios::scientific` or `ios::fixed` then the precision refers to the number of digits after the decimal place.

NOTE This means that `ios::scientific` will have one more significant digit than `ios::floatfield`, and `ios::fixed` will have a varying number of digits.

Return A `smanip` type, that is an implementation defined type.

See Also `ios_base::setf()`, `ios_base::precision()`

Formatting and Manipulators

27.6.3 Standard manipulators

Listing 18.31 Example of setprecision() usage:

```
#include <iostream>
#include <iomanip>

int main()
{
using namespace std;

    cout << "Original: " << 321.123456 << endl;
    cout << "Precision set: " << setprecision(8)
        << 321.123456 << endl;
    return 0;
}
```

Result:

```
Original: 321.123
Precision set: 321.12346
```

setw

To set the width of the output field.

Prototypes `smanip<int> setw(int)`

Remarks Use the manipulator `setw()` directly in a stream to set the field size for output.

Return A pointer to `ostream`

See Also `ios_base::width()`

Listing 18.32 Example of setw() usage:

```
#include <iostream>
#include <iomanip>

int main()
{
using namespace std;

    cout << setw(8)
        << setfill('*')
```

```
    << "Hi!" << endl;
    return 0;
}
```

Result:
Hi!*****

Overloaded Manipulator

To store a function pointer and object type for input.

Prototype Overloaded input manipulator for `int` type.

```
istream &imanip_name
        (istream &stream, type param)
{
    // body of code
    return stream;
}
```

Overloaded output manipulator for `int` type.

```
ostream &omanip_name
        (ostream &stream, type param)
{
    // body of code
    return stream;
}
```

For other input/output types

```
smanip<type> mainip_name(type param)
{
    return smanip<type> (manip_name, param);
}
```

Remarks Use an overloaded manipulator to provide special and unique input handling characteristics for your class.

Return A pointer to stream object.

Listing 18.33 Example of overloaded manipulator usage:

```
#include <iostream>
#include <cstring>
#include <cstdlib>
```

Formatting and Manipulators

27.6.3 Standard manipulators

```
#include <cctype>

char buffer[80];
char *Password = "Metrowerks";

struct verify
{
    explicit verify(char* check) : check_(check) { }
    char* check_;
};

char *StrUpr(char * str);
std::istream& operator >> (std::istream& stream, const verify& v);

int main()
{
using namespace std;

    cin >> verify(StrUpr(Password));
    cout << "Log in was Completed ! \n";

    return 0;
}

std::istream& operator >> (std::istream& stream, const verify& v)
{
using namespace std;

    short attempts = 3;

    do {
        cout << "Enter password: ";
        stream >> buffer;

        StrUpr(buffer);
        if (! strcmp(v.check_, buffer)) return stream;
        cout << "\a\aa";
        attempts--;
    } while(attempts > 0);

    cout << "All Tries failed \n";
```

```
    exit(1);
    return stream;
}

char *StrUpR(char * str)
{
    char *p = str; // dupe string
    while(*p) *p++ = static_cast<char>(std::toupper(*p));
    return str;
}
```

Result:

```
Enter password: <codewarrior>
Enter password: <mw>
Enter password: <metrowerks>
Log in was Completed !
```

Formatting and Manipulators

27.6.3 Standard manipulators

String Based Streams

This chapter discusses string-based streams in the standard C++ library.

The String Based Stream Library (clause 27.7)

There are four template classes and 6 various types defined in the header `<sstream>` that are used to associate stream buffers with objects of class `basic_string`.

The chapter is constructed in the following sub sections and mirrors clause 27.7 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header `<sstream>`” on page 597](#)
- [“27.7.1 Template class `basic_stringbuf`.” on page 598](#)
- [“27.7.2 Template class `basic_istringstream`” on page 604](#)
- [“27.7.3 Class `basic_stringstream`” on page 614](#)

Header `<sstream>`

Overview The header `<sstream>` includes classes and typed that associate stream buffers with string objects for input and output manipulations.

Prototype

```
namespace std{
    template
        <class charT, class traits = char_traits<charT> >
    class basic_stringbuf;
    typedef basic_stringbuf<char>     stringbuf;
    typedef basic_stringbuf<wchar>    wstringbuf;
```

String Based Streams

27.7.1 Template class `basic_stringbuf`.

```
template
    <class charT, class traits =           char_traits<charT> >
class basic_istringstream;
typedef basic_istringstream<char>      istringstream;
typedef basic_istringstream<wchar>     wistringstream;

template
    <class charT, class traits =           char_traits<charT> >
class basic_ostringstream;
typedef basic_ostringstream<char>      ostringstream;
typedef basic_ostringstream<wchar>     wostringstream;
};

}
```

Remarks The class `basic_string` is discussed in previous chapters.

27.7.1 Template class `basic_stringbuf`.

Overview The template class `basic_stringbuf` is derived from `basic_streambuf` and is used to associate both input and output streams with an object of class `basic_string`.

The other topics in this section are:

- [“27.7.1.1 basic_stringbuf constructors” on page 599](#)
- [“27.7.1.2 Member functions” on page 601](#)
- [“27.7.1.3 Overridden virtual functions” on page 602](#)

Prototype

```
template
    <class charT, class traits =           char_traits<charT> >
class basic_stringbuf: public      basic_streambuf<charT, traits> >
{
public:

    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    explicit basic_stringbuf
```

```
(ios_base::openmode which = ios_base::in | ios_base::out);
explicit basic_stringbuf
  (const basic_string<char_type> &str,
  ios_base::openmode which = ios_base::in | ios_base::out);

basic_string<char_type> str() const;
void str(const basic_string<char_type>&s);

protected
virtual int_type underflow();
virtual int_type pbackfail
  (int_type c = traits::eof());
virtual int_type overflow
  (int_type c = traits::eof());

virtual pos_type seekoff
  (off_type off,
  ios_base::seekdir way,
  ios_base::openmode which = ios_base::in | ios_base::out);
virtual pos_type seekpos
  (pos_type sp,
  ios_base::openmode which = ios_base::in | ios_base::out);

private:
ios_base::openmode mode; exposition only
};

}
```

Remarks The class `basic_stringbuf` is derived from `basic_streambuf` to associate a stream with a `basic_string` object for in-core memory character manipulations.

27.7.1.1 `basic_stringbuf` constructors

The `basic_stringbuf` has two constructors:

- `explicit basic_stringbuf(ios_base::openmode);`
- `explicit basic_stringbuf(const basic_string,
ios_base::openmode);`

Constructor

To create a string buffer for characters for input/output.

String Based Streams

27.7.1 *Template class basic_stringbuf.*

Prototype `explicit basic_stringbuf
 (ios_base::openmode which =
 ios_base::in | ios_base::out);
explicit basic_stringbuf
 (const basic_string <char_type> &str,
 ios_base::openmode which =
 ios_base::in | ios_base::out);`

Remarks The `basic_stringbuf` constructor is used to create an object usually as an intermediate storage object for input and output. The overloaded constructor is used to determine the input or output attributes of the `basic_string` object when it is created.

No array object is allocated.

Listing 19.1 Example of `basic_stringbuf::basic_stringbuf()` usage:

```
#include <iostream>
#include <sstream>

const int size = 100;

int main()
{
using namespace std;

    stringbuf strbuf;
    strbuf.pubsetbuf('\0', size);
    strbuf.sputn("ABCDE",50);

    char ch;
        // look ahead at the next character
    ch =strbuf.snextc();
    cout << ch;
        // get pointer was not returned after peeking
    ch = strbuf.snextc();
    cout << ch;

    return 0;
}
```

Result:

BC

27.7.1.2 Member functions

The class `basic_stringbuf` has one member functions:

- `str()`

basic_stringbuf::str

To return or clear the `basic_string` object stored in the buffer.

Prototype `basic_string<char_type> str() const;`
 `void str(const basic_string<char_type>&s);`

Remarks The function `str()` freezes the buffer then returns a
`basic_string` object.

The function `str(const string s)` assigns the value of the
string 's' to the `stringbuf` object.

Return The no argument version returns a `basic_string` if successful.
The function with an argument has no return.

Listing 19.2 Example of `basic_stringbuf::str()` usage:

```
#include <iostream>
#include <sstream>
#include <cstring>

char CW[ ] = "Metrowerks CodeWarrior";
char AW[ ] = " - \"Software at Work\" ";

int main()
{
using namespace std;

    string buf;
    stringbuf strbuf(buf, ios::in | ios::out);

    int size;
```

String Based Streams

27.7.1 Template class `basic_stringbuf`.

```
size = strlen(CW);
strbuf.sputn(CW, size);
size = strlen(AW);
strbuf.sputn(AW, size);
cout << strbuf.str() << endl;

// Clear the buffer then fill it with
// new information and then display it
string clrBuf = "";
string ANewLine = "We Listen we Act";

strbuf.str(clrBuf);
strbuf.sputn(ANewLine.c_str(), ANewLine.size());

cout << strbuf.str() << endl;
return 0;
}
```

Results

```
Metrowerks CodeWarrior - "Software at Work"
We Listen we Act
```

27.7.1.3 Overridden virtual functions

The base class `basic_streambuf` has several virtual functions that are to be overloaded by derived classes. The are:

- `underflow()`
- `pbackfail()`
- `overflow()`
- `seekoff()`
- `seekpos()`

`basic_stringbuf::underflow`

To show an underflow condition and not increment the get pointer.

Prototype `virtual int_type underflow();`

Remarks The function `underflow` overrides the `basic_streambuf` virtual function.

Return The first character of the pending sequence and does not increment the get pointer. If the position is `null` returns `traits::eof()` to indicate failure.

See Also `basic_streambuf::underflow()`

`basic_stringbuf::pbackfail`

To show a failure in a put back operation.

Prototype `virtual int_type pbackfail
(int_type c = traits::eof());`

Remarks The function `pbackfail` overrides the `basic_streambuf` virtual function.

Return The function `pbackfail()` is only called when a put back operation really has failed and returns `traits::eof`. If success occurs the return is undefined.

See Also `basic_streambuf::pbackfail()`

`basic_stringbuf::overflow`

Consumes the pending characters of an output sequence.

Prototype `virtual int_type overflow
(int_type c = traits::eof());`

Remarks The function `overflow` overrides the `basic_streambuf` virtual function.

Return The function returns `traits::eof()` for failure or some unspecified result to indicate success.

See Also `basic_streambuf::overflow()`

`basic_stringbuf::seekoff`

To return an offset of the current pointer in an input or output streams.

Prototype `virtual pos_type seekoff
(off_type off,
ios_base::seekdir way,`

String Based Streams

27.7.2 Template class `basic_istringstream`

```
ios_base::openmode which =
    ios_base::in | ios_base::out);
```

Remarks The function `seekoff` overrides the `basic_streambuf` virtual function.

Return A `pos_type` value, which is an invalid stream position.

See Also `basic_streambuf::seekoff()`

`basic_stringbuf::seekpos`

To alter an input or output stream position.

Prototype

```
virtual pos_type seekpos
    (pos_type sp,
     ios_base::openmode which =
        ios_base::in | ios_base::out);
```

Remarks The function `seekoff` overrides the `basic_streambuf` virtual function.

Return A `pos_type` value, which is an invalid stream position.

See Also `basic_streambuf::seekoff()`

27.7.2 Template class `basic_istringstream`

Overview The template class `basic_istringstream` is derived from `basic_istream` and is used to associate input streams with an object of class `basic_string`.

The prototype is listed below. The other topics in this section are:

- [“27.7.2.1 basic_istringstream constructors” on page 605](#)
- [“27.7.2.2 Member functions” on page 607](#)

Prototype

```
namespace std {
    template
        <class charT, class traits = char_traits<charT> >
    class basic_istringstream : public basic_istream<charT,
traits>
{
```

```

public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    explicit basic_istringstream
        (ios_base::openmode which = ios_base::in);
    explicit basic_istringstream
        (const basic_string<charT> &str,
         ios_base::openmode which = ios_base::in);

    basic_stringbuf<charT, traits>* rdbuf() const;

    basic_string<charT> str() const;
    void str(const basic_string<charT> &s);

private:
    basic_stringbuf<charT,traits> sb; exposition only
};

}

```

Remarks The class `basic_istringstream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also `basic_ostringstream`, `basic_string`,
`basic_stringstream`, `basic_filebuf`.

27.7.2.1 `basic_istringstream` constructors

The class `basic_istringstream` has two constructors.

- `basic_istringstream`
`(ios_base::openmode)`
- `basic_istringstream`
`(const basic_string, ios_base::openmode)`

Constructor

The `basic_istringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

String Based Streams

27.7.2 Template class `basic_istringstream`

Prototype `explicit basic_istringstream
 (ios_base::openmode which = ios_base::in);
explicit basic_istringstream
 (const basic_string<charT> &str,
 ios_base::openmode which = ios_base::in);`

Remarks The `basic_istringstream` constructor is overloaded to accept a
an object of class `basic_string` for input.

See Also `basic_ostringstream`, `basic_stringstream`

Listing 19.3 Example of `basic_istringstream::basic_istringstream()` usage

```
#include <iostream>
#include <string>
#include <sstream>

int main()
{
using namespace std;

string sBuffer = "3 12.3 line";
int num = 0;
double flt = 0;
char szArr[20] = "\0";

istringstream Paragraph(sBuffer, ios::in);
Paragraph >> num;
Paragraph >> flt;
Paragraph >> szArr;

cout << num << " " << flt << " "
<< szArr << endl;

return 0;
}

Result
3 12.3 line
```

27.7.2.2 Member functions

The class `basic_istringstream` has two member functions

- `rdbuf()`
- `str()`

`basic_istringstream::rdbuf`

To retrieve a pointer to the stream buffer.

Prototype `basic_stringbuf<charT, traits>* rdbuf() const;`

Remarks To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer.

Return A pointer to an object of type `basic_stringbuf` **sb** is returned by the `rdbuf` function.

See Also `basic_ostringstream::rdbuf()`

`basic_ios::rdbuf()`

`basic_stringstream::rdbuf()`

Listing 19.4 Example of `basic_istringstream::rdbuf()` usage.

```
#include <iostream>
#include <sstream>

std::string buf = "Metrowerks CodeWarrior - \"Software at
work\"";
char words[50];

int main()
{
using namespace std;

istringstream ist(buf);
istream in(ist.rdbuf());
in.seekg(25);

in.get(words,50);
cout << words;
```

String Based Streams

27.7.2 Template class `basic_istringstream`

```
    return 0  
}
```

Result

```
"Software at work"
```

`basic_istringstream::str`

To return or assign the `basic_string` object stored in the buffer.

Prototype `basic_string<charT> str() const;`
`void str(const basic_string<charT> &s);`

Remarks The function `str()` freezes the buffer then returns a `basic_string` object.

The function `str(const string s)` assigns the value of the string 's' to the `stringbuf` object.

Return The no argument version returns a `basic_string` if successful.
The function with an argument has no return.

See Also `basic_streambuf::str()`
`basic_ostringstream.str()`
`basic_stringstream::str()`

Listing 19.5 Example of `basic_istringstream::str()` usage.

```
#include <iostream>  
#include <sstream>  
  
std::string buf = "Metrowerks CodeWarrior - \"Software at  
Work\"";  
  
int main()  
{  
using namespace std;  
  
    istringstream istr(buf);  
    cout << istr.str();  
    return 0;  
}
```

Result:

Metrowerks CodeWarrior - "Software at Work"

27.7.2.3 Class *basic_ostringstream*

Overview The template class `basic_ostringstream` is derived from `basic_ostream` is use to associate output streams with an object of class `basic_string`.

The prototype is listed below. The other topics in this section are:

- 27.7.2.4 `basic_ostringstream` constructors
- 27.7.2.5 Member functions

Prototype

```
namespace std {
    template
        <class charT, class traits = char_traits<charT> >
        class basic_ostringstream : public basic_ostream<charT,
traits>{

public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    explicit basic_ostringstream
        (ios_base::openmode which = ios_base::out);
    explicit basic_ostringstream
        (const basic_string<charT> &str,
        ios_base::openmode which = ios_base::out);

    basic_stringbuf<charT, traits>* rdbuf() const;

    basic_string<charT> str() const;
    void str(const basic_string<charT> &s);

private:
    basic_stringbuf<charT,traits> sb; exposition only
}
```

String Based Streams

27.7.2.3 Class `basic_ostringstream`

```
};  
}
```

Remarks The class `basic_ostringstream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also `basic_istringstream`, `basic_string`,
`basic_stringstream`, `basic_filebuf`.

27.7.2.4 `basic_ostringstream` constructors.

The class `basic_ostringstream` has two constructors

- `basic_ostringstream`
(`ios_base::openmode`)
- `basic_ostringstream`
(`const basic_string`, `ios_base::openmode`)

Constructor

The `basic_ostringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

Prototype `explicit basic_ostringstream`
(`ios_base::openmode` which = `ios_base::out`);
`explicit basic_ostringstream`
(`const basic_string<charT>` &`str`,
`ios_base::openmode` which = `ios_base::out`);

Remarks The `basic_stringstream` constructor is overloaded to accept a an object of class `basic_string` for output.

See Also `basic_istringstream`, `basic_stringstream`

Listing 19.6 Example of `basic_ostringstream::basic_ostringstream()` usage

The file MW Reference contains
Metrowerks CodeWarrior - "Software at Work"
Registered Trademark

```
#include <iostream>  
#include <fstream>
```

```
#include <sstream>
#include <cstdlib>

int main()
{
using namespace std;

ifstream in( "MW Reference" );
if(!in.is_open())
{cout << "can't open file for input"; exit(1);}

ostringstream Paragraph;
char ch = '\0';

while((ch = in.get()) != EOF)
{
    Paragraph << ch;
}

cout << Paragraph.str();

in.close();
return 0;
}
```

Result:

Metrowerks CodeWarrior - "Software at Work"
Registered Trademark

27.7.2.5 Member functions

The class `basic_ostream` has two member functions:

- `rdbuf()`
- `str()`

`basic_ostream::rdbuf`

To retrieve a pointer to the stream buffer.

Prototype `basic_stringbuf<charT, traits>* rdbuf() const;`

String Based Streams

27.7.2.3 Class `basic_ostringstream`

| | |
|----------|---|
| Remarks | To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function <code>rdbuf()</code> allows you to retrieve this pointer. |
| Return | A pointer to an object of type <code>basic_stringbuf</code> sb is returned by the <code>rdbuf</code> function. |
| See Also | <code>basic_ostringstream::rdbuf()</code> <code>basic_ios::rdbuf()</code> <code>basic_stringstream::rdbuf()</code> |

Listing 19.7 example of `basic_ostringstream::rdbuf()` usage

```
#include <iostream>
#include <sstream>
#include <string>

std::string motto = "Metrowerks CodeWarrior - \"Software at
Work\"";

int main()
{
using namespace std;

ostringstream ostr(motto);
streampos cur_pos(0), start_pos(0);

cout << "The original array was :\n"
    << motto << "\n\n";
    // associate buffer
stringbuf *strbuf(ostr.rdbuf());

streamoff str_off = 10;
cur_pos = ostr.tellp();
cout << "The current position is "
    << static_cast<streamoff>(cur_pos);
    << " from the beginning\n";

ostr.seekp(str_off);

cur_pos = ostr.tellp();
cout << "The current position is "
```

```
<< static_cast<streamoff>(cur_pos);
<< " from the beginning\n";

strbuf->sputc('\0');

cout << "The stringbuf array is\n"
    << strbuf->str() << "\n\n";
cout << "The ostringstream array is still\n"
    << motto;

return 0;
}
```

Results:

```
The original array was :
Metrowerks CodeWarrior - "Software at Work"
```

```
The current position is 0 from the beginning
The current position is 10 from the beginning
The stringbuf array is
Metrowerks
Metrowerks CodeWarrior - "Software at Work"
```

basic_ostringstream::str

To return or assign the `basic_string` object stored in the buffer.

Prototype `basic_string<charT> str() const;`
`void str(const basic_string<charT> &s);`

Remarks The function `str()` freezes the buffer then returns a `basic_string` object.

The function `str(const string s)` assigns the value of the string 's' to the `stringbuf` object.

Return The no argument version returns a `basic_string` if successful.
The function with an argument has no return.

See Also `basic_streambuf::str()`, `basic_istringstream.str()`
`basic_stringstream::str()`

String Based Streams

27.7.3 Class `basic_stringstream`

Listing 19.8 Example of `basic_ostringstream::str()` usage.

```
#include <iostream>
#include <sstream>

int main()
{
using namespace std;

ostringstream out;
out << "Ask the teacher anything\n";
out << "OK, what is 2 + 2?\n";
out << 2 << " plus " << 2 << " equals "
    << 4 << ends;

cout << out.str();
return 0;
}
```

Result:

```
Ask the teacher anything
OK, what is 2 + 2?
2 plus 2 equals 4?
```

27.7.3 Class `basic_stringstream`

Overview The template class `basic_stringstream` is derived from `basic_iostream` is use to associate input and output streams with an object of class `basic_string`.

The prototype is listed below. The other topics in this section are:

- 27.7.3.4 `basic_stringstream` constructors
- 27.7.3.5 Member functions

Prototype

```
namespace std {
    template
        <class charT, class traits = char_traits<charT> >
    class basic_stringstream : public basic_iostream<charT,
traits>
```

```
{  
public:  
    typedef charT char_type;  
    typedef typename traits::int_type int_type;  
    typedef typename traits::pos_type pos_type;  
    typedef typename traits::off_type off_type;  
  
    explicit basic_stringstream  
        (ios_base::openmode which = ios_base::out | ios_base::out);  
    explicit basic_stringstream  
        (const basic_string<charT> &str,  
         ios_base::openmode which = ios_base::out | ios_base::out);  
  
    basic_stringbuf<charT, traits>* rdbuf() const;  
  
    basic_string<charT> str() const;  
    void str(const basic_string<charT> &s);  
  
private:  
    basic_stringbuf<charT, traits> sb; exposition only  
};  
}
```

Remarks The class `basic_stringstream` uses an object of type `basic_stringbuf` to control the associated storage.

See Also `basic_istringstream`, `basic_string`,
`basic_stringstream`, `basic_filebuf`

27.7.3.4 `basic_stringstream` constructors

The class `basic_stringstream` has two constructors:

- `explicit basic_stringstream
(ios_base::openmode)`
- `explicit basic_stringstream
(const basic_string, ios_base::openmode)`

String Based Streams

27.7.3 Class `basic_stringstream`

Constructor

The `basic_stringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

Prototype `explicit basic_stringstream
 (ios_base::openmode which =
 ios_base::out | ios_base::out);
explicit basic_stringstream
 (const basic_string<charT> &str,
 ios_base::openmode which =
 ios_base::out | ios_base::out);`

Remarks The `basic_stringstream` constructor is overloaded to accept a an object of class `basic_string` for input or output.

See Also `basic_ostringstream`, `basic_istringstream`

Listing 19.9 Example of `basic_stringstream::basic_stringstream()` usage

```
#include <iostream>
#include <sstream>

char buf[50] = "ABCD 22 33.33";
char words[50];

int main()
{
using namespace std;

stringstream iost;

char word[20];
long num;
double real;

iost << buf;
iost >> word;
iost >> num;
iost >> real;

cout << word << " "
```

```
    << num << " "
    << real << endl;

    return 0;
}
```

Result
ABCD 22 33.33

27.7.3.5 Member functions

The class `basic_stringstream` has two member functions:

- `rdbuf()`
- `str()`

`basic_stringstream::rdbuf`

To retrieve a pointer to the stream buffer.

Prototype `basic_stringbuf<charT, traits>* rdbuf() const;`

Remarks To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer.

Return A pointer to an object of type `basic_stringbuf` **sb** is returned by the `rdbuf` function.

See Also `basic_ostringstream::rdbuf()` `basic_ios::rdbuf()`
`basic_stringstream::rdbuf()`

Listing 19.10 Example of `basic_stringstream::rdbuf()` usage

```
#include <iostream>
#include <iostream>
#include <sstream>

std::string buf = "Metrowerks CodeWarrior - \"Software at
Work\" ";
char words[50];

int main()
```

String Based Streams

27.7.3 Class basic_stringstream

```
{  
using namespace std;  
  
stringstream ist(buf, ios::in);  
istream in(ist.rdbuf());  
in.seekg(25);  
  
in.get(words,50);  
cout << words;  
  
return 0;  
}
```

Result

"Software at Work"

basic_stringstream::str

To return or assign the `basic_string` object stored in the buffer.

Prototype `basic_string<charT> str() const;`
 `void str(const basic_string<charT> &s);`

Remarks The function `str()` freezes the buffer then returns a `basic_string` object.

The function `str(const string s)` assigns the value of the string 's' to the `stringbuf` object.

Return The no argument version returns a `basic_string` if successful.
The function with an argument has no return.

See Also `basic_streambuf::str()`
 `basic_ostringstream.str()`
 `basic_istringstream::str()`

Listing 19.11 Example of basic_stringstream::str() usage

```
#include <iostream>  
#include <sstream>  
  
std::string buf = "Metrowerks CodeWarrior - \"Software at  
Work\"";
```

```
char words[50];

int main()
{
using namespace std;

    stringstream iost(buf, ios::in);

    cout << iost.str();

    return 0;
}
```

Result
Metrowerks CodeWarrior - "Software at Work"

String Based Streams

27.7.3 Class *basic_stringstream*

File Based Streams

Association of stream buffers with files for file reading and writing.

The File Based Streams Library (clause 27.8)

The chapter is constructed in the following sub sections and mirrors clause 27.8 of the ISO (the International Organization for Standardization) C++ Standard :

- [“Header <fstream>” on page 621](#)
- [“27.8.1 File streams” on page 621](#)
- [“27.8.1.1 Template class basic_filebuf” on page 622](#)
- [“27.8.1.5 Template class basic_ifstream” on page 630](#)
- [“27.8.1.8 Template class basic_ofstream” on page 636](#)
- [“27.8.1.11 Template class basic_fstream” on page 642](#)

Header <fstream>

The header <fstream> defines template classes and types to assist in reading and writing of files.

27.8.1 File streams

Prototype

```
namespace std{
template
<class charT, class traits = ios_traits<charT> >
class basic_filebuf;
typedef basic_filebuf<char> filebuf;
typedef basic_filebuf<wchar_t> wfilebuf;
```

File Based Streams

27.8.1.1 Template class `basic_filebuf`

```
template
    <class charT, class traits = ios_traits<charT> >
class basic_ifstream;
    typedef basic_ifstream<char> ifstream;
    typedef basic_ifstream<wchar_t> wifstream;

template
    <class charT, class traits = ios_traits<charT> >
class basic_ofstream;
    typedef basic_ofstream<char> ofstream;
    typedef basic_ofstream<wchar_t> wofstream;
}
```

Remarks A FILE refers to the type FILE as defined in the Standard C Library and provides an external input or output stream with the underlying type of char or byte. A stream is a sequence of char or bytes.

27.8.1.1 Template class `basic_filebuf`

A class to provide for input and output file stream buffering mechanisms.

The prototype is listed below. Other topics in this section are:

- [“27.8.1.2 basic_filebuf Constructors” on page 624](#)
- [“27.8.1.3 Member functions” on page 625](#)
- [“27.8.1.4 Overridden virtual functions” on page 627](#)

Prototype

```
namespace std{
    template
        <class charT, class traits = ios_traits<charT> >
    class basic_filebuf : public basic_streambuf <charT, traits>
    {
    public:

        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
```

```
typedef typename traits::off_type off_type;

basic_filebuf();
virtual ~basic_filebuf();

bool is_open() const;
basic_filebuf<charT, traits>* open
    (const char* c, ios_base::openmode mode);
basic_filebuf<charT, traits>* close();

protected:

virtual int showmany();

virtual int_type underflow();
virtual int_type pbackfail
    (int_type c = traits::eof());
virtual int_type overflow
    (int_type c = traits::eof());

virtual basic_streambuf<charT traits>* setbuf
    (char_type* s, streamsize n);

virtual pos_type seekoff
    (off_type off,
     ios_base::seekdir way,
     ios_base::in | ios_base::out);
virtual pos_type seekpos
    (pos_type sp,
     ios_base::openmode which,
     ios_base::in | ios_base::out);

virtual int sync();
virtual void imbue(const locale& loc);
};

}
```

Remarks The *filebuf* class is derived from the *streambuf* class and provides a buffer for file output and or input.

File Based Streams

27.8.1.1 Template class `basic_filebuf`

27.8.1.2 `basic_filebuf` Constructors

Default Constructor

To construct and initialize a `filebuf` object.

Prototype `basic_filebuf()`

Remarks The constructor opens a `basic_filebuf` object and initializes it with `basic_streambuf<charT, traits>()` and if successful `is_open()` is false.

Destructor

To remove the `basic_filebuf` object from memory.

Prototype `virtual ~basic_filebuf();`

Listing 20.1 For example of `basic_filebuf::basic_filebuf()` usage:

The file MW Reference before operation contains .

Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdio>
#include <cstring>

char inFile[ ] = "MW Reference";

int main()
{
using namespace std;

FILE *fp = fopen( inFile, "a+" );
filebuf in(fp);
if( !in.is_open() )
    { cout << "could not open file"; exit(1); }
char str[] = "\n\ttrademark";
in.sputn(str, strlen(str));

in.close();
return 0;
}
```

Result:

The file MW Reference now contains:
Metrowerks CodeWarrior "Software at Work"
trademark

27.8.1.3 Member functions

basic_filebuf::is_open

Test to ensure `filebuf` stream is open for reading or writing.

Prototype `bool is_open() const`

Remarks Use the function `is_open()` for a `filebuf` stream to ensure it is open before attempting to do any input or output operation on the stream.

Return True if stream is available and open.

Listing 20.2 For example of `basic_filebuf::is_open()` usage

See: `basic_filebuf::basic_filebuf`

basic_filebuf::open

Open a `basic_filebuf` object and associate it with a file.

Prototype `basic_filebuf<charT, traits>* open
(const char* c,
ios_base::openmode mode);`

Remarks You would use the function `open()` to open a `filebuf` object and associate it with a file. You may use `open()` to reopen a buffer and associate it if the object was closed but not destroyed.

NOTE If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

File Based Streams

27.8.1.1 Template class `basic_filebuf`

Table 20.1 Legal `basic_filebuf` file opening modes

| Opening Modes | stdio equivalent |
|--|---|
| Input Only | |
| <code>ios::in</code> | "r" |
| <code>ios::binary ios::in</code> | "rb" |
| Output only | |
| <code>ios::out</code> | "w" |
| <code>ios::binary ios::out</code> | "wb" |
| <code>ios::out ios::trunc</code> | "w" |
| <code>ios::binary ios::out ios::trunc</code> | "wb" |
| <code>ios::out ios::app</code> | "a" |
| Input and Output | |
| <code>ios::in ios::out</code> | "r+" |
| <code>ios::binary ios::in ios::out</code> | "r+b" |
| <code>ios::in ios::out ios::trunc</code> | "w+" |
| <code>ios::binary ios::in ios::out ios::trunc</code> | "w+b" |
| <code>ios::binary ios::out ios::app</code> | "ab" |
| Return | If successful the <code>this</code> pointer is returned, if <code>is_open()</code> equals true then a null pointer is returned. |

Listing 20.3 Example of `filebuf::open()` usage:

The file MW Reference before operation contained:

Metrowerks CodeWarrior "Software at Work"

```
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main(){
using namespace std;
```

```
filebuf in;
in.open(inFile, ios::out | ios::app);
if(!in.is_open())
    {cout << "could not open file"; exit(1);}
char str[] = "\n\tregistered trademark";
in.sputn(str, strlen(str));

in.close();
return 0;
}
```

Result:

The file MW Reference now contains:
Metrowerks CodeWarrior "Software at Work"
 registered trademark

basic_filebuf::close

To close a *filebuf* stream without destroying it.

Prototype *basic_filebuf<charT, traits>** *close()*;

Remarks The function *close()* would remove the stream from memory but will not remove the *filebuf* object. You may re-open a *filebuf* stream that was closed using the *close()* function.

Return The *this* pointer with success otherwise a null pointer.

Listing 20.4 For example of basic_filebuf::close() usage

See *basic_filebuf::open()*

27.8.1.4 Overridden virtual functions

basic_filebuf::showmanyC

Overrides *basic_streambuf::showmanyC()*.

Prototype *virtual int showmanyC()*;

Remarks Behaves the same as *basic_streambuf::showmanyC()*.

File Based Streams

27.8.1.1 Template class `basic_filebuf`

`basic_filebuf::underflow`

Overrides `basic_streambuf::underflow()`;

Prototype `virtual int_type underflow() ;`

Remarks Behaves the same as `basic_streambuf::underflow` with the specialization that a sequence of characters is read as if they were read from a file into an internal buffer.

`basic_filebuf::pbackfail`

Overrides `basic_streambuf::pbackfail()`.

Prototype `virtual int_type pbackfail
 (int_type c = traits::eof());`

Remarks This function puts back the characters designated by `c` to the input sequence if possible.

Return `traits::eof()` if failure and returns either the character put back or `traits::not_eof(c)` for success.

`basic_filebuf::overflow`

Overrides `basic_streambuf::overflow()`

Prototype `virtual int_type overflow
 (int_type c = traits::eof());`

Remarks Behaves the same as `basic_strreambuf<charT, traits>::overflow(c)` except the behavior of consuming characters is performed by conversion.

Return `traits::eof()` with failure.

`basic_filebuf::seekoff`

Overrides `basic_streambuf::seekoff()`

Prototype `virtual pos_type seekoff
 (off_type off,
 ios_base::seekdir way,
 ios_base::in | ios_base::out);`

| | |
|---------|---|
| Remarks | Sets the offset position of the stream as if using the C standard library function <code>fseek(file, off, whence)</code> . |
| Return | Seekoff function returns a newly formed <code>pos_type</code> object which contains all information needed to determine the current position if successful. An invalid stream position if it fails. |

`basic_filebuf::seekpos`

Overrides `basic_streambuf::seekpos()`

Prototype `virtual pos_type seekpos
 (pos_type sp,
 ios_base::openmode which,
 ios_base::in | ios_base::out);`

| | |
|---------|---|
| Remarks | Description undefined in standard at the time of writing. |
| Return | Seekpos function returns a newly formed <code>pos_type</code> object which contains all information needed to determine the current position if successful. An invalid stream position if it fails. |

`basic_filebuf::setbuf`

Overrides `basic_streambuf::setbuf()`

Prototype `virtual basic_streambuf<charT traits>* setbuf
 (char_type* s, streamsize n);`

| | |
|---------|---|
| Remarks | Description undefined in standard at the time of writing. |
|---------|---|

`basic_filebuf::sync`

Overrides `basic_streambuf::sync`

Prototype `virtual int sync();`

| | |
|---------|---|
| Remarks | Description undefined in standard at the time of writing. |
|---------|---|

`basic_filebuf::imbue`

Overrides `basic_streambuf::imbue`

Prototype `virtual void imbue(const locale& loc);`

File Based Streams

27.8.1.5 Template class basic_ifstream

Remarks Description undefined in standard at the time of writing.

27.8.1.5 Template class basic_ifstream

A class to provide for input file stream mechanisms.

The prototype is listed below. Other topics in this section are:

- [“27.8.1.6 basic_ifstream Constructor” on page 631](#)
- [“27.8.1.7 Member functions” on page 632](#)

Prototype

```
namespace std{
    template
        <class charT, class traits = ios_traits<charT> > {
    class basic_ifstream : public basic_istream<charT, traits>
{
    public:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;

        basic_ifstream();
        explicit basic_ifstream
            (const char *s, openmode mode = in);

        basic_filebuf<charT, traits>* rdbuf() const;
        bool is_open();
        void open(const char* s, openmode mode = in);
        void close();

    private:
        basic_filebuf<charT, traits> sb; exposition only
    };
}
```

NOTE If the `basic_ifstream` supports reading from file. It uses a `basic_filebuf` object to control the sequence. That object is represented here as `basic_filebuf sb`.

Remarks The `basic_ifstream` provides mechanisms specifically for input file streams.

27.8.1.6 `basic_ifstream` Constructor

Default Constructor and Overloaded Constructor

Create a file stream for input.

Prototype `basic_ifstream();`
 `explicit basic_ifstream`
 `(const char *s, openmode mode = in);`

Remarks The constructor creates a stream for file input; it is overloaded to either create and initialize when called or to simply create a class and be opened using the `open()` member function. The default opening mode is `ios::in`. See `basic_filebuf::open()` for valid open mode settings.

NOTE See `basic_ifstream::open` for legal opening modes.

See also `basic_ifstream::open()` for overloaded form usage.

Listing 20.5 Example of `basic_ifstream::basic_ifstream()` constructor usage:

The MW Reference file contains:
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    ifstream in(inFile, ios::in);
    if(!in.is_open())
        {cout << "can't open input file"; exit(1);}
}
```

File Based Streams

27.8.1.5 Template class `basic_ifstream`

```
char c = '\0';
while(in.good())
{
    if(c) cout << c;
    in.get(c);

}

in.close();
return 0;
}
```

Result:
Metrowerks CodeWarrior "Software at Work"

27.8.1.7 Member functions

`basic_ifstream::rdbuf`

The `rdbuf()` function retrieves a pointer to a `filebuf` type buffer.

Prototype `basic_filebuf<chart, traits>* rdbuf() const;`

Remarks In order to manipulate for random access or use an `ifstream` stream for both input and output you need to manipulate the base buffer. The function `rdbuf()` returns a pointer to this buffer for manipulation.

Return A pointer to type `basic_filebuf`.

Listing 20.6 Example of `basic_ifstream::rdbuf()` usage:

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";
```

```
int main()
{
using namespace std;

ifstream inFile(inFile, ios::in | ios::out);
if(!inFile.is_open())
    {cout << "Could not open file"; exit(1);}

ostream Out(inFile.rdbuf());

char str[] = "\n\tRegistered Trademark";

inFile.rdbuf()->pubseekoff(0, ios::end);

Out << str;

inFile.close();

return 0;
}
```

Result:

The File now reads:

Metrowerks CodeWarrior "Software at Work"
 Registered Trademark

basic_ifstream::is_open

Test for open stream.

Prototype `bool is_open() const`

Remarks Use `is_open()` to test that a stream is indeed open and ready for input from the file.

Return True if file is open.

Listing 20.7 For example of basic_ifstream::is_open() usage

See `basic_ifstream::basic_ifstream()`

File Based Streams

27.8.1.5 Template class `basic_ifstream`

`basic_ifstream::open`

Open is used to open a file or reopen a file after closing it.

Prototype `void open(const char* s, openmode mode = in);`

Remarks The default open mode is `ios::in`, but can be one of several modes. (see below) A stream is opened and prepared for input or output as selected.

Return No return

Table 20.2 17.4.1.1.4 Legal `basic_ifstream` file opening modes

| Opening Modes | stdio equivalent |
|--|-------------------------|
| Input Only | |
| <code>ios::in</code> | "r" |
| <code>ios::binary ios::in</code> | "rb" |
| Input and Output | |
| <code>ios::in ios::out</code> | "r+" |
| <code>ios::binary ios::in ios::out</code> | "r+b" |
| <code>ios::in ios::out ios::trunc</code> | "w+" |
| <code>ios::binary ios::in ios::out ios::trunc</code> | "w+b" |
| <code>ios::binary ios::out ios::app</code> | "ab" |

NOTE If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings

Listing 20.8 Example of `basic_ifstream::open()` usage:

The MW Reference file contains:

Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
```

```
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
    using namespace std;

    ifstream in;
    in.open(inFile);
    if(!in.is_open())
        {cout << "can't open input file"; exit(1);}

    char c = NULL;
    while((c = in.get()) != EOF)
    {
        cout << c;
    }

    in.close();
    return 0;
}
```

Result:
Metrowerks CodeWarrior "Software at Work"

basic_ifstream::close

Closes the file stream.

Prototype void close();

Remarks The `close()` function closes the stream for operation but does not destroy the `ifstream` object so it may be re-opened at a later time. If the function fails calls `setstate(failbit)` which may throw an exception.

Return: No return.

Listing 20.9 Example of basic_ifstream::close() usage:

See `basic_ifstream::basic_ifstream()`

27.8.1.8 Template class basic_ofstream

A class to provide for output file stream mechanisms.

The prototype is listed below. Other topics in this section are:

- [“27.8.1.9 basic_ofstream constructor” on page 637](#)
- [“27.8.1.10 Member functions” on page 638](#)

Prototype

```
namespace std{
    template <class charT, class traits = ios_traits<charT> >
    class basic_ofstream : public basic_ostream<charT, traits>
    {
public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    basic_ofstream();
    explicit basic_ofstream
        (const char *s, openmode mode = out | trunc);

    basic_filebuf<charT, traits>* rdbuf() const;
    bool is_open();
    void open(const char* s, openmode mode = out);
    void close();

private:
    basic_filebuf<charT, traits> sb; exposition only
    };
}
```

NOTE The `basic_ofstream` supports writing to file. It uses a `basic_filebuf` object to control the sequence. That object is represented here as `basic_filebuf sb`.

Remarks The *basic_ofstream* class provides for mechanisms specific to output file streams.

27.8.1.9 ***basic_ofstream* constructor**

Default and Overloaded Constructors

To create a file stream object for output.

Prototype `basic_ofstream();`
`explicit basic_ofstream`
`(const char *s, openmode mode = out | trunc);`

Remarks The class *basic_ofstream* creates an object for handling file output. It may be opened later using the *ofstream::open()* member function. It may also be associated with a file when the object is declared. The default open mode is *ios::out*.

NOTE There are only certain valid file opening modes for an *ofstream* object see *basic_ofstream::open()* for a list of valid opening modes.

Listing 20.10 Example of *basic_ofstream::ofstream()* usage:

Before the operation the file MW Reference may or may not exist.

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char outFile[] = "MW Reference";

int main()
{
using namespace std;

ofstream out(outFile);
if(!out.is_open())
{cout << "file not opened"; exit(1);}

out << "This is an annotated reference that "
```

File Based Streams

27.8.1.8 Template class `basic_ofstream`

```
<< "contains a description\n"
<< "of the Working ANSI C++ Standard "
<< "Library and other\nfacilities of "
<< "the Metrowerks Standard Library. ";

out.close();
return 0;
}
```

Result:

This is an annotated reference that contains a description of the Working ANSI C++ Standard Library and other facilities of the Metrowerks Standard Library.

27.8.1.10 Member functions

`basic_ofstream::rdbuf`

To retrieve a pointer to the stream buffer.

Prototype `basic_filebuf<charT, traits>* rdbuf() const;`

Remarks In order to manipulate a stream for random access or other operations you must use the streams base buffer. The member function `rdbuf()` is used to return a pointer to this buffer.

Return A pointer to `filebuf` type.

Listing 20.11 Example of `basic_ofstream::rdbuf()` usage:

The file MW Reference before the operation contains:

This is an annotated reference that contains a description of the Working ANSI C++ Standard Library and other facilities of the Metrowerks Standard Library

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char outFile[] = "MW Reference";

int main()
```

```

{
using namespace std;

ofstream out(outFile, ios::in | ios::out);
if(!out.is_open())
    {cout << "could not open file for output"; exit(1);}
istream inOut(out.rdbuf());

char ch;
while((ch = inOut.get()) != EOF)
{
    cout.put(ch);
}

out << "\nAnd so it goes...";

out.close();

return 0;
}

```

Result:

This is an annotated reference that contains a description of the Working ANSI C++ Standard Library and other facilities of the Metrowerks Standard Library.

This is an annotated reference that contains a description of the Working ANSI C++ Standard Library and other facilities of the Metrowerks Standard Library.

And so it goes...

basic_ofstream::is_open

To test whether the file was opened.

Prototype `bool is_open();`

Remarks The `is_open()` function is used to check that a file stream was indeed opened and ready for output. You should always test with this function after using the constructor or the `open()` function to open a stream.

File Based Streams

27.8.1.8 Template class `basic_ofstream`

Return True if file stream is open and available for output.

Listing 20.12 For example of `basic_ofstream::is_open()` usage

See `basic_ofstream::ofstream()`

basic_ofstream::open

To open or re-open a file stream for output.

Prototype `void open(const char* s, openmode mode = out);`

Remarks The function `open()` opens a file stream for output. The default mode is `ios::out`, but may be any valid open mode (see below.) If failure occurs `open()` calls `setstate(failbit)` which may throw an exception.

Return No return

Table 20.3 Legal `basic_ofstream` file opening modes.

| Opening Modes | stdio equivalent |
|--|------------------|
| Output only | |
| <code>ios::out</code> | "w" |
| <code>ios::binary ios::out</code> | "wb" |
| <code>ios::out ios::trunc</code> | "w" |
| <code>ios::binary ios::out ios::trunc</code> | "wb" |
| <code>ios::out ios::app</code> | "a" |
| Input and Output | |
| <code>ios::in ios::out</code> | "r+" |
| <code>ios::binary ios::in ios::out</code> | "r+b" |
| <code>ios::in ios::out ios::trunc</code> | "w+" |
| <code>ios::binary ios::in ios::out ios::trunc</code> | "w+b" |
| <code>ios::binary ios::out ios::app</code> | "ab" |

NOTE If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

Listing 20.13 Example of `basic_ofstream::open()` usage:

Before operation, the file MW Reference contained:
Chapter One

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char outFile[ ] = "MW Reference";

int main()
{
using namespace std;

ofstream out;
out.open(outFile, ios::out | ios::app);
if(!out.is_open())
    {cout << "file not opened"; exit(1);}

out << "\nThis is an annotated reference that "
    << "contains a description\n"
    << "of the Working ANSI C++ Standard "
    << "Library and other\nfacilities of "
    << "the Metrowerks Standard Library.';

out.close();
return 0;
}
```

Result:

After the operation MW Reference contained
Chapter One
This is an annotated reference that contains a description
of the Working ANSI C++ Standard Library and other
facilities of the Metrowerks Standard Library.

basic_ofstream::close

The member function closes the stream but does not destroy it.

Prototype `void close();`

Remarks Use the function `close()` to close a stream. It may be re-opened at a later time using the member function `open()`. If failure occurs `open()` calls `setstate(failbit)` which may throw an exception.

Return No return.

Listing 20.14 For example of basic_ofstream::close() usage.

```
basic_ofstream()
```

27.8.1.11 Template class basic_fstream

A template class for the association of a file for input and output

The prototype is listed below. The other topic in this section is:

- [“27.8.1.12 basic_fstream Constructor” on page 643](#)
- [“27.8.1.13 Member Functions” on page 644](#)

Prototype

```
namespace std {
    template
        <class charT, class traits=ios_traits<charT> >
    class basic_fstream : public basic_iostream<charT, traits>
{
public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    basic_fstream();
    explicit basic_fstream
        (const char *s,
         ios_base::openmode = ios_base::in | ios_base::out);
```

```

basic_filebuf<charT, traits>* rdbuf() const;

bool is_open();
void open
  (const char* s,
   ios_base::openmode = ios_base::in | ios_base::out);
void close();

private:
basic_filebuf<charT, traits> sb; exposition only
};

}

```

Remarks The template class `basic_fstream` is used for both reading and writing from files.

NOTE The `basic_fstream` supports writing to file. It uses a `basic_filebuf` object to control the sequence. That object is represented here as `basic_filebuf sb`.

27.8.1.12 basic_fstream Constructor

Default and Overloaded Constructor

To construct an object of `basic_ifstream` for input and output operations.

Prototypes

```

basic_fstream();
explicit basic_fstream
  (const char *s,
   ios_base::openmode =
     ios_base::in | ios_base::out);

```

Remarks The `basic_fstream` class is derived from `basic_iostream` and that and a `basic_filebuf` object are initialized at construction.

Listing 20.15 Example of `basic_fstream:: basic_fstream()` usage

The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

File Based Streams

27.8.1.11 Template class `basic_fstream`

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[ ] = "MW Reference";

int main()
{
using namespace std;

fstream inOut(inFile, ios::in | ios::out);
if(!inOut.is_open())
    {cout << "Could not open file"; exit(1);}

char str[ ] = "\n\tRegistered Trademark";

char ch;
while((ch = inOut.get())!= EOF)
{
    cout << ch;
}
inOut.clear();
inOut << str;
inOut.close();

return 0;
}
```

Result:

```
Metrowerks CodeWarrior "Software at Work"
The File now reads:
Metrowerks CodeWarrior "Software at Work"
    Registered Trademark
```

27.8.1.13 Member Functions

`basic_fstream::rdbuf`

The `rdbuf()` function retrieves a pointer to a `filebuf` type buffer.

| | |
|-----------|--|
| Prototype | <code>basic_filebuf<charT, traits>* rdbuf() const;</code> |
| Remarks | In order to manipulate for random access or use of an <code>fstream</code> stream you may need to manipulate the base buffer. The function <code>rdbuf()</code> returns a pointer to this buffer for manipulation. |
| Return | A pointer to type <code>basic_filebuf</code> . |

Listing 20.16 Example of `basic_fstream::rdbuf()` usage

The MW Reference file contains originally Metrowerks CodeWarrior "Software at Work"

```
#include <iostream>
#include <fstream>
#include <cstdlib>

char inFile[] = "MW Reference";

int main()
{
using namespace std;

fstream inout;
inout.open(inFile, ios::in | ios::out);
if(!inout.is_open())
    {cout << "Could not open file"; exit(1);}

char str[] = "\n\tRegistered Trademark";

inout.rdbuf()->pubseekoff(0,ios::end);
inout << str;
inout.close();

return 0;
}
```

Result:

The File now reads:

Metrowerks CodeWarrior "Software at Work"
Registered Trademark

File Based Streams

27.8.1.11 Template class `basic_fstream`

`basic_fstream::is_open`

Test to ensure `basic_fstream` file is open and available for reading or writing.

Prototype `bool is_open() const`

Remarks Use the function `is_open()` for a `basic_fstream` file to ensure it is open before attempting to do any input or output operation on a file.

Return True if a file is available and open.

For an example, see [“Example of `basic_fstream:: basic_fstream\(\)` usage” on page 643](#).

`basic_fstream::open`

To open or re-open a file stream for input or output.

Prototypes `void open
 (const char* s,
 ios_base::openmode =
 ios_base::in | ios_base::out);`

Remarks You would use the function `open()` to open a `basic_fstream` object and associate it with a file. You may use `open()` to reopen a file and associate it if the object was closed but not destroyed.

NOTE If an attempt is made to open a file in an inappropriate file opening mode, the file will not open and a test for the object will not give false, therefore use the function `is_open()` to check for file openings.

Table 20.4 Legal file opening modes

| Opening Modes | stdio equivalent |
|-------------------------------------|-------------------------|
| Input Only | |
| <code>ios:: in</code> | “r” |
| <code>ios:: binary ios::in</code> | “rb” |

| Opening Modes | stdio equivalent |
|---|---|
| Output only | |
| ios::out | "w" |
| ios::binary ios::out | "wb" |
| ios::out ios::trunc | "w" |
| ios::binary ios::out ios::trunc | "wb" |
| ios::out ios::app | "a" |
| Input and Output | |
| ios::in ios::out | "r+" |
| ios::binary ios::in ios::out | "r+b" |
| ios::in ios::out ios::trunc | "w+" |
| ios::binary ios::in ios::out ios::trunc | "w+b" |
| ios::binary ios::out ios::app | "ab" |
| Return | No return. |
| | For an example, see " Example of basic_fstream::rdbuf() usage " on page 645 . |

basic_fstream::close

The member function closes the stream but does not destroy it.

Prototype `void close();`

Remarks Use the function `close()` to close a stream. It may be re-opened at a later time using the member function `open()`. If failure occurs `open()` calls `setstate(failbit)` which may throw an exception.

Return No return.

For an example, see "[Example of basic_fstream::basic_fstream\(\)](#)" on [page 643](#).

File Based Streams

27.8.1.11 *Template class basic_fstream*

C Library Files

The header `<cstdio>` contains the C++ implementation of the Standard C Headers.

The C Library Files (clause 27.9)

The chapter is constructed in the following sub sections and mirrors clause 27.9 of the ISO (the International Organization for Standardization) C++ Standard :

`<cstdio>` Macros

Macros:

| | | |
|-----------|----------|--------------|
| BUFSIZ | EOF | FILENAME_MAX |
| FOPEN_MAX | L_tmpnam | NULL |
| SEEK_CUR | SEEK_END | SEEK_SET |
| stderr | stdin | stdout |
| TMP_MAX | _IOFBF | _IOLBF |
| _IONBF | | |

`<cstdio>` Types

Types:

| | | |
|------|--------|--------|
| FILE | fpos_t | size_t |
|------|--------|--------|

<cstdio> Functions

Functions:

| | | |
|----------|---------|----------|
| clearerr | fclose | feof |
| ferror | fflush | fgetc |
| fgetpos | fgets | fopen |
| fprintf | fputc | fputs |
| fread | freopen | fscanf |
| fseek | fsetpos | ftell |
| fwrite | getc | getchar |
| gets | perror | printf |
| putc | putchar | puts |
| remove | rename | rewind |
| scanf | setbuf | setvbuf |
| sprintf | scanf | tmpnam |
| ugetc | vprintf | vfprintf |
| vsprintf | tmpfile | |

Strstream

The header `<strstream>` defines streambuf derived classes that allow for the formatting and storage of character array based buffers, as well as their input and output.

The Strstream Class Library (Annex D)

The chapter is constructed in the following sub sections and mirrors annex D of the ISO (the International Organization for Standardization) C++ Standard :

- [“Strstreambuf Class” on page 654](#), a base class for strstream classes
 - [“Strstreambuf constructors and Destructors” on page 656](#)
 - [“Strstreambuf Public Member Functions” on page 657](#)
 - [“Protected Virtual Member Functions” on page 660](#)
- [“Istrstream Class” on page 662](#), a strstream class for input
 - [“Constructors and Destructor” on page 663](#)
 - [“Public Member Functions” on page 664](#)
- [“Ostrstream Class” on page 666](#), a strstream class for output
 - [“Constructors and Destructor” on page 666](#)
 - [“Public Member Functions” on page 668](#)
- [“Strstream Class” on page 670](#), a class for input and output
 - [“Constructors and Destructor” on page 671](#)
 - [“Public Member Functions” on page 672](#)

Header <strstream>

The include file strstream includes three classes, for in memory character array based stream input and output.

Listing 22.1 Class declarations for header <strstream>

```
class strstreambuf
    : public streambuf
{
public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t),
                 void (*pfree_arg)(void*));
    strstreambuf(char* gnex_arg, streamsize n, char* pbeg_arg =
0);
    strstreambuf(const char* gnex_arg, streamsize n);
    strstreambuf(signed char* gnex_arg, streamsize n,
                 signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnex_arg, streamsize n);
    strstreambuf(unsigned char* gnex_arg, streamsize n,
                 unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnex_arg, streamsize n);
    virtual ~strstreambuf();
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
protected:
    virtual int_type overflow (int_type c = EOF);
    virtual int_type pbackfail(int_type c = EOF);
    virtual int_type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                           ios_base::openmode which = ios_base::in |
                           ios_base::out);
    virtual pos_type seekpos(pos_type sp,
                           ios_base::openmode which = ios_base::in |
                           ios_base::out);
    virtual streambuf* setbuf(char* s, streamsize n);
private:
    typedef unsigned char strstate;
    static const strstate allocated = 1 << 0;
    static const strstate constant   = 1 << 1;
```

```
static const strstate dynamic      = 1 << 2;
static const strstate frozen       = 1 << 3;
static const streamsize default_alsize = 128;
streamsize alsizel_;
void* (*palloc_)(size_t);
void (*pfree_)(void*);
strstate strmode_;

void init(char* gnnext_arg, streamsize n, char* pbeg_arg = 0);
};

class istrsstream : public basic_istream<char>
{
public:
    explicit istrsstream(const char* s);
    explicit istrsstream(char* s);
    istrsstream(const char* s, streamsize n);
    istrsstream(char* s, streamsize n);
    virtual ~istrsstream();
    strstreambuf* rdbuf() const;
    char* str();
private:
    strstreambuf strbuf_;
};

class ostrsstream : public basic_ostream<char>
{
public:
    ostrsstream();
    ostrsstream(char* s, int n, ios_base::openmode mode =
ios_base::out);
    virtual ~ostrsstream();
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
private:
    strstreambuf strbuf_;
};
```

Strstream

Strstreambuf Class

```
class strstream : public basic_iostream<char>
{
public:
    // Types
    typedef
    char                                char_type;
    typedef typename char_traits<char>::int_type int_type;
    typedef typename char_traits<char>::pos_type pos_type;
    typedef typename char_traits<char>::off_type off_type;
    // constructors/destructor
    strstream();
    strstream(char* s, int n, ios_base::openmode mode =
ios_base::in|ios_base::out);
    virtual ~strstream();
    // Members:
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    int pcount() const;
    char* str();
private:
    strstreambuf strbuf_;
};
```

Strstreambuf Class

The class `strstreambuf` is derived from `streambuf` to associate a stream with an in memory character array.

The `strstreambuf` class includes virtual protected and public member functions

- [“freeze” on page 657](#), freezes the buffer
- [“pcount” on page 658](#), determines the buffer size
- [“str” on page 659](#), returns a string
- [“setbuf” on page 660](#), a virtual function to set the buffer
- [“seekoff” on page 660](#), a virtual function for stream offset
- [“seekpos” on page 661](#), a virtual function for stream position
- [“underflow” on page 661](#), a virtual function for input error
- [“pbackfail” on page 661](#), a virtual function for put back error

- [“overflow” on page 662](#), a virtual function for output error

Listing 22.2 The strstreambuf class declaration.

```
class strstreambuf
    : public streambuf
{
public:
    explicit strstreambuf(streamsize alsiz_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t),
                 void (*pfree_arg)(void*));
    strstreambuf(char* gnex_arg, streamsize n, char* pbeg_arg =
0);
    strstreambuf(const char* gnex_arg, streamsize n);
    strstreambuf(signed char* gnex_arg, streamsize n,
                 signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnex_arg, streamsize n);
    strstreambuf(unsigned char* gnex_arg, streamsize n,
                 unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnex_arg, streamsize n);
    virtual ~strstreambuf();
    void   freeze(bool freezefl = true);
    char* str();
    int   pcount() const;
protected:
    virtual int_type overflow (int_type c = EOF);
    virtual int_type pbackfail(int_type c = EOF);
    virtual int_type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                            ios_base::openmode which = ios_base::in |
                            ios_base::out);
    virtual pos_type seekpos(pos_type sp,
                            ios_base::openmode which = ios_base::in |
                            ios_base::out);
    virtual streambuf* setbuf(char* s, streamsize n);
private:
    typedef unsigned char strstate;
    static const strstate allocated = 1 << 0;
    static const strstate constant   = 1 << 1;
    static const strstate dynamic    = 1 << 2;
    static const strstate frozen     = 1 << 3;
    static const streamsize default_alsize = 128;
```

Strstream

Strstreambuf Class

```
streamsize alsizel;  
void* (*palloc_)(size_t);  
void (*pfree_)(void*);  
strrstate strmode_;  
  
void init(char* gnext_arg, streamsize n, char* pbeg_arg = 0);  
};
```

Remarks The template class `streambuf` is an abstract class for deriving various stream buffers whose objects control input and output sequences.

Strstreambuf constructors and Destructors

Default Constructor and Overloaded Constructors

Construct and destruct an object of type `streambuf`.

Dynamic Prototypes

```
explicit strstreambuf(streamsize alsizel_arg = 0);  
strstreambuf(void* (*palloc_arg)(size_t),  
            void (*pfree_arg)(void*));
```

Character Array Prototypes

```
strstreambuf(char* gnext_arg, streamsize n,  
            char* pbeg_arg = 0);  
strstreambuf(const char* gnext_arg, streamsize n);  
strstreambuf(signed char* gnext_arg,  
            streamsize n, signed char* pbeg_arg = 0);  
strstreambuf(const signed char* gnext_arg,  
            streamsize n);  
strstreambuf(unsigned char* gnext_arg,  
            streamsize n, unsigned char* pbeg_arg = 0);  
strstreambuf(const unsigned char* gnext_arg,  
            streamsize n);
```

Remarks The constructor sets all pointer member objects to null pointers.

The `strstreambuf` object is used usually for a intermediate storage object for input and output. The overloaded constructor that is used determines the attributes of the array object when it is created. These might be allocated, or dynamic and are stored in a bitmask type. The first two constructors listed allow for dynamic allocation. The constructors with character array arguments will use that character array for a buffer.

Destructor

To destroy a strstreambuf object.

Prototype `virtual ~strstreambuf();`

Remarks Removes the object from memory.

Strstreambuf Public Member Functions

The public member functions allow access to member functions from derived classes.

freeze

To freeze the allocation of strstreambuf.

Prototype `void freeze(bool freezefl = true);`

Remarks The function `freeze()` stops allocation if the strstreambuf object is using dynamic allocation and prevents the destructor from freeing the allocation. The function `freeze(0)` when used with zero as an argument releases the freeze to allow for destruction.

Return No return

Listing 22.3 Example of strstreambuf::freeze() usage:

```
#include <iostream>
#include <strstream>
#include <string.h>

const int size = 100;

int main()
{
    // dynamic allocation minimum allocation 100
    strstreambuf strbuf(size);

    // add a string and get size
    strbuf.sputn("Metrowerks ", strlen("Metrowerks "));
    cout << "The size of the stream is: "
        << strbuf.pcount() << endl;
```

Strstream

Strstreambuf Class

```
// add a string and get size
strbuf.sputn( "CodeWarrior", strlen("CodeWarrior") );
cout << "The size of the stream is: "
     << strbuf.pcount() << endl;

strbuf.sputc('\0');      // null terminate for output

// now freeze for no more growth
strbuf.freeze();
// try to add more
strbuf.sputn( " -- Software at Work --",
              strlen(" -- Software at Work --"));

cout << "The size of the stream is: "
     << strbuf.pcount() << endl;
cout << "The buffer contains:\n"
     << strbuf.str() << endl;
return 0;
}
```

Result:

```
The size of the stream is: 11
The size of the stream is: 22
The size of the stream is: 23
The buffer contains:
Metrowerks CodeWarrior
```

pcount

To determine the effective length of the buffer,

Prototype **int pcount() const;**

Remarks The function **pcount()** is used to determine the offset of the next character position from the beginning of the buffer.

Return A null terminated character array.

Listing 22.4 Example of *strstreambuf::pcount()* usage.

See: *strstreambuf::freeze*

str

To return the char array stored in the buffer.

Prototype `char* str();`

Remarks The function `str()` freezes the buffer and appends a null character then returns the array. The user is responsible for destruction of any dynamically allocated buffer.

Return A null terminated character array.

Listing 22.5 Example of `strstreambuf::str()` usage

```
#include <iostream>
#include <strstream>

const int size = 100;
char buf[size];
char arr[size] = "Metrowerks CodeWarrior - Software at Work";

int main()
{
    ostrstream ostr(buf, size);
    ostr << arr;

        // associate buffer
    strstreambuf *strbuf(ostr.rdbuf());

        // do some manipulations
    strbuf->pubseekoff(10,ios::beg);
    strbuf->sputc('\0');
    strbuf->pubseekoff(0, ios::beg);

    cout << "The original array was\n" << arr << "\n\n";
    cout << "The strstreambuf array is\n"
        << strbuf->str() << "\n\n";
    cout << "The ostrstream array is now\n" << buf;

    return 0;
}
```

Result:

The original array was
Metrowerks CodeWarrior - Software at Work

The strstreambuf array is
Metrowerks

The ostrstream array is now
Metrowerks

Protected Virtual Member Functions

Protected member functions that are overridden for stream buffer manipulations by the `strstream` class and derived classes from it.

setbuf

To set a buffer for stream input and output sequences.

Prototype `virtual streambuf* setbuf(char* s, streamsize n);`

Remarks The function `setbuf()` is overridden in `strstream` classes.

Return The `this` pointer.

seekoff

To return an offset of the current pointer in an input or output streams.

Prototype `virtual pos_type seekoff(
 off_type off,
 ios_base::seekdir way,
 ios_base::openmode
 which = ios_base::in | ios_base::out);`

Remarks The function `seekoff()` is overridden in `strstream` classes.

Return A `pos_type` value, which is an invalid stream position.

seekpos

To alter an input or output stream position.

Prototype `virtual pos_type seekpos(
 pos_type sp,
 ios_base::openmode
 which = ios_base::in | ios_base::out);`

Remarks The function `seekpos()` is overridden in `strstream` classes.

Return A `pos_type` value, which is an invalid stream position.

underflow

To show an underflow condition and not increment the get pointer.

Prototype `vvirtual int_type underflow();`

Remarks The virtual function `underflow()` is called when a character is not available for input.

There are many constraints for `underflow()`.

- The pending sequence of characters is a concatenation of end pointer minus the get pointer plus some sequence of characters to be read from input.
- The result character if the sequence is not empty the first character in the sequence or the next character in the sequence.
- The backup sequence if the beginning pointer is null, the sequence is empty, otherwise the sequence is the get pointer minus the beginning pointer.

Return The first character of the pending sequence and does not increment the get pointer. If the position is null returns `traits::eof()` to indicate failure.

pbackfail

To show a failure in a put back operation.

Prototype `virtual int_type pbackfail(int_type c = EOF);`

Remarks The resulting conditions are the same as the function `underflow()`.

Return The function `pbackfail()` is only called when a put back operation really has failed and returns `traits::eof`. If success occurs the return is undefined.

overflow

Consumes the pending characters of an output sequence.

Prototype `virtual int_type overflow (int_type c = EOF);`

Remarks The pending sequence is defined as the concatenation of the `put` pointer minus the `beginning` pointer plus either the sequence of characters or an empty sequence, unless the `beginning` pointer is null in which case the pending sequence is an empty sequence.

This function is called by `sputc()` and `sputn()` when the buffer is not large enough to hold the output sequence.

Overriding this function requires that:

- When overridden by a derived class how characters are consumed must be specified.
- After the overflow either the `beginning` pointer must be null or the `beginning` and `put` pointer must both be set to the same non-null value.
- The function may fail if appending characters to an output stream fails or failure to set the previous requirement occurs.

Return The function returns `traits::eof()` for failure or some unspecified result to indicate success.

Istrstream Class

The class `istrstream` is used to create and associate a stream with an array for input.

The `istrstream` class includes the following facilities

- “[Constructors and Destructor](#)” on page 663, to create and remove an `istrstream` object
- “[rdbuf](#)” on page 664, to access the buffer
- “[str](#)” on page 665, returns the buffer

Listing 22.6 The istrstream class declaration

```
class istrstream : public basic_istream<char>
{
public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();
    strstreambuf* rdbuf() const;
    char* str();
private:
    strstreambuf strbuf_;
};
```

Constructors and Destructor

The `istrstream` class has an overloaded constructor.

Constructors

Create an array based stream for input

Prototype `explicit istrstream(const char* s);
explicit istrstream(char* s);
istrstream(const char* s, streamsize n);
istrstream(char* s, streamsize n);`

Remarks The `istrstream` constructor is overloaded to accept a dynamic or pre-allocated character based array for input. It is also overloaded to limit the size of the allocation to prevent accidental overflow

Listing 22.7 Example of usage.

```
#include <iostream>
#include <strstream>

char buf[100] ="double 3.21 string array int 321";

int main()
```

```
{  
    char arr[4][20];  
    double d;  
    long i;  
  
    istrstream istr(buf);  
    istr >> arr[0] >> d >> arr[1] >> arr[2] >> arr[3] >> i;  
  
    cout << arr[0] << " is " << d << "\n"  
        << arr[1] << " is " << arr[2] << "\n"  
        << arr[3] << " is " << i << endl;  
  
    return 0;  
}
```

Result:
double is 3.21
string is array
int is 321

Destructor

To destroy an `istrstream` object.

Prototype `virtual ~istrstream();`

Remarks The `istrstream` destructor removes the `istrstream` object from memory.

Public Member Functions

There are two public member functions.

rdbuf

Returns a pointer to the `strsteambuf`

Prototype `strsteambuf* rdbuf() const;`

| | |
|---------|--|
| Remarks | To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function rdbuf() allows you to retrieve this pointer. |
| Return | A pointer to strstreambuf. |

Listing 22.8 Example of *istrstream::rdbuf()* usage.

See: `strstreambuf::str()`

str

Return a pointer to the stored array.

Prototype `char* str();`

| | |
|---------|--|
| Remarks | The function str() freezes and terminates the character array stored in the buffer with a null character. It then returns the null terminated character array. |
|---------|--|

| | |
|--------|------------------------------|
| Return | A null terminated char array |
|--------|------------------------------|

Listing 22.9 Example of *istrstream::str()* usage.

```
#include <iostream>
#include <strstream>

const int size = 100;
char buf[size] = "Metrowerks CodeWarrior - Software at Work";

int main()
{
    istrstream istr(buf, size);
    cout << istr.str();
    return 0;
}
```

Result:

Metrowerks CodeWarrior - Software at Work

Ostrstream Class

The class `strstreambuf` is derived from `streambuf` to associate a stream with an array buffer for output.

The `ostrstream` class includes the following facilities

- [“Constructors and Destructor” on page 666](#)
- [“freeze” on page 668](#)
- [“pcount” on page 669](#)
- [“rdbuf” on page 670](#)
- [“str” on page 670](#)

Listing 22.10 The ostrstream class declaration

```
class ostrstream : public basic_ostream<char>
{
public:
    ostrstream();
    ostrstream(char* s, int n, ios_base::openmode mode =
ios_base::out);
    virtual ~ostrstream();
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
private:
    strstreambuf strbuf_;
};
```

Constructors and Destructor

The `ostrstream` class has an overloaded constructor.

Constructors

Creates a stream and associates it with a char array for output.

Prototype `ostrstream();`
 `ostrstream(char* s, int n,`

```
ios_base::openmode mode = ios_base::out);
```

Remarks The `ostrstream` array is overloaded for association a pre allocated array or for dynamic allocation.

NOTE When using an `ostrstream` object the user must supply a null character for termination. When storing a string which is already null terminated that null terminator is stripped off to allow for appending.

Listing 22.11 Example of ostrstream usage.

```
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "Ask the teacher anything you want to know" << ends;

    istream inOut(out.rdbuf());
    char c;
    while( inOut.get(c) ) cout.put(c);

    return 0;
}
```

Result:

```
Ask the teacher anything you want to know
```

Destructor

Destroys an `ostrstream` object.

Prototype `virtual ~ostrstream();`

Remarks A `ostrstream` destructor removes the `ostrstream` object from memory.

Public Member Functions

The `ostrstream` class has four public member functions.

freeze

Freezes the dynamic allocation or destruction of a buffer.

Prototype `void freeze(bool freezefl = true);`

Remarks To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer.

Return A pointer to `strstreambuf`.

Listing 22.12 Example of `ostrstream freeze()` usage.

```
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "Metowerks " << 1234;
    out <<     "the size of the array so far is "
        << out.pcount() <<     " characters \n";

    out << " Software" << '\0';
    out.freeze();           //   freezes so no more growth can occur

    out << " at work" << endl;
    out <<     "the final size of the array    is "
        << out.pcount() <<     " characters \n";

    cout << out.str() << endl;

    return 0;
}
```

Result:
the size of the array so far is 14 characters

```
he final size of the array    is 24 characters
Metowerks 1234 Software
```

pcount

Determines the number of bytes offset of the current stream position to the beginning of the array.

Prototype int pcount() const;

Remarks The function pcount() is used to determine the offset of the array. This may not equal to the number of characters inserted due to possible positioning operations.

Return An int_type, the length of the array.

Listing 22.13 Example of ostrstream pcount() usage.

```
#include <iostream>
#include <strstream>

int main()
{
    ostrstream out;
    out << "Metowerks " << 1234    << ends;
    out <<    "the size of the array so far is "
    << out.pcount() <<    " characters \n";

    out << " Software at work" << ends;
    out <<    "the final size of the array    is "
    << out.pcount() <<    " characters \n";

    cout << out.str() << endl;

    return 0;
}
```

Result:

```
the size of the array so far is 15 characters
the final size of the array    is 33 characters
Metowerks 1234
```

rdbuf

To retrieve a pointer to the streams buffer.

Prototype `strstreambuf* rdbuf() const;`

Remarks To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function `rdbuf()` allows you to retrieve this pointer.

Return A pointer to `strstreambuf`.

Listing 22.14 Example of ostrstream rdbuf() usage.

See: `streambuf::pubseekoff()`

str

Returns a pointer to a character array.

Prototype `char* str();`

Remarks The function `str()` freezes any dynamic allocation.

Return A null terminated character array.

Listing 22.15 Example of ostrstream str() usage.

See `ostrstream::freeze()`,

Strstream Class

The class `strstream` is derived from `streambuf` to associate a stream with an array buffer for output

The `strstream` class includes the following facilities

- [“Constructors and Destructor” on page 671](#)
- [“freeze” on page 672](#)
- [“pcount” on page 672](#)
- [“rdbuf” on page 672](#)
- [“str” on page 673](#)

Listing 22.16 The strstream class declaration

```
class strstream : public basic_iostream<char>
{
public:
    // Types
    typedef
    char                                     char_type;
    typedef typename char_traits<char>::int_type int_type;
    typedef typename char_traits<char>::pos_type pos_type;
    typedef typename char_traits<char>::off_type off_type;
    // constructors/destructor
    strstream();
    strstream(char* s, int n, ios_base::openmode mode =
ios_base::in|ios_base::out);
    virtual ~strstream();
    // Members:
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    int pcount() const;
    char* str();
private:
    strstreambuf strbuf_;
};
```

Strstream Types

The strstream class type defines a `char_type`, `int_type`, `pos_type` and `off_type`, for stream positioning and storage.

Constructors and Destructor

Constructors

Creates a stream and associates it with a char array for input and output.

Prototype `strstream();`
 `strstream(char* s, int n, ios_base::openmode mode`
 `=`
 `ios_base::in|ios_base::out);`

Remarks The `strstream` array is overloaded for association a pre allocated array or for dynamic allocation

Destructor

Destroys a `strstream` object.

Prototype `virtual ~strstream();`

Remarks Removes the `strstream` object from memory.

Public Member Functions

The class `strstream` has four public member functions.

freeze

Freezes the dynamic allocation or destruction of a buffer.

Prototype `void freeze(bool freezefl = true);`

Remarks The function `freeze` stops dynamic allocation of a buffer.

pcount

Determines the number of bytes offset of the current stream position to the beginning of the array.

Prototype `int pcount() const;`

Remarks The function `pcount()` is used to determine the offset of the array. This may not equal to the number of characters inserted due to possible positioning operations.

Return An `int_type`, the length of the array.

rdbuf

Retrieves a pointer to the streams buffer

Prototype `strstreambuf* rdbuf() const;`

| | |
|---------|--|
| Remarks | To manipulate a stream for random access or synchronization it is necessary to retrieve a pointer to the streams buffer. The function rdbuf() allows you to retrieve this pointer. |
| Return | A pointer to strstreambuf. |

str

Returns a pointer to a character array.

Prototype `char* str();`

| | |
|---------|--|
| Remarks | The function str() freezes any dynamic allocation. |
| Return | A null terminated character array. |

Return A null terminated character array.

Strstream

Strstream Class

Msl_mutex.h

This chapter is a reference guide to the mutex support library.

The Msl_mutex Support Library

The mutex “mutual exclusion” support library provides the mechanism for implementing thread-safe applications.

It is important to remember that <mutex.h> does not attempt to be a complete thread library. It merely implements just enough for the C++ library’s implementation. Programs with needs greater than this should use the facilities of the native operating system.

The mutex support library consists of the following classes:

- [“Mutex” on page 676](#), a class for implementing a basic mutual exclusion area.
- [“Mutex lock” on page 678](#), an exception safe execution of the mutex object.

Header <msl_mutex.h>

The header mutex.h is used for mutual exclusion to provide thread-safe applications.

Listing 23.1 Class Mutex Synopsis

```
namespace Metrowerks{
class mutex
{
public:
    mutex();
    ~mutex();
    void lock();
```

```
    void unlock( );
private:
    mutex(const mutex&);                                // Not defined
    mutex& operator=(const mutex&);      // Not defined
};
```

Listing 23.2 Class mutex_lock Synopsis

```
class mutex_lock
{
public:
    explicit mutex_lock(mutex& m) : m_(m) {m_.lock();}
    ~mutex_lock() {m_.unlock();}
private:
    mutex& m_;

    mutex_lock(const mutex_lock&);
    mutex_lock& operator=(const mutex_lock&);

}; // end namespace Metrowerks
```

Mutex

The mutex class provides for the basic mutual exclusion mechanism.

```
synopsis class mutex {
public:
    mutex();
    ~mutex();
    void lock();
    void unlock();
private:
    mutex(const mutex&);
    mutex& operator=(const mutex&);
};
```

For example of using the mutex class see: "[Example of mutex lock](#)" on page 679.

Constructors and Assignment Operator

Initialize the mutex object.

Prototype `mutex () ;`
 `private:`
 `mutex(const mutex&);`

A default and a copy constructor are defined.

The copy constructor is declared private and not defined to prevent the `mutex` object from being copied.

Operator=

The assignment operator for the Mutex Class.

Prototype `private:`
 `mutex& operator=(const mutex&);`

The assignment operator is declared private and not defined to prevent the `mutex` object from being copied.

Destructor

Used for implicit mutex destruction.

Prototype `~mutex () ;`

Destroys the `mutex` object.

Public Member Functions

Public members that provide for mutual exclusion.

Lock

Locks the object into exclusion.

Prototype `void lock() ;`

If locked the object is prevented from being used.

For example of using `mutex::lock()` see: "[Example of mutex lock](#)" on [page 679](#).

Unlock

Unlocks the exclusion.

Prototype `void unlock();`

If unlocked the object is allowed to be reused.

Mutex_lock

The class mutex_lock is used so that a mutex object won't accidentally be left locked after an exception is thrown.

Prototype `class mutex_lock {`
 `public:`
 `explicit mutex_lock(mutex& m);`
 `~mutex_lock();`
 `private:`
 `mutex& m_;`
 `mutex_lock(const mutex_lock&);`
 `mutex_lock& operator=(const mutex_lock&);`
 `}`;

Constructors and Assignment Operator

Initialize the mutex_lock object.

Prototype `inline explicit`
 `mutex_lock(mutex& m) : m_(m) m_.lock(); }`
 `private:`
 `mutex_lock(const mutex_lock&);`

The copy constructor is declared private and not defined to prevent the mutex_lock object from being copied.

Operator=

The assignment operator for mutex_lock.

Prototype `mutex_lock& operator=(const mutex_lock&);`

The assignment operator is declared private and not defined to prevent the mutex_lock object from being copied.

Destructor

Destroys the mutex_lock object.

Prototype mutex_lock() {m_.unlock();}

Destroys the mutex_lock object.

Listing 23.3 Example of mutex_lock

```
class A
{
public:
    void foo();
private:
    mutex mut_;
};

void A::foo()
{
    mut_.lock();
    // do some stuff
    mut_.unlock();    // not exception safe!
}
```

If "do some stuff" should throw an exception then mut_ will remain locked with no good way to unlock it. So you should instead use:

```
void A::foo()
{
    mutex_lock lock(mut_);
    // do some stuff
}
```

Msl_mutex.h
Header <msl_mutex.h>

MSL_Utility

This chapter is a reference guide to the General utility support in the Metrowerks standard libraries.

The MSL Utilities Library

This chapter consists of utilities for support of non standard headers.

- [“Basic Compile-Time Transformations” on page 682](#)
- [“Type Query” on page 684](#)
- [“CV Query” on page 685](#)
- [“Type Classification” on page 685](#)
- [“POD classification” on page 688](#)
- [“Miscellaneous” on page 689](#)

The <msl_utility> Header

The purpose of this header is to offer a collection of non-standard utilities collected under the namespace Metrowerks. These utilities are of a fundamental nature, and are typically used in other utilities, rather than top level code. Example usage assumes that a using declaration or directive has been previously issued.

NOTE This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

Concepts and ideas co-developed on Boost

<http://www.boost.org/>

Basic Compile-Time Transformations

A collection of templated strut types which can be used for simple compile-time transformations of types.

remove_const

Will remove the top level const (if present) from a type.

Prototype `typedef typename
 remove_const<T>::type non_const_type;`

The resulting “non_const_type” will be the same as the input type T, except that if T is const qualified, that constant qualification will be removed.

Listing 24.1 Example of remove_const

```
typedef typename remove_const <const int>::type Int;  
Int has type int.
```

remove_volatile

Will remove the top level volatile (if present) from a type.

Prototype `typedef typename
 remove_volatile<T>::type non_volatile_type;`

The resulting “non_volatile_type” will be the same as the input type T, except that if T is volatile qualified, that volatile qualification will be removed.

Listing 24.2 Example of remove_volatile

```
typedef typename remove_volatile <volatile int>::type Int;  
Int has type int.
```

remove_cv

Will remove the top level qualifiers (const and/or volatile, if present) from a type.

Prototype `typedef typename
 remove_cv<T>::type non_qualified_type;`

The resulting “non_qualified_type” will be the same as the input type T, except that if T is cv qualified, the qualifiers will be removed.

Listing 24.3 Example of remove_cv

```
typedef typename remove_cv <const int>::type Int;  
Int has type int.
```

remove_pointer

If given a pointer, returns the type being pointed to. If given a non-pointer type, simply returns the input.

Prototype `typedef typename
 remove_pointer<T>::type pointed_to_type;`

Listing 24.4 Example of remove_pointer

```
typedef typename  
remove_pointer<const int*volatile*const>::type IntPtr;  
typedef typename remove_pointer<IntPtr>::type Int;  
IntPtr will have type type const int*volatile. Int will have the  
type const int.
```

remove_reference

If given a reference, returns the type being referenced. If given a non-reference, simply returns the input.

Prototype `typedef typename
 remove_reference<T>::type referenced_type;`

Listing 24.5 Example of remove_reference

```
typedef typename remove_reference<int&>::type Int;
typedef typename remove_reference<const int&>::type ConstInt;
Int has the type int, and ConstInt has the type const int.
```

remove_bounds

If given an array type, will return the type of an element in the array. If given a non-array type, simply returns the input.

Prototype `typedef typename remove_bounds<T>::type Element;`

Listing 24.6 Example of remove_bounds

```
typedef int IntArray[4];
typedef typename remove_bounds<IntArray>::type Int;
Int has the type int.
```

remove_all

This transformation will recursively remove cv qualifiers, pointers, references and array bounds until the type is a fundamental type, enum, union, class or member pointer.

Prototype `typedef typename remove_all<T>::type fundamental_type;`

Listing 24.7 Example of remove_all

```
typedef const int** Array[4];
typedef typename remove_all<Array*>::type Int;
Int has the type int.
```

Type Query

The following structs perform basic queries on one or more types and return a bool value.

is_same

This struct can be used to tell if two types are the same type or not.

Prototype `bool b = is_same<T, U>::value;`

Listing 24.8 Example of `is_same`

```
bool b = is_same<const int, int>::value;  
The resulting value is false. int and const int are two distinct  
types.
```

CV Query

`is_const`

Returns true if type has a top level const qualifier, else false.

Prototype `bool b = is_const<T>::value;`

Listing 24.9 Example of `is_const`

```
bool b = is_const<const int>::value;  
The resulting value is true.
```

`is_volatile`

Returns true if type has a top level volatile qualifier, else false.

Prototype `bool b = is_volatile<T>::value;`

Listing 24.10 Example of `is_volatile`

```
bool b = is_volatile<const int>::value;  
The resulting value is false.
```

Type Classification

The following structs implement classification as defined by section 3.9 in the C++ standard. All types can be classified into one of ten basic categories:

1. integral
2. floating
3. void
4. pointer

5. member pointer
6. reference
7. array
8. enum
9. union
- 10.class

Top level cv qualifications do not affect type classification. For example, both const int and int are considered to be of integral type.

Prototype `bool b = is_XXX<T>::value;`

where XXX is one of the ten basic categories.

1. is_integral
2. is_floating
3. is_void
4. is_pointer
5. is_member_pointer
6. is_reference
7. is_array
8. is_enum
9. is_union
- 10.is_class

Listing 24.11 Example of is_integral

```
bool b = is_integral<volatile int>::value;  
The value of b is true.
```

The classifications: is_enum and is_union do not currently work automatically. Enumerations and unions will be mistakenly classified as class type. This can be corrected on a case by case basis by specializing is_enum_imp or is_union_imp. These specializations are in the Metrowerks::details namespace.

Listing 24.12 Example of Metrowerks::details namespace

```
enum MyEnum {zero, one, two};

template <>
struct Metrowerks::details::is_enum_imp<MyEnum>
{static const bool value = true;};
```

Now MyEnum will be correctly classified as an enum instead of a class (via the is_enum struct). You do not need to worry about providing this specialization unless you are explicitly using the is_enum query and wanting your enumeration to answer to it correctly.

There are also five “super” categories that are made up of combinations of the ten basic categories:

1. is_arithmetic - is_integral or is_floating
2. is_fundamental - is_arithmetic or is_void
3. is_scalar - is_arithmetic or is_pointer or is_member_pointer or is_enum
4. is_compound - not is_fundamental
5. is_object - anything but a void or reference type

is_extension is also provided for those types that we provide as an extension to the C++ standard. is_extension<T>::value will be false for all types except for long long and unsigned long long.

has_extension is a modified form of is_extension that answers to true if a type either is an extension or contains an extension.

Listing 24.13 Example of is_extension and has_extension

```
is_extension<long long*&>::value; // false
has_extension<long long*&>::value; // true
```

is_signed / is_unsigned

These structs only work on arithmetic types. The type must be constructable by an int and be less-than comparable.

Listing 24.14 Example of `is_signed` and `is_unsigned` use

```
bool b1 = is_signed<char>::value;
bool b2 = is_unsigned<char>::value;
```

NOTE At the risk of restarting a great debate, bool tests as unsigned

POD classification

Four structs classify types as to whether or not they have trivial special members as defined in section 12 of the C++ standard:

- `has_trivial_default_ctor`
- `has_trivial_copy_ctor`
- `has_trivial_assignment`
- `has_trivial_dtor`

This library will answer correctly for non-class types. But user defined class types will always answer false to any of these queries. If you create a class with trivial special members, and you want that class to be able to take advantage of any optimizations that might arise from the assumption of trivial special members, you can specialize these structs:

Listing 24.15 Example of specialized structs

```
template <>
struct Metrowerks::details::class_has_trivial_default_ctor<MyClass>
{ static const bool value = true; };

template <>
struct Metrowerks::details::class_has_trivial_copy_ctor<MyClass>
{ static const bool value = true; };

template <>
struct Metrowerks::details::class_has_trivial_assignment<MyClass>
{ static const bool value = true; };
```

```
template <>
struct Metrowerks::details::class_has_trivial_dtor<MyClass>
    {static const bool value = true;};
```

Note that these specializations need not worry about cv qualifications. The higher level has_trival_XXX structs do that for you.

Finally there is an `is_POD` struct that will answer true if a type answers true on all four of the above queries.

Miscellaneous

compile_assert

This is a compile time assert. This is a very basic version of this idea. Can be used to test assumptions at compile time.

Listing 24.16 Example of compile_assert use

```
#include <msl_utility>

template <class T>
T
foo(const T& t)
{
    Metrowerks::compile_assert<sizeof(T) >= sizeof(int)>
T_Must_Be_At_Least_As_Big_As_int;
    //...
    return t;
}

int main()
{
    int i;
    foo(i); // ok
    char c;
    foo(c); // Error      : illegal use of incomplete struct/union/
class
                //                                'Metrowerks::compile_assert<0>'
```

array_size

Given an array type, you can get the size of the array with `array_size`.

Listing 24.17 Example usage of array_size

```
typedef int Array[10];
size_t n = array_size<Array>::value;
n has the value of 10.
```

can_derive_from

A simple union of class type and union types.

Prototype `bool b = can_derive_from<T>::value;`

store_as - Container optimization

Starting with Pro 4.1 the standard sequences vector, deque and list implemented the “void* optimization” as described in section 13.5 of Stroustrup’s 3rd. In a nutshell, this optimization allows all `Container<T*>` to share an implementation with `Container<void*>` for the purpose of reducing template code bloat.

Starting with Pro 6 containers will take this idea further and with more flexibility. Using `store_as`, one can specify what types can be stored as alternate types in the containers. For example, a `vector<unsigned char>` can be implemented as a `vector<char>`, thus saving on the instantiation of `vector<unsigned char>`. Only vector uses `store_as` in Pro 6, but other containers will have this capability in future releases.

Prototype `template <> struct store_as<Type_to_be_optimized> {typedef Implementaiton_Type type;};`

For example, to specify that a `container<long>` be implemented in terms of a `container<unsigned long>`:

```
template <> struct store_as<long>
{typedef unsigned long type;};
```

If a specialization of `store_as` does not appear for a type, then that type will be implemented as itself in containers.

This header has a default table of `store_as` specializations suitable for your platform. But if for some reason you are not happy with the shipping configuration, you can alter the behavior here.

Additionally, this optimization can be turned off by #defining `_Inhibit_Container_Optimization` in `<mslconfig>`, or in a prefix file.

The requirements for a type to appear in a `store_as` specialization are:

- It must have a trivial copy constructor, assignment operator and destructor.
- Its default constructor must do nothing but cause all bytes in the object to be zeroed (if this constructor is used).
- The two types in a `store_as` specialization must have the same `sizeof`.

User defined types can also take advantage of this optimization if they meet the above requirements (recommended only for those that really know what they are doing). Client code must declare that the user defined type is a POD, and then create a `store_as` entry. For example, here is a struct that will share its vector implementation with `vector<unsigned long>`:

Listing 24.18 A struct that shares its vector implementation

```
#include <msl_utility>
#include <vector>

struct A
{
    int data_;
};

template <> struct Metrowerks::is_POD<A>
    {static const bool value = true;};
template <> struct Metrowerks::store_as<A>
    {typedef unsigned long type;};

int main()
{
    std::vector<A> a(5);
```

MSL_Utility

The <msl_utility> Header

```
// Shares implementation with vector<unsigned long>
}
```

Note that this is strictly a code size optimization. Functionality does not change. And it only saves on code size if you already have to use a `vector<unsigned long>` for some other reason (or if you are using a vector that is stored as a `vector<unsigned long>`).

call_traits

This struct is a collection of type definitions that ease coding of template classes when the template parameter may be a non-array object, an array, or a reference. The type definitions specify how to pass a type into a function, and how to pass it back out either by value, reference or const reference. The interface is:

```
call_traits<T>::value_type
call_traits<T>::reference
call_traits<T>::const_reference
call_traits<T>::param_type
```

The first three types are suggestions on how to return a type from a function by value, reference or const reference. The fourth type is a suggestion on how to pass a type into a method.

The `call_traits` struct is most useful in avoiding references to a reference which are currently illegal in C++. Another use is in helping to decay array-type parameters into pointers. In general, use of `call_traits` is limited to advanced techniques, and will not require specializations of `call_traits` to be made. For example uses of `call_traits` see `compressed_pair`. For an example specialization see `alloc_ptr`.

is_empty

Answers true if the type is a class or union that has no data members, otherwise answers false. This is a key struct for determining if the space for an “empty” object can be optimized away or not.

```
Prototype    bool b = is_empty<T>::value;
```

compressed_pair

Like std::pair, but attempts to optimize away the space for either the first or second template parameter if the type is "empty". And instead of the members being accessible via the public data members first and second, they are accessible via member methods first() and second(). compressed_pair handles reference types as well as other types thanks to the call_traits template. This is a good example to study if you're wanting to see how to take advantage of either call_traits or is_empty. To see an example of how compressed_pair is used see alloc_ptr.

Listing 24.19 Synopsis of compressed_pair

```
template <class T1, class T2>
class compressed_pair<T1, T2>
{
public:
    typedef
    T1
        first_type;
    typedef
    T2
        second_type;
    typedef typename call_traits<first_type>::param_type
    first_param_type;
    typedef typename
    call_traits<second_type>::param_type           second_param_type;
    typedef typename
    call_traits<first_type>::reference            first_reference;
    typedef typename call_traits<second_type>::reference
    second_reference;
    typedef typename
    call_traits<first_type>::const_reference     first_const_reference;
    typedef typename call_traits<second_type>::const_reference
    second_const_reference;

    compressed_pair();
    compressed_pair(first_param_type x, second_param_type y);
    explicit compressed_pair(first_param_type x);
    explicit compressed_pair(second_param_type y);
```

MSL_Utility

The `<msl_utility>` Header

```
first_reference           first();
first_const_reference first() const;

second_reference          second();
second_const_reference second() const;

void swap(compressed_pair& y);
};

template <class T1, class T2> void swap
(compressed_pair<T1, T2>& x, compressed_pair<T1, T2>& y);
```

Use of the single argument constructors will fail at compile time (ambiguous call) if `first_type` and `second_type` are the same type.

The swap specialization will call swap on each member if and only if its size has not been optimized away. The call to swap on each member will look both in std, and in the member's namespace for the appropriate swap specialization. Thus clients of `compressed_pair` need not put swap specializations into namespace std.

A good use of `compressed_pair` is in the implementation of a container that must store a function object. Function objects are typically zero-sized classes, but are also allowed to be ordinary function pointers. If the function object is a zero-sized class, then the container can optimize its space away by using it as a base class. But if the function object instantiates to a function pointer, it can not be used as a base class. By putting the function object into a `compressed_pair`, the container implementor need not worry whether it will instantiate to a class or function pointer.

Listing 24.20 Example of compressed_pair

```
#include <iostream>
#include <functional>
#include <msl_utility>

template <class T, class Compare>
class MyContainer
{
public:
```

```
    explicit MyContainer(const Compare& c = Compare()) : data_(0,
c) {}

        T*                      pointer()                  {return
data_.first();}

        const T*                 pointer() const          {return
data_.first();}

        Compare&                compare()                {return
data_.second();}

        const Compare& compare() const          {return
data_.second();}

        void                     swap(MyContainer& y)
{data_.swap(y.data_);}

private:
    Metrowerks::compressed_pair<T*, Compare> data_;
};

int main()
{
    typedef MyContainer<int, std::less<int> >      MyContainer1;
    typedef MyContainer<int, bool (*)(int, int)> MyContainer2;
    std::cout << sizeof(MyContainer1) << '\n';
    std::cout << sizeof(MyContainer2) << '\n';
}
```

MyContainer1 uses a zero-sized Compare object. On a 32 bit machine, the sizeof MyContainer1 will be 4 bytes as the space for Compare is optimized away by compressed_pair. But MyContainer2 instantiates Compare with an ordinary function pointer which can't be optimized away. Thus the sizeof MyContainer2 is 8 bytes.

alloc_ptr

An extension of std::auto_ptr. alloc_ptr will do everything that auto_ptr will do with the same syntax. Additionally alloc_ptr will deal with array new/delete:

Listing 24.21 Alloc_ptr will deal with array new/delete

```
alloc_ptr<int, array_deleter<int> > a(new int[4]);  
// Ok, destructor will use delete[]
```

By adding the `array_deleter<T>` template parameter you can enable `alloc_ptr` to correctly handle pointers to arrays of elements.

`alloc_ptr` will also work with allocators which adhere to the standard interface. This comes in very handy if you are writing a container that is templated on an allocator type. You can instantiate an `alloc_ptr` to work with an allocator with:

```
alloc_ptr<T, Allocator<T>, typename Allocator<T>::size_type> a;
```

The third parameter can be omitted if the allocator is always going to allocate and deallocate items one at a time (e.g. node based containers).

`alloc_ptr` takes full advantage of `compressed_pair` so that it is as efficient as `std::auto_ptr`. The `sizeof(alloc_ptr<int>)` is only one word. Additionally `alloc_ptr` will work with a reference to an allocator instead of an allocator (thanks to `call_traits`). This is extremely useful in the implementation of node based containers.

Listing 24.22 Synopsis of class alloc_ptr

```
template<class T, class Allocator = single_deleter<T>,  
        class Size =  
number<call_traits<Allocator>::value_type::size_type, 1> >  
class alloc_ptr  
{  
public:  
    typedef T element_type;  
  
    typedef typename  
call_traits<Allocator>::value_type           allocator_type;  
    typedef typename  
call_traits<Allocator>::param_type          allocator_param_type;  
    typedef typename call_traits<Allocator>::reference  
allocator_reference;  
    typedef typename call_traits<Allocator>::const_reference  
allocator_const_reference;
```

```
typedef typename allocator_type::size_type size_type;
typedef typename allocator_type::difference_type difference_type;
typedef typename allocator_type::pointer pointer;
typedef typename allocator_type::const_pointer const_pointer;
typedef typename allocator_type::reference reference;
typedef typename allocator_type::const_reference const_reference;

explicit alloc_ptr(pointer p = 0);
alloc_ptr(pointer p, allocator_param_type alloc, Size sz =
Size());
alloc_ptr(alloc_ptr& x);
template<class U>
alloc_ptr(alloc_ptr<U, allocator_type::rebind<U>::other,
Size>& x);
~alloc_ptr();
alloc_ptr& operator =(alloc_ptr& x);
template<class U>
alloc_ptr& operator=(alloc_ptr<U,
allocator_type::rebind<U>::other, Size>& x);
reference operator*() const;
pointer operator->() const;
reference operator[](difference_type n);
reference operator[](size_type n);
const_reference operator[](difference_type n) const;
const_reference operator[](size_type n) const;
pointer get() const;
pointer release();
void reset(pointer p = 0, Size size);
alloc_ptr(alloc_ptr_ref<T, Allocator, Size> r);
alloc_ptr& operator=(alloc_ptr_ref<T, Allocator, Size> r);
template<class U> operator alloc_ptr_ref<U,
allocator_type::rebind<U>::other, Size>();
template<class U> operator alloc_ptr<U,
allocator_type::rebind<U>::other, Size>();
allocator_reference allocator();
allocator_const_reference allocator() const;
```

MSL_Utility

The <msl_utility> Header

```
Size&          get_size();
size_type     capacity() const;
};
```

This is essentially the std::auto_ptr interface with a few twists to accommodate allocators and size parameters.

MSL C++ Debug Mode

This chapter describes the MSL Debug Mode for code diagnostics.

Overview of MSL C++ Debug Mode

The STL portion of MSL C++ has a debug mode that can be used to diagnose common mistakes in code that uses the MSL C++ containers and their iterators. When an error is detected, a `std::logic_error` is thrown with an appropriate error message.

Types of Errors Detected

Given a container (such as `vector`), the following errors are detected in MSL Debug mode:

- Incrementing an iterator beyond `end()`.
- Decrementing an iterator before `begin()`.
- Dereferencing an iterator that is not dereferenceable.
- Any use of an invalid iterator besides assigning a valid value to it.
- Passing an iterator to a container method when that iterator does not point into that container.
- Comparison of two iterators that don't point into the same container.

How to Enable Debug Mode

To enable MSL C++ Debug mode simply uncomment this line in the MSL Configuration header `<mslconfig>` See “[C++ Switches, Flags and Defines](#)” on page 745 for more information.

MSL C++ Debug Mode

Overview of MSL C++ Debug Mode

```
#define _MSL_DEBUG
```

Alternatively you can `#define _MSL_DEBUG` in a prefix file. Either way, you must rebuild your C++ library after flipping this switch. Convenience projects are provided under `MSL(MSL_Build_Projects)/` to make this task easier. After rebuilding the C++ library, rebuild your application and run it. If there are any errors, a `std::logic_error` will be thrown. If exceptions are disabled, then instead the error function `_msl_error(const char*)` is called. This function can be defined by client code. There are some sample implementations in `<mslconfig>`. The default simply calls `fprintf` and `abort`.

Debug Mode Implementations

The debug facilities are available for the standard containers as well as the Metrowerks extension containers:

- [“deque” on page 703](#)
- [“list” on page 704](#)
- [“string” on page 705](#)
- [“vector” on page 706](#)
- [“tree-based containers - map, multimap, set, multiset” on page 707](#)
- [“cdeque” on page 707](#)
- [“slist” on page 708](#)
- [“hash-based containers - map, multimap, set, multiset” on page 709](#)

Each container has methods that will invalidate some or all outstanding iterators. If those iterators are invalidated, then their use (except for assigning a new valid iterator) will generate an error. An iterator is considered invalidated if it no longer points into the container, or if the container's method silently causes the iterator to point to a new element within the container. Some methods (such as `swap`, or `list::splice`) will transfer ownership of outstanding iterators from one container to another, but otherwise leave them valid.

Listing 25.1 Example of dereference at end:

```
#include <iostream>
#include <vector>
#include <stdexcept>

int main()
{
    try
    {
        std::vector<int> v(10);
        std::vector<int>::iterator i = v.begin() + 9;
        *i = 9; // ok
        ++i; // ok
        *i = 10; // error
    } catch (std::exception& e)
    {
        std::cerr << e.what() << '\n';
    }
    catch (...)
    {
        std::cerr << "Unknown exception caught\n";
    }
}
```

In this example the iterator `i` is incremented to the end of the vector and then dereferenced and assigned through. In release mode this is undefined behavior and may overwrite other important information in your application. However in debug mode this example prints out:

MSL DEBUG: dereferenced invalid iterator

Listing 25.2 Example of iterator/list mismatch:

```
#include <iostream>
#include <list>
#include <stdexcept>

int main()
{
    try
```

MSL C++ Debug Mode

Overview of MSL C++ Debug Mode

```
{  
    std::list<int> l1(10), l2(10);  
    std::list<int>::iterator i = l1.begin();  
    l2.erase(i); // error  
}  
catch (std::exception& e)  
{  
    std::cerr << e.what() << '\n';  
}  
catch (...)  
{  
    std::cerr << "Unknown exception caught\n";  
}  
}
```

An iterator is initialized to point into the first list. But then this iterator is mistakenly used to erase an element from a second list. This is normally undefined behavior. In debug mode this example prints out:

```
MSL DEBUG: invalid iterator given to list
```

Listing 25.3 Example of use of invalidated iterator:

```
#include <iostream>  
#include <deque>  
#include <stdexcept>  
  
int main()  
{  
    try  
    {  
        std::deque<int> d(10);  
        std::deque<int>::iterator i = d.begin(), e = d.end();  
        for (; i != e; ++i)  
            d.push_back(0);  
    }  
    catch (std::exception& e)  
    {  
        std::cerr << e.what() << '\n';  
    }  
    catch (...)
```

```
{  
    std::cerr << "Unknown exception caught\n";  
}  
}
```

The `push_back` method on `deque` invalidates all iterators. When the loop goes to increment `i`, it is operating on an invalidated iterator. This is normally undefined behavior. In debug mode this example prints out:

```
MSL DEBUG: increment end or invalid iterator
```

Debug Mode Containers

The list below documents when iterators are invalidated for each container, and for each method in that container:

deque

assign

All `assign` methods (including `operator=`) invalidate all iterators.

push_front/back

Invalidate all iterators.

pop_front/back

Only the iterators to the erased elements are invalidated.

insert

All iterators are invalidated.

erase

If erasing at either end, only iterators to elements erased are invalidated, else all iterators are invalidated.

resize

If the size increases, all iterators are invalidated. Else only iterators to the erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

additional notes

The index operator is range checked just like the at() method.

list**assign**

All assign methods (including operator=) invalidate all iterators.

push_front/back

No iterators are invalidated.

pop_front/back

Only the iterators to the erased elements are invalidated.

insert

No iterators are invalidated.

erase

Only the iterators to the erased elements are invalidated.

resize

Only the iterators to the erased elements are invalidated.

clear

Invalidate all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

splice, merge

Iterators remain valid, but iterators into the argument list now point into this.

string

assign

All assign methods (including operator=) invalidate all iterators.

push_back

If capacity is not exceeded no iterators are invalidated, else all iterators are invalidated.

pop_back

Only the iterators to the erased element is invalidated.

insert

If capacity is not exceeded iterators to elements beyond the insertion point are invalidated, else all iterators are invalidated.

erase

Iterators to elements at and beyond the erased elements are invalidated.

resize

If capacity is exceeded all iterators are invalidated, else iterators to any erased elements are invalidated.

clear

Invalidate all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

additional notes

The index operator is range checked just like the at() method.

vector**assign**

All assign methods (including operator=) invalidate all iterators.

push_back

If capacity is not exceeded no iterators are invalidated, else all iterators are invalidated.

pop_back

Only the iterators to the erased element is invalidated.

insert

If capacity is not exceeded iterators to elements beyond the insertion point are invalidated, else all iterators are invalidated.

erase

Iterators to elements at and beyond the erased elements are invalidated.

resize

If capacity is exceeded all iterators are invalidated, else iterators to any erased elements are invalidated.

clear

Invalidate all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

additional notes

The index operator is range checked just like the at() method.

tree-based containers - map, multimap, set, multiset

assign

Invalidate all iterators.

insert

No iterators are invalidated.

erase

Only the iterators to the erased elements are invalidated.

clear

Invalidate all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

cdeque

assign

All assign methods (including operator=) invalidate all iterators.

push_front/back

If capacity exceeded invalidates all iterators, else no iterators are invalidated.

pop_front/back

Only the iterators to the erased elements are invalidated.

insert

If capacity exceeded or if insert position is not at the front or back, invalidates all iterators, else no iterators are invalidated.

erase

If erasing at either end, only iterators to elements erased are invalidated, else all iterators are invalidated.

resize

If capacity exceeded invalidates all iterators, else iterators to any erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

additional notes

The index operator is range checked just like the at() method.

slist**assign**

All assign methods (including operator=) invalidate all iterators.

push_front/back

No iterators are invalidated.

pop_front/back

Only the iterators to the erased elements are invalidated.

insert

No iterators are invalidated.

erase

Only the iterators to the erased elements are invalidated.

resize

Only the iterators to the erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

splice, splice_after, merge

Iterators remain valid, but iterators into the argument list now point into this.

additional notes

Incrementing end() is not an error, it gives you begin().

hash-based containers - map, multimap, set, multiset

assign

Invalidates all iterators.

insert

If `load_factor()` attempts to grow larger than `load_factor_limit()`, then the table is rehashed which invalidates all iterators, else no iterators are invalidated.

erase

Only the iterators to the erased elements are invalidated.

clear

Invalidates all iterators.

swap

Iterators remain valid, but they now point into the swapped container.

Invariants

In addition to the iterator checking described above, each container (except `string`) has a new member method:

```
bool invariants() const;
```

This method can be called at any time to assess the container's class invariants. If the method returns false, then the container has somehow become corrupted and there is a bug (most likely in client code, but anything is possible). If the method returns true, then no errors have been detected. This can easily be used in debug code like:

```
#include <vector>
#include <cassert>

int main()
{
    int iarray[4];
    std::vector<int> v(10);
    assert(v.invariants());
    for (int i = 0; i <= 4; ++i)
        iarray[i] = 0;
    assert(v.invariants());
```

}

The for loop indexing over `iarray` goes one element too far and steps on the vector. The assert after the loop detects that the vector has been compromised and fires.

Be warned that the `invariants` method for some containers can have a significant computational expense ($O(N)$), so this method is not advised for release code (nor are any of the debug facilities).

MSL C++ Debug Mode

Debug Mode Containers

Hash Libraries

This chapter is a reference guide to the hash support in the Metrowerks standard libraries.

The Hash Containers Library

This chapter on Metrowerks implementation of hashes is made up of.

- [“General Hash Issues” on page 713](#)
- [“Hash set” on page 723](#)
- [“Hash map” on page 732](#)
- [“Hash fun” on page 742](#)

A separate chapter [“The <msl_utility> Header” on page 681](#) is also useful when understanding the methodology.

General Hash Issues

This document reflects issues that are common to `hash_set`, `hash_multiset`, `hash_map` and `hash_multimap`. Rather than repeat each of these issue for each of the four hash containers, they are discussed here once and for all.

Introduction

These classes are analogous to `std::set`, `std::multiset`, `std::map` and `std::multimap`, but are based on a hash table. The design and implementation of these classes has the following goals:

- High CPU performance
- Minimum memory usage

- Ease of use
- Control over hashing details
- Backward compatibility with previous Metrowerks hash containers
- Compatibility with hash containers supplied by SGI and Microsoft

Not all of these goals can be simultaneously met. For example, optimizations often require a trade-off between size and speed. “Ease of use” can pull the design in opposite directions from “control over details”. And it is not possible to be 100% compatible with two or more other implementations, when they are not compatible among themselves. Nevertheless, thought and concessions have been made toward all of these goals.

Namespace Issues

These classes are a Metrowerks extension to the standard C++ library. So they have been implemented within the namespace Metrowerks. There are several techniques available for accessing these classes:

Fully Qualified Reference:

One technique is to fully qualify each use of a Metrowerks extension with the full namespace. For example:

Listing 26.1 Qualified Reference

```
#include <hash_set>

int main()
{
    Metrowerks::hash_set<int> a;
}
```

Namespace Alias

“Metrowerks” is quite a long name and can get tiresome continually typing. But it is not likely to conflict with other library's

namespaces. You can easily shorten the Metrowerks namespace while still retaining the protection of namespaces through the use of an alias. For example, here is how to refer to the Metrowerks namespace as “mw”:

Listing 26.2 Namespace Alias

```
#include <hash_map>

namespace mw = Metrowerks;

int main()
{
    mw::hash_map<int, int> a;
}
```

The short name “mw” is much more likely to conflict with other's libraries, but as the implementor of your code you can choose your aliases such that there is no conflict.

Using Declaration

Using declarations can bring individual names into the current namespace. They can be used either at namespace scope (outside of functions) or at function scope (inside of functions). Here is an example use of a using declaration at namespace scope:

Listing 26.3 Namespace Scope

```
#include <hash_set>

using Metrowerks::hash_multiset;

int main()
{
    hash_multiset<int> a;
}
```

Anywhere below the using declaration, `hash_set` can be referred to without the use of the Metrowerks qualifier.

Using Directive

Using directives will import every name in one namespace into another. These can be used to essentially “turn off” namespaces so that you don't have to deal with them. They can be used at namespace scope, or to limit their effect, can also be used at function scope. For example:

Listing 26.4 Function Scope

```
#include <hash_map>

int main()
{
    using namespace Metrowerks;
    hash_multimap<int, int> a;
}
```

In the above example, any name in the Metrowerks namespace can be used in main without qualification.

Compatibility Headers

Most headers with the name <name> have an associated compatibility header <name.h>. These compatibility headers simply issue using declarations for all of the names they contain. Here is an example use:

Listing 26.5 Using Declarations for Names

```
#include <hash_set.h>
#include <hash_map.h>

int main()
{
    hash_set<int> a;
    hash_map<int, int> b;
}
```

Constructors

Each hash container has a constructor which takes the following arguments, with the following defaults:

```
size_type num_buckets = 0
const key_hasher& hash = key_hasher()
const key_compare& comp = key_compare()
float load_factor_limit = 2
float growth_factor = 4
const allocator_type& a = allocator_type()
```

Since all arguments have defaults, the constructor serves as a default constructor. It is also declared explicit to inhibit implicit conversions from the first argument: `size_type`. The first argument is a way to specify the initial number of buckets. This was chosen as the first parameter in order to remain compatible both with previous versions of Metrowerks hash containers, as well as the SGI hash containers.

The second and third parameters allow client code to initialize the hash and compare function objects if necessary. This will typically only be necessary if ordinary function pointers are being used. When function objects are used, the default constructed function object is often sufficient.

The fourth and fifth parameters allow you to set the initial values of `load_factor_limit` and `growth_factor`. Details on how these parameters interact with the `size()` and `bucket_count()` of the container can be found in the capacity section.

A second constructor also exists that accepts templated input iterators for constructing a hash container from a range. After the pair of iterators, the 6 parameters from the first constructor follow in the same order, and with the same defaults.

Iterator Issues

The hash iterators are of the forward type. You can increment them via prefix or postfix `++`, but you can not decrement them. This is compatible with our previous implementation of the hash containers, and with the hash containers provided by SGI. But the hash iterators provided by Microsoft are bidirectional. Code that

takes advantage of the decrement operators offered by Microsoft will fail at compile time in the Metrowerks implementation.

Forward iterators were chosen over bidirectional iterators to save on memory consumption. Bidirectional iterators would add an additional word of memory to each entry in the hash container. Furthermore a hash container is an unordered collection of elements. This “unorder” can even change as elements are added to the hash container. The ability to iterate an unordered collection in reverse order has a diminished value.

Iterators are invalidated when the number of buckets in the hash container change. This means that iteration over a container while adding elements must be done with extra care (see Capacity for more details). Despite that iterators are invalidated in this fashion, pointers and references into the hash container are never invalidated except when the referenced element is removed from the container.

Capacity

empty, size and max_size have semantics identical with that described for standard containers.

The load factor of a hash container is the number of elements divided by the number of buckets:

$$\text{load_factor} = \frac{\text{size}()}{\text{bucket_count}()}$$

During the life time of a container, the load factor is at all times less than or equal to the load factor limit:

$$\frac{\text{size}()}{\text{bucket_count}()} \leq \text{load_factor_limit}()$$

This is a class invariant. When both size() and bucket_count() are zero, the load_factor is interpreted to be zero. size() can not be greater than zero if bucket_count() is zero. Client code can directly or indirectly alter size(), bucket_count() and load_factor_limit(). But at all times, bucket_count() may be adjusted so that the class invariant is not compromised.

- If client code increases size() via methods such as insert such that the invariant is about to be violated, bucket_count() will be increased by growth_factor().
- If client code decreases size() via methods such as erase, the invariant can not be violated.
- If client code increases load_factor_limit(), the invariant can not be violated.
- If client code decreases load_factor_limit() to the point that the invariant would be violated, then bucket_count() will be increased just enough to satisfy the invariant.
- If client code increases bucket_count(), the invariant can not be violated.
- If client code decreases bucket_count() to the point that the invariant would be violated, then bucket_count() will be decreased only to the minimum amount such that the invariant will not be violated.

The final item in the bulleted list results amounts to a “shrink to fit” statement.

```
myhash.bucket_count(0); // shrink to fit
```

The above statement will reduce the bucket count to the point that the load_factor() is just at or below the load_factor_limit().

```
bucket_count()  
returns the current number of buckets in the container.
```

bucket_count(size_type num_buckets) sets the number of buckets to the first prime number that is equal to or greater than num_buckets, subject to the class invariant described above. It returns the actual number of buckets that were set. This is a relatively expensive operation as all items in the container must be rehashed into the new container. This routine is analogous to vector's reserve. But it does not reserve space for a number of elements. Instead it sets the number of buckets which in turn reserves space for elements, subject to the setting of load_factor_limit().

```
load_factor()  
returns size()/bucket_count() as a float.  
load_factor_limit()  
returns the current load_factor_limit.
```

`load_factor_limit(float lf)` sets the load factor limit. If the new load factor limit is less than the current load factor limit, the number of buckets may be increased if the new load factor limit would violate the class invariant as described above. You can completely block the automatic change of `bucket_count` with:

```
myhash.load_factor_limit(INFINITY);
```

This may be important if you are wanting outstanding iterators to not be invalidated while inserting items into the container. The argument to `load_factor_limit` must be positive, else an exception of type `std::out_of_range` is thrown.

The `growth_factor` functions will read and set the `growth_factor`. When setting, the new growth factor must be greater than 1 else an exception of type `std::out_of_range` is thrown.

The `collision(const_iterator)` method will count the number of items in the same bucket with the referred to item. This may be helpful in diagnosing a poor hash distribution.

Insert For Unique Hashed Containers

insert

`hash_set` and `hash_map` have the following insert method:

```
std::pair<iterator, bool>
insert(const value_type& x);
```

If `x` does not already exist in the container, it will be inserted. The returned iterator will point to the newly inserted `x`, and the `bool` will be true. If `x` already exists in the container, the container is unchanged. The returned iterator will point to the element that is equal to `x`, and the `bool` will be false.

```
iterator insert(iterator, const value_type& x);
```

Operates just like the version taking only a `value_type`. The `iterator` argument is ignored. It is only present for compatibility with standard containers.

```
template <class InputIterator> void insert
(InputIterator first, InputIterator last);
```

Inserts those elements in [first, last) that don't already exist in the container.

Insert For Multi Hashed Containers

insert

The functions `hash_multiset` and `hash_multimap` have the following insert method:

Prototype `iterator insert(const value_type& x);`
`iterator insert(iterator p, const value_type& x);`
`template <class InputIterator> void insert`
 `(InputIterator first, InputIterator last);`

In the first `insert` prototype `x` is inserted into the container and an iterator pointing to the newly inserted value is returned. If values equal to `x` already exist in the container, then the new element is inserted after all other equal elements. This ordering is stable throughout the lifetime of the container.

In the second prototype `insert` first checks to see if `*p` is equivalent to `x` according to the compare function. If it is, then `x` is inserted before `p`. If not then `x` is inserted as if the `insert` without an iterator was used. An iterator is returned which points to the newly inserted element.

The final `insert` prototype inserts [first, last) into the container. Equal elements will be ordered according to which was inserted first.

erase

Prototype `void erase(iterator position);`
`size_type erase(const key_type& x);`
`void erase(iterator first, iterator last);`

The first `erase` function erases the item pointed to by `position` from the container. The second erases all items in the container that compare equal to `x` and returns the number of elements erased. The third `erase` erases the range [first, last) from the container.

Prototype `swap(hash_set& y);`

Swaps the contents of *this with y in constant time.

Prototype `clear();`

Erases all elements from the container.

Observers

Prototype `get_allocator() const;`

Returns the allocator the hash container was constructed with.

Prototype `key_comp() const`

Returns the comparison function the hash container was constructed with.

Prototype `value_comp() const`

Returns the comparison function used in the underlying hash table.
For hash_set and hash_multiset, this is the same as key_comp().

Prototype `key_hash()`

Returns the hash function the hash container was constructed with.

Prototype `value_hash()`

Returns the hash function used in the underlying hash table. For hash_set and hash_multiset, this is the same as key_hash().

Set Operations

Prototype `iterator find(const key_type& x) const;`

Returns an iterator to the first element in the container that is equal to x, or if x is not in the container, returns end().

Prototype `count(const key_type& x) const`

Return Returns the number of elements in the container equal to x.

Prototype `std::pair<iterator, iterator> equal_range(const key_type& x);`

Returns a pair of iterators indicating a range in the container such that all elements in the range are equal to x. If no elements equal to x are in the container, an empty range is returned.

Global Methods

Prototype `swap(x, y)`

Same semantics as `x.swap(y)`.

Prototype `operator == (x, y)`

Return Returns true if `x` and `y` contain the same elements in the same order. To accomplish this they most likely must have the same number of buckets as well.

Prototype `operator != (x, y)`

Return Returns `!(x == y)`

Incompatibility with Previous versions Metrowerks Hash Containers

The current hash containers are very compatible with previous versions except for a few methods:

You can no longer compare two hash containers with the ordering operators: `<`, `<=`, `>`, `>=`. Since hash containers are unordered sets of items, such comparisons have little meaning.

`lower_bound` is no longer supported. Use `find` instead if you expect the item to be in the container. If not in the container, `find` will return `end()`. As there is no ordering, finding the position which an item could be inserted before has no meaning in a hash container.

`upper_bound` is no longer supported. Again because of the fact that hash containers are unordered, `upper_bound` has questionable semantics.

Despite the lack of `lower_bound` and `upper_bound`, `equal_range` is supported. In a pinch, `equal_range().first` suffices for `lower_bound`, and `equal_range().second` suffices for `upper_bound`.

Hash_set

This header contains two classes:

- `hash_set`

- hash_multiset.

hash_set is a container that holds an unordered set of items, and no two items in the container can compare equal. hash_multiset permits duplicate entries. Also see the General Hash Issues Introduction.

NOTE

This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

Introduction.

These containers are in the namespace Metrowerks. See Namespace Issues for details and hints about how to best take advantage of this fact.

hash_set and hash_multiset are largely compatible with previous versions of these classes which appeared in namespace std. But see Incompatibility for a short list of incompatibilities.

Old HashSet Headers

Previous versions of CodeWarrior placed hash_set and hash_multiset in the headers <hashset.h> and <hashmset.h> respectively. These headers are still available, but should be used only for transition purposes. They will disappear in a future release. These headers import the contents of <hash_set> into the std namespace (as previous versions of hash_(multi)set were implemented in std).

Listing 26.6 Old HashSet Headers

```
#include <hashset.h>

int main()
{
    std::hash_set<int> a;
}
```

Template Parameters

Both `hash_set` and `hash_multiset` have the following template parameters and defaults:

Listing 26.7 Template Parameters and Defaults

```
template <class T, class Hash = hash<T>, class Compare =
    std::equal_to<T>, class Allocator = std::allocator<T> >
class hash_(multi)set;
```

The first parameter is the type of element the set is to contain. It can be almost any type, but must be copyable.

The second parameter is the hash function used to look up elements. It defaults to the hash function in `<hash_fun>`. Client code can use `hash<T>` as is, specialize it, or supply completely different hash function objects or hash function pointers. The hash function must accept a `T`, and return a `size_t`.

The third parameter is the comparison function which defaults to `std::equal_to<T>`. This function should have equality semantics. A specific requirement is that if two keys compare equal according to `Compare`, then they must also produce the same result when processed by `Hash`.

The fourth and final parameter is the allocator, which defaults to `std::allocator<T>`. The same comments and requirements that appear in the standard for allocators apply here as well.

Nested Types

`hash_set` and `hash_multiset` define a host of nested types similar to standard containers. Several noteworthy points:

- `key_type` and `value_type` are the same type and represent the type of element stored.
- `key_hasher` and `value_hasher` are the same type and represent the hash function.
- `key_compare` and `value_compare` are the same type and represent the comparison function.

-
- iterator and const_iterator are the same type and have semantics common to a forward const_iterator.

Iterator Issues

See Iterator Issues that are common to all hash containers.

Iterators of hash_set and hash_multiset are not mutable. They act as const_iterators. One can cast away the const qualification of references returned by iterators, but if the element is modified such that the hash function now has a different value, the behavior is undefined.

See Capacity for details on how to control the number of buckets.

hash_set

hash_set is a container based on a hash table that supports fast find, insert and erase. The elements in a hash_set are unordered. A hash_set does not allow multiple entries of equivalent elements.

Listing 26.8 hash_set synopsis

```
namespace Metrowerks {

template <class T, class Hash = hash<T>, class Compare =
std::equal_to<T>,
          class Allocator = std::allocator<T> >
class hash_set
{
public:
    //  types:
    typedef
T                                         key_type;
    typedef
T                                         value_type;
    typedef
Hash                                      key_hasher;
};
```

```
typedef value_hash
Hasher;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type difference_type;
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef Compare key_compare;
typedef Compare value_compare;
typedef typename hash_type::const_iterator iterator;
typedef typename hash_type::const_iterator const_iterator;

// lib.set.cons construct/copy/destroy:
explicit hash_set(size_type num_buckets = 0,
                   const key_hasher& hash = key_hasher(),
                   const key_compare& comp = key_compare(),
                   float load_factor_limit = 2, float growth_factor = 2,
                   const allocator_type& a = allocator_type());

template <class InputIterator>
hash_set(InputIterator first, InputIterator last, size_type num_buckets = 0,
          const key_hasher& hash = key_hasher(),
          const key_compare& comp = key_compare(),
          float load_factor_limit = 2, float growth_factor = 2,
          const allocator_type& a = allocator_type());

allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
```

Hash Libraries

Hash_set

```
const_iterator end() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;
size_type bucket_count() const;
size_type bucket_count(size_type num_buckets);
float load_factor() const;
void load_factor_limit(float lf);
float load_factor_limit() const;
void growth_factor(float gf);
float growth_factor() const;
size_type collision(const_iterator i) const;

// modifiers:
std::pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);
void swap(hash_set& y);

void clear();

// observers:
key_compare key_comp() const;
value_compare value_comp() const;
key_hasher key_hash() const;
value_hasher value_hash() const;

// set operations:
iterator find(const key_type& x) const;
size_type count(const key_type& x) const;

std::pair<iterator, iterator> equal_range(const key_type& x)
const;
};
```

```
template <class T, class Hash, class Compare, class Allocator>
void swap(hash_set<T, Hash, Compare, Allocator>& x,
          hash_set<T, Hash, Compare, Allocator>& y);

template <class T, class Hash, class Compare, class Allocator>
bool
operator==(const hash_set<T, Hash, Compare, Allocator>& x,
            const hash_set<T, Hash, Compare, Allocator>& y);

template <class T, class Hash, class Compare, class Allocator>
bool
operator!=(const hash_set<T, Hash, Compare, Allocator>& x,
            const hash_set<T, Hash, Compare, Allocator>& y);

} // Metrowerks
```

Listing 26.9 hash_multiset synopsis

```
namespace Metrowerks

template <class T, class Hash = hash<T>, class Compare =
std::equal_to<T>,
           class Allocator = std::allocator<T> >
class hash_multiset
{
public:
    // types:
    typedef key_type;
    T
    typedef value_type;
    T
    typedef key_hasher;
    value_type;
    Hash
    ;
    typedef value_hasher;
    Hash
    er;
    typedef Allocator
    allocator_type;
    typedef typename Allocator::reference
    reference;
```

Hash Libraries

Hash_set

```
typedef typename Allocator::const_reference const_reference;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type difference_type;
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef Compare key_compare;
typedef Compare value_compare;
typedef typename hash_type::const_iterator iterator;
typedef typename hash_type::const_iterator const_iterator;

// lib.set.cons construct/copy/destroy:
explicit hash_multiset(size_type num_buckets = 0,
    const key_hasher& hash = key_hasher(), const key_compare& comp = key_compare(),
    float load_factor_limit = 2, float growth_factor = 2,
    const allocator_type& a = allocator_type());

template <class InputIterator>
hash_multiset(InputIterator first, InputIterator last,
size_type num_buckets = 0,
    const key_hasher& hash = key_hasher(), const key_compare& comp = key_compare(),
    float load_factor_limit = 2, float growth_factor = 2,
    const allocator_type& a = allocator_type());

allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;
```

```
size_type bucket_count() const;
size_type bucket_count(size_type num_buckets);
float      load_factor() const;
void       load_factor_limit(float lf);
float      load_factor_limit() const;
void       growth_factor(float gf);
float      growth_factor() const;
size_type collision(const_iterator i) const;

// modifiers:
iterator insert(const value_type& x);
iterator insert(iterator p, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void         erase(iterator position);
size_type    erase(const key_type& x);
void         erase(iterator first, iterator last);
void         swap(hash_multiset& y);

void clear();

// observers:
key_compare   key_comp() const;
value_compare value_comp() const;
key_hasher    key_hash() const;
value_hasher  value_hash() const;

// set operations:
iterator    find(const key_type& x) const;
size_type    count(const key_type& x) const;

std::pair<iterator,iterator> equal_range(const key_type& x)
const;
};

template <class T, class Hash, class Compare, class Allocator>
void swap(hash_multiset<T, Hash, Compare, Allocator>& x,
          hash_multiset<T, Hash, Compare, Allocator>& y);

template <class T, class Hash, class Compare, class Allocator>
bool
```

Hash Libraries

Hash_map

```
operator==(const hash_multiset<T, Hash, Compare, Allocator>& x,
            const hash_multiset<T, Hash, Compare, Allocator>&
y);

template <class T, class Hash, class Compare, class Allocator>
bool
operator!=(const hash_multiset<T, Hash, Compare, Allocator>& x,
            const hash_multiset<T, Hash, Compare, Allocator>&
y);

} // namespace Metrowerks
```

Hash_map

The `hash_map` is a container that holds an unordered set of key-value pairs, and no two keys in the container can compare equal. `hash_multimap` permits duplicate entries. Also see the General Hash Issues Introduction.

This header contains two classes:

- `hash_map`
- `hash_multimap`

NOTE This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

Introduction

These containers are in the namespace Metrowerks. See Namespace Issues for details and hints about how to best take advantage of this fact.

`hash_map` and `hash_multimap` are largely compatible with previous versions of these classes which appeared in namespace `std`. But see Incompatibility for a short list of incompatibilities.

Old Hashmap Headers

Previous versions of CodeWarrior placed `hash_map` and `hash_multimap` in the headers `<hashmap.h>` and `<hashmm.h>` respectively. These headers are still available, but should be used only for transition purposes. They will disappear in a future release. These headers import the contents of `<hash_map>` into the std namespace (as previous versions of `hash_(multi)map` were implemented in std).

Listing 26.10 Old Hashmap Headers

```
#include <hashmap.h>

int main()
{
    std::hash_map<int, int> a;
}
```

Template Parameters

Both `hash_map` and `hash_multimap` have the following template parameters and defaults:

Listing 26.11 Hashmap Template Parameters

```
template <class Key, class T, class Hash = hash<Key>,
          class Compare = std::equal_to<Key>,
          class Allocator = std::allocator<std::pair<const Key, T>>>
class hash_(multi)map;
```

The first parameter is the type of key the map is to contain. It can be almost any type, but must be copyable.

The second parameter is the type of the value that will be associated with each key. It can be almost any type, but must be copyable.

The third parameter is the hash function used to look up elements. It defaults to the hash function in `<hash_fun>`. Client code can use `hash<Key>` as is, specialize it, or supply completely different hash

function objects or hash function pointers. The hash function must accept a Key, and return a size_t.

The fourth parameter is the comparison function which defaults to std::equal_to<Key>. This function should have equality semantics. A specific requirement is that if two keys compare equal according to Compare, then they must also produce the same result when processed by Hash.

The fifth and final parameter is the allocator, which defaults to std::allocator<std::pair<const Key, T>>. The same comments and requirements that appear in the standard for allocators apply here as well.

Nested Types

hash_map and hash_multimap define a host of nested types similar to standard containers. Several noteworthy points:

- key_type and value_type are not the same type. value_type is a pair<const Key, T>.
- key_hasher and value_hasher are not the same type. key_hasher is the template parameter Hash. value_hasher is a nested type which converts key_hasher into a function which accepts a value_type.

– value_hasher has the public typedef's

```
typedef value_type argument_type;  
typedef size_type    result_type;
```

This qualifies it as a std::unary_function (as defined in <functional>) and so could be used where other functionals are used.

– value_hasher has these public member functions:

```
size_type operator()(const value_type& x) const;  
size_type operator()(const key_type& x) const;
```

These simply return the result of key_hasher, but with the first operator extracting the key_type from the value_type before passing the key_type on to key_hasher.

- Key_compare and value_compare are not the same type.
key_compare is the template parameter Compare.
value_compare is a nested type which converts key_compare into a function which accepts a value_type.

– value_compare has the public typedef's

```
typedef value_type first_argument_type;
typedef value_type second_argument_type;
typedef bool result_type;
```

This qualifies it as a std:: binary_function (as defined in <functional>) and so could be used where other functionals are used.

– value_compare has these public member functions:

```
bool operator()(const value_type& x,
                 const value_type& y) const;
bool operator()(const key_type& x,
                 const value_type& y) const;
bool operator()(const value_type& x,
                 const key_type& y) const;
```

These pass their arguments on to key_compare, extracting the key_type from value_type when necessary.

Iterator Issues

See Iterator Issues that are common to all hash containers.

See Capacity for details on how to control the number of buckets.

Listing 26.12 hash_map synopsis

```
namespace Metrowerks {

template <class Key, class T, class Hash = hash<Key>, class
Compare = std::equal_to<Key>,
          class Allocator = std::allocator<std::pair<const
Key, T> > >
class hash_map
{
public:
```

Hash Libraries

Hash_map

```
// types:
typedef Key
key_type;
typedef
T
mapped_type;
typedef std::pair<const Key,
T> value_type;
typedef
Hash key_hasher;
;
typedef Compare
key_compare;
typedef Allocator
allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference
const_reference;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type
difference_type;
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef value_hash_imp<is_empty<key_hasher>::value>
value_hasher;
typedef value_compare_imp<is_empty<key_compare>::value>
value_compare;
typedef typename hash_type::iterator iterator;
typedef typename
hash_type::const_iterator const_iterator;

// construct/copy/destroy:
explicit hash_map(size_type num_buckets = 0,
    const key_hasher& hash = key_hasher(), const key_compare&
comp = key_compare(),
    float load_factor_limit = 2, float growth_factor = 2,
    const allocator_type& a = allocator_type());

template <class InputIterator>
```

```
hash_map(InputIterator first, InputIterator last, size_type
num_buckets = 0,
         const key_hasher& hash = key_hasher(), const
key_compare& comp = key_compare(),
         float load_factor_limit = 2, float growth_factor = 2,
         const allocator_type& a = allocator_type());

allocator_type get_allocator() const;

// iterators:
iterator      begin();
const_iterator begin() const;
iterator      end();
const_iterator end() const;

// capacity:
bool          empty() const;
size_type     size() const;
size_type     max_size() const;
size_type     bucket_count() const;
size_type     bucket_count(size_type num_buckets);
float         load_factor() const;
void          load_factor_limit(float lf);
float         load_factor_limit() const;
void          growth_factor(float gf);
float         growth_factor() const;
size_type     collision(const_iterator i) const;

// element access:
mapped_type& operator[](const key_type& x);

// modifiers:
std::pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);

void          erase(iterator position);
size_type     erase(const key_type& x);
void          erase(iterator first, iterator last);

void          swap(hash_map& y);
```

Hash Libraries

Hash_map

```
void clear();

// observers:
key_compare      key_comp() const;
value_compare    value_comp() const;
key_hasher       key_hash() const;
value_hasher     value_hash() const;

// set operations:
iterator          find(const key_type& x);
const_iterator    find(const key_type& x) const;
size_type         count(const key_type& x) const;

std::pair<iterator, iterator>
equal_range(const key_type& x);
std::pair<const_iterator, const_iterator> equal_range(const
key_type& x) const;
};

template <class Key, class T, class Hash, class Compare, class
Allocator>
void swap(hash_map<Key, T, Hash, Compare, Allocator>& x,
          hash_map<Key, T, Hash, Compare, Allocator>& y);

template <class Key, class T, class Hash, class Compare, class
Allocator>
bool
operator==(const hash_map<Key, T, Hash, Compare, Allocator>& x,
            const hash_map<Key, T, Hash, Compare, Allocator>&
y);

template <class Key, class T, class Hash, class Compare, class
Allocator>
bool
operator!=(const hash_map<Key, T, Hash, Compare, Allocator>& x,
            const hash_map<Key, T, Hash, Compare, Allocator>&
y);
} // Metrowerks
```

Element Access

Prototype `mapped_type& operator[](const key_type& x);`

If the key `x` already exists in the container, returns a reference to the `mapped_type` associated with that key. If the key `x` does not already exist in the container, inserts a new entry: `(x, mapped_type())`, and returns a reference to the newly created, default constructed `mapped_type`.

Listing 26.13 hash_multimap synopsis

```
namespace Metrowerks {

    template <class Key, class T, class Hash = hash<Key>, class
Compare = std::equal_to<Key>,
            class Allocator = std::allocator<std::pair<const
Key, T> > >
    class hash_multimap
    {
public:
    // types:
    typedef Key
key_type;
    typedef
T
mapped_type;
    typedef std::pair<const Key,
T>                     value_type;
    typedef
Hash
key_hasher;
;
    typedef Compare
key_compare;
    typedef Allocator
allocator_type;
    typedef typename Allocator::reference
reference;
    typedef typename Allocator::const_reference
const_reference;
    typedef typename Allocator::size_type
size_type;
    typedef typename Allocator::difference_type
difference_type;
    typedef typename Allocator::pointer
pointer;
```

Hash Libraries

Hash_map

```
typedef typename Allocator::const_pointer      const_pointer;
typedef value_hash_imp<is_empty<key_hasher>>::value>

           value_hasher;
typedef value_compare_imp<is_empty<key_compare>>::value>

           value_compare;
typedef typename hash_type::iterator          iterator;
typedef typename
hash_type::const_iterator       const_iterator;

//  construct/copy/destroy:
explicit hash_multimap(size_type num_buckets = 0,
    const key_hasher& hash = key_hasher(), const key_compare&
comp = key_compare(),
    float load_factor_limit = 2, float growth_factor = 2,
    const allocator_type& a = allocator_type());

template <class InputIterator>
    hash_multimap(InputIterator first, InputIterator last,
size_type num_buckets = 0,
    const key_hasher& hash = key_hasher(), const
key_compare& comp = key_compare(),
    float load_factor_limit = 2, float growth_factor = 2,
    const allocator_type& a = allocator_type());

allocator_type get_allocator() const;

//  iterators:
iterator      begin();
const_iterator begin() const;
iterator      end();
const_iterator end() const;

//  capacity:
bool      empty() const;
size_type size() const;
size_type max_size() const;
size_type bucket_count() const;
size_type bucket_count(size_type num_buckets);
float      load_factor() const;
void      load_factor_limit(float lf);
```

```
float      load_factor_limit() const;
void       growth_factor(float gf);
float      growth_factor() const;
size_type   collision(const_iterator i) const;

//  modifiers:
iterator   insert(const value_type& x);
iterator   insert(iterator p, const value_type& x);
template <class InputIterator>
void       insert(InputIterator first, InputIterator last);

void       erase(iterator position);
size_type  erase(const key_type& x);
void       erase(iterator first, iterator last);

void       swap(hash_multimap& y);

void clear();

//  observers:
key_compare key_comp() const;
value_compare value_comp() const;
key_hasher   key_hash() const;
value_hasher value_hash() const;

//  set operations:
iterator   find(const key_type& x);
const_iterator find(const key_type& x) const;
size_type   count(const key_type& x) const;

std::pair<iterator, iterator>
equal_range(const key_type& x);
    std::pair<const_iterator, const_iterator> equal_range(const
key_type& x) const;
};

template <class Key, class T, class Hash, class Compare, class
Allocator>
void swap(hash_multimap<Key, T, Hash, Compare, Allocator>& x,
          hash_multimap<Key, T, Hash, Compare, Allocator>&
y);
```

Hash Libraries

Hash_fun

```
template <class Key, class T, class Hash, class Compare, class
Allocator>
bool
operator==(const hash_multimap<Key, T, Hash, Compare, Allocator>&
x,
            const hash_multimap<Key, T, Hash, Compare,
Allocator>& y);

template <class Key, class T, class Hash, class Compare, class
Allocator>
bool
operator!=(const hash_multimap<Key, T, Hash, Compare, Allocator>&
x,
            const hash_multimap<Key, T, Hash, Compare,
Allocator>& y);

} // namespace Metrowerks
```

Hash_fun

<hash_fun> declares a templated struct which serves as a function object named hash. This is the default hash function for all hash containers. As supplied, hash works for integral types, basic_string types, and char* types (c-strings).

NOTE This header is non-standard. The classes herein are offered as extensions to the C++ standard. They are marked as such by the namespace Metrowerks.

Client code can specialize hash to work for other types. For example:

Listing 26.14 <hash_fun>

```
namespace Metrowerks
{
    template <>
    struct hash<MyType>
```

```
: _STD::unary_function<MyType, std::size_t>
{
    std::size_t operator()(const MyType& key) const;
};

template <>
std::size_t
hash<MyType>::operator()(const MyType& key) const
{
    std::size_t h;
    // compute h (the hash of key)
    return h;
}

} // Metrowerks
```

Alternatively, client code can simply supply customized hash functions to the hash containers via the template parameters.

The returned `size_t` should be as evenly distributed as possible in the range `[0, numeric_limits<size_t>::max()]`. Logic in the hash containers will take care of folding this output into the range of the current number of buckets.

Hash Libraries

Hash_fun

Mslconfig

The MSL header <mslconfig> contains a description of the macros and defines that are used as switches or flags in the MSL C++ library.

C++ Switches, Flags and Defines

The MSL C++ library has various flags that may be set to customize the library to users specifications these include:

- [“ CSTD” on page 746](#)
- [“ Inhibit Container Optimization” on page 746](#)
- [“ Inhibit Optimize RB bit” on page 746](#)
- [“ MSL DEBUG” on page 747](#)
- [“ msl_error” on page 747](#)
- [“ MSL ARRAY AUTO PTR” on page 747](#)
- [“ MSL CPP ” on page 748](#)
- [“ MSL EXTENDED BINDERS” on page 748](#)
- [“ MSL EXTENDED PRECISION OUTP” on page 749](#)
- [“ MSL FORCE ENABLE BOOL SUPPORT” on page 749](#)
- [“ MSL FORCE ENUMS ALWAYS INT” on page 750](#)
- [“ MSL IMP EXP” on page 751](#)
- [“ MSL LONGLONG SUPPORT ” on page 752](#)
- [“ MSL MINIMUM NAMED LOCALE” on page 752](#)
- [“ MSL MULTITHREAD” on page 753](#)
- [“ MSL NO BOOL” on page 753](#)
- [“ MSL NO CONSOLE IO” on page 754](#)
- [“ MSL NO CPP NAMESPACE” on page 754](#)

- “ [MSL NO EXCEPTIONS](#)” on page 754
- “ [MSL NO EXPLICIT FUNC TEMPLATE ARG](#)” on page 754
- “ [MSL NO FILE IO](#)” on page 755
- “ [MSL NO IO](#)” on page 755
- “ [MSL NO LOCALE](#)” on page 755
- “ [MSL NO REFCOUNT STRING](#)” on page 756
- “ [MSL NO VECTOR BOOL](#)” on page 756
- “ [MSL NO WCHART](#)” on page 756
- “ [MSL NO WCHART LANG SUPPORT](#)” on page 757
- “ [MSL NO WCHART C SUPPORT](#)” on page 757
- “ [MSL NO WCHART CPP SUPPORT](#)” on page 757
- “ [MSL USE AUTO PTR 96](#)” on page 757
- “ [STD](#)” on page 757

_CSTD

The `_CSTD` macro evaluates to `::std` if the MSL C library is compiled in the std namespace, and to nothing if the MSL C library is compiled in the global namespace.

`_STD` and `_CSTD` are meant to prefix C++ and C objects in such a way that you don't have to care whether or not the object is in std or not. For example:

`_STD::cout`, or `_CSTD::size_t`.

_Inhibit_Container_Optimization

If this flag is defined it will disable pointer specializations in the containers. This may make debugging easier.

NOTE

You must recompile the C++ lib when flipping this switch.

_Inhibit_Optimize_RB_bit

Normally the red/black tree used to implement the associative containers has a space optimization that compacts the red/black

flag with the parent pointer in each node (saving one word per entry). By defining this flag, the optimization is turned off, and the red/black flag will be stored as an enum in each node of the tree.

_MSL_DEBUG

This switch when enabled and the library rebuilt will put MSL Standard C++ library into debug mode. For full information see [“Overview of MSL C++ Debug Mode” on page 699.](#)

NOTE You must recompile the C++ lib when flipping this switch.

__msl_error

This feature is included for those wishing to use the C++ lib with exceptions turned off. In the past, with exceptions turned off, the lib would call fprintf and abort upon an exceptional condition. Now you can configure what will happen in such a case by filling out the definition of __msl_error(). Two example definitions are given. One example will call fprintf and abort, the other will do nothing at all.

_MSL_ARRAY_AUTO_PTR

When defined auto_ptr can be used to hold pointers to memory obtained with the array form of new.

The syntax looks like:

```
auto_ptr<string, _Array<string> >
    pString(new string[3]);
    pString.get()[0] = "pear";
    pString.get()[1] = "peach";
    pString.get()[2] = "apple";
```

Without the _Array tag, auto_ptr behaves in a standard fashion. This extension to the standard is not quite conforming as it can be detected through the use of template template arguments.

NOTE This extension can be disabled by not defining
`_MSL_ARRAY_AUTO_PTR`.

__MSL_CPP__

Evaluates to an integer value which represents the C++ lib's current version number.

TIP This value is best when read in hexadecimal format

_MSL_EXTENDED_BINDERS

Defining this flag adds defaulted template parameters to binder1st and binder2nd. This allows client code to alter the type of the value that is stored. This is especially useful when you want the binder to store the value by const reference instead of by value to save on an expensive copy construction.

For example:

```
#include <string>
#include <functional>
#include <algorithm>

struct A
{
public:
    A(int data = 0) : data_(data) {}
    friend bool operator < (const A& x, const A& y) {return x < y;}
private:
    int data_;
    A(const A&);
};

int main()
{
using namespace std;
A a[5];
```

```
A* i = find_if(a, a+5, binder2nd<less<A> >(less<A>(), A(5)));
}
```

This causes the compile-time error, because binder2nd is attempting to store a copy of A(5). But with `_MSL_EXTENDED_BINDERS` you can request that binder2nd store a const A& to A(5).

```
A* i = find_if(a, a+5,
    binder2nd<less<A>, const A&>(less<A>(), A(5)));

```

TIP This may be valuable when A is expensive to copy.

This also allows for the use of polymorphic operators by specifying reference types for the operator.

NOTE This extension to the standard is detectable with template template parameters so it can be disabled by not defining `_MSL_EXTENDED_BINDERS`.

_MSL_EXTENDED_PRECISION_OUTP

When defined this allows the output of floating point output to be printed with precision greater than `DECIMAL_DIG`. With this option, an exact binary to decimal conversion can be performed (by bumping precision high enough).

TIP The cost is about 5-6Kb in code size.

NOTE You must recompile the C++ lib when flipping this switch.

_MSL_FORCE_ENABLE_BOOL_SUPPORT

This tri-state flag has the following properties

- If not defined, then the C++ library and headers will react to the settings in the language preferences panel (as in the past).

- If the flag is set to zero, then the C++ lib/header will force “Enable bool support” to be off while processing the header (and then reset at the end of the header).
- If the flag is set to one, then the C++ library and header will force “Enable bool support” to be on while processing the header (and then reset at the end of the header).

If `_MSL_FORCE_ENABLE_BOOL_SUPPORT` is defined, the C++ library will internally ignore the “Enable bool support” setting in the application’s language preferences panel, despite the fact that most of the C++ library is compiled into the application (since it is in headers) instead of into the binary C++ library.

The purpose of this flag is (when defined) to avoid having to recompile the C++ library when “Enable bool” support is changed in the language preferences panel.

With `_MSL_FORCE_ENABLE_BOOL_SUPPORT` defined to one, `std::methods` will continue to have a real `bool` in their signature, even when `bool` support is turned off in the application. But the user won’t be able to form a `bool` (or a `true/false`). The user won’t be able to:

```
bool b = std::ios_base::sync_with_stdio(false);
// error: undefined bool and false
```

but this will work:

```
unsigned char b =
std::ios_base::sync_with_stdio(0);
```

And the C++ lib will link instead of getting the ctype link error.

NOTE

Changing this flag will require a recompile of the C++ library.

_MSL_FORCE_ENUMS_ALWAYS_INT

This tri-state flag has the following properties

- If not defined, then the C++ library and headers will react to the settings in the language preferences panel (as in the past).

- If the flag is set to 0, then the C++ lib/header will force “Enums always int” to be off while processing the header (and then reset at the end of the header).
- If the flag is set to 1, then the C++ library and header will force “Enums always int” to be on while processing the header (and then reset at the end of the header).

If `_MSL_FORCE_ENUMS_ALWAYS_INT` is defined, the C++ library will internally ignore the “Enums always int” setting in the application's language preferences, despite the fact that most of the C++ library is compiled into the application (since it is in headers) instead of into the binary C++ library.

The purpose of this flag is (when defined) to avoid having to recompile the C++ lib when “Enums always int” is changed in the language preferences panel.

For example, with `_MSL_FORCE_ENUMS_ALWAYS_INT` defined to zero, and if the user turns “enums always int” on in his language preferences panel, then any enums the user creates himself will have an underlying `int` type.

This can be exposed by printing out the `sizeof(the enum)` which will be four. However if the user prints out the `sizeof(a std::enum)`, then the size will be one(because all `std::enums` fit into 8 bits) despite the `enums_always_int` setting.

NOTE Changing this flags will require a recompile of the C++ library.

_MSL_IMP_EXP

The C, C++, SIOUX and runtime shared libraries have all been combined into one shared library locating under the appropriate OS support folder.

The exports files (.exp) have been removed. The prototypes of objects exported by the shared lib are decorated with a macro:

`_MSL_IMP_EXP_xxx`
or
`_MSL_IMP_EXP_xxx`
//where xxx is the library designation.

which can be defined to `__declspec(dllimport)`.

This replaces the functionality of the `.exp/.def` files. Additionally, the C, C++, SIOUX and runtimes can be imported separately by defining the following 4 macros differently:

```
_MSL_IMP_EXP_C  
_MSL_IMP_EXP_CPP  
_MSL_IMP_EXP_SIOUX  
_MSL_IMP_EXP_RUNTIME
```

Define these macros to nothing if you don't want to import from the associated lib, otherwise they will pick up the definition of `_MSL_IMP_EXP`.

NOTE There is a header `<UseDLLPrefix.h>` that can be used as a prefix file to ease the use of the shared lib. It is set up to import all 4 sections.

NOTE There is a problem with non-const static data members of templated classes when used in a shared lib. Unfortunately `<locale>` is full of such objects. Therefore you should also define `_MSL_NO_LOCALE` which turns off locale support when using the C++ lib as a shared lib. This is done for you in `<UseDLLPrefix.h>`. See [“_MSL_NO_LOCALE” on page 755](#) for more details.

__MSL_LONGLONG_SUPPORT__

When defined, C++ supports long long and unsigned long long integral types. Recompile the C++ lib when flipping this switch.

_MSL_MINIMUM_NAMED_LOCALE

When defined, turns off all of the named locale stuff except for "C" and "" (which will be the same as "C"). This reduces both lib size and functionality, but only if you are already using named locales. If your code does not explicitly use named locales, this flag has no effect.

_MSL_MULTITHREAD

The thread safety of MSL C++ can be controlled by the flag `_MSL_MULTITHREAD`.

If you explicitly use `std::mutex` objects in your code, then they will become empty do-nothing objects when multi-threading is turned off (`_MSL_MULTITHREAD` is undefined). Thus the same source can be used in both single thread and multi-thread projects.

The `_MSL_MULTITHREAD` flag causes some mutex objects to be set up in the library internally to protect data that is not obviously shared. For example, `std::basic_string` is refcounted. It is possible that two threads might each have their own `basic_string`, and that `basic_string` might share data among threads under the covers via the refcount mechanism. Therefore `basic_string` protects its refcount with a mutex object so that client code (even multi-threaded client code) can not detect that a refcounting implementation is in use.

See “[Multi-Thread Safety](#)” on page 29 for a full description of Metrowerks Standard Library multi-threading safety policy.

_MSL_NO_BOOL

If defined then `bool` will not be treated as a built-in type by the library. Instead it will be a `typedef` to `unsigned char` (with suitable values for true and false as well). If `_MSL_FORCE_ENABLE_BOOL_SUPPORT` is not defined then this flag will set itself according to the “Enable bool support” switch in the language preferences panel.

NOTE The C++ lib must be recompiled when flipping this switch.

NOTE When `_MSL_NO_BOOL` is defined, `vector<bool>` will really be a `vector<unsigned char>`, thus it will take up more space and not have flip methods. Also there will not be any traits specializations for `bool` (i.e. `numeric_limits`).

_MSL_NO_CONSOLE_IO

This flag allows you to turn off console support while keeping memory mapped streams (stringstream) functional.

See Also

[“MSL NO FILE IO” on page 755](#)

_MSL_NO_CPP_NAMESPACE

If defined then the C++ lib will be defined in the global namespace.

NOTE

You must recompile the C++ lib when flipping this switch.

_MSL_NO_EXCEPTIONS

If defined then the C++ lib will not throw an exception in an exceptional condition. Instead `void __msl_error(const char*);` will be called. You may edit this inline in `<mslconfig>` to do whatever is desired. Sample implementations of `__msl_error` are provided in `<mslconfig>`.

NOTE

The operator new (which is in the runtime libraries) is not affected by this flag.

Remarks

This flag detects the language preferences panel “Enable C++ exceptions” and defines itself if this option is not on.

NOTE

The C++ lib must be recompiled when changing this flag (including if the language preference panel is changed).

_MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG

When defined assumes that the compiler does not support calling function templates with explicit template arguments.

On Windows, when ARM Conformance is selected in the language preferences panel, then this switch is automatically turned on. The Windows compiler goes into a MS compatible mode with ARM on.

This mode does not support explicit function template arguments.

In this mode, the signatures of has_facet and use_facet change.

Listing 27.1 Example of _MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG usage:

Standard setting:

```
template <class Facet>
    const Facet& use_facet(const locale& loc);
template <class Facet>
    bool has_facet(const locale& loc) throw();
```

```
_MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG setting.
template <class Facet>
    const Facet& use_facet(const locale& loc, Facet* );
template <class Facet>
    bool has_facet(const locale& loc, Facet* ) throw();
```

NOTE You must recompile the C++ lib when flipping this switch.

_MSL_NO_FILE_IO

This flag allows you to turn off file support while keeping memory mapped streams (stringstream) functional.

See Also [“_MSL_NO_CONSOLE_IO” on page 754](#)

_MSL_NO_IO

If this flag is defined the C++ will not support any I/O (not even stringstream).

_MSL_NO_LOCALE

When this flag is defined, locale support is stripped from the library. This has tremendous code size benefits.

All C++ I/O will implicitly use the “C” locale. You may not create locales or facets, and you may not call the imbue method on a stream. But otherwise all streams are completely functional.

NOTE The C++ lib must be recompiled when flipping this switch.

_MSL_NO_REFCOUNT_STRING

This flag will reconfigure basic_string so that it is not refcounted. This may have code size savings. It may or may not have performance benefits. Benefits of this switch are highly application dependent.

When compiled for multi-thread (_MSL_MULTITHREAD) the flag _MSL_NO_REFCOUNT_STRING will automatically be defined.

NOTE You must recompile the C++ lib when flipping this switch.

_MSL_NO_VECTOR_BOOL

If this flag is defined it will disable the standard `vector<bool>` partial specialization. You can still instantiate `vector<bool>`, but it will not have the space optimization of one `bool` per bit.

NOTE There is no need to recompile the C++ lib when flipping this switch, but you should remake any precompiled headers you might be using.

_MSL_NO_WCHART

This flag has been replaced by three new flags,

- “[_MSL_NO_WCHART_LANG_SUPPORT](#)”.
- “[_MSL_NO_WCHART_C_SUPPORT](#)” and
- “[_MSL_NO_WCHART_CPP_SUPPORT](#)”.

_MSL_NO_WCHART_LANG_SUPPORT

This flag is set if the compiler does not recognize wchar_t as a separate data type (no wchar_t support in the language preference panel). The C++ lib will still continue to support wide character functions. wchar_t will be typedef'd to another built-in type.

NOTE The C++ library must be recompiled when turning this switch on (but need not be recompiled when turning it off).

_MSL_NO_WCHART_C_SUPPORT

This flag is set if the underlying C lib does not support wide character functions. This should not be set when using MSL C.

NOTE The C++ library must be recompiled when turning this switch on (but need not be recompiled when turning it off).

_MSL_NO_WCHART_CPP_SUPPORT

This flag can be set if wide character support is not desired in the C++ lib. Setting this flag can cut the size of the I/O part of the C++ lib in half.

NOTE The C++ library must be recompiled when turning this switch on (but need not be recompiled when turning it off).

_MSL_USE_AUTO_PTR_96

Defining this flag will disable the standard auto_ptr and enable the version of auto_ptr that appeared in the Dec.'96 CD2.

_STD

This macro evaluates to `::std` if the C++ lib is compiled in the std namespace, and to nothing if the C++ lib is compiled in the global namespace.

Mslconfig

C++ Switches, Flags and Defines

See Also [“CSTD” on page 746](#)

Index

map 311

Symbols

<cassert> 73
<cerrno> 73
<functional
 negate> 85
_MSL_CPP_ 748
_msl_error 747
_MSL_LONGLONG_SUPPORT_ 752
_CSTD 746
_Inhibit_Container_Optimization 746
_Inhibit_Optimize_RB_bit 746
_MSL_ARRAY_AUTO_PTR 747
_MSL_CX_LIMITED_RANGE 433
_MSL_DEBUG 747
_MSL_EXTENDED_BINDERS 748
_MSL_EXTENDED_PRECISION_OUTP 749
_MSL_FORCE_ENABLE_BOOL_SUPPORT 749
_MSL_FORCE_ENUMS_ALWAYS_INT 750
_MSL_IMP_EXP 751
_MSL_IMP_EXP_C 752
_MSL_IMP_EXP_CPP 752
_MSL_IMP_EXP_RUNTIME 752
_MSL_IMP_EXP_SIOUX 752
_MSL_MINIMUM_NAMED_LOCALE 752
_MSL_NO_BOOL 753
_MSL_NO_CONSOLE_IO 754
_MSL_NO_CPP_NAMESPACE 754
_MSL_NO_EXCEPTIONS 754
_MSL_NO_EXPLICIT_FUNC_TEMPLATE_ARG
 754
_MSL_NO_FILE_IO 755
_MSL_NO_IO 755
_MSL_NO_LOCALE 755
_MSL_NO_REFCOUNT_STRING 756
_MSL_NO_VECTOR_BOOL 756
_MSL_NO_WCHART 756
_MSL_NO_WCHART_C_SUPPORT 757
_MSL_NO_WCHART_CPP_SUPPORT 757
_MSL_NO_WCHART_LANG_SUPPORT 757
_MSL_USE_AUTO_PTR_96 757

_STD 757

A

Abnormal Termination 63
abort
 Numeric_limits 51
abs 446
access 407
 valarray 407
Accumulate 428
Adaptors for Pointers to Functions 91
Adaptors for pointers to functions
 Functional 91
Adaptors for Pointers to Members 92
Adaptors for pointers to members
 Functional 92
address 99
adjacent_difference 430
Adjacent_find
 algorithm 374
Advance 341
Algorithm 361
 adjacent_find 374
 binary_search 388
 copy 377
 copy_backward 377
 count 374
 count_if 375
 equal 375
 equal_range 388
 fill 380
 fill_n 380
 find 372
 find_end 373
 find_first_of 373
 find_if 373
 for_each 372
 generate 380
 generate_n 380
 includes 389
 inplace_merge 389
 iter_swap 378
 lexicographical_compare 395
 lower_bound 387
 make_heap 393

max 394
max_element 394
merge 388
min 393
min_element 394
mismatch 375
next_permutation 395
nth_element 386
partial_sort 386
partial_sort_copy 386
partition 384
pop_heap 392
prev_permutation 396
push_heap 392
random_shuffle 384
remove 381
remove_copy 381
remove_copy_if 381
remove_if 381
replace 379
replace_copy_if 379
reverse 382
reverse_copy 383
rotate 383
rotate_copy 383
search 376
search_n 376
set_difference 391
set_intersection 390
set_symmetric_difference 391
set_union 390
sort 385
sort_heap 393
stable_partition 384
stable_sort 385
swap 377
swap_ranges 378
transform 378
unique 382
unique_copy 382
upper_bound 387
Algorithms Library 361–397
allocate 99
allocator 98
allocator globals 100
Allocator requirements 76
Always_noconv
 codecvt 184
Any

bitset 331
Apply
 valarray 411
Arbitrary-Positional Stream 26
arg 447
Arithmetic operations
 Functional 84
assert.h 73
Assertions 73
Assign
 deque 283
 list 287
 vector 299
assign 112
Assignment Operator
 bad_alloc 56
 bad_cast 59
 bad_typeid 60
 complex 440
 type_info 58
Assignment operator
 auto_ptr 107
 bad_exception 62
 exception 61
 gslice_array 422
 mask_array 424
 slice_array 419
Assignment operators
 indirect_array 426
Associative Containers 304
Associative Containers Requirements 274
atexit
 Numeric_limits 51
auto_ptr 108
 destructor 107
Auto_ptr 103, 109
 Assignment operator 107
 Constructors 107
 Members 107
 Operator = 107
Auto_ptr conversions 108
Auto_ptr_ref 108

B

Back_insert_iterator
 back_inserter 348
 constructors 347

| | |
|--------------------------|----------------------------|
| operator = 348 | basic_ifstream 630 |
| operators 348 | close 635 |
| Back_inserter | constructor 631 |
| back_insert_iterator 348 | is_open 633 |
| bad 498 | open 634 |
| bad_alloc | Open Modes 634 |
| assignment operator 56 | rdbuf 632 |
| constructors 55 | basic_ios 484 |
| destructor 56 | bad 498 |
| what 56 | clear 493 |
| Bad_cast | constructors 486 |
| assignment operator 59 | copyfmt 491 |
| constructor 59 | eof 496 |
| what 59 | exceptions 500 |
| Bad_exception | fail 497 |
| assignment operator 62 | fill 490 |
| constructor 62 | good 496 |
| what 62 | imbue 490 |
| Bad_typeid | Operator ! 491 |
| assignment operator 60 | Operator bool 491 |
| constructor 60 | rdbuf 488 |
| what 60 | rdstate 491 |
| Base | setstate 495 |
| reverse_iterator 344 | tie 487 |
| Basic Iterator 340 | basic_iostream 566 |
| basic_filebuf 622 | constructor 566 |
| close 627 | destructor 566 |
| constructors 624 | basic_istream 536 |
| destructor 624 | constructors 538 |
| imbue 629 | destructor 538 |
| is_open 625 | extractors, arithmetic 541 |
| open 625 | extractors, characters 542 |
| Open Modes 626 | gcount 547 |
| overflow 628 | get 549 |
| pbackfail 628 | getline 551 |
| seekoff 628 | ignore 553 |
| seekpos 629 | peek 554 |
| setbuf 629 | putback 558 |
| showmany 627 | read 555 |
| sync 629 | readsome 556 |
| underflow 628 | seekg 562 |
| basic_fstream 642 | sentry 539 |
| close 647 | sync 561 |
| constructor 643 | tellg 562 |
| is_open 646 | unget 559 |
| open 646 | ws 564 |
| Open Modes 646 | basic_istringstream 604 |
| rdbuf 644 | constructors 605 |
| | rdbuf 607 |

str 608
basic_ofstream 636
close 642
constructors 637
is_open 639
open 640
Open Modes 640
rdbuf 638
basic_ostream 567
constructor 568
destructor 569
endl 584
ends 584
flush 581
flush,flush 585
Inserters, arithmetic 571
Inserters, characters 573
put 579
resetiosflags 588
seekp 577
sentry 570
setbase 589
setfill 590
setiosflags 589
setprecision 591
setw 592
tellp 577
write 580
basic_ostringstream 609
constructors 610
rdbuf 611
str 613
basic_streambuf 508
constructors 511, 599
destructor 511
eback 524
egptr 524
eptr 525
gbump 524
getloc 512
gptr 524
imbuf 526
in_avail 517
Locales 512
overflow 530, 603
pbbackfail 529, 603
pbase 525
pbump 525
pptr 525
pubseekoff 513
pubseekpos 514
pubsetbuf 512
pubsync 516
pubuimbue 512
sbumpc 518
seekoff 527, 603
seekpos 527, 604
setbuf 526
setg 524
setp 525
sgetc 519
sgetn 520
showmanc 528
snextc 517
sputback 520
sputc 522
sputn 523
str 601
sungetc 522
sync 527
uflow 529
underflow 528, 602
xsgetn 528
xsputn 530
basic_string
append 133
assign 134
assignment operator 130
at 133
begin 130
c_str 137
capacity 131, 132
clear 132
compare 140
Constructors 128
copy 136
data 137
destructor 130
Element Access 132
empty 132
end 131
erase 135
extractor 146
find 137
find_first_not_of 139
find_first_of 138
find_last_of 138
get_allocator 137

getline 147
 insert 134
 inserter 146
 Inserters and extractors 146
 iterator support 130
 max_size 131
 Modifiers 133
 Non-Member Functions and Operators 141
 Null Terminated Sequence Utilities 147
 operator 143, 144, 146
 operator!= 142
 operator+ 141
 operator+= 133
 operator== 142
 operator> 144
 operator>= 145
 operator>> 146
 rbegin 131
 rend 131
 replace 135
 reserve 132
 rfind 137
 size 131
 String Operations 136
 substr 140
 swap 136
 basic_stringbuf 598
 basic_stringstream 614
 constructors 616
 rdbuf 617
 str 618
 before
 type_info 58
 Bidirectional Iterators 337
 Binary_function 83
 Binary_negate 88
 Binary_search
 algorithm 388
 bind1st
 Functional 90
 bind2nd 91
 binder1st
 Functional 89
 binder2nd
 Functional 90
 Binders
 Functional 89
 Bitset 327
 any 331
 constructors 327
 count 330
 flip 330
 none 331
 operator 331, 333
 operator != 328, 331
 operator & 332
 operator &= 328
 operator <<=> 328
 operator == 330
 operator >> 331, 332
 operator >>= 329
 operator ^ 332
 operator ^= 328
 operator | 332
 Operator ~ 329
 reset 329
 set 329
 size 330
 test 331
 to_string 330
 to_ulong 330
 boolalpha 501
 Bsearch 396
 Buffer management 512
 Buffering 462

C

C Library Files 649–650
 C Library Locales 269
 C++ Library 25–38
 Capacity
 vector 299
 Category
 Locale 159
 cerr 463
 char_type 114
 Character 26
 character 112
 Character Classification
 locale 165
 character container type 112
 Character Conversions
 locale 165
 Character Sequences 26
 Character Trait Definitions 112, 114

Character traits definitions 112
Class
 Back_insert_iterator 347
 basic_filebuf 622
 basic_fstream 642
 basic_ifstream 630
 basic_ios 484
 basic_iostream 566
 basic_istream 536
 sentry 539
 basic_istringstream 604
 basic_ofstream 636
 basic_ostream 567
 sentry 570
 basic_ostringstream 609
 basic_streambuf 508
 basic_stringbuf 598
 basic_stringstream 614
Bitset 327
complex 440
Deque 280
fpos 467
Front_insert_iterator 348
gslice 419
gslice_array 421
indirect_array 425
Insert_iterator 350
ios_base 468
 failure 470
 Init 474
Istream_iterator 352
Istreambuf_iterator 355
list 284
Map 307
mask_array 423
Multimap 313
Multiset 321
Ostream_iterator 354
Ostreambuf_iterator 357
Priority_queue 292
Queue 291
Reverse_iterator 342
Set 318
Stack 294
Vector 296, 301
class
 Mutex_lock 678
Class Auto_ptr 103
Class bad_alloc 55
 Class bad_cast 58
 Class bad_typeid 59
 Class Ctype
 locale 167
 Class Ctype_byname 181
 Class ctype_byname
 locale 171
 Class exception 61
 class mutex 676
 Class slice 416
 Class Slice_array 418
 Class type_info 57
 Classic
 locale 164
 Classic_table
 ctype 180
Clear
 list 289
clear 493
clog 463
Close
 messages 262
close
 basic_filebuf 627
 basic_fstream 647
 basic_ifstream 635
 basic_ofstream 642
Cmath 430
Codevt
 Virtual Functions 185
Codevtc
 always_noconv 184
 in 184
 length 184
 max_length 184
 out 183
 unshift 184
Collate
 compare 202
 hash 203
 member functions 202
 transform 203
 Virtual Functions 203
Collate Category 202
Combine
 locale 163
Compare

collate 202
 compare 112
 Comparison Function 26
 Comparisons
 Functional 85
 complex 440
 abs 446
 arg 447
 conj 447
 constructor 440
 cos 448
 cosh 448
 exp 448
 imag 441, 446
 log 448
 log10 449
 norm 447
 operator 445
 operator - 444
 operator != 445
 operator * 444
 operator *= 442
 operator + 443
 operator += 441
 operator / 444
 operator /= 442
 operator -= 442
 operator = 440
 operator == 445
 operator >> 445
 polar 447
 pow 449
 real 441, 446
 sin 449
 sinh 449
 sqrt 450
 tan 450
 tanh 450
 Complex Class Library 433–450
 Component 27
 Conforming Implementations 37
 conj 447
 const_mem_fun_ref_t 96
 template function 96
 const_mem_fun_t 94
 template function 95
 const_mem_fun1_ref_t
 template class 96
 const_mem_fun1_t 95
 Constraints on programs 35
 Constructors
 insert_iterator 351
 construct 100
 Constructor
 list 287
 locale 162
 mutex_lock 678
 Constructor,ctype_byname 172
 Constructors 128
 Auto_ptr 107
 back_insert_iterator 347
 bad_alloc 55
 bad_cast 59
 bad_exception 62
 bad_typeid 60
 basic_filebuf 624
 basic_fstream 643
 basic_ifstream 631
 basic_ios 486
 basic_iostream 566
 basic_istream 538
 basic_istringstream 605
 basic_ofstream 637
 basic_ostream 568
 basic_ostringstream 610
 basic_streambuf 511, 599
 basic_stringstream 616
 bitset 327
 ctype 179
 deque 282
 domain_error 69
 exceptions 61
 failure, ios_base 471
 front_insert_iterator 349
 gslice 420
 gslice_array 422
 indirect_array 426
 invalid_argument 69
 ios_base 484
 istream_iterator 353
 istreambuf_iterator 356
 istrstream 663
 length_error 70
 logic_error 68
 map 311
 mask_array 424
 multimap 316

multiset 324
ostream_iterator 355
ostreambuf_iterator 358
ostrstream 666
out_of_range 70
overflow_error 72
pair 80
priority_queue 293
range_error 71
raw_storage_iterator 101
reverse_iterator 344
runtime_error 71
sentry, basic_istream 540
sentry, basic_ostream 570
set 320
slice 417
slice_array 418
stack 295
strstream 671
strstreambuf 656
type_info 58
underflow_error 72
valarray 406
vector 299
Container adaptors 291
Container Requirements 271
Containers Library 271–333
Conversion Constructor 108
Copy
 algorithm 377
copy 113
Copy construction 76
Copy_backward
 algorithm 377
copyfmt 491
cos 448
cosh 448
Count
 algorithm 374
 bitset 330
Count_if
 algorithm 375
cout 462
Cshift
 valarray 411
cstdio
 Functions 650
 Macros 649
 Types 649
Cstdlib 431
Ctype
 classic_table 180
 constructors 179
 destructor 179
 member functions 168
Ctype Category
 locale 166
Ctype Specializations
 locale 176
Ctype_byname,constructor 172
Curr_symbol
 moneypunct 252

D

Date and Time functions 109
date_order 215
deallocate 99
Debug Mode Implementations 700
dec 503
Decimal_point
 moneypunct 251
 numpunct 196
Default Behavior 27
Default construction 76
Delete 54
denorm_min
 Numeric_limits 48
Deque 280
 assign 283
 constructors 282
 erase 284
 insert 283
 resize 283
 swap 284
destroy 100
Destructor
 auto_ptr 107
 bad_alloc 56
 ctype 179
 exception 61
 istream_iterator 353
 istrstream 664
 mutex_lock 679
 ostream_iterator 355
 ostrstream 667

strstr 672
 valarray 407
Destructors
 basic_filebuf 624
 basic_ios 486
 basic_iostream 566
 basic_istream 538
 basic_oiostream 569
 basic_streambuf 511
 Init, ios_base 474
 ios_base 484
 sentry, basic_istream 540
 sentry, basic_oiostream 570
 strstreambuf 657
Diagnostics Library 67–73
**d
 Numeric_limits 45
Distance 341
divides
 functional 84
do_date_order 216
do_get_date 216
do_get_monthname 217
do_get_time 216
do_get_weekday 216
do_get_year 217
Do_is
 locale 170
Do_narrow
 locale 171
do_put, time_put 227
Do_scan_is
 locale 170
Do_scan_not
 locale 170
Do_tolower
 locale 171
Do_toupper
 locale 171
Do_widen
 locale 171
Domain_error 68
 constructor 69
Dynamic memory management 52

E
 eback 524

 egptr 524
 Empty
 stack 295
 endl 584
 ends 584
 eof 114, 496
 eptr 525
 epsilon
 Numeric_limits 45
 eq 112
 eq_int_type 114
Equal
 algorithm 375
 istreambuf_iterator 357
Equal_range
 algorithm 388
 map 312
 multimap 317
equal_to
 functional 85
Equality Comparisons 75
Erase
 deque 284
 list 288
 vector 300
errno.h 73
Error numbers 73
Exception
 assignment operator 61
 destructor 61
 what 61
Exception classes 67
Exception handling 60
Exceptions
 constructor 61
exceptions
 basic_ios 500
exit
 Numeric_limits 52
exp 448
External "C" Linkage 36
Extractors
 basic_istream, arithmetic 541
 basic_istream, characters 542
 overloading 544**

F

fail 497
Failed
 `ostreambuf_iterator` 359
Falseename
 `numpunct` 197
File Based Streams 621–648
Fill
 algorithm 380
fill 490
Fill_n
 algorithm 380
Find
 algorithm 372
 map 311
 multimap 316
find 113
Find_end
 algorithm 373
Find_first_of
 algorithm 373
Find_if
 algorithm 373
fixed 503
flags 475
Flip
 `bitset` 330
float_denorm_style
 `Numeric_limits` 49
float_round_style
 `Numeric_limits` 49
flush 581
fmtflags 471
For_each
 algorithm 372
Formatting and Manipulators 533–595
Forward Declarations 455–460
Forward Iterators 336
Fpos 467
Frac_digits
 `moneypunct` 253
Freestanding Implementations 34
freeze
 `ostrstream` 668
 `strstream` 672
 `strstreambuf` 657

Front_insert_iterator
 constructor 349
 `front_inserter` 350
 operator = 349
 operators 349
Front_inserter
 `front_insert_iterator` 350
fstream 621
Functional 81
 Adaptors for pointers to functions 91
 Adaptors for pointers to members 92
 Arithmetic operations 84
 bind1st 90
 bind2nd 91
 binder1st 89
 binder2nd 90
 Binders 89
 Comparisons 85
 Logical operations 87
 mem_fun_t 92
 mem_fun1_t 93
 Negators 87
 pointer_to_binary_function 91, 92
 pointer_to_unary_function 91
functional
 divides 84
 equal_to 85
 greater 86
 greater_equal 86
 less 86
 less_equal 86
 logical_and 87
 logical_not 87
 logical_or 87
 minus 84
 modulus 85
 multiplies 84
 not_equal_to 85
 plus 84

G

gbump 524
gcount 547
General Utilities Library 75–109
Generate
 algorithm 380
Generate_n
 algorithm 380

G
 Get
 messages 262
 money_get 248
 num_get 192
 get 108, 549
 get_date 215
 get_monthname 215
 get_state 114
 get_temporary_buffer 102
 get_time 215
 get_weekday 215
 get_year 216
 getline 551
 getloc
 basic_streambuf 512
 ios_base 482
 Global
 locale 164
 good 496
 gptr 524
 greater
 functional 86
 greater_equal
 functional 86
 Grouping
 moneypunct 252
 numpunct 197
 Gslice 419
 constructors 420
 size 420
 start 420
 stride 421
 Gslice_array 421
 assignment operations 422
 assignment operator 422
 constructors 422
 fill operator 423

H
 Handler Function 27
 has_denorm
 Numeric_limits 47
 has_denorm_loss
 Numeric_limits 47
 Has_facet
 locale 164
 has_infinity

Numeric_limits 46
 has Quiet_NaN
 Numeric_limits 46
 has_Signaling_NaN
 Numeric_limits 46
H
 Hash
 collate 203
 Hash Libraries 713–743
 Headers 116
 algorithm 361
 cmath 430
 cstdlib 431
 fstream 621
 functional 81
 ios 465
 iosfwd 455
 iostream 461
 istream 533
 iterator 337
 msl_mutex.h 675
 numeric 427
 streambuf 507
 stringfwd 458
 strstream 652
 utility 78
 valarray 401
 hex 503

I
 I/O Library Summary 451
 ignore 553
 imag 446
 complex 441
 imbue
 basic_filebuf 629
 basic_ios 490
 basic_streambuf 526
 iosbase 482
In
 codecvt 184
 in_avail 517
 Includes
 algorithm 389
 Indirect_array 425
 assignment operations 426
 assignment operator 426
 constructors 426
 indirect_array

fill operator 427
infinity
 Numeric_limits 47
Inner_product 428
Inplace_merge
 algorithm 389
Input and Output Library 451–453
Input iterators 336
Insert
 deque 283
 list 288
 vector 300
Insert Iterators 347
Insert_iterator
 constructors 351
 inserter 351
 operator * 351
 operator = 351
Inserter
 insert_iterator 351
Inserters
 basic_ostream, arithmetic 571
 basic_ostream, characters 573
 overloading 575
int_type 114
internal 502
Introduction 21–23
Invalid_argument 69
 constructor 69
ios 465
ios_base 468
 constructors 484
 failure 470
 constructor 471
 what 471
 flags 475
 fmtflags 471
 getloc 482
 imbue 482
 Init 474
 destructor 474
 iostate 472
 iword 482
Open Modes 473
precision 479
pword 483
register_callback 483
seekdir 473
setf 477
sync_with_stdio 484
unsetf 478
width 480
xalloc 482
iosfwd 455
iostate 472
iostream 461
Iostream Base Class 465–505
Iostream Class Templates 27
Iostream Objects 461–464
Iostreams Definitions 452
Iostreams requirements 452
Is
 locale 168
is_bounded
 Numeric_limits 48
is_exact
 Numeric_limits 45
is_iec559
 Numeric_limits 48
is_integer
 Numeric_limits 45
is_modulo
 Numeric_limits 48
is_open
 basic_filebuf 625
 basic_fstream 646
 basic_ifstream 633
 basic_ofstream 639
is_signed
 Numeric_limits 45
is_specialized
 Numeric_limits 44
istream 533
Istream_iterator
 constructors 353
 destructor 353
 operations 353
Istreambuf_iterator
 constructor 356
 equal 357
 operators 357
istrstream 662
 constructor 663
 destructor 664
rdbuf 664

str 665
 Iter_swap
 algorithm 378
 Iterator 337
 advance 341
 distance 341
 Iterator Primitives 339
 Iterator Traits 339
 Iterators
 basic 340
 bidirectional 337
 forward 336
 input 336
 insert iterators 347
 Operation 341
 output 336
 predefined 342
 Random Access 337
 requirements 336
 reverse 342
 Iterators Library 335–359
 iword 482

L

Language Support Library 41–66
 Leading Underscores 35
 left 502
 Length
 codecvt 184
 length 113
 Length_error 69
 constructor 70
 less
 functional 86
 Less than comparison 76
 less_equal
 functional 86
 Lexicographical_compare
 algorithm 395
 Library-wide Requirements 33
 Linkage 35
 List 284
 assign 287
 clear 289
 constructor 287
 erase 288
 insert 288

merge 290
 pop_back 289
 pop_front 288
 push_back 288
 push_front 288
 remove 289
 resize 287
 reverse 290
 sort 290
 splice 289
 swap 290
 unique 290

Locale

facet 161
 category 159
 character classification 165
 character conversions 165
 class ctype 167
 class type_byname 171
 classic 164
 combine 163
 constructor 162
 ctype category 166
 ctype specializations 176
 do_is 170
 do_narrow 171
 do_scan_is 170
 do_scan_not 170
 do_tolower 171
 do_toupper 171
 do_widen 171
 global 164
 has_facet 164
 is 168
 name 163
 narrow 170
 operator != 163
 operator () 163
 Operator == 163
 scan_is 169
 scan_not 169
 tolower 166, 169
 toupper 165, 169
 use_facet 164
 widen 169

locale

id 161

Locale Types 159

Locales

 basic_streambuf 512

Localization Library 151–269

Lock

 mutex 677

log 448

log10 449

Logic_error 68

 constructor 68

Logical operations

 Functional 87

logical_and

 functional 87

logical_not

 functional 87

logical_or

 functional 87

Lower_bound

 algorithm 387

 map 311

 multimap 317

lt 112

M

Make_heap

 algorithm 393

Make_pair

 pair 81

Manipulator

 Overloading 593

 scientific 503

Manipulators

 adjustfield 502

 basefield 502

 boolalpha 501

 dec 503

 endl 584

 ends 584

 fixed 503

 floatfield 503

 flush 585

 fmtflags 501

 hex 503

Instantiations 588

internal 502

ios_base 500

left 502

noboolalpha 501

noshowbase 501

noshowpoint 501

noshowpos 501

noskipws 501

nounitbuf 502

nouppercase 502

oct 503

overloaded 504

right 502

showbase 501

showpoint 501

showpos 501

skipws 501

uppercase 501

ws 564

Map 307

 constructor 311

 equal_range 312

 find 311

 lower_bound 311

 swap 312

 upper_bound 312

Mask_array 423

 Assignment operations 424

 assignment operator 424

 constructors 424

 fill operator 425

Max

 algorithm 394

 valarray 410

max

 Numeric_limits 44

Max_element

 algorithm 394

max_exponent

 Numeric_limits 46

max_exponent10

 Numeric_limits 46

Max_length

 codecvt 184

max_size 99

mem_fun_ref_t 93

 template function 94

mem_fun_t

 Functional 92

 template function 93

mem_fun1_ref_ 94
mem_fun1_t
 Functional 93
 template function 93
Memory
 address 99
 allocate 99
 allocator globals 100
 auto_ptr conversions 108
 construct 100
 deallocate 99
 destroy 100
 get 108
 get_temporary_buffer 102
 max_size 99
 operator
 auto_ptr 109
 operator * 101
 operator auto_ptr_ref 108
 operator!= 100
 operator* 107
 operator== 100
 operator-> 107
 raw_storage_iterator 100
 constructor 101
 release 108
 reset 108
 return_temporary_buffer 102
Specialized Algorithms 102
uninitialized_copy 103
uninitialized_fill 103
Merge
 algorithm 388
 list 290
Message Retrieval Category 260
Messages
 close 262
 get 262
 open 262
Min
 algorithm 393
 valarray 410
min
 Numeric_limits 44
Min_element
 algorithm 394
min_exponent
 Numeric_limits 46
min_exponent10
 Numeric_limits 46
minus
 functional 84
Mismatch
 algorithm 375
Modifier Function 27
modulus
 functional 85
Monetary Category 241
Money_get
 get 248
Money_put
 put 250
Moneypunct
 curr_symbol 252
 decimal_point 251
 frac_digits 253
 grouping 252
 negative_sign 252
 pos_format 253
 positive_sign 252
 thousands_sep 252
move 113
MSL C++ Debug Mode 699
MSL Debug Mode 699–711
Msl_mutex.h 675–679
msl_mutex.h 675
Msl.Utility 681–698
Mslconfig 745–758
Multimap 313
 constructors 316
 equal_range 317
 find 316
 lower_bound 317
 swap 317
multiplies
 functional 84
Multiset 321
 constructor 324
 swap 324
Mutex 676
 lock 677
 operator = 677
 Public Member Functions 677
 unlock 678
mutex
 Constructor 677

 Destructor 677
Mutex_lock 678
 constructor 678
 destructor 679
 operator = 678

N

Name
 locale 163
name
 type_info 58
Narrow
 locale 170
Narrow-oriented Iostream Classes 27
neg_format 253
negate
 85
Negative_sign
 moneypunct 252
Negators
 Functional 87
New 53
new_handler 56
Next_permutation
 algorithm 395
noboolalpha 501
None
 bitset 331
Non-member functions
 valarray 411
norm 447
noshowpoint 501
noshowpos 501
noskipws 501
not_eof 113
not_equal_to
 functional 85
not1 88
not2 89
nounitbuf 502
nouppercase 502
NTCTS 27, 112
Nth_element
 algorithm 386
Num_get
 get 192
 Num_put
 put 195
 Numeric Category 190
 Numeric limits 42
 Numeric Punctuation Facet 195
 Numeric_limits
 abort 51
 atexit 51
 denorm_min 48
 digits 45
 epsilon 45
 exit 52
 float_denorm_style 49
 float_round_style 49
 has_denorm 47
 has_denorm_loss 47
 has_infinity 46
 has_quiet_NaN 46
 has_signaling_NaN 46
 infinity 47
 is_bounded 48
 is_exact 45
 is_ie559 48
 is_integer 45
 is_modulo 48
 is_signed 45
 is_specialized 44
 max 44
 max_exponent 46
 max_exponent10 46
 min 44
 min_exponent 46
 min_exponent10 46
 quiet_NaN 47
 radix 45
 round_error 45
 round_style 49
 signaling_NaN 47
 Static Members 44
 tinyness_before 48
 traps 48
 Numerics Library 399–432
 Numpunct
 decimal_point 196
 falseename 197
 grouping 197
 thousands_sep 197
 trueename 197

O

Object State 27
 Observer Function 28
 oct 503
 off_type 114
 Open
 basic_filebuf 625
 basic_fstream 646
 basic_ifstream 634
 basic_ofstream 640
 messages 262
 Open Modes
 basic_filebuf 626
 basic_fstream 646
 basic_ifstream 634
 basic_ofstream 640
 ios_base 473
 Operator 79, 79, 80, 292, 328, 331, 333, 445
 != 79
 *
 Memory 101
 > 79
 >= 79
 delete 54
 placement 55
 new 53
 placement 55
 Operator -
 complex 444
 Operator !
 basic_ios 491
 Operator !=
 bitset 331
 complex 445
 locale 163
 Operator &
 bitset 332
 Operator &=
 bitset 328
 Operator ()
 locale 163
 Operator * 101, 107
 complex 444
 insert_iterator 351
 Operator *=
 complex 442
 Operator +
 complex 443
 Operator +=
 complex 441
 Operator /
 complex 444
 Operator /=
 complex 442
 Operator -=
 complex 442
 Operator =
 complex 440
 front_insert_iterator 349
 insert_iterator 351
 Operator ==
 bitset 330
 complex 445
 locale 163
 pair 80
 queue 292
 Operator >>
 bitset 331, 332
 complex 445
 Operator >>=
 bitset 329
 Operator ^
 bitset 332
 Operator ^=
 bitset 328
 Operator |=
 bitset 332
 Operator |==
 bitset 328
 Operator ~
 bitset 329
 Operator bool
 basic_ios 491
 sentry, basic_istream 540
 sentry, basic_ostream 571
 Operator!=
 Memory 100
 type_info 58
 utility 79
 Operator()
 Functional 89, 90, 91, 92
 Operator++ 102
 Operator= 101
 Auto_ptr 107
 mutex 677

mutex_lock 678
Operator==
 Memory 100
 type_info 57
Operator-> 107
Operator>
 utility 79
Operator>=
 utility 79
Operators
 back_insert_iterator 348
 reverse_iterator 344
 Utility 79
ostream cerr 463
ostream clog 463
ostream cout 462
Ostream_iterator
 constructors 355
 destructor 355
Ostream_iterator Operations 355
Ostreambuf_iterator
 constructor 358
 failed 359
Ostreambuf_iterator Operations 358
ostrstream 666
 constructor 666
 destructor 667
 freeze 668
 pcount 669
 rdbuf 670
 str 670
Other Conventions 31
Other Runtime Support 64
Out
 codecvt 183
Out_of_range 70
 constructor 70
Output Iterators 336
overflow 530, 662
 basic_filebuf 628
 basic_streambuf 603
Overflow_error 71
 constructor 72
Overloaded
 manipulators 504
Overloading
 Extractors 544
Inserters 575
Manipulator 593
P
Pair
 Constructors 80
 make_pair 81
 Operator 80
 Operator == 80
 Utility 80
Partial_sort
 algorithm 386
Partial_sort_copy
 algorithm 386
partial_sum 429
Partition
 algorithm 384
pbackfail 529, 661
 basic_filebuf 628
 basic_streambuf 603
pbase 525
pbump 525
pcount
 ostrstream 669
 strstream 672
 strstreambuf 658
peek 554
Placement Operator Delete 55
Placement Operator New 55
plus
 functional 84
pointer_to_binary_function
 Functional 91, 92
pointer_to_unary_function
 Functional 91
polar 447
Pop
 priority_queue 294
 stack 296
Pop_back
 list 289
Pop_front
 list 288
Pop_heap
 algorithm 392
Pos_format
 moneypunct 253

pos_type 114
Positive_sign
 moneypunct 252
pow 449
pptr 525
precision 479
Predefined Iterators 342
Predicate
 not1 88
 not2 89
Prev_permutation
 algorithm 396
Priority_queue 292
 constructors 293
 pop 294
 push 294
Program-defined Facets 269
pubimbue 512
pubseekoff 513
pubseekpos 514
pubsetbuf 512
pubsync 516
Push
 priority_queue 294
 stack 296
Push_back
 list 288
Push_front
 list 288
Push_heap
 algorithm 392
Put
 money_put 250
 num_put 195
put 579
put_time_put 227
putback 558
pword 483

Q

Qsort 397
Queue 291
 operator 292
 operator == 292
quiet_NaN
 Numeric_limits 47

R

radix
 Numeric_limits 45
Random Access Iterators 337
Random_shuffle
 algorithm 384
Range_error 71
 constructor 71
Raw storage iterator 100
Raw_storage_iterator
 constructor 101
 operator = 101
 operator++ 102
raw_storage_iterator 101, 102
rdbuf 488
 basic_fstream 644
 basic_ifstream 632
 basic_istringstream 607
 basic_ofstream 638
 basic_ostringstream 611
 basic_stringstream 617
 istrstream 664
 ostrstream 670
 strstream 672
rdstate 491
read 555
readsome 556
real 446
 complex 441
Reentrancy 38
register_callback 483
release 108
Remove
 algorithm 381
 list 289
Remove_copy
 algorithm 381
Remove_copy_if
 algorithm 381
Remove_if
 algorithm 381
Replace
 algorithm 379
Replace_copy
 algorithm 379
Replace_copy_if
 algorithm 379

Replacement Function 28
Replacement Functions 36
Repositional Stream 28
Required Behavior 28
Reserved Function 28
Reserved Names 35
Reset
 bitset 329
reset 108
resetiosflags 588
Resize
 deque 283
 list 287
 valarray 411
 vector 300
Restrictions On Exception Handling 38
return_temporary_buffer 102
Reverse
 algorithm 382
 list 290
Reverse iterators 342
Reverse_copy
 algorithm 383
Reverse_iterator
 base 344
 constructor 344
 operators 344
right 502
Rotate
 algorithm 383
Rotate_copy
 algorithm 383
round_error
 Numeric_limits 45
round_style
 Numeric_limits 49
Runtime_error 70
 constructor 71

S

sbunc 518
Scan_is
 locale 169
Scan_not
 locale 169
scientific 503
Search

algorithm 376
Search_n
 algorithm 376
seekdir 473
seekg 562
seekoff 527
 basic_filebuf 628
 basic_streambuf 603
 strstreambuf 660
seekp 577
seekpos 527
 basic_filebuf 629
 basic_streambuf 604
 strstreambuf 661
sentry 539, 570
constructor
 basic_istream 540
 basic_ostream 570
destructor
 basic_istream 540
 basic_ostream 570
Operator bool
 basic_istream 540
 basic_ostream 571
Sequences 274
Sequences Requirements 272
Set 318
 bitset 329
 constructors 320
 swap 321
Set_difference
 algorithm 391
Set_intersection
 algorithm 390
Set_symmetric_difference
 algorithm 391
Set_union
 algorithm 390
setbase 589
setbuf 526, 629
 strstreambuf 660
setf 477
setfill 590
setg 524
setiosflags 589
setp 525
setprecision 591

setstate 495
setw 592
sgetc 519
sgetn 520
Shift
 valarray 410
showmanc 528
showmanyc
 basic_filebuf 627
showpoint 501
showpos 501
Sice_array 418
signaling_NaN
 Numeric_limits 47
sin 449
sinh 449
Size
 bitset 330
 gslice 420
 slice 417
 stack 295
 valarray 410
skipws 501
Slice
 constructors 417
 size 417
 start 417
 stride 417
Slice_array
 assignment operations 419
 assignment operator 419
 constructor 418
 fill operator 419
snextc 517
Sort
 algorithm 385
 list 290
Sort_heap
 algorithm 393
Splice
 list 289
sputback 520
sputc 522
sputn 523
sqrt 450
Stable_partition
 algorithm 384
Stable_sort
 algorithm 385
Stack 294
 constructors 295
 empty 295
 pop 296
 push 296
 size 295
 top 296
Standard Iterator Tags 340
Standard Locale Categories 166
Start
 gslice 420
 slice 417
state_type 114
Static Members
 Numeric_limits 44
str
 basic_istringstream 608
 basic_ostringstream 613
 basic_streampbuf 601
 basic_stringstream 618
 istrstream 665
 ostrstream 670
 strstream 673
 strstreibuf 659
Stream
 buffering 462
Stream Buffers 507–531
Stream Iterators 351
streampbuf 507
Stride
 gslice 421
 slice 417
string 116
String Based Streams 597–619
Stringfwd 458
Strings Library 111–150
Strstream 651–673
strstream 652, 666
 constructor 656, 671
 destructor 672
 freeze 672
 pcount 672
 rdbuf 672
 str 673
strstreibuf 654
 freeze 657

-
- pcount 658
 - seekoff 660
 - seekpos 661
 - setbuf 660
 - str 659
 - stiostream
 - overflow 662
 - strstream
 - pbackfail 661
 - underflow 661
 - strtreambuf
 - destructor 657
 - struct char_traits 115
 - subset 408
 - valarray 408
 - Sum
 - valarray 410
 - sungetc 522
 - Supported locale names 151
 - Swap
 - algorithm 377
 - deque 284
 - list 290
 - map 312
 - multimap 317
 - multiset 324
 - set 321
 - vector 300
 - swap
 - basic_string 146
 - Swap_ranges
 - algorithm 378
 - sync 561
 - basic_filebuf 629
 - basic_streambuf 527
 - sync_with_stdio
 - ios_base 484
 - T**
 - tan 450
 - tanh 450
 - tellg 562
 - tellp 577
 - Template Class Codecvt 182
 - Template Class Codecvt_byname 186
 - Template Class Collate 202
 - Template Class Collate_byname 204
 - Template Class Messages 260
 - Template Class Messages_byname 265
 - Template Class Money_get 247
 - Template Class Money_put 249
 - Template Class Moneypunct 250
 - Template Class Moneypunct_byname 256
 - Template Class Num_get 190
 - Template Class Num_put 193
 - Template Class Num_punct 196
 - Template Class Num_punct_byname 198
 - Template Class Time_get 213
 - Template Class Time_get_byname 225
 - Template Class Time_put 226
 - Template Class Time_put_byname 227
 - terminate 64
 - terminate_handler 63
 - Test
 - bitset 331
 - Thousands_sep
 - moneypunct 252
 - num_punct 197
 - tie 487
 - Time Category 213
 - Time_put
 - Virtual functions 227
 - time_put,do_put 227
 - timeput,put 227
 - tinyness_before
 - Numeric_limits 48
 - to_char_type 113
 - to_int_type 114
 - To_string
 - bitset 330
 - To_ulong
 - bitset 330
 - Tolower
 - locale 166, 169
 - Top
 - stack 296
 - Toupper
 - locale 165, 169
 - Traits 28
 - traits 112
 - Transform
 - collate 203
 - Translation Units 35

| | |
|---|--|
| traps Numeric_limits 48 | Use_facet locale 164 |
| Truename numpunct 197 | Using the library 35 |
| Transform algorithm 378 | Utility 78 Operator 79 Operator!= 79 Operator> 79 Operator>= 79 Operators 79 Pair 80 |
| Type identification 57 | Valarray 401 apply 411 assignment operations 409 binary operators 411 constructors 406 cshift 411 destructor 407 logical operators 413 max 410 min 410 non-member functions 411 resize 411 shift 410 size 410 sum 410 transcendentals 415 unary operators 408 |
| Type_info assignment operator 58 before 58 constructor 58 name 58 operator != 58 operator== 57 | Vector 296, 301 assign 299 capacity 299 constructors 299 erase 300 insert 300 resize 300 swap 300 |
| U | wcerr 464 wcin 463 wclog 464 wcout 463 |
| uflow 529 | What bad_cast 59 bad_exception 62 bad_typeid 60 exception 61 |
| Unary operators valarray 408 | what bad_alloc 56 |
| Unary_function 83 | Widen locale 169 |
| Unary_negate 87 | |
| uncaught_exception 64 | |
| underflow 528 basic_filebuf 628 basic_streambuf 602 strstreambuf 661 | |
| Underflow_error 72 constructor 72 | |
| unexpected 63 | |
| unexpected_handler 63 | |
| unget 559 | |
| uninitialized_copy 103 | |
| uninitialized_fill 103 | |
| Unique algorithm 382 list 290 | |
| Unique_copy algorithm 382 | |
| Unlock mutex 678 | |
| unsetf 478 | |
| Unshift codecvt 184 | |
| Upper_bound algorithm 387 map 312 | |
| uppercase 501 | |

Index

Wide-oriented Iostream Classes 28

width 480

wistream wcin 463

wostream wcerr 464

wostream wclog 464

wostream wcout 463

write 580

ws 564

xalloc 482

xsgetn 528

xsputn 530