CodeWarrior®
MSL C++ Reference

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 2201 Donley Drive, Suite 310 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Mail order	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`

America Online	keyword: `Metrowerks`
CompuServe	goto `Metrowerks`

Table of Contents

1

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.

Chapter 2 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. See "Overview of the MSL C++ Reference" on page 51.

Chapter 3 discusses the ANSI/ISO language support library. See "Overview of Language Support Libraries" on page 55.

Chapter 4 elaborates on the diagnostics library that provides a consistent framework for reporting errors in a C++ program, including predefined exception classes. See "Overview of Diagnostics LIbrary" on page 71.

Chapter 5 talks about the general utilities library, which includes components used by other library elements, such as predefined storage allocator for dynamic storage management. See "Overview of General Utilities Libraries" on page 79.

Chapter 6 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. See "Overview of Strings Classes" on page 101.

Chapter 7 covers the localization components extend internationalization support for character classification, numeric, monetary, and date/time formatting and parsing among other things. See "Overview of the Localization Library" on page 123.

Chapter 8 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. See "Overview of the Containers Library" on page 161.

Chapter 9 discusses iterator classes. See "Overview of Iterators" on page 263.

Chapter 10 discusses the algorithms library. This library provides sequence, sorting, and general numerics algorithms. See "Overview of the Algorithms Library" on page 295.

Chapter 11 discusses the numerics library. It describes the components for complex number types, numeric arrays, generalized numeric algorithms and facilities included from the ISO C library. See "Overview of the Numerics Library" on page 347.

Chapters 12-20 discuss the iostreams components that are the primary mechanism for C++ program input/output. They can be used with other elements of the library, particularly strings, locales, and iterators.

2

MSL C++ Reference

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

Overview of the MSL C++ Reference

This section introduces you to the definitions, conventions, terminology, and other aspects of the MSL C++ library. The topics discussed are:

- "Definitions" on page 51
- "Features not implemented in MSL C++" on page 52
- "Multi-Thread Safety" on page 54

Definitions

This section discusses the meaning of certain terms in the MSL C++ library. The topics discussed are:

- "Character Sequences" on page 51
- "Byte Strings" on page 52
- "Multibyte Strings" on page 52

Character Sequences

The Standard C/C++ library makes widespread use of characters and character sequences that follow a few uniform conventions:

A letter is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.

The decimal-point character is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented by a period,'.', which is also its value in the `"C"` locale, but may change during program execution by a call to `setlocale(int, const char*)`, or by a change to a locale object.

A character sequence is an array object A that can be declared as `T A[N]`, where `T` is any of the types `char, unsigned char,` or `signed char,` optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value `S` that points to its first element.

Byte Strings

A null-terminated byte string, or `NTBS`, is a character sequence which has a value zero appended to its contents. 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 an including the terminating null character.

Multibyte Strings

A null-terminated multibyte string, or `NTMBS`, is an `NTBS` that constitutes a sequence of valid multibyte characters, beginning and ending in the initial shift state. An `NTBS` that contains characters only from the basic character set is also an `NTMBS.` Each multibyte character then consists of a single byte.

Features not implemented in MSL C++

The following sections of April 1995 draft will not be implemented in the MSL C++ product.

- “Wide-Character Sequences” on page 53
- “Template Functionality” on page 53
- “ANSI/ISO Library Functionality” on page 54

Chapter 18 of the Draft Working Paper refers to Language support library, and depends on how a compiler:

- associates type information with an object
- handles placement new and placement delete
- takes care of terminate, unexpected, exit, atexit functions

Wide-Character Sequences

A wide-character sequence is an array object A that can be declared as `T A[N]`, where `T` is type `wchar_t`, optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value `S` that designates its first element.

A null-terminated wide-character string, or `NTWCS`, is a wide-character sequence which has a value zero appended to its contents.

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.

Template Functionality

There are several areas of the Standard library that depend on C++ features like member templates 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 and time related facets in Locale
- Support for wchar_t in basic_filebuf class
- Multi-thread safety.

Multi-Thread Safety

MSL C++ Library will be multi-thread safe provided the operating system supports thread-safe system calls. Library will have locks at appropriate places in the code for thread safety. The locks will be implemented as a mutex class -- the implementation of which differs from platform to platform.

This will ensure that the library is MT-Safe internally. For example, if a buffer is shared between two ios class objects, then only one ios 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. At present, `basic_string class` is only made multi-thread safe.

3

Language Support Library

This chapter is a reference guide to the ANSI/ISO language support library.

Overview of Language Support Libraries

This clause in the April 1995 WPD describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. This chapter describes the headers that declare these function signatures and define any related types. It also describes common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, and other runtime support.

The topics discussed are:

- "Types and Macros" on page 55
- "Numeric Limits" on page 58
- "Dynamic Memory Storage Allocation Errors" on page 66

Types and Macros

This section discusses types and macros. The topics are:

- "Macros" on page 56
- "Type Implementations" on page 56

Macros

The macros defined are:

- "NULL" on page 56
- "offsetoff" on page 56
- "sizeof" on page 56

NULL

Description The macro **`NULL`** is an implementation-defined C++ null-pointer constant in this International Standard. In MSL this evaluates to zero.

Source file `<cstddef.h>`

offsetoff

Description The macro **`offsetoff`** accepts a restricted set of `type` arguments in this International Standard. `type` shall be a POD[1] structure or a POD union.

Source file `<cstddef.h>`

sizeof

Description The **`sizeof`** operator yields the number of bytes in the object representation of its operand. The operand can be either an expression, which is not evaluated, or a parenthesized `type-id`.

Source file `<cstddef.h>`

Type Implementations

There are several fundamental types. They are:

1. A POD structure or union is a union that has no members that are of type pointer to members.The acronym for POD stands for "Plain Old Data".

Character types

Objects declared as characters(**`char`**) shall be large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character object, its value shall be equivalent to the integer code of that character. It is implementation-defined whether a char object can take on negative values. Characters can be explicitly declared `unsigned` or `signed`. Plain `char`, `signed char`, and `unsigned char` are three distinct types. A `char`, a `signed char`, and an `unsigned char` occupy the same amount of storage and have the same alignment requirements; that is, they have the same object representation. For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers.

Enumeration

An `enumeration` comprises a set of named `integer` constant values, which form the basis for an integral subrange that includes those values. Each distinct `enumeration` constitutes a different enumerated type. Each constant has the type of its `enumeration`.

Integral types

There are four signed `integer` types: `signed char`, `short int`, `int` and `long int`. In this list, each type provides at least as much storage as those preceding it in the list, but the implementation can otherwise make any of them equal in storage size. Plain `ints` have the natural size suggested by the machine architecture; the other signed integer types are provided to meet special needs.

For each of the `signed integer` types, there exists a corresponding `unsigned integer` type: `unsigned char`, `unsigned short int`, `unsigned int` and `unsigned long int`, each of which occupies the same amount of storage and has the same alignment requirements as the corresponding signed integer type.

Wide character types

Type `wchar_t` is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

Bool type

Values of type `bool` are either `true(1)` or `false(0)`. There are no `signed`, `unsigned`, `short`, or `long bool` types or values.

Floating point types

There are three `floating point` types: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much as precision as `double`.

Void type

The `void` type has an empty set of values. It is used as the return type for functions that do not return a value.

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 will be provided for each fundamental type, both floating point and integer, including `bool`. The member `is_specialized` shall be `true` for all such specializations of `numeric_limits`.

The only class discussed is:

- “Template Class numeric_limits” on page 59

Template Class numeric_limits

This section discusses the numeric_limits template class. The functions discussed are:

Table 3.1 numeric_limits functions

numeric_limits::is_specialized	numeric_limits::max_exponent10
numeric_limits::min	numeric_limits::has_infinity
numeric_limits::max	numeric_limits::has_quiet_NaN
numeric_limits::digits	numeric_limits::has_signaling_NaN
numeric_limits::digits10	numeric_limits::has_denorm
numeric_limits::is_signed	numeric_limits::infinity
numeric_limits::is_integer	numeric_limits::quiet_NaN
numeric_limits::is_exact	numeric_limits::signaling_NaN
numeric_limits::radix	numeric_limits::denorm_min
numeric_limits::epsilon	numeric_limits::is_bounded
numeric_limits::round_error	numeric_limits::is_modulo
numeric_limits::min_exponent	numeric_limits::traps
numeric_limits::min_exponent10	numeric_limits::tinyness_before
numeric_limits::max_exponent	numeric_limits::round_style

numeric_limits::is_specialized

Description The member `is_specialized` makes it possible to distinguish between scalar types, which have specializations, and non-scalar types, which do not.

Definition

```
static const bool is_specialized;
```

numeric_limits::min

Description This function returns the minimum finite value that can be `CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN`, etc. For this function, floating types with denormalization, return the minimum positive normalized value, `denorm_min`. The value returned by this function is meaningful for all specializations in which `is_bounded==true`, or `is_bounded==false` and `is_signed==false`.

Prototype

```
static T min();
```

numeric_limits::max

Description This function returns the maximum finite value that can be `CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX`, etc. The value returned by this function is meaningful for all specializations in which `is_bounded==true`.

Prototype

```
static T max();
```

numeric_limits::digits

Description It stores the number of radix digits that can be represented without change. For built-in integer types, it denotes the number of non-sign bits in the representation. For floating point types, it denotes the number of radix digits in the mantissa which can be `FLT_MANT_DIG`, `DBL_MANT_DIG` or `LDBL_MANT_DIG`.

Definition `static const int digits;`

numeric_limits::digits10

Description It stores the number of base 10 digits that can be represented without change. The value can be one among `FLT_DIG, DBL_DIG` or `LDBL_DIG`. It is meaningful for all specializations in which `is_bounded==true`.

Definition `static const int digits10;`

numeric_limits::is_signed

Description It stores a value `true` if the type is signed. It is meaningful for all specializations.

Definition `static const bool is_signed;`

numeric_limits::is_integer

Description It stores a value that is `true`, if the type is integer. It is meaningful for all specializations.

Definition `static const bool is_integer;`

numeric_limits::is_exact

Description It stores a value `true` if the type uses an exact representation. All integer types are exact, but not vice versa. For example, rational and fixed-exponent representations are exact but not integer. It is meaningful for all specializations.

Definition `static const bool is_exact;`

numeric_limits::radix

Description It stores a value that specifies the base of radix of the exponent representation for floating types. Its value is usually 2 or can be `FLT_RADIX`. For integer types it specifies the base of the representation. It is meaningful for all specializations.

Definition

```
static const int radix;
```

numeric_limits::epsilon

Description It returns the difference between 1 and the least value greater than 1 that is representable. The value can be either `FLT_EPSILON`, `DBL_EPSILON` or `LDBL_EPSILON`. The value returned is meaningful only for floating point types.

Prototype

```
static T epsilon();
```

numeric_limits::round_error

Description It returns a value that denotes the maximum rounding error that is permitted. It is meaningful only for floating point types.

Prototype

```
static T round_error();
```

numeric_limits::min_exponent

Description It stores the minimum negative integer such that `radix` raised to that power is in range. It is meaningful only for floating point types and the values can be `FLT_MIN_EXP`, `DBL_MIN_EXP`,`LDBL_MIN_EXP`.

Definition

```
static const int min_exponent;
```

numeric_limits::min_exponent10

Description It stores the minimum negative integer such that 10 raised to that power is in range. It is meaningful only for floating point types and the values can be `FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP.`

Definition

```
static const int min_exponent10;
```

numeric_limits::max_exponent

Description It stores the maximum positive integer such that radix raised to that power is in range. It is meaningful only for floating point types and the values can be `FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.`

Definition

```
static const int max_exponent;
```

numeric_limits::max_exponent10

Description It stores the maximum positive integer such that 10 raised to that power is in range. It is meaningful only for floating point types and the values can be FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP.

Definition

```
static const int max_exponent10;
```

numeric_limits::has_infinity

Description It stores a value `true` if the type has a representation for positive infinity. It is meaningful only for floating point types. It will be true for all specialization in which `is_iec559==true`.

Definition

```
static const bool has_infinity;
```

numeric_limits::has_quiet_NaN

Description It stores a value `true` if the type has a representation for a quiet(non-signaling) `Not a Number`. It is meaningful only for floating point types. The value will be true for all specializations in which `is_iec559==true`.

Definition

```
static const bool has_quiet_NaN;
```

numeric_limits::has_signaling_NaN

Description It stores a value `true` if the type has a representation for a signaling `Not a Number`. It is meaningful only for floating point types. The value will be true for all specializations in which `is_iec559==true`.

Definition

```
static const bool has_signaling_NaN;
```

numeric_limits::has_denorm

Description It stores `true` if the type allows denormalized values (i.e. variable number of exponent bits). The value is meaningful only for floating point types.

Definition

```
static const bool has_denorm;
```

numeric_limits::infinity

Description It returns the representation of positive infinity, if available. It is meaningful only for specializations for which `has_infinity==true`. Required in specializations for which `is_iec559==true`.

Prototype

```
static T infinity();
```

numeric_limits::quiet_NaN

Description This function returns the representation of a quiet `Not a Number` if `has_quiet_NaN==true`.

Prototype `static T quiet_NaN();`

numeric_limits::signaling_NaN

Description This function returns the representation of a signaling `Not a Number` if `has_signaling_NaN==true`.

Prototype `static T signaling_NaN();`

numeric_limits::denorm_min

Description It returns the minimum positive denormalized value. It is meaningful for all floating point types.   In this function specialization for which `has_denorm==false`, returns the minimum positive normalized value.

Prototype `static T denorm_min();`

numeric_limits::is_bounded

Description It stores a value `true` if the set of values representable by the type is finite. All built-in types are bounded, this member would be false for arbitrary precision types. It is meaningful for all specializations.

Definition `static const bool is_bounded;`

numeric_limits::is_modulo

Description It stores a value 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 than either of the two numbers.

This has a value `false` for all floating types, `true` for unsigned integers, and `true` for signed integers on most machines and is meaningful for all specializations.

Definition

```
static const bool is_modulo;
```

numeric_limits::traps

Description

It stores a value `true` if trapping is implemented for the type. It is meaningful for all specializations.

Definition

```
static const bool traps;
```

numeric_limits::tinyness_before

Description

It stores a value `true` if tinyness is detected before rounding. It is meaningful only for floating point types.

Definition

```
static const bool tinyness_before;
```

numeric_limits::round_style

Description

It stores the rounding style for the type. The value stored is meaningful for all floating point types. Specializations for integer types shall return `round_toward_zero`.

Definition

```
static const float_round_style round_style;
```

Dynamic Memory Storage Allocation Errors

This section discusses classes related to memory allocation errors. The classes discussed are:

- “Class bad_alloc” on page 67
- “Class bad_cast” on page 68
- “Class bad_typeid” on page 69

Class bad_alloc

The class `bad_alloc`, derived from the `class exception`, defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

Table 3.2 The member functions discussed are:

Constructor–bad_cast	Copy Constructor–bad_alloc
Assignment Operator–bad_alloc	bad_alloc::what

Constructor–bad_alloc

Description This function constructs an object of class `bad_alloc`.

Prototype
```
bad_alloc() throw();
```

Copy Constructor–bad_alloc

Description This function copies an object of class `bad_alloc`.

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

Assignment Operator–bad_alloc

Description This function copies an object of class `bad_alloc`.

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

bad_alloc::what

Description Our implementation returns a `string` object instead of `const char*`. This function returns the string associated with the exception if some allocation is done, else it returns `string()`.

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

Class bad_cast

The class `bad_cast`, derived from the `class exception`, defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid dynamic-cast expression.

Table 3.3 The member functions discussed are:

Constructor–bad_cast	Copy Constructor–bad_cast
Assignment Operator–bad_cast	bad_cast::what

Constructor–bad_cast

Description This function constructs an object of class `bad_cast`.

Prototype `bad_cast() throw();`

Copy Constructor–bad_cast

Description This function copies or constructs an object of class `bad_cast`.

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

Assignment Operator–bad_cast

Description This function copies an object of class `bad_cast`.

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

bad_cast::what

Description Our implementation returns a `string` object instead of `const char*`. This function returns the string associated with the exception if some allocation is done, else it returns `string()`.

Prototype

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

Class bad_typeid

The class `bad_typeid`, derived from the `class exception`, defines the type of objects thrown as exceptions by the implementation to report a null pointer in a typeid expression.

Table 3.4 The member functions discussed are:

Constructor–bad_typeid	Copy Constructor–bad_typeid
Assignment Operator–bad_typeid	bad_typeid::what

Constructor–bad_typeid

Description This function constructs an object of class `bad_typeid`.

Prototype

```
bad_typeid() throw();
```

Copy Constructor–bad_typeid

Description These functions copy an object of class `bad_typeid`.

Prototype

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

Assignment Operator–bad_typeid

Description This function copies an object of class `bad_typeid`.

Prototype

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

bad_typeid::what

Description

Our implementation returns a `string` object instead of `const char*`. This function returns the string associated with the exception if some allocation is done, else it returns `string()`.

Prototype

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

4

Diagnostics Library

This chapter is a reference guide to the ANSI/ISO exception classes, which are used for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes.

Overview of Diagnostics LIbrary

The standard C++ library provides classes to be used to report errors in a C++ program. In the error model reflected in these classes, errors are divided into two broad categories: logic errors and runtime errors. The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program, and are preventable. In contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header `<stdexcept.h>` defines several types of predefined exceptions for reporting errors in C++ program. These exceptions are related via inheritance.

The only group of classes in the diagnostics library are:

- “Exception Classes” on page 71

Exception Classes

There are several exception-related classes in the diagnostics library. The base class is Class exception. Other exceptoin classes derive from exception.

The classes are:

- “Class exception” on page 72
- “Class domain_error” on page 74

- “Class invalid_argument” on page 74
- “Class logic_error” on page 73
- “Class out_of_range” on page 75
- “Class runtime_error” on page 76
- “Class length_error” on page 75
- “Class overflow_error” on page 77
- “Class range_error” on page 76

Class exception

Description

The class exception defines the base class for the types of objects thrown as exceptions by C++ standard library components, and certain expressions, to report errors detected during program execution.

Table 4.1 The member functions discussed are:

Constructor–exception	Copy Constructor–exception
Assignment Operator–exception	Destructor–exception
exception::what	

Constructor–exception

Description

This function constructs an object of class `exception`. It does not throw any exceptions.

Prototype

```
exception() throw();
```

Copy Constructor–exception

Description

This function copies an exception object.

Prototype

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

Assignment Operator–exception

Description This function copies an exception object.

Prototype

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

Destructor–exception

Description This function destroys an object of `class exception`.

Prototype

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

exception::what

Description Our implementation returns a `string` object instead of `const char*`. This function returns the string associated with the exception if some allocation is done, else it returns `string()`.

Prototype

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

Class logic_error

Description The class `logic_error`, derived from "Class exception" on page 72, defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

Table 4.2 The member functions discussed are:

Constructor–logic_error

Constructor–logic_error

Description This function constructs an object of class `logic_error`.

Prototype `logic_error(const string& what_arg);`

Class domain_error

Description The class `domain_error`, derived from “Class logic_error” on page 73, defines the type of objects thrown as exceptions to report domain errors.

Table 4.3 The member functions discussed are:

Constructor–domain_error

Constructor–domain_error

Description This function constructs an object of class `domain_error`.

Prototype `domain_error(const string& what_arg);`

Class invalid_argument

Description The class `invalid_argument` defines the type of objects thrown as exceptions to report an invalid argument.

Table 4.4 The member functions discussed are:

Constructor–invalid_argument

Constructor–invalid_argument

Description This function constructs an object of class `invalid_argument`.

Prototype `invalid_argument(const string& what_arg);`

Class length_error

Description The class `length_error`, derived from “Class logic_error” on page 73, defines the type of objects thrown as exceptions to report an attempt to produce an object whose length equals or exceeds its maximum allowable size.

Table 4.5 The member functions discussed are:

Constructor–length_error

Constructor–length_error

Description This function constructs an object of class `length_error`.

Prototype

```
length_error(const string& what_arg);
```

Class out_of_range

Description The class `out_of_range`, derived from “Class logic_error” on page 73, defines the type of objects thrown as exceptions to report an argument value not in its expected range.

Table 4.6 The member functions discussed are:

Constructor–out_of_range

Constructor–out_of_range

Description This function constructs an object of class `out_of_range`.

Prototype

```
out_of_range(const string& what_arg);
```

Class runtime_error

Description The class runtime_error, derived from “Class exception” on page 72, defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

Table 4.7 The member functions discussed are:

Constructor–runtime_error

Constructor–runtime_error

Description This function constructs an object of class `runtime_error`.

Prototype

```
runtime_error(const string& what_arg);
```

Class range_error

Description The class `range_error`, derived from “Class runtime_error” on page 76, defines the type of objects thrown as exceptions to report range errors.

Table 4.8 The member functions discussed are:

Constructor–range_error

Constructor–range_error

Description This function constructs an object of class `range_error`.

Prototype

```
range_error(const string& what_arg);
```

Class overflow_error

Description

The class `overflow_error`, derived from “Class runtime_error” on page 76, defines the type of objects thrown as exceptions to report an arithmetic overflow error.

Table 4.9 The member functions discussed are:

Constructor–overflow_error

Constructor–overflow_error

Description

This function constructs an object of class `overflow_error`.

Prototype

```
overflow_error(const string& what_arg);
```

5

General Utilities Libraries

This chapter discusses the Allocator class.

Overview of General Utilities Libraries

Every STL container class uses an `Allocator class` to encapsulate information about the memory model being used by the program. The topics in this chapter are:

Allocator Classes

Different memory models have different requirements for pointers, references, integer sizes, etc. The `Allocator class` encapsulates information about pointers, constant pointers, references, constant references, sizes of objects, difference types between pointers, allocation and deallocation functions, and some other functions. The exact set of types and functions defined within the allocator are explained in the Default Allocator Interface, later in this chapter.

Since memory model information can be encapsulated in an allocator, STL containers can work with different memory models by simply providing different allocators. All operations on allocators are expected to be amortized constant time.

Additional topics are:

- "Passing Allocators to STL Containers" on page 80
- "Extracting Information from an Allocator Object" on page 80

Passing Allocators to STL Containers

Once an allocator class for a particular memory model has been written, it must be passed on to the STL container for that container to work properly in the concerned memory model. This is done by passing the allocator to the STL container as a template parameter.

For example, the `vector` container has the following interface:

```
template <class T, class Allocator = allocator>
class vector;
```

Here the `Allocator` parameter defaults to `allocator`. All the STL containers are not yet parameterized on Allocator. In all places where Allocator parameter is specified in the draft, default `allocator` is used.

Extracting Information from an Allocator Object

Allocator is obtained by a container by a macro of the same name. Once the allocator is known, the container must somehow extract the memory model information from the Allocator class. This information is extracted by simply accessing the `typedefs` and member functions of the `Allocator` class.

For example, the public interface of the vector class mentioned above contains the following `typedefs` to extract information about references and pointers from the `Allocator` class:

```
typedef Allocator<T>::reference reference;
typedef Allocator<T>::const_reference
  const_reference;
typedef Allocator<T>::pointer iterator;
typedef Allocator<T>::const_pointer
  const_iterator;
```

NOTE: If for different memory models, the types `Allocator<T>::pointer`, `Allocator<T>::size_type`, etc., will in general be different. However, the point is that these differences do not affect the vector container (or any other STL container), since the changes are completely encapsulated in the `Allocator` class.

The information passed on from the `Allocator` class to the STL container includes the types of `pointers, constant pointers, references, constant references`, etc., together with some member functions. Complete details of the types and functions encapsulated by the default allocator are provided in the next section.

The Default Allocator Interface

"Class allocator declaration" on page 81 contains the Class allocator declaration.

Listing 5.1 Class allocator declaration

```
class allocator {
   public:
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
```

```
    template <class T>
    class types {
      typedef T* pointer;
      typedef const T* const_pointer;
      typedef T& reference;
      typedef const T& const_reference;
      typedef T value_type;
    };

    allocator();
    ~allocator();

    template<class T> typename
      types<T>::pointer
      address(types<T>::reference x) const;

    template<class T> typename
      types<T>::const_pointer
      address(types<T>::const_reference x) const;

    template<class T, class U> typename
      types<T>::pointer allocate(size_type,
      types<U>::const_pointer hint);

    template<class T>
      void deallocate(types<T>::pointer p);

    size_type max_size() const;
};
  class allocator::types<void> { // specialization
     public:
      typedef void* pointer;
      typedef const void* const_pointer;
      typedef void value_type;
  };
void* operator new(size_t N, allocator& a);
```

Description In the allocator interface it can be seen that the type information is encapsulated in the nested template class `types`. Since nested class templates are not yet supported by any compilers, our implementation provides the following workaround by templatizing the class `allocator`.

The topics in this section discuss:

- "Typedef Declarations" on page 83
- "Allocator Member Functions" on page 84
- "Custom Allocators" on page 87
- "Allocator Requirements" on page 88

Typedef Declarations

Table 5.1 The following typedef's are defined in the class allocator<T>.

allocator::pointer	allocator::const_pointer
allocator::reference	allocator::const_reference
allocator::value_type	allocator::const_address

allocator::pointer

Description The type of a pointer in the memory model.

Definition

```
typedef T* pointer;
```

allocator::const_pointer

Description The type of a constant pointer in the memory model.

Definition

```
typedef const T* const_pointer;
```

allocator::reference

Description The type of a reference in the memory model.

Definition `typedef T& reference;`

allocator::const_reference

Description The type of a constant reference in the memory model.

Definition `typedef const T& const_reference`

allocator::value_type

Description `value_type` refers to the type of the objects in the container. By default, containers contain objects of the type with which they are instantiated. For example, `vector<int*>` is a declaration of a vector of `pointers to integers`.

Definition `typedef T value_type;`

Allocator Member Functions

Table 5.2 The Class allocator has the following member functions

Constructor–allocator	Destructor–allocator
allocator::size_type	allocator::difference_type
allocator::address	allocator::const_address
allocator::allocate	allocator::deallocate
allocator::max_size	

Constructor–allocator

Description Constructs an allocator object.

Prototype `allocator();`

Destructor–allocator

Description Destroys an allocator object.

Prototype `~allocator();`

The class `allocator` has the following other members:

allocator::size_type

Description size_type is the type that can represent the size of the largest object in the memory model.

Definition

```
typedef size_t size_type;
```

allocator::difference_type

Description This is the type that can represent the difference between any two pointers in the memory model.

Definition

```
typedef ptrdiff_t difference_type;
```

allocator::address

Description Returns a pointer to the referenced object x.

Prototype

```
template <class T>
  allocator<T>::pointer
    address(allocator<T>::reference x);
```

allocator::const_address

Description Returns a constant pointer to the referenced object `x`.

Prototype

```
template <class T>
  allocator<T>::const_pointer
    const_address(allocator<T>::const_reference x);
```

allocator::allocate

Description This member function allocates memory for `n` objects of type size_type but the objects are not constructed. It uses the global `operator new.`

NOTE: If that different memory models require different allocate functions (which is why the function has been encapsulated in the memory allocator class). `allocate` may raise an appropriate exception.

Prototype

```
template <class T>
  allocator<T>::pointer
      allocate(size_type n);
```

allocator::deallocate

Description Deallocates all of the storage pointed to by the pointer p using the global `operator delete`. All objects in the area pointed to by `p` should be destroyed prior to the call of deallocate. The function is templatized to allow it to be specialized for particular types in custom allocators.

Prototype

```
template <class T> void deallocate(
      allocator<T>::pointer p);
```

allocator::max_size

Description Returns the largest positive value of `difference_type`. This is the same as the largest number of elements that the container can hold in the given memory model.

Prototype `size_type max_size();`

Custom Allocators

The data types representing pointers (`size_type`) as well as the difference between two pointers (`difference_type`) differ across memory models.

Table 5.3 The data types representing pointers

allocator::vec_default	allocator::vec_large
allocator::vec_huge	

allocator::vec_default

Description A vector of 100 integers using the default allocator.

Prototype `vector<int> vec_default(100);`

allocator::vec_large

Description A vector of 1000 integers using the far allocator.

Prototype `vector<int, far_allocator> vec_large(1000);`

Remarks In case of the far allocator, the addressable range is 64 K. The `size_type` is a 32 bit value and `difference_type` is a 16-bit value.

allocator::vec_huge

Description A vector of 100,000 integers using the huge allocator.

Prototype `vector<int, huge_allocator> vec_huge(100000);`

Allocator Requirements

This section discusses

- "Function Objects" on page 88
- "Function Adaptors" on page 89

Function Objects

Description

A function object encapsulates a function in an object for use by other components. This is done by overloading the function call operator, `operator()`, of the corresponding class.

Passing a function object to an algorithm is similar to passing a pointer to a function, with one important difference. Function objects are classes that have `operator()` overloaded, which makes it possible to

- pass function objects to algorithms at compile time, and
- increase efficiency, by inlining the corresponding call.

These costs make a difference when the functions involved are very simple ones, such as integer additions or comparisons.

For example, if we want to have a by-element addition of two integer `vectors a` and `b`, with the result being placed in a, we can do:

```
transform(a.begin(), a.end(), b.begin(),
    a.begin(), plus<int>());
```

If we want to negate every element of a, we do:

```
transform(a.begin(), a.end(),
    a.begin(),negate<int>());
```

In both of the above examples, the addition and negation will be inlined.

Function Adaptors

Description

Function adaptors are STL classes that allow users to construct a wider variety of function objects. Using function adaptors is often easier than direct construction of a new function object type with a struct or class definition.

For example, a binder is a kind of function adapter that converts binary function objects into unary function objects by binding an argument to some particular value. The code fragment:

```
int x[1000];
int* where = find_if(x, x+1000,
  bind2nd(greater<int>(), 200));
```

finds the first integer in array x that is greater than 200. The base function object greater<int>() takes two arguments x and y and returns the greater value. By applying the binder bind2nd to this function object and the number 200, we produce a function object that defines a unary function that takes a single argument x and returns true if x > 200. This function object is then used as a parameter to find_if, to find the first element in the array that is greater than 200.

Besides binders, the library defines two other kinds of function adaptors:

- Negators are function objects that reverse the sense of predicate function objects.
- Adaptors for pointers to functions allow pointers to (unary and binary) functions to work with function adaptors that the library provides.

Arithmetic Operations

Files

```
#include <function.h>
```

Description

STL provides basic function object classes for all of the arithmetic operators in the language. The functionality of the operators is described below. The function object classes are:

Table 5.4 The functionality of the operators

plus	times
divides	minus
modulus	negate

plus

Description The plus function object accepts two operands x and y of type T, and returns the result of the computation of x + y.

Prototype

```
template <class T> struct plus<T>
```

minus

Description The minus function object accepts two operands x and y of type T, and returns the result of the computation of x - y.

Prototype

```
template <class T> struct minus<T>
```

times

Description The times function object accepts two operands x and y of type T, and returns the result of the computation of x * y.

Prototype

```
template <class T> struct times<T>
```

divides

Description The divides function object accepts two operands x and y of type T, and returns the result of the computation of x / y.

Prototype

```
template <class T> struct divides<T>
```

modulus

Description The modulus function object accepts two operands, x and y, of type T, and returns their result of the computation x % y.

Prototype `template <class T> struct modulus<T>`

negate

Description `negate` is a unary function that accepts a single operand x of type T, and returns the result of the computation of -x.

Prototype `template <class T> struct negate<T>`

Comparison Operations

Files #include <function.h>

Description STL provides basic function object classes for all of the comparison operators in the language. The basic functionality of the comparison objects is described below.

Table 5.5 The operation classes are:

equal_to	not_equal_to
greater	less
greater_equal	less_equal

equal_to

Description An object of this type accepts two parameters, x and y, of type T, and returns true if x == y.

Prototype `template <class T> struct equal_to<T>`

not_equal_to

Description

An object of this type accepts two parameters, x and y, of type T, and returns true if x != y.

Prototype

```
template <class T> struct not_equal_to<T>
```

greater

Description

An object of this type accepts two parameters, x and y, of type T, and returns true if x > y.

Prototype

```
template <class T> struct greater<T>
```

less

Description

An object of this type accepts two parameters, x and y, of type T, and returns true if x < y.

Prototype

```
template <class T> struct less<T>
```

greater_equal

Description

An object of this type accepts two parameters, x and y, of type T, and returns true if x >= y.

Prototype

```
template <class T> struct greater_equal<T>
```

less_equal

Description

An object of this type accepts two parameters, x and y, of type T, and returns true if x <= y.

Prototype

```
template <class T> struct less_equal<T>
```

Logical Operations

Files `#include <function.h>`

Description STL provides basic function object classes for the following logical operators in the language: and, or, not. The basic functionality of the logical operators is described below.

Table 5.6 The logical operation classes are:

logical_and	logical_or
logical_not	

logical_and

Description An object of this type accepts two parameters, x and y, of type T, and returns the boolean result of the logical and operation: `x && y`.

Prototype

```
template <class T> struct logical_and<T>
```

logical_or

Description An object of this type accepts two parameters, `x` and `y`, of type `T`, and returns the boolean result of the logical or operation: `x || y`.

Prototype

```
template <class T> struct logical_or<T>
```

logical_not

Description An object of this type accepts a single parameter, `x`, of type T, and returns the boolean result of the logical not operation: `!x`.

Prototype

```
template <class T> struct logical_not<T>
```

Negator Adaptors

Files

```
#include <function.h>
```

Description

Negators are function adaptors that take a predicate and return its complement. STL provides the negators `not1` and `not2` that take a unary and binary predicate respectively and return their complements.

Table 5.7 The negator function adaptors are:

not1	not2

not1

Description

This function accepts a `unary` predicate x as input and returns its `complement !x`.

Prototype

```
template <class Predicate>
unary_negate<Predicate> not1(const predicate& x);
```

not2

Description

This function accepts a `binary` predicate `x` as input and returns its `complement`, `!x`.

Prototype

```
template <class Predicate>
binary_negate<Predicate> not2(const predicate& x);
```

Binder Adaptors

Files

```
#include <function.h>
```

Description

Binders are function adaptors that convert binary function objects into unary function objects by binding an argument to some particular value. STL provides two binders `bind1st` and `bind2nd`, which are described below.

Table 5.8 The binder function adaptors are:

bind1st	bind2nd

bind1st

Description

This adapter accepts a function object `op` of two arguments and a value `x`, of type `T`. It returns a function object of one argument, constructed out of `op` with the *first* argument bound to `x`.

Prototype

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

bind2nd

Description

This adapter accepts a function object `op` of two arguments and a value `x`, of type `T`. It returns a function object of one argument, constructed out of `op` with the second argument bound to `x`.

Prototype

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

Adaptors for Pointers to Functions

Files

```
#include <function>
```

Description

Adaptors for pointers to functions are provided to allow pointers to unary and binary functions to work with the function adaptors provided in the library. They also can help avoid the “code bloat” problem arising from multiple template instances in the same program.

Table 5.9 The function pointer adaptors are:

ptr_fun–unary	ptr_fun–binary

Description

STL provides two adaptors for pointers to functions: one for unary and the other for binary functions. Both the functions have the same name (which is overloaded).

ptr_fun–unary

Description

This function adapter accepts a pointer to a unary function that takes an argument of type Arg and returns a result of type Result. A function object of type pointer_to_unary_function<Arg, Result> is constructed out of this argument, and returned.

Prototype

```
template <class Arg, class Result>
      ptr_fun(Result (*x) (Arg));
```

ptr_fun–binary

Description

This function adapter accepts a pointer to a binary function that accepts arguments of type Arg1 and Arg2 and returns a result of type Result. A function object of type pointer_to_binary_function<Arg, Result> is constructed out of this argument, and returned.

Prototype

```
template <class Arg, class Result>
      ptr_fun(Result (*x) (Arg1, Arg2));
```

Specialized Algorithms

All the iterators that are used as formal template parameters in these algorithms are required to have their operator* return an object for which operator& is defined and returns a pointer to T. See “Overview of Iterators” on page 263 where algorithms are discussed in detail.

Table 5.10 The uninitialized copy and fill functions are:

uninitialized_copy	uninitialized_fill
uninitialized_fill_n	

uninitialized_copy

Description

This function behaves as follows: `while (first != last) construct (&*result++, *first++);` This function returns `result`.

Prototype

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

uninitialized_fill

Description

This function behaves as follows:

```
while (first != last)
  construct (&*first++, x);
```

Prototype

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

uninitialized_fill_n

Description

This function behaves as follows:

```
while (n--)
  construct (&*first++, x);
```

Prototype

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

Template class auto_ptr

Description

Template class auto_ptr holds onto a pointer obtained via new and deletes that object when it itself is destroyed when it goes out of scope. The auto_ptr provides semantics of strict ownership. An object may be safely pointed to by only one auto_ptr, so copying an auto_ptr copies the pointer and transfers ownership to the destination. The declaration of auto_ptr is given below.

```
template <class X> class auto_ptr {//...};
```

Table 5.11 **The auto_ptr functions are:**

Constructor–auto_ptr	Copy Constructor–auto_ptr
auto_ptr::reset	auto_ptr::get
auto_ptr::release	Assignment Operator–auto_ptr
Dereferencing Operator–auto_ptr	Association Operator–auto_ptr

Constructor–auto_ptr

Description

This function requires p to point to an object of class X or a class derived from X for which delete p is defined and accessible. or else p is a null pointer.

Prototype

```
auto_ptr (X* p = 0);
```

Copy Constructor–auto_ptr

Description

This copies the argument a to *this.

Prototype

```
auto_ptr (auto_ptr& a);
```

auto_ptr::reset

Description This function requires `p` to point to an object of `class X` or a class derived from `X` for which `delete p` is defined and accessible, or else `p` is a null pointer.

Prototype

```
X* reset (X* p = 0);
```

auto_ptr::get

Description This function returns the pointer p specified as the argument to the constructor `auto_ptr (X* p)` or as the argument to the most recent call to `reset (X* p)`.

Prototype

```
X* get () const;
```

auto_ptr::release

Description This function releases the pointer and after a call to this function `get()` will return `0`.

Prototype

```
X* release ();
```

Assignment Operator–auto_ptr

Description Copies the argument `a` to `*this`.

Prototype

```
void operator= (auto_ptr& a);
```

Dereferencing Operator–auto_ptr

Description This returns `*get ()` provided `get()` does not return `0`.

Prototype

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

Association Operator–auto_ptr

Description

Returns the pointer associated.

Prototype

```
X* operator-> () const;
```

6

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.

Overview of Strings Classes

The strings library is based on the ANSI/ISO string class description in the April 1995 Draft Working Paper of the ANSI/ISO Committee. The classes include:

- “Class basic_string” on page 101
- “Class string_char_traits<charT>” on page 118

Class basic_string

Files `#include <bstring.h>`

Description Defining the string class as a template has the advantage that strings of various types can be easily instantiated. For example, a regular C character string is simply instantiated as `basic_string<char>`, while a wide-character string is instantiated as `basic_string<wchar_t>`.

For the rest of this chapter, we use the term `string` to refer to the class template `basic_string<charT>`, where `charT` may be of type `char, wchar_t,` or of a type defined in a C++ program.

The string class has many powerful string processing features. The class contains members to assign, append, insert, remove, replace, compare, find and concatenate an input string to a given string. Input, output and relational member functions are also provided. All of the string member functions are described in detail in this chapter.

The functions defined for the string class report two kinds of errors: a length error is associated with exceptions of type `length_error` and an `out-of-range` error is associated with exceptions of type `out_of_range`.

All of the member function descriptions make references to a variable `string::npos`. This is the maximum possible size of any string on a given implementation, and is defined as `size_t(-1)`.

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

The topics in this section include:

Constructors and Destructors–basic_string

Constructors

Description The various basic_string constructors construct a string object for character sequence manipulations.

Prototype

```
basic_string();
```

Remarks Default constructor. Constructs an empty string.

Prototype

```
string (const string& str, size_t pos = 0, size_t
n = npos);
```

Remarks

Constructs a string from the given input string `str`. The effective length `rlen` of the constructed string is the smaller of `n` and `str.size() - pos`, and the string is constructed by copying `rlen` characters starting at position `pos` of the input string `str`. The function throws an out-of-range error if `pos > str.size()`.

Prototype

```
string (const charT* s, size_t n);
```

Remarks

The input pointer `s` is assumed to point to an array of `charT` of length `n`. It is assumed that `s` is not a null pointer.

This constructor copies `n` characters starting at `s`, and constructs a string object initialized with the corresponding characters.

NOTE: If `n > length(s)`, then junk characters are appended to the end of the string. i.e. `n` characters are copied regardless of the exact length of the array pointed to by the input pointer `s`.

Prototype

```
string (const charT* s);
```

Remarks

Constructs a string object from the array pointed to by the input pointer `s`. It is assumed that `s` is not a null pointer.

Prototype

```
string (size_t rep, charT c);
```

Remarks

Constructs a string object with the character `c` repeated `rep` times. Reports a length error if `rep` equals `string::npos`.

Destructor

Description

Deallocates the memory referenced by the string object.

Prototype

```
~string ();
```

Public Member Operators basic_string

Assignment Operator basic_string

Description Assigns the input string `str` to the current string.

Prototype

```
string& operator= (const string& str);
string& operator= (const charT* s);
string& operator= (charT s);
```

Remarks Both the overloaded members assign the string constructed from the input `s` to the current string.

Assignment & Addition Operator basic_string

Description Appends the string `rhs` to the current string.

Prototype

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

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

Public Member Functions basic_string

Table 6.1 The functions are:

basic_string::append	basic_string::assign
basic_string::insert	basic_string::erase
basic_string::replace	

basic_string::append

Description

Appends characters from the input string `str` to the current string object. At most `n` characters, starting at position `pos` of `str`, are appended.

Prototype

```
string& append (const string& str,
      size_t pos = 0,size_t n = npos);
```

Remarks

The function reports an out-of-range error if `pos > str.size()`.

```
string& append (const charT* s, size_t n);
string& append (const charT* s);
string& append (size_t rep, charT c = charT());
```

Remarks

All of the above functions construct a string object from the input `s` and `append` it to the current string.

basic_string::assign

Description

Assigns characters from the input string `str` to the current string object. At most `n` characters, starting at position `pos` of `str` are assigned to the current string. The function reports an out-of-range error if `pos > str.size()`.

Prototype

```
string& assign (const string& str,
size_t pos = 0,size_t n = npos);
string& assign (const charT* s, size_t n);
string& assign (const charT* s);
string& assign (size_t rep, charT c = charT());
```

Remarks

All of the overloaded `assign` functions construct a string object from the input `s` and `assign` it to the current string.

basic_string::insert

Description

Inserts at most `n` characters, starting at position `pos2` of the input string `str`, into the current string. The characters are inserted starting at position `pos1` in the current string.

Prototype

```
string& insert (size_t pos1, const string& str,
      size_t pos2 = 0, size_t n = npos);
string& insert (size_t pos,
      const charT* s, size_t n);
string& insert (size_t pos, const charT* s);
string& insert (size_t pos, size_t rep,
      charT s = charT());
```

Remarks

All of the overloaded insert functions construct a string object from the input `s` and `insert` it into the current string.

basic_string::erase

Description

Removes up to `n` characters from the string starting from position `pos`.

Prototype

```
string& erase (size_t pos = 0, size_t n = npos);
```

basic_string::replace

Description

This function replaces a range of characters in the current string with a range of characters taken from the input string `str`. The range to be `replace`d starts at position `pos1` in the current string, and extends for `n1` characters, or up to the end of the string, whichever comes first.

Prototype

```
string& replace (size_t pos1, size_t n1,
      const string& str, size_t pos2 = 0,
      size_t n2 = npos);
```

Remarks The range of characters `inserted` starts at position `pos2` of the input string `str`, and extends for `n2` characters, or up to the end of the string `str`, whichever comes first.

```
string& replace (size_t pos, size_t n,
      const charT* s, size_t n2);
string& replace (size_t pos, size_t n,
      const charT* s);
string& replace (size_t pos, size_t n,
      size_t rep, charT s = charT());
```

Remarks All of the overloaded functions construct a string object from the input `s` and `replace` the range `[pos, n]` in the current string with the constructed string.

Access Members–basic_string

Table 6.2 The functions are:

basic_string::at	Subset Operator []–basic_string
basic_string::c_str	basic_string::size
basic_string::resize	basic_string::capacity
basic_string::reserve	basic_string::copy
basic_string::substr	basic_string::swap
basic_string::compare	basic_string::find
basic_string::rfind	basic_string::find_first_of
basic_string::find_last_of	basic_string::find_first_not_of
basic_string::find_not_of	

basic_string::at

Description This function returns a constant reference to the character at position `pos` of the current string.

Prototype `const charT& at (size_t pos) const;`

```
charT& at (size_t pos);
```

Remarks In the overloaded `at()` function if `pos < size()` returns a reference to the character at position `pos`, else throws `out_of_range` exception.

Subset Operator []–basic_string

Description Returns the element at position `pos` of the current string. Returns `traits::eos()` if `pos == size()`.

Prototype

```
charT operator[] (size_t pos) const;
charT& operator[] (size_t pos);
```

Remarks In the overloaded `operator[]` if `pos < size()`, returns the element at position `pos` of the current string. The reference returned is invalid after a subsequent call to any non-const member function for the object.

basic_string::c_str

Description Returns a pointer to the initial element of an array of length `size()+1` whose first `size()` elements equal the corresponding elements of the current string and whose last element is a null character specified by `eos()` of the corresponding character traits type.

Prototype

```
const charT* c_str () const;
```

basic_string::size

Description If `size()` is non-zero, returns `c_str()`. Else, returns 0.

Prototype

```
const charT* data () const;
size_t size () const;
```

Remarks The overloaded `size()` function returns a count of the number of char-like objects currently in the string.

basic_string::resize

Description If `n <= size()`, truncates the string to length `n` else it pads the extra locations with `c`. Reports a length error if `n` equals `string::npos`.

Prototype

```
void resize (size_t n, charT c);
void resize (size_t n);
```

Remarks The overloaded `resize()` function returns `resize(n, eos())`, where `eos()` is the null character of the traits type of the character set of the string.

basic_string::capacity

Description Returns the size of the allocated storage in the string.

Prototype

```
size_t capacity () const;
```

basic_string::reserve

Description This function is a directive that informs a `string` of a planned changed in size, so that it can manage the storage allocation accordingly. Reallocation of a string happens if and only if the current capacity is less than `res_arg`. After this call, `capacity()` is greater than or equal to `res_arg` if reallocation happens and equal to the previous value of `capacity()` otherwise.

Prototype

```
void reserve (size_t res_arg);
```

basic_string::copy

Description

This function replaces the string designated by `s` with a copy of a range of characters from the current string. The range copied begins at position `pos` of the current string and extends for `n` characters or up to the end of the current string, whichever comes first.

Prototype

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

basic_string::substr

Description

This function returns a copy the substring consisting of at most `n` characters starting at position `pos` of the current string.

Prototype

```
string substr (size_t pos = 0,
      size_t n = npos) const;
```

basic_string::swap

Description

This function swaps the contents of the two strings. The time complexity of this function is linear.

Prototype

```
void swap(string& s);
```

basic_string::compare

Description

This function compares a range of characters from the current string to the input string `str`. The range to be compared starts at position `pos` of the current string and extends for `n` characters or up to the end of the current string, whichever comes first. Returns an integer value that defines the result of the comparison.

Prototype

```
int compare (const string& str, size_t pos = 0,
      size_t n = npos) const;
int compare (charT* s, size_t pos, size_t n) const;
```

```
int compare (charT* s, size_t pos) const;
```

Remarks All of the overloaded functions construct a string from the input `s`, and `compare` it with the current string.

basic_string::find

Description This member function determines the earliest occurrence of the input `pattern` in the current string object, starting from position `pos` in the current string. If `find` can determine such an occurrence, it returns the starting index of `pattern` in the current string. Otherwise, it returns `string::npos`.

Prototype

```
size_t find (const string& pattern,
      size_t pos = 0) const;
size_t find (const charT* pattern, size_t pos,
       size_t n) const;
size_t find (const charT* pattern,
      size_t pos = 0) const;
size_t find (charT pattern,
      size_t pos = 0) const;
```

Remarks All of the overloaded functions construct a string from the input `pattern`, and try to `find` the `pattern` in the current string.

basic_string::rfind

Description This function scans the current string backwards, and finds the first occurrence of `pattern` in the string (from the back). The starting index of the matched position in the current string should be greater than or equal to the parameter `pos`. If a match is found, the starting index is returned; otherwise, the function returns `string::npos`.

Prototype

```
size_t rfind (const string& pattern,
      size_t pos = npos) const;
size_t rfind (const charT* pattern, size_t pos,
      size_t n) const;
size_t rfind (const charT* pattern,
```

```
      size_t pos = npos) const;
size_t rfind (charT pattern,
      size_t pos = npos) const;
```

Remarks All of the overloaded functions construct a string from the input `pattern`, and then call `rfind` on the current string with the constructed input.

basic_string::find_first_of

Description This function determines the first location, `loc`, between `pos` and the end of the current string, such that the character at `loc` matches at least one character from the parameter string `str`. If such a location can be determined, it is returned. Otherwise, the function returns `string::npos`.

Prototype

```
size_t find_first_of (const string& str,
      size_t pos = 0) const;
size_t find_first_of (const charT* s, size_t pos,
      size_t n) const;
size_t find_first_of (const charT* s,
      size_t pos = 0) const;
size_t find_first_of (charT s,
      size_t pos = 0) const;
```

Remarks All of the overloaded member functions construct a string from the input `s`, and then call `find_first_of` on the current string with the constructed input.

basic_string::find_last_of

Description This function determines the highest location, `loc`, up to `pos`, such that the character at `loc` matches at least one character from the parameter string `str`. If such a location can be determined, it is returned. Otherwise, the function returns `string::npos`.

Prototype

```
size_t find_last_of (const string& str,
      size_t pos = npos) const;
```

```
size_t find_last_of (const charT* s, size_t pos,
        size_t n) const;
size_t find_last_of (const charT* s,
       size_t pos = npos) const;
size_t find_last_of (charT s,
       size_t pos = npos) const;
```

Remarks All of the overloaded member functions construct a string from the input `s`, and then call `find_last_of` on the current string with the constructed input.

basic_string::find_first_not_of

Description This function determines the first location `loc`, between `pos` and the end of the current string, such that the character at `loc` does not match any character from the parameter string str. If such a location is found, it is returned. Otherwise, the function returns `string::npos`.

Prototype

```
size_t find_first_not_of (const string& str,
       size_t pos = 0) const;
size_t find_first_not_of (const charT* s,
       size_t pos, size_t n) const;
size_t find_first_not_of (const charT* s,
       size_t pos = 0) const;
size_t find_first_not_of (charT c,
       size_t pos = 0) const;
```

Remarks All of the overloaded member functions construct a string from the input `s`, and then call `find_first_not_of` on the current string with the constructed input.

basic_string::find_not_of

Description This function scans the current string up to the position `pos` and determines the highest location, `loc`, such that the character at `loc` does not match any character from the parameter string `str`. If such a location is found, it is returned. Otherwise, the function returns `string::npos`.

Prototype

```
size_t find_last_not_of (const string& str,
      size_t pos = npos) const;
size_t find_last_not_of (const charT* s,
      size_t pos, size_t n) const;
size_t find_last_not_of (const charT* s,
      size_t pos = npos) const;
size_t find_last_not_of (charT c,
      size_t pos = npos) const;
```

Remarks All of these member functions construct a string from the input `s`, and then call `find_last_not_of` on the current string with the constructed input.

Global Operators

Table 6.3 The operators are:

Inserter Operator << basic_string	Extractor Operator >> basic_string
Concatenation Operator + basic_string	Equality Operator == basic_string
Less Than Operator < basic_string	Less Than or Equal Operator <= basic_string
Not Equal Operator != basic_string	Greater Than Operator > basic_string
Greater Than or Equal Operator >= basic_string	

Inserter Operator << basic_string

Description Puts the string `s` on the output stream `o`.

Prototype

```
ostream& operator<< (ostream& o, const string& s);
```

Extractor Operator >> basic_string

Description Reads a string of characters from input stream `i` into `s`. Any whitespace is treated as a string terminator.

Prototype
```
istream& operator>> (istream& i, string& s);
```

Concatenation Operator + basic_string

Description Appends the string `rhs` to `lhs`.

Prototype
```
string operator+ (const string& lhs,
      const string& rhs);
```

Prototype
```
string operator+ (const charT* lhs,
      const string& rhs);
string operator+ (charT lhs, const string& rhs);
string operator+ (const string& lhs,
      const charT* rhs);
string operator+ (const string& lhs, charT rhs);
```

Remarks All of the overloaded member functions construct strings from `lhs` and `rhs`, and then append `rhs` to `lhs`.

Equality Operator == basic_string

Description Returns true if `lhs.compare(rhs)` is zero, otherwise false.

Prototype
```
bool operator== (const string& lhs,
      const string& rhs);
bool operator== (const charT* lhs,
      const string& rhs);
bool operator== (charT lhs, const string& rhs);
bool operator== (const string& lhs,
      const charT* rhs);
```

```
bool operator== (const string& lhs, charT rhs);
```

Remarks

All of the overloaded == member functions construct strings from `lhs` and `rhs`, and then call `operator== (string&, string&)` for string equality.

Less Than Operator < basic_string

Description

Returns true if `lhs.compare(rhs) < 0`, otherwise false.

Prototype

```
bool operator< (const string& lhs,
      const string& rhs);
bool operator<(const charT* lhs,
      const string& rhs);
bool operator< (charT lhs, const string& rhs);
bool operator< (const string& lhs,
      const charT* rhs);
bool operator< (const string& lhs, charT rhs);
```

Remarks

All of the less than member functions construct strings from `lhs` and `rhs`, and then call `operator<` for strings, defined above.

Less Than or Equal Operator <= basic_string

Description

All of the less than equal to member functions construct strings from `lhs` and `rhs` (if needed), and then return `!(rhs < lhs)`.

Prototype

```
bool operator<= (const string& lhs,
      const string& rhs);
bool operator<= (const charT* lhs,
      const string& rhs);
bool operator<= (charT lhs, const string& rhs);
bool operator<= (const string& lhs,
      const charT* rhs);
bool operator<= (const string& lhs, charT rhs);
```

Not Equal Operator != basic_string

Description

All of the not equal to member functions construct strings from `lhs` and `rhs` (if needed), and then return `!(lhs == rhs)`.

Prototype

```
bool operator!= (const string& lhs,
      const string& rhs);
bool operator!= (const charT* lhs,
      const string& rhs);
bool operator!= (charT lhs, const string& rhs);
bool operator!= (const string& lhs,
      const charT* rhs);
bool operator!= (const string& lhs, charT rhs);
```

Greater Than Operator > basic_string

Description

All of the greater than member functions construct strings from `lhs` and `rhs` (if needed), and then return `(rhs < lhs)`.

Prototype

```
bool operator> (const string& lhs,
      const string& rhs);
bool operator> (const charT* lhs,
      const string& rhs);
bool operator> (charT lhs, const string& rhs);
bool operator> (const string& lhs,
      const charT* rhs);
bool operator> (const string& lhs, charT rhs);
```

Greater Than or Equal Operator >= basic_string

Description

All of the greater than equal to member functions construct strings from `lhs` and `rhs` (if needed), and then return `!(lhs < rhs)`.

Prototype

```
bool operator>= (const string& lhs,
      const string& rhs);
bool operator>= (const charT* lhs,
```

```
      const string& rhs);
bool operator>= (charT lhs, const string& rhs);
bool operator>= (const string& lhs,
      const charT* rhs);
bool operator>= (const string& lhs, charT rhs);
```

Class string_char_traits<charT>

Description

The user can use the template class string for a specialized character type. For this the user needs to define the member function and data members in the template struct string_char_traits<charT> for the particular character type charT. The interface for struct string_char_traits<charT> is outlined below.

The topics in this section are:

- "Typedef Declarations string_char_traits" on page 118
- "Member Functions string_char_traits" on page 118

Typedef Declarations string_char_traits

Description

The following typedef is defined in the class string_char_traits.

string_char_traits::char_type

Description

A type defined for character types.

Definition

```
typedef charT char_type;
```

Member Functions string_char_traits

Table 6.4 **The member functions are:**

string_char_traits::char_type	string_char_traits::assign
string_char_traits::eq	string_char_traits::ne

string_char_traits::lt	string_char_traits::eos
string_char_traits::is_del	string_char_traits::compare
string_char_traits::length	string_char_traits::copy
string_char_traits::char_in	string_char_traits::char _out

string_char_traits::assign

Description Used for character type assignment.

Prototype

```
static void assign (char_type& c1,
      const char_type& c2);
```

string_char_traits::eq

Description Used for `bool` equality checking.

Prototype

```
static bool eq (const char_type& c1,
      const char_type& c2);
```

string_char_traits::ne

Description Used for `bool` inequality checking.

Prototype

```
static bool ne(const char_type& c1,
      const char_type& c2);
```

string_char_traits::lt

Description Used for `bool` less than checking.

Prototype

```
static bool lt (const char_type& c1,
      const char_type& c2);
```

string_char_traits::eos

Description Used to supply an end of sentence character.

Prototype

```
static char_type eos();
```

string_char_traits::char_in

Description Used for character input.

Prototype

```
static istream& char_in (istream& is, charT& a);
```

string_char_traits::char _out

Description Used for character output.

Prototype

```
static ostream& char_out (ostream& os, charT a);
```

string_char_traits::is_del

Description Used to test for deletion.

Prototype

```
static bool is_del(charT a);
```

string_char_traits::compare

Description Used for null terminated Character array comparison.

Prototype

```
static int compare (const char_type* s1,
      const char_type* s2, size_t n);
```

string_char_traits::length

Description Used when determining the length of a Null terminated character array.

Prototype

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

string_char_traits::copy

Description Used for copying a null terminated character array.

Prototype

```
static char_type* copy (char_type* s1,
      const char_type* s2, size_t n);
```

7

Localization Library

This chapter is a reference guide to the ANSI/ISOstandard C++ Localization library and is based on the April 1995 Draft Working Paper of the ANSI/ISO committee.

Overview of the Localization Library

C++ localization library extends the internationalization facilities provided by the C library in such a way that will help programmers to encapsulate the cultural differences.

Files

```
#include <mlocale.h>
#include <locale.h>
```

Description

The localization library is a set of classes that help C++ programmers to encapsulate the cultural differences that come up especially while porting an application across different user communities. Hence, this library provides a set of classes that include internationalization support for character classification, string collation, formatting and parsing of date, time, numeric and monetary quantities, message retrieval etc.

In different countries people use different formats for formatting date, time, currency etc., For example in India, the date is written using the DD/MM/YY(D-Day, M-Month and Y-Year) format while in the USA, it is the MM/DD/YY format. This difference (though, “minor”) may cause problems when any software system is being used by people belonging to different communities. So, as soon as the ISO C standard was published, the work on improvising the internationalization support began and this work was published as the ISO C Amendment I.

The C++ support for internationalization is summarized in two header files. The header `<mlocale>` declares the set of classes like locale, facet etc., which are part of the C++ standard library and the header `<locale>` declares the elements of localization library from the standard C library.

This section summarizes the classes and convenience interfaces provided by the C++ library for internationalization. The topics in this chapter are:

Class locale

Class `locale` is used for implementing a type-safe polymorphic collection of facets (feature-sets) indexed by facet types. The following sections illustrate the components of the class `locale`.

The topics are:

Typedef Declarations

Description

`Category`, when given a legal value, represents a collection of facets. Valid `category` values include 0 and the `locale` member bitmask elements `collate`, `ctype`, `monetary`, `numeric`, `time` and `messages`. In addition, `locale` member `all` is defined such that the expression

```
(collate | ctype | monetary | numeric | time |
messages) == all
```

The categories and their corresponding facets are given below.

Definition

```
typedef unsigned category;
```

Table 7.1 Categories and corresponding facets

Category	Include Facets
collate	collate<char>, collate<wchar_t>
ctype	ctype<char>, ctype<wchar_t>, codecvt<char, wchar_t, mbstate_t>, codecvt<wchar_t, char, mbstate_t>
monetary	moneypunct <char, bool>, moneypunct <wchar_t, bool>, money_get<char, bool, II>, money_get<wchar_t, bool, II>, money_put<char, bool, II>, money_put<wchar_t, bool, II>

Category	Include Facets
numeric	numpunct<char>, numpunct<wchar_t>, num_get<char, II>, num_get<wchar_t, II>, num_put<char, OI>, num_put<wchar_t, OI>
time	time_get<char, bool, II>, time_get<wchar_t, bool, II> time_put<char, bool, II>, time_put<wchar_t, bool, II>
messages	messages<char>, messages<wchar_t>

Public Data Members

The class `locale` has two nested classes in it, `“Class facet” on page 132` and `“Class id” on page 134`. The `“Class facet” on page 132` acts as a base class for all locale facets.

See “Class facet” on page 132 , “Class id” on page 134.

Constructors

Default Constructor

Description Default constructor constructs a locale which is a snapshot of the current global locale. The current global locale can be set by calling either the standard “C” function `setlocale()` or `locale::global(const locale&)`.

Prototype

```
locale () throw();
```

Overloaded and Copy Constructors

Prototype

```
locale (const locale& other) throw();
```

Description Copy constructor constructs an instance of the class locale which is a copy of the locale `other`.

Prototype

```
explicit locale (const char* std_name);
```

Remarks Constructs a locale using the standard C locale names (for example, "POSIX"). If the argument `std_name` passed to it is not a valid name, then a `runtime_error` exception is thrown. The valid set of `std_name` include "C", null string and any other implementation-defined value.

Prototype

```
locale (const locale& other,
      const char* std_name, category cat);
```

Remarks Creates a locale as a copy of `other` except for the facets identified by the category argument `cat`, for which the semantics will be the same as that of the second argument `std_name`. The resulting locale will have a name if and only if `other` has a name.

Prototype

```
template <class Facet>
      locale(const locale& other, Facet* f);
```

Remarks A locale is constructed which does `not` have a name. The constructed locale has all the features set from the first locale argument `other`, except that of the type `Facet`, for which the second argument is used.

Prototype

```
template <class Facet>
      locale (const locale& other,
      const locale& one);
```

Remarks A locale which does `not` have a name is constructed such that all the facets are incorporated from the first argument `other` and that facet which is identified by the template parameter `Facet`, is incorporated from the second argument `one`. If the second argument does not have a facet of that particular type, `runtime_error` exception is thrown.

Prototype

```
locale (const locale& other,
      const locale& one, category cat);
```

Remarks

A locale is constructed which has all the facets from the first argument `other`, except those facet(s) that are specified by the category argument `cat`. These facets that are specified by the categories are installed from the second argument. The constructed locale will have a name if and only if the first two locales are named.

Public member operators

Table 7.2 The public member operators are:

Assignment Operator= locale	Equality Operator== locale
Not Equal Operator!= locale	Grouping Operator locale

Assignment Operator= locale

Description

Creates a copy of `other`, replacing the current value.

Prototype

```
const locale& operator =
      (const locale& other) const;
```

Equality Operator== locale

Description

The function returns true if and only if any of the following conditions are satisfied.

- Both arguments are the same locale
- If one is a copy of the other
- Both locales have a name and they are identical

Prototype

```
bool operator == (const locale& other) const;
```

Not Equal Operator!= locale

Description Evaluates the following expression and returns the result: `!(*this == other)`.

Prototype

```
bool operator != (const locale& other) const;
```

Grouping Operator locale

Description Compares two strings according to the `collate<charT>` facet of locale. This member operator satisfied requirements for a comparator predicate template argument as applied to strings.

Prototype

```
template <class charT, class IS_Traits>
    bool operator()
    (const basic_string<charT,
      IS_Traits>& s1, const basic_string<charT,
      IS_Traits>& s2) const;
```

Remarks Since member templates are not supported, our library does not provide this function yet.

Public Member Functions

Table 7.3 The public member functions are:

locale::name	locale::has
locale::use	

locale::name

Description The name of `*this`, if it has one; otherwise, the string “*”.

Prototype

```
const basic_string<char>& name() const;
```

locale::has

Description

An indication whether the facet requested is present in `*this`. Also, see `use`.

Prototype

```
template <class Facet> bool has() const;
template <class Facet> bool has (
      const locale& loc, Facet* f);
```

Remarks

The semantics of this function is the same as explained above, except that locale argument is also passed as an argument, instead of being implicit.

See Also

“locale::use” on page 130.

locale::use

Description

The function returns a reference to the Facet of the locale. If the facet requested in not present in the locale on which the function was applied but present in the current global locale, returns the global locale’s instance of Facet.

Prototype

```
template <class Facet> const Facet& use() const;
template <class Facet> const Facet& use (
      const locale& loc, Facet* f);
```

Remarks

The semantics of this function is the same as explained above, except that locale argument is also passed as an argument, instead of being implicit.

See Also

“locale::has” on page 130.

Static Member Functions

Table 7.4 The static member functions are:

locale::global	locale::classic
locale::transparent	

locale::global

Description The function sets the global locale to its argument. Subsequent calls to the default constructor and of other library functions affected by the function `setlocale()`, use the locale loc until the next call to this function or `setlocale()`.

Prototype

```
static locale global (const locale& loc);
```

locale::classic

Description A call to this member returns the standard “C” locale. This is equivalent to constructing a locale using a call to the constructor `locale(“C”)`.

Prototype

```
static const locale& classic();
```

locale::transparent

Description This function returns the continuously updated global locale. A locale which implements semantics that vary dynamically as the global locale is changed.

Prototype

```
static locale transparent();
```

Global Operators

Table 7.5 The global operators are:

Extractor Operator>> locale	Inserter Operator << locale

Extractor Operator>> locale

Description

This function tries to read a line into a string and construct a locale from it. If either operation fails, sets failbit of streams. This operator is not yet implemented in our library.

Prototype

```
template <class charT, class Traits>
  basic_istream<charT, Traits>& operator >>
    (basic_istream<charT, Traits>& s, locale& loc);
```

Inserter Operator << locale

Description

The usual stream output operator for locales.

Prototype

```
template <class charT, class Traits>
basic_ostream<charT, Traits>& operator <<
    (basic_ostream<charT, Traits>& s, locale& loc);
```

Class facet

Description

Class `facet` is the base class for locale feature sets. Declaration of the class facet is given below.

Prototype

```
class locale::facet {
  protected :
    facet (size_t refs = 0);
    virtual ~facet ();
  private :
    facet (const facet&); // not defined
    void operator= (const facet&);   // not defined
```

```
};
```

Remarks

A class is a `facet` if it is publicly derived from another facet, or if it is a class derived from `locale::facet` and containing a declaration as follows:

```
static locale::id id;
```

See also

"Class id" on page 134

A program that passes a type that is not a facet (explicit or deduced) as a parameter to a locale function expecting a facet, is ill-formed.

The `refs` argument to the constructor is used for lifetime management. If `(refs == 0)` the facet's lifetime is managed by the locale or locales it is incorporated into. If `(refs == 1)` its lifetime is until explicitly deleted.

For some standard facets (say `FACET`), a standard "`FACET_byname`" class, derived from it, implements the semantics equivalent to that facet of the locale constructed by locale (const char*). Each `FACET_byname` facet provides a constructor,

```
FACET_byname(const char* name, size_t refs = 0);
```

where `name` names the locale, and the `refs` argument is passed to the base class constructor. If there is no "..._byname" version of a particular facet, the base class implements such semantics itself.

Prototype

```
class Facet : public locale::facet {
  protected :
  virtual <return_type> do_F (...) const;
  //....
  public :
  <return_type> F(...) const
          { return do_F (...);}
  //...

  Facet(size_t refs = 1) : locale::facet(refs){}

  protected :
  ~Facet () {}
```

```
};
```

Remarks

There are 6 different categories in which the facets provided by C++ localization library are divided. They are `collate`, `ctype`, `monetary`, `numeric`, `time`, `messages`. Each of the standard categories includes a family of facets. See also:

- “The Numerics Category” on page 134
- “The Collate Category” on page 140
- “Ctype Category” on page 142
- “The Monetary Category” on page 152

Class id

Description

This class is used for identification of a locale facet. A mandatory object of this class used as an index in every derived facet facilitates lookup and initialization of facets. This type-id mechanism ensures that every facet type installed in a locale is assigned a unique id. Id is for every facet type and not for every facet object.

See also

“Class facet” on page 132

The Numerics Category

The numerics category consists of three templatized classes. These classes handle all the numeric formatting and parsing. These classes are publicly derived from `locale::facet`.

The classes are:

- “Class numpunct” on page 135
- “Class num_get” on page 137
- “Class num_put” on page 138

Classes `num_get` and `num_put` use `numpunct` for numeric formatting and parsing.

Class numpunct

Description

This class specifies numeric punctuation. The base class provides classic "C" numeric formats, while numpunct_byname version supports named locale (e.g. POSIX, X/OPEN) numeric formatting semantics. Other two numeric facets, `num_get` and `num_put` use `numpunct` facet installed in a particular locale to parse/format numeric quantities.

The topics discussed for this class are:

Typedef Declarations numpunct

Description

The following typedef's are defined in the class `numpunct`.

numpunct::char_type

Definition

```
typedef charT char_type;
```

numpunct::string

Definition

```
typedef basic_string<charT> string;
```

Public Member Functions numpunct

Table 7.6 The public member functions are

numpunct::decimal_point	numpunct::thousands_sep
numpunct::grouping	numpunct::truename
numpunct::falsename	

numpunct::decimal_point

Description The function returns a string for use as the decimal radix separator. The base class implementation of this member returns a ".". The `num_get<charT,InputIterator>` class is not required to recognize numbers formatted using a decimal radix separator if it is not a one-character string.

Prototype `string decimal_point() const;`

numpunct::thousands_sep

Description The function returns a string which is the thousand separator. The base class implementation of this member returns the empty string. The `num_get<charT, InputIterator>` class is not required to recognize the numbers formatted using a thousand separator if is not a one-character string.

Prototype `string thousands_sep() const;`

numpunct::grouping

Description This function returns a vector `vec` in which `vec[i]` represents the number of digits in the group at position `i` starting with 0 as the rightmost group. *If* `vec.size() <= i`, the number is the same as group `(i-1)`; if `(i < 0 || vec[i] <= 0)`, the size of the digit group is unlimited. The base-class implementation this returns the empty vector.

Prototype `vector<char> grouping() const;`

numpunct::truename

Description Returns a string representing the name of the boolean value `true`. The base class implementation return the strings "`true`".

Prototype `string truename() const;`

numpunct::falsename

Description Returns a string representing the name of the boolean value `false`. The base class implementation return the strings “`false`”.

Prototype `string falsename() const;`

Class num_get

Description Template class `num_get` has the following set of typedef’s and public member functions.

- “Typedef Declarations num_get” on page 137
- “Public Member Functions num_get” on page 138

Typedef Declarations num_get

Description The following typedef’s are defined in the class `num_get`.

char_type

Definition `typedef charT     char_type;`

iter_type

Definition `typedef InputIterator iter_type;`

ios

Definition `typedef basic_ios<charT> ios;`

Public Member Functions num_get

There is one function.

num_get::get

Description

All the above functions read characters from `in`, interpreting them according to flags set in `str` and the `ctype<charT>` facet and `numpunct<charT>` facet installed in the locale `loc`. These functions ignore the state of `str` initially. But they indicate failure by calling `str.setstate (ios_base::failbit)`. If an error occurs, `v` is not changed; otherwise it is set to the resulting value. Digit grouping separators are optional; if present, digit grouping is checked after the entire number is read. When reading a non-numeric boolean value, the names are compared exactly.

Prototype

```
iter_type get(iter_type in, iter_type end,
  ios& str, const locale&, bool& v) const;
iter_type get(iter_type in, iter_type end,
  ios& str, const locale&, long& v) const;
iter_type get(iter_type in, iter_type end,
  ios& str, const locale&,
  unsigned long& v) const;
iter_type get(iter_type in, iter_type end,
  ios& str, const locale&, double& v) const;
iter_type get(iter_type in, iter_type end,
  ios& str,const locale&,long double& v) const;
```

Class num_put

Description

Template class `num_put` has the following set of `typedef`'s and public member functions.

- "Typedef Declarations num_put" on page 139
- "Public Member Functions num_put" on page 139

Typedef Declarations num_put

Description The following typedef's are defined in the class `num_put`.

char_type

Definition

```
typedef charT char_type;
```

iter_type

Definition

```
typedef OutputIterator iter_type;
```

ios

Definition

```
typedef basic_ios<charT> ios;
```

Public Member Functions num_put

There is one function.

num_put::put

Description All the above functions write characters to the sequence `out`, formatting `val` according to the flags set in `str`, `ctype<charT>` and `numpunct<charT>` facets installed inside locale `loc`. These functions insert digit group separators as specified by
`numpunct<charT>::grouping().`

Prototype

```
iter_type put(iter_type out, ios& str,
      const locale& loc, bool val) const;
iter_type put(iter_type out, ios& str,
      const locale& loc, long val) const;
iter_type put(iter_type out, ios& str,
      const locale& loc, unsigned long val) const;
iter_type put(iter_type out, ios& str,
      const locale& loc, double val) const;
```

```
iter_type put(iter_type out, ios& str,
      const locale& loc, long double val) const;
```

Remarks All the above functions ignore and do not change the stream state. They return an iterator pointing immediately after the last character produced.

The Collate Category

Collate category consists of one template class `collate` which provides features for use in the collation (comparison) and hashing of strings.

See Also "Class Collate" on page 140

Class Collate

Description This class provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (see Chapter 9) and containers operating on strings. The base class implementation applied lexicographic ordering (see 9.30).

- "Typedef Declarations collate" on page 140
- "Public Member Functions collate" on page 141

Typedef Declarations collate

Description The following typedef's are defined in the class `collate`.

char_type

Definition

```
typedef   charT                               char_type;
```

string

Definition

```
typedef   basic_string<charT>             string;
```

Public Member Functions collate

Table 7.7 The public member functions are:

collate::compare	collate::transform
collate::hash	

collate::compare

Description

Compares the two strings and returns 1 if the first string is greater than the second, -1 if less; zero otherwise.

Prototype

```
int compare (const char_type* low1,
    const char_type* high1, const char_type* low2,
    const char_type* high2) const;
```

collate::transform

Description

This function returns a string that, when compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `compare()` on the same two strings.

Prototype

```
string transform (const char* low,
      const char* high) const;
```

collate::hash

Description

Returns an integer value equal to the result of calling `hash()` on any other string for which `compare()` returns `0`(i.e equal) when passed the two strings.

Prototype

```
long hash (const char* low,
      const char* high) const;
```

Ctype Category

In this category, there are two templatized classes, `ctype<charT>` and `codecvt<fromtT, toT, stateT>`. There is also a specialization of `ctype<charT>` for the `char` data type.

Class `locale::ctype` encapsulates the C library `ctype` features. This class is typically used by rest of the library classes for character classing. A specialization `locale::ctype<char>` is provided, so that the member functions on type char may be implemented inline.

The two templatized classes in this category are:

- "Class ctype<charT>" on page 142
- "Class ctype<char> Specialization" on page 146
- "Class codecvt" on page 150

Class ctype<charT>

Description

The definition of class `ctype_base` which is the base class for `ctype` is given below.

Prototype

```
class ctype_base {
  public :
  enum ctype_mask
  {
    space, print, cntrl, upper, lower,
    alpha, digit, punct, xdigit,
    alnum = alpha | digit, graph = alnum | punct
  };
};
```

NOTE: The type `ctype_mask` is a bitmask type. As noted before, locale's `ctype` facet encapsulates the C library `ctype` features. This facet is derived from `locale::facet` and `ctype_base`.

- “Typedef Declarations ctype<charT>” on page 143
- “Public Member Functions ctype<charT>” on page 143

Typedef Declarations ctype<charT>

Description The following typedef’s are defined in the class `Ctype`.

char_type

Definition

```
typedef charT char_type;
```

Public Member Functions ctype<charT>

Table 7.8 The public member functions are

ctype::is	ctype::do_is
ctype::scan_is	ctype::scan_not
ctype::toupper	ctype::tolower
ctype::widen	ctype::narrow

ctype::is

Description This function classify a character or sequence of characters. For each argument character, this function identifies a value `M` of type `ctype_mask`. The function `is()` returns the result `(M & mask) != 0`.

Prototype

```
bool is (ctype_mask mask, char_type c) const;
```

See Also `do_is()`

ctype::do_is

Description

This function classify a character or sequence of characters. For each argument character, this function identifies a value `M` of type `ctype_mask`. The `do_is()` function simply places `M` for all `*p` where `(low <= p && p < high)`, into `vec[p-low]` and returns `high`.

Prototype

```
const char_type* do_is (const char_type* low,
    const char_type* high, ctype_mask* vec) const;
```

See Also

is()

ctype::scan_is

Description

This function locates a character in the buffer `[low, high)` that conforms to the classification mask, mask. It returns the smallest pointer `p` in the range `[low, high)` such that `(*p)` would return `true`; otherwise, returns `high`.

Prototype

```
const char* scan_is
  (ctype_mask mask,
  const char_type* low,
  const char_type* high) const;
```

ctype::scan_not

Description

This function locates a character in the buffer `[low, high)` that fails to the classification mask, mask. It returns the smallest pointer `p` in the range `[low, high)` such that `(*p)` would return `false`; otherwise, returns `high`.

Prototype

```
const char* scan_not (
  ctype_mask mask,
  const char_type* low,
  const char_type* high) const;
```

ctype::toupper

Description

These functions convert a character or sequence of characters to upper-case. The first function returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form replaces each character `*p` in the range `[low, high)` for which a corresponding upper-case character exists, with that character and returns `high`.

Prototype

```
charT toupper (char_type c) const;
const char_type* toupper (char_type* low,
      const char_type* high) const;
```

ctype::tolower

Description

These functions convert a character or sequence of characters to lower-case. The first function returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form replaces each character `*p` in the range `[low, high)` for which a corresponding lower-case character exists, with that character and returns `high`.

Prototype

```
charT tolower (char_type c) const;
const char_type* tolower (char_type* low,
      const char_type* high) const;
```

ctype::widen

Description

These function apply the simplest reasonable transformation from a `char` value or sequence of `char` values to the corresponding `char_type` value or values. The only characters for which unique transformations are required are the digits, alphabetic characters, ‘-’, ‘+’, newline and space. For any character `c`, the transformed character of `c` is not a member of any character classification that `c` is not also a member of. The first function returns the transformed value and the second form transforms each character `*p` in the range `[low, high)` placing the result in `dest[p-low]` and returns `high`.

Prototype

```
char_type widen (char c) const;
const char* widen (const char* low,
      const char* high char_type* dest) const;
```

ctype::narrow

Description

These function apply the simplest reasonable transformation from a `char_type` value or sequence of `char_type` values to the corresponding `char` value or values. The only characters for which unique transformations are required are the digits, alphabetic characters, '-', '+', newline and space. For any character `c`, the transformed character of `c` is not a member of any character classification that `c` is not also a member of. In addition, the expression `(narrow(c)-'0')` evaluates to the digit value of the character. The first function returns the transformed value or `dfault` is no mapping is readily available and the second form transforms each character `*p` in the range `[low, high)` placing the result(or `dfault` if no simplest transformation is readily available) in `dest[p-low]` and returns `high`.

Prototype

```
char narrow (char_type c, char dfault) const;
const char_type* narrow (const char_type* low,
      const char_type* high,
      char dfault, char* dest) const;
```

Class ctype<char> Specialization

`ctype<char>` specialization is provided so that the member functions on type `char` can be implemented `inline`. This specialization is provided because if affects the derivation interface for ctype<char>.

- "Data Members ctype<char>" on page 147
- "Public Member Functions ctype<char>" on page 147

Data Members ctype<char>

The following are the protected or private data members of ctype<char>.

ctype::ctype_mask

Definition `static const ctype_mask;`

ctype::table_

Definition `const ctype_ mask* const table_;`

ctype::classic_table_

Definition `static const mask classic_table_[UCHAR_MAX+1];`

ctype::delete_it_

Definition `bool delete_it_;`

Remarks The type `delete_it_` is a private flag,

Public Member Functions ctype<char>

Table 7.9 The public member functions are:

`constructor ctype<char>`	`ctype::is for ctype<char>`
`ctype::scan_is for ctype<char>`	`ctype::scan_not for ctype<char>`
`ctype::tolower for ctype<char>`	`ctype::toupper for ctype<char>`
`ctype::widen for ctype<char>`	`ctype::narrow for ctype<char>`

constructor ctype<char>

Description

This constructor initializes the protected member `table_` with the tab argument if nonzero, or the static value classic_table_ otherwise, and initializes the private member `delete_it_ to (tab && del).` The `refs` argument is passed to the base class constructor.

Prototype

```
ctype (const ctype_mask* tab = 0,
      bool del = false, size_t refs = 0);
```

ctype::is for ctype<char>

Description

The first function returns `table_[(unsigned char)c] & mask`. The second function, for all `*p` in the range `[low, high)` assigns `vec[p-low]` to `table_[(unsigned char)*p]` and it returns `high`.

Prototype

```
bool is (ctype_mask mask, char c) const;
const char* is (const char* low,
      const char* high, ctype_mask* vec) const;
```

ctype::scan_is for ctype<char>

Description

This function returns the smallest `p` in the range `[low, high)` such that `table_[(unsigned char)*p] & mask) == true`.

Prototype

```
const char* scan_is (ctype_mask mask,
      const char* low, const char* high) const;
```

ctype::scan_not for ctype<char>

Description

This function returns the smallest `p` in the range `[low, high)` such that `table_[(unsigned char)*p] & mask) == false`.

Prototype

```
const char* scan_not (ctype_mask mask,
      const char* low, const char* high) const;
```

ctype::tolower for ctype<char>

Description

These functions convert a character or sequence of characters to lower-case. The first function returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form replaces each character `*p` in the range `[low, high)` for which a corresponding lower-case character exists, with that character and returns `high`.

Prototype

```
char tolower (char c) const;
const char* toupper (char* low,
  const char* high) const;
```

ctype::toupper for ctype<char>

Description

These functions convert a character or sequence of characters to upper-case. The first function returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form replaces each character `*p` in the range `[low, high)` for which a corresponding upper-case character exists, with that character and returns `high`.

Prototype

```
char toupper (char c) const;
const char* toupper (char* low,
  const char* high) const;
```

ctype::widen for ctype<char>

Description

The second function copies contents of the range `[low, high)` to `dest` and returns `high`. The first function returns `c`.

Prototype

```
char widen (char c) const;
const char* widen (const char* low,
      const char* high, char* dest) const;
```

ctype::narrow for ctype<char>

Description The second function copies contents of the range `[low, high)` to `dest` and returns `high`. The first function returns `c`.

Prototype

```
char narrow (char c, char /* dfault */) const;
const char* narrow (
  const char* low,
  const char* high, char /* dfault */,
  char* dest) const;
```

Class codecvt

Description Definition of class codecvt_base, used as a base class for codecvt facet is given below.

Prototype

```
class codecvt_base {
  public :
    enum result {ok, partial, error, noconv};
};
```

The class `codecvt<fromT, toT, stateT>` is for use when converting from one codeset to another, such as from wide characters to multibyte characters, or vice versa. Instances of this facet are typically used in pairs instantiated oppositely. The `stateT` argument selects the pair of codesets being mapped between. This implementation provides the following two specializations for codecvt facet `codecvt<char, wchar_t, mbstate_t>` and `codecvt<wchar_t, char, mbstate_t>`.

- "Typedef Declarations codecvt" on page 150
- "Member Functions codecvt" on page 151

Typedef Declarations codecvt

Description The following typedef's are defined in the class `codecvt`.

from_type

Definition

```
typedef fromT from_type;
```

to_type

Definition

```
typedef toT to_type;
```

state_type

Definition

```
typedef stateT state_type;
```

Member Functions codecvt

There is one function.

Prototype

```
codecvt::result convert (
      state_type& state,
      const from_type* from,
      const from_type* from_end,
      const from_type*& from_next,
      to_type* to,
      to_type* to_limit,
      to_type*& to_next) const;
```

NOTE: This function requires the following conditions to hold true. `(from <= from_end && to <= to_end)` and state initialized if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

This function translates characters in the range `(from, from_end)`, placing the results starting at `to`. It stops when it runs out of character to translate or space to put the results, or if it encounters a character it cannot convert. It always leaves the `from_next` and `to_next` pointers pointing one beyond the last character successfully converted. If no translation is needed (returns `codecvt_base::noconv`), sets `to_next` equal to, to. In any case, this function does not write into `*to_limit`. This function returns an enumeration value of type `codecvt_base::result` as summarized below:

Table 7.10 convert result values

Value	Meaning
ok	completed the conversion
partial	ran out of space in the destination
error	encountered a from_type character it could not convert
noconv	no conversion was needed

The Monetary Category

The monetary category consists of three template classes `moneypunct`, `money_put` and `money_get` to handle monetary formatting and parsing. In all the three classes, a template parameter indicates whether local or international monetary formats are to be used. The `money_get<>` and `money_put<>` facets use `moneypunct<>` members to determine all formatting details. `moneypunct<>` provides basic format information for money processing.

The declaration of the class `money_base`, which is a base class for moneypunct is given below.

Prototype

```
class money_base {
  public :
    enum part {none, space, symbol, sign, value};
    struct pattern { char field[4];};
};
```

The classes in the monetary category are:

- “Class moneypunct” on page 153
- “Class money_put” on page 156

- "Class money_get" on page 157

Class moneypunct

Description

This class provides money punctuation, similar to `numpunct<>` of the numerics category.

- "Typedef Declarations moneypunct" on page 153
- "Public Member Functions moneypunct" on page 153

Typedef Declarations moneypunct

Description

The following typedef's are defined in the class `moneypunct`.

char_type

Definition

```
typedef charT char_type;
```

string_type

Definition

```
typedef basic_string<charT> string_type;
```

Public Member Functions moneypunct

Table 7.11 The public member functions are:

moneypunct::decimal_point	moneypunct::thousands_sep
moneypunct::groupint	moneypunct::curr_symbol
moneypunct::positive_sign	moneypunct::negative_sign
moneypunct::frac_digits	moneypunct::neg_format
moneypunct::frac_digits	

moneypunct::decimal_point

Description This function returns the radix separator to use in case `frac_digits()` is greater than zero.

Prototype

```
basic_string<charT,traits> decimal_point() const;
```

moneypunct::thousands_sep

Description This function returns the digit group separator to use in case `grouping()` specifies a digit grouping pattern.

Prototype

```
basic_string<charT,traits> thousands_sep() const;
```

Remarks The two functions above have been changed to return `charT`, in the April 1995 draft. But this implementation still returns a `string`.

moneypunct::groupint

Description This function returns a vector `vec` in which `vec[i]` represents the number of digits in the group at position `i` starting with 0 as the rightmost group. *If* `vec.size() <= i`, the number is the same as group `(i-1)`; if `(i < 0 || vec[i] <= 0)`, the size of the digit group is unlimited. The base-class implementation this returns the empty vector.

Prototype

```
vector<char> grouping () const;
```

moneypunct::curr_symbol

Description This function returns the string to use as the currency identifier symbol.

Prototype

```
string_type curr_symbol () const;
```

moneypunct::positive_sign

Description This function returns the string to use to indicate a positive monetary value.

Prototype `string_type positive_sign () const;`

moneypunct::negative_sign

Description This function returns the string to use to indicate a negative monetary value. If this is a one-char-string containing '(', it is paired with a matching ')'.

Prototype `string_type negative_sign () const;`

moneypunct::frac_digits

Description This function returns the number of digits after the decimal radix separator, if any.

Prototype `int frac_digits () const;`

moneypunct::pos_format

Description This function returns a four-element array specifying the order in which the syntactic elements appear in the monetary format. In this array, each element is an enumeration value of type `money_base::part`. Each enumeration value appears exactly once. `none`, if present, is not first; `space`, if present, is neither first nor last. Otherwise, the elements may appear in any order. In international instantiations, the result is always `{ symbol, sign, none, value}`.

Prototype `money_base::pattern pos_format () const;`

moneypunct::neg_format

Description

This function returns a four-element array specifying the order in which the syntactic elements appear in the monetary format. In this array, each element is an enumeration value of type `money_base::part`. Each enumeration value appears exactly once. `none`, if present, is not first; `space`, if present, is neither first nor last. Otherwise, the elements may appear in any order. In international instantiations, the result is always `{ symbol, sign, none, value}`.

Prototype

```
money_base::pattern neg_format () const;
```

Class money_put

Description

Class `money_put` contains the following set of typedefs and public member functions.

- "Typedef Declarations money_put" on page 156
- "Public Member Functions money_put" on page 157

Typedef Declarations money_put

Description

The following typedef's are defined in the class `money_put`.

char_type

Definition

```
typedef charT char_type;
```

iter_type

Definition

```
typedef OutputIterator iter_type;
```

string

Definition

```
typedef basic_string<charT> string;
```

ios

Definition

```
typedef basic_ios<charT> ios;
```

Public Member Functions money_put

There is one function.

money_put::put

Description

Writes characters to s, according to the format specified by the `moneypunct<charT>` facet of `loc`, and `f.flags ()`. Ignores any fractional part of units, or any characters in digits beyond the(optional) leading '-' and immediately subsequent digits. If format flags specify filling with internal space, the fill characters are placed where none or space appears in the formatting pattern. Returns an iterator pointing immediately after the last character produced.

Prototype

```
iter_type put (iter_type s, ios& f,
      const locale& loc, double units&) const;
iter_type put (iter_type s, ios& f,
      const locale& loc,
      const string& SixDgts) const
```

Class money_get

Description

Class `money_get` contains the following set of typedefs and public member functions.

- "Typedef Declarations money_get" on page 158
- "Public Member Functions money_get" on page 158

Typedef Declarations money_get

Description The following typedef's are defined in the class `money_get`.

char_type

Definition

```
typedef charT char_type;
```

iter_type

Definition

```
typedef InputIterator iter_type;
```

string

Definition

```
typedef basic_string<charT> string;
```

ios

Definition

```
typedef basic_ios<charT> ios;
```

Public Member Functions money_get

There is one function.

money_get::get

Description These functions read characters from `s` until they have constructed a monetary value, as specified in `str.flags()` and the `money-punct<charT>` facet of `loc`, or until it encounters an error or runs out of characters. The result is a pure sequence of digits, representing a count of the smallest unit of currency representable. Digit group separators are optional; if present, digit grouping is checked after all syntactic elements have been read. Where `space` or `none` appear in the format pattern, except at the end optional whitespace is consumed. These functions set `units` or `digits` from the sequence of digits found. `units` is negated, or `digits` is preceded by

'-' for a negative value. These functions indicate a failure by calling `str.setstate (failbit)`. On error, `units` or `digits` argument is unchanged. These function return an iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

Prototype

```
iter_type get (
  iter_type s, iter_type end,
  ios& str, const locale& loc,
  double& units) const
iter_type get (
  iter_type s, iter_type end,
  ios& str, const locale& loc,
  string& digits) const
```

8

Containers Library

This chapter discusses the containers library. These classes support lists, sets, maps, stacks, queues, and more.

Overview of the Containers Library

STL containers are divided into two broad categories: `sequence` containers and `associative` containers.

The topics in this chapter are:

What Are STL Containers

This section discusses the concept of a container, and how they work in the containers library. There are several common features of all the container classes. These features are discussed in this section as well.

The topics in this section are:

- "Basic Design and Organization of STL Containers" on page 162
- "Common Type Definitions in All Containers" on page 164
- "Common Members of All Containers" on page 163
- "Common Member Functions in all Containers" on page 166
- "Sequence Container Requirements" on page 170
- "Associative Container Requirements" on page 172
- "Associative Container Types and Member Functions" on page 174

Basic Design and Organization of STL Containers

Sequence containers include vectors, lists and deques. These contain elements of a single type, organized in a strictly linear arrangement. Although only the three most basic sequence containers are provided in this version of STL, it is possible to construct other sequence containers efficiently using these basic containers through the use of`container adaptors`, which are STL classes that provide interface mappings. STL provides adaptors for stacks, queues and priority queues.

The second STL container category consists of associative containers, which include sets, multisets, maps and multimaps. Associative containers allow for the fast retrieval of data based on keys. For example, a map (also known as an `associative array`) allows a user to retrieve an object of type T based on a key of some other type, while sets allow for the fast retrieval of the keys themselves.

All of the STL containers have three important characteristics:

- Every container allocates and manages its own storage.

- Every container provides a minimal set of operations (as member functions) to access and maintain its storage. The provided set of member functions includes:
  - `Constructors and destructors:` these functions allow users to construct and destroy instances of the container. Most containers have several kinds of constructors.
  - `Element access member functions:` these allow users to access the container elements. In most instances, the element access member functions do not change the container.
  - `Insertion member functions:` these are used to insert elements into the container.
  - `Erase member functions:` used to delete elements from the container.
- Each container has an allocator object associated with it. This allocator object encapsulates information about the memory model currently being used, and allows the classes to be portable across various platforms.

The same naming convention is used for the member functions of all containers, resulting in a very uniform interface to all the classes.

There is a great deal of similarity in the interfaces of all STL containers. Some differences exist between sequence and associative container interfaces, which we examine after taking a look at the common components.

Common Members of All Containers

The public members of STL containers fall into a two level hierarchy. The first level defines members that are common to *all* containers, while the second level contains two categories:

- members common to sequence containers (vectors, lists, deques).
- members common to associative containers (sets, maps, multisets, multimaps).

The common members of all STL containers fall into two distinct categories: `type definitions` and `member functions.` We take a look at each in turn.

Common Type Definitions in All Containers

The common type definitions found in each STL container are presented below. It is assumed that `X` is a container class containing objects of type `T, a` and `b` are values of `X`, `u` is an identifier and `r` is a value of `X&`.

value_type

Description Type of values the container holds.

Definition `X::value_type`

reference

Description Type that can be used for storing into X::value_type objects. This type is usually X::value_type&.

Definition `X::reference`

const_reference

Description

A constant reference type identical to X::reference.

Definition `X::const_reference`

pointer

Description A pointer type pointing to X::reference.

Definition `X::pointer`

iterator

Description An iterator type that points to X instances. It is either a random access iterator type (for vector or deque) or a bidirectional iterator type (for other containers).

Definition `X::iterator`

const_iterator

Description A iterator type that can be used with constant instances of type X. It is either a constant random access iterator type (for vector or deque) or a constant bidirectional iterator type (for other containers).

Definition `X::const_iterator`

reverse_iterator

Description An iterator type identical to X::iterator except that traversal direction is reversed (X::reverse_iterator::operator++ is X::iterator::operator--, etc.).

Definition `X::reverse_iterator`

const_reverse_iterator

Description A constant iterator type identical to X::const_iterator except that traversal direction is reversed.

Definition `X::const_reverse_iterator`

difference_type

Description The type that can represent the difference between any two X iterator objects (varies with the memory model).

Definition `X::difference_type`

size_type

Description The type that can represent the size of any X instance (varies with the memory model).

Definition `X::size_type`

Common Member Functions in all Containers

The common member functions required to be in each STL container are outlined below. In the descriptions, it is assumed that X is a container class containing objects of type T, a and b are values of X, u is an identifier and r is a value of X&.

Default Constructor

Description The default constructor. Takes constant time.

Prototype `X()`

Overloaded and Copy Constructors

Prototype `X(a);`

Remarks Constructor. Takes linear time.

Prototype `X u(a);`

Remarks Copy Constructor. Takes linear time.

Destructor

Description Destructor. The destructor is applied to every element of a, and all the memory is returned. Takes linear time.

Prototype `(&a)->~X();`

begin

Description Returns an iterator (const_iterator for constant a), that can be used to begin traversing all locations in the container.

Prototype `a.begin();`

end

Description Returns an iterator (const_iterator for constant a), that can be used in a comparison for ending traversal through the container.

Prototype `a.end();`

rbegin

Description Returns a reverse_iterator (const_reverse_iterator for constant a) that can be used to begin traversing through all locations of the container in the reverse of the normal order.

Prototype `a.rbegin();`

rend

Description Returns a reverse_iterator (const_reverse_iterator for constant a) that can be used in a comparison for ending a reverse-direction traversal through all locations in the container.

Prototype
```
a.rend();
```

Equality Operator ==

Description Equality operation on containers of the same type. Returns true when the sequences of elements in a and b are element wise equal (using X::value_type::operator==). Takes linear time.

Prototype
```
a == b
```

Not Equal Operator !=

Description The opposite of the equality operation. Takes linear time.

Prototype
```
a != b
```

Assignment Operator =

Description The assignment operator for containers. Takes linear time.

Prototype
```
r = a
```

size

Description Returns the number of elements in the container.

Prototype
```
a.size();
```

max_size

Description size() of the largest possible container.

Prototype `a.max_size();`

empty

Description Returns true if the container is empty (i.e., if a.size() == 0).

Prototype `a.empty();`

Less Than Operator <

Description Compares two containers lexicographically. Takes linear time.

> **NOTE:** lexicographical comparisons are described in chapter 10 Set Operations on Sorted Structures

Prototype `a < b`

Greater Than Operator >

Description Returns true if b < a, as defined above. Takes linear time.

Prototype `a > b`

Less Than or Equal Operator <=

Description Returns true if !(a > b). Takes linear time.

Prototype `a <= b`

Greater Than or Equal>=

Description Returns true if !(a < b). Takes linear time.

Prototype

```
a >= b
```

swap

Description Swaps two containers of the same type in constant time.

Prototype

```
a.swap(b);
```

Remarks From the above definitions, we note that several comparison functions, constructors, type definitions and other member functions are shared between all STL containers, allowing their interfaces to be very uniform.

NOTE: Shallow Copies: It must be noted that the assignment operators of all STL containers make shallow copies. This means that the assignment operator will simply copy the assigned container exactly as is, and will not traverse pointers downwards to make copies recursively of elements that the container elements might possibly point to.

Sequence Container Requirements

All STL Sequence containers define two constructors, three insert member functions and two erase member functions in addition to the common types and member functions mentioned in the previous section.

The additional members are defined below. In the definitions, `X` is assumed to be a sequence class (e.g., a vector, list or deque), i and j satisfy input iterator requirements, `[i,j)` is a valid range, `n` is a value of `X::size_type, p` is a valid iterator to `a, q, q1` and `q2` are valid dereferenceable iterators to a, `[q1, q2)` is a valid range, `t` is a value of `X::value_type`.

Prototype

```
X(n,t);
X a(n,t);
```

Remarks Constructs a sequence with n copies of t.

```
X(i,j);
X a(i,j);
```

Remarks Constructs a sequence equal to the range [i,j).

insert

Description Inserts a copy or copies of an element.

Prototype `a.insert(p,t);`

Remarks Inserts a copy of t before p. Returns an iterator pointing to the inserted copy.

Prototype `a.insert(p,n,t);`

Remarks Inserts n copies of t before p.

Prototype `a.insert(p,i,j);`

Remarks Inserts copies of elements in [i,j) before p.

erase

Description `Erases the element or elements.`

Prototype `a.erase(q);`

Erases the element pointed to by q.

Prototype `a.erase(q1,q2);`

Erases the elements in the range [q1,q2).

NOTE: These additional member functions only define some basic insert, erase and construction operations. All other operations on the containers (such as sorting, searching, transformations, etc.) are carried out by `generic algorithms` provided by the library.

Associative Container Requirements

STL provides four basic kinds of associative containers: set, multiset, map and multimap.

Before taking a detailed look at the type definitions and member functions provided by the associative containers, we need to define a few terms and explain some of the ideas behind the design.

Basic Design and Organization

All associative containers are parameterized on a type Key and an ordering relation `Compare` that induces a total ordering on elements of Key. In addition, map and multimap associate an arbitrary type T with the Key. An object of type `Compare` is called the *c*`omparison object` of the container.

Equality of Keys

For associative containers, it is important to note that equality of keys depends on the equivalence relation imposed by the comparison, and not on the operator== on keys. Thus, two keys k1 and k2 are considered equal if, for a comparison object comp, `comp(k1,k2)==false && comp(k2,k1)==false`.

Additional Definitions

The set and map containers support unique keys; they can store at most one element of each key. multiset and multimap containers support equal keys; i.e., they can store multiple elements that have the same key.

For set and multiset, the value type is the same as the key type; i.e., the values stored in sets and multisets are basically the keys themselves. For map and multimap, the value type is `pair<const Key,T>`; i.e., the elements stored in maps and multimaps are pairs whose first elements are const Key values and whose second elements are T values.

Finally, an iterator of an associative container is of the bidirectional iterator category. insert operations do not affect the validity of iterators and references to the container, and erase operations only invalidate iterators and references to the erased elements.

Listed below are the type definitions and member functions defined by associative containers in addition to the common container members outlined previously.

In all the definitions, we assume that X is an associative container class, a is a value of `X`, a_uniq is a value of `X` when `X` supports unique keys and a_eq is a value of `X` when `X` supports equal keys. `i` and `j` satisfy input iterator requirements and refer to elements of `value_type. [i,j)` is a valid range, `p` is a valid iterator to a, `q, q1,` and `q2` are valid dereferenceable iterators to `a, [q1,q2)` is a valid range, t is a value of `X::value_type` and `k` is a value of `X::key_type.`

Associative Container Types and Member Functions

key_type

Description The type of keys, Key, with which the container is instantiated.

Definition `X::key_type`

key_compare

Description The comparison object type, `Compare`, with which the container is instantiated.

Definition `X::key_compare`

value_compare

Description A type for comparing objects of `X::value_type`. This is the same as `key_compare` for set and multiset, while for map and multimap it is a type that compares pairs of `Key` and `T` values by comparing their keys using `X::key_compare`.

Definition `X::value_compare`

Constructors

Description `Constructs a container object.`

Prototype

```
X();
X a;
```

Remarks Constructs an empty container, using `Compare()` as a comparison object. Takes constant time.

Prototype

```
X(c);
X a(c);
```

Remarks

Constructs an empty container, using `c` as a comparison object. Takes constant time.

```
X(i,j,c);
X a(i,j,c);
```

Remarks

Constructs an empty container, using c as a comparison object, and inserts elements from the range [i,j) in it. Takes NlogN time in general, where N is the distance from i to j; linear if [i,j) is sorted with value_comp().

```
X(i,j);
X a(i,j);
```

Remarks

Same as above, but uses `Compare()` as a comparison object.

key_comp

Description

Returns the comparison object, of type `X::key_compare`, out of which a was constructed.

Prototype

```
a.key_comp();
```

value_comp

Description

Returns an object of type `X::value_compare` constructed out of the comparison object.

Prototype

```
a.value_comp();
```

insert

Description

Inserts elements under various and specific conditions.

Prototype

```
a_uniq.insert(t);
```

Remarks Inserts t if and only if there is no element in the container with key equal to the key of t. Returns a `pair<iterator,bool>` whose bool component indicates whether the insertion was made and whose iterator component points to the element with key equal to the key of t. Takes time logarithmic in the size of the container.

```
a_eq.insert(p,t);
```

Remarks Inserts t into the container and returns the iterator pointing to the newly inserted element.

```
a.insert(p,t);
```

Remarks Inserts t if and only if there is no element with key equal to the key of t in containers that support unique keys (i.e., sets and maps). Always inserts t in containers that support equal keys (i.e., multisets and multimaps). Iterator p is a hint pointing to where the insert should start to search. Takes time logarithmic in the size of the container in general, but amortized constant if t is inserted right after p.

```
a.insert(i,j);
```

Remarks Inserts the elements from the range [i,j) into the container. Takes NlogN time in general, where N is the distance from i to j. Linear if [i,j) is sorted according to value_comp().

erase

Description Erases and element under various specific conditions.

Prototype

```
a.erase(k);
```

Remarks Erases all elements in the container with key equal to k. Returns the number of erased elements.

```
a.erase(q);
```

Remarks Erases the element pointed to by q.

```
a.erase(q1,q2);
```

Remarks Erases all the elements in the range [q1,q2). Takes log(size() + N) time, where N is the distance from q1 to q2.

find

Description Returns an iterator pointing to an element with key equal to k, or a.end() if such an element is not found.

Prototype
```
a.find(k);
```

count

Description Returns the number of elements with key equal to k.

Prototype
```
a.count(k);
```

lower_bound

Description Returns an iterator pointing to the first element with key not less than k.

Prototype
```
a.lower_bound(k);
```

upper_bound

Description Returns an iterator pointing to the first element with key greater than k.

Prototype
```
a.upper_bound(k);
```

equal_range

Description Returns a pair of iterators (const iterators if a is constant), the first equal to `lower_bound(k)` and the second equal to `upper_bound(k)`.

Prototype

```
a.equal_range(k);
```

NOTE: It must be noted that associative containers provide two constructors to copy ranges: X(i,j,c) and X(i,j). The first version, X(i,j,c), uses c as a comparison object, while the second constructor, X(i,j), uses the comparison object Compare() constructed from the Compare type with which X is instantiated.

Organization of the Container Class Descriptions

Description

The remaining sections of this chapter describe the specific requirements for the three sequence containers (vector, list, deque) and associative containers (set, multiset, map, multimap). Each of these container class descriptions contains the following subsections:

- Files— shows the header file to be included in programs that use the class.
- Class Declaration—the class name and template parameters are shown.
- Description—describes the basic functionality of the class. It serves as a short introduction to the particular container being described.
- Type Definitions—explains the type definitions in the public interface of the class.
- Constructors, Destructors and Related Functions—contains descriptions of constructors and destructors in the class. Some classes also have other related functions that deal with allocation and deallocation issues, and these are explained wherever required.
- Element Access Member Functions—explains the functionality of all member functions that are used to access elements in the container.
- Insert Member Functions—explains all member functions that are used to insert elements into the container.
- Erase Member Functions—details all member functions that are used to erase elements from the container.

- Additional Notes Section(s)—this section or sections contain details such as implementation dependencies, time complexity discussions for insert and erase member functions, memory model dependencies, etc. Any important information that is not included in the other sections is included in the notes sections.

Template class vector<T>

Files

```
#include <vector.h>
```

Declaration

```
template <class T>
class vector;
```

Description

Vectors are containers that arrange elements of a given type in a strictly linear arrangement, and allow fast random access to any element (i.e., any element can be accessed in constant time).

Vectors allow constant time insertions and deletions at the end of the sequence. Inserting and/or deleting elements in the middle of a vector requires linear time. Further details of the time complexity of vector insertion can be found in the notes section.

The topics in this section are:

Type Definitions vector

Iterator

Description

`iterator is a random access iterator type referring to T.`

Definition

```
typedef iterator;
```

const_iterator

Description

const_iterator is a constant random access iterator type referring to const T.

Definition

```
typedef const_iterator;
```

Remarks

It is guaranteed that there is a constructor for const_iterator out of iterator.

T*

Description

The type T* (pointer to T).

Definition

```
typedef Allocator<T>::pointer pointer;
```

reference

Description

`The type T& that can be used for storing into T objects.`

Definition

```
typedef Allocator<T>::reference reference;
```

const_reference

Description

```
typedef Allocator<T>::const_reference
```

```
const_reference;
```

Definition The type const T& that can be used for storing into T objects.

size_type

Description size_type is an unsigned integral type that can represent the size of any vector instance.

Definition

```
typedef size_type;
```

difference_type

Description A signed integral type that can represent the difference between any two pointers to vector::iterator objects.

Definition

```
typedef difference_type;
```

value_type

Description The type of values the vector holds. This is simply T.

Definition

```
typedef T value_type;
```

reverse_iterator

Description Non-constant reverse random access iterator.

Definition

```
typedef reverse_iterator;
```

const_reverse_iterator

Description Constant reverse random access iterator.

Definition `typedef const_reverse_iterator;`

Constructors, Destructors and Related Functions vector

Default Constructor

Description The default constructor. Constructs a vector of size zero.

Prototype

```
vector();
```

Overloaded and Copy Constructors

Prototype

```
explicit vector(size_type n, const T& value = T());
```

Remarks Constructs a vectorof size n and initial izes all its elementswith value. If the second argument is not supplied, `value` is obtained with the default constructor, T(), for the element value type T.

Prototype

```
vector(const vector<T>& x);
```

Remarks The vector copy constructor. Constructs a vector and initializes it with copies of the elements of vector x.

Prototype

```
vector(const_iterator first, const_iterator last);
```

Remarks Constructs a vector of sizelast-first and initializes it with copies of elements in the range [first,last).

Assignment Operator =

Description The vector assignment operator. Replaces the contents of the current vector with a copy of the parameter vector x.

Prototype

```
vector<T>& operator=(const vector<T>& x);
```

Reserve

Description

This member function is a directive that informs the vector of a planned change in size, so storage can be managed accordingly. It does not change the size of the vector, and it takes time at most linear in the size of the vector. Reallocation happens at this point if and only if the current capacity is less than the argument of reserve (capacity is a vector member function that returns the size of the allocated storage in the vector). After a call to reserve, the capacity is greater than or equal to the argument of reserve if reallocation happens, and equal to the previous capacity otherwise.

Prototype

```
void reserve(size_type n);
```

Remarks

Reallocation invalidates all the references, pointers, and iterators referring to the elements in the vector. It is guaranteed that no reallocation takes place during the insertions that happen after reserve takes place till the time when the size of the vector reaches the size specified by reserve.

Destructor

Description

The vector destructor. Returns all allocated storage back to the free store.

Prototype

```
~vector();
```

vector::swap

Description

Swaps the contents of the current vector with those of the input vector x. The current vector replaces x and vice versa.

Prototype

```
void swap(vector<T>& x);
```

Comparison Operations vector

Equality Operator ==

Description

Equality operation on vectors. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator==(const vector<T>& x,
      const vector<T>& y);
```

Less Than Operator <

Description

Returns true if x is lexicographically less than y, false otherwise. Takes linear time.

Prototype

```
bool operator<(const vector<T>& x,
      const vector<T>& y);
```

Element Access Member Functions vector

vector::begin

Description

Returns an iterator (const_iterator for constant vector) that can be used to begin traversing through the vector.

Prototype

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

vector::end

Description

Returns an iterator (const_iterator for constant vector) that can be used in a comparison for ending traversal through the vector.

Prototype

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

vector::rbegin

Description

Returns a reverse_iterator (const_reverse_iterator for constant vectors) that can be used to begin traversing the vector in the reverse of the normal order.

Prototype

```
reverse_iterator rbegin();
const_reverse_iterator rbegin();
```

vector::rend

Description

Returns a reverse_iterator (const_reverse_iterator for constant vectors) that can be used in a comparison for ending reverse-direction traversal through the vector.

Prototype

```
reverse_iterator rend();
const_reverse_iterator rend();
```

vector::size

Description

Returns the number of elements currently stored in the vector.

Prototype

```
size_type size() const;
```

vector::max_size

Description

Returns the maximum possible size of the vector.

Prototype

```
size_type max_size() const;
```

vector::capacity

Description

Returns the largest number of elements that the vector can store without reallocation. See also the `reserve` member function.

Prototype
```
size_type capacity() const;
```

vector::empty

Description Returns true if the vector contains no elements (i.e., if begin() == end()), false otherwise.

Prototype
```
bool empty() const;
```

Subset Operator []

Description Returns the nth element from the beginning of the vector in constant time.

Prototype
```
reference operator[](size_type n);
const reference operator[](size_type n) const;
```

vector::front

Description Returns the first element of the vector; i.e., the element referred to by the iterator begin(). Undefined if the vector is empty.

Prototype
```
reference front();
const_reference front() const;
```

vector::back

Description Returns the last element of the vector; i.e., the element pointed to by the iterator(end()-1) .Undefined if the vector is empty.

Prototype
```
reference back();
const_reference back() const;
```

Insert Member Functions vector

The time complexities of all insert member functions are described in the notes subsections at the end of this section.

vector::push

Description Adds the element x at the end of the vector.

Prototype

```
void push_back(const T& x);
```

vector::insert

Description Inserts an element or elements into position or positions referred to in the vecotor object.

Prototype

```
iterator insert(iterator position, const T& x);
```

Remarks Inserts the element x at the position in the vector referred to by the iterator `position` Elements already in the vector are moved as required. The iterator returned refers to the position at which the element was inserted.

Prototype

```
void insert(iterator position, size_type n,
    const T& x = T());
```

Remarks Inserts n copies of the element x starting at the position referred to by the iterator position.

Prototype

```
void insert(iterator position, const T* first,
    const T* last);
```

Remarks Copies of elements in the range [first,last) are inserted into the vector at the position referred to by the iterator position.

Erase Member Functions vector

vector::pop_back

Description Erases the last element of the vector.

Prototype
```
void pop_back();
```

vector::erase

Description Erases one or more elements from a vector.

Prototype
```
void erase(iterator position);
```

Remarks Erases the element of the vector pointed to by the iterator position.

Prototype
```
void erase(iterator first, iterator last);
```

Remarks The iterators first and last are assumed to point into the vector, and all elements in the range [first,last) are erased from the vector.

Notes on Insert and Erase Member Functions vector

Inserting a single element into a vector is linear in the distance from the insertion point to the end of the vector. The amortized complexity of inserting a single element at the end of a vector is constant (see Section 1.4.2 for discussion of amortized complexity).

Insertion of multiple elements into a vector with a single call of the insert member function is linear in the sum of the number of elements plus the distance to the end of the vector. This means that it is much faster to insert many elements into the middle of a vector at once than to do the insertions one at a time.

All insert member functions cause reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid.

erase invalidates all iterators and references after the point of the erase.The destructor of T is called for each erased element and the assignment operator of T is called the number of times equal to the number ofelements in the vector after the erased elements.

Specialization Class vector<bool>

Files `#include <vector.h>`

Description This class is a specialization of the template class `vector<T>`so as to optimize space allocation. All the member functions of template class `vector<T>`are defined for this class also and it has an extra member function `swap()`.

In this implementation vector<bool> has been replaced by a class `bit_vector`since bool has not been implemented.

Public Member Functions

vector::swap

Description In this implementation, this function is a `global function` which swaps the contents of `x`and `y`

Prototype `void swap(reference x, reference y);`

Template class deque<T>

Files `#include <deque.h>`

Declaration `template <class T> class deque;`

Description

This class implements a deque of objects of type T. Deques are very much like vectors, except that they can be expanded in both directions: they allow constant time insertion and deletion ob objects at either end.

Like vectors, deques allow fast random access to any element in constant time.

The topics in this section are:

Typedef Declarations deque

Description

The following typedef's are defined in the class `deque<T>`.

iterator

Definition

```
typedef iterator;
```

const_iterator

Definition

```
typedef const_iterator;
```

iterator is a random access iterator referring to T. const_iterator is a constant random access iterator referring to const T.

pointer

Description The type T* (pointer to T).

Definition

```
typedef Allocator<T>::pointer pointer;
```

reference

Description The type T& that can be used for storing into T objects.

Definition

```
typedef Allocator<T>::reference reference;
```

const_reference

Description The type const T& for const references that can be used for storing into T objects.

Definition

```
typedef Allocator<T>::const_reference
const_reference;
```

size_type

Description size_type is an unsigned integral type that can represent the size of any deque instance.

Definition

```
typedef size_type;
```

difference_type

Description A signed integral type that can represent the difference between any two deque::iterator objects.

Definition

```
typedef difference_type;
```

value_type

Description The type of values the deque holds. This is simply T.

Definition

```
typedef T value_type;
```

reverse_iterator

Description Non-constant reverse random access iterators

Definition

```
typedef reverse_iterator;
```

const_reverse_iterator

Description Constant reverse random access iterators

Definition

```
typedef const_reverse_iterator;
```

Constructors, Destructors and Related Functions deque

Default Constructor

Description The default constructor. Constructs a deque with size zero.

Prototype

```
deque();
```

Overloaded and Copy Constructors

Prototype

```
explicit deque(size_type n, const T& value = T());
```

Remarks Constructs a deque of size n, and initializes all its elements with value. The default for value is set to T(), where T() is the default constructor of the type passed to the deque class template.

Prototype `deque(const deque<T>& x);`

Remarks The deque copy constructor. Constructs a deque and initializes it with copies of the elements of deque x.

Prototype `deque(const_iterator first, const_iterator last);`

Remarks Constructs a deque of size last-first and initializes it with copies of elements in the range [first,last).

Assignment Operator =

Prototype `deque<T>& operator=(const deque<T>& x);`

Remarks The deque assignment operator. Replaces the contents of the current deque with a copy of the parameter deque x.

Destructor

Description The set destructor. Returns all allocated storage back to the free store.

Prototype `~deque();`

deque::swap

Description Swaps the contents of the current deque with those of the input deque x. The current deque replaces x and vice versa.

Prototype `void swap(deque<T>& x);`

Comparison Operations deque

Equality Operator ==

Description Equality operation on deques. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator==(const deque<T>& x,
      const deque<T>& y);
```

Less Than Operator <

Description Returns true if x is*lexicographically* less than y, false otherwise . Takes linear time.

Prototype

```
bool operator<(const deque<T>& x,
const deque<T>& y);
```

Element Access Member Functions deque

deque::begin

Description Returns an iterator (const_iterator for constant deque) that can be used to begin traversing through all locations in the deque.

Prototype

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

deque::end

Description Returns an iterator (const_iterator for constant deque) that can be used in a comparison for ending traversal through the deque.

Prototype

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

deque::rbegin

Description Returns a reverse_iterator (const_reverse_iterator for constant deques), that can be used to begin traversing all locations in the deque in the reverse of the normal order.

Prototype

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

deque::rend

Description Returns a reverse_iterator (const_reverse_iterator for constant deques), that can be used in a comparison for ending reverse-direction traversal through all locations in the deque.

Prototype

```
reverse_iterator rend();
const_reverse_iterator rend();
```

deque::size

Description Returns the number of elements in the deque.

Prototype

```
size_type size() const;
```

deque::max_size

Description Returns the maximum possible size of the deque.

Prototype

```
size_type max_size() const;
```

deque::empty

Description Returns true if the deque contains no elements (i.e., if begin() == end()), false otherwise

Prototype

```
bool empty() const;
```

Subset Operator []

Description

Allows constant time access to then th element of the deque.

Prototype

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

deque::front

Description

Returns the first element of the deque; i.e., the element pointed to by the iterator begin().

Prototype

```
reference front();
const_reference front() const;
```

deque::back

Description

Returns the last element of the deque; i.e., the element pointed to by the iterator(end()-1) .

Prototype

```
reference back();
const_reference back() const;
```

Insert Member Functions deque

deque::push_front

Description

Adds the element x at the beginning of the deque.

Prototype

```
void push_front(const T& x);
```

deque::push_back

Description Adds the element x at the end of the deque.

Prototype

```
void push_back(const T& x);
```

deque::insert

Description Inserts one or more elements into a position in the deque.

Prototype

```
iterator insert(iterator position, const T& x);
```

Remarks Inserts the element x at the position in the deque pointed to by the iterator position. The iterator returned points to the position that contains the inserted element.

Prototype

```
void insert(iterator position, size_type n,
      const T& x = T());
```

Description Inserts n copies of the element x starting at the position pointed to by the iterator position.

Prototype

```
void insert(iterator position,
      const T* first, const T* last;
```

Description Inserts elements into the deque before the position pointed to by the iterator position. Copies of elements in the range [first,last) are inserted into the deque.

Erase Member Functions deque

deque::pop_front

Description

Prototype `void pop_front();`

Erases the first element of the deque.

deque::pop_back

Description Erases the last element of the deque.

Prototype `void pop_back();`

deque::erase

Description Erases one or more elements of the deque.

Prototype `void erase(iterator position);`

Remarks Erases the element of the deque pointed to by the iterator position.

Prototype `void erase(iterator first, iterator last);`

Remarks The iterators first and last are assumed to point into the deque, and all elements in the range [first,last) are erased from the deque.

Deque Class Notes

Storage Management

As with all STL containers, all storage management in deques is handled automatically.Deques are implemented using segmented storage (unlike vectors). This means that all deque elements are not necessarily kept in contiguous locations in memory.

Complexity of Insertion

Deques are specially optimized for insertion of single elements at either the beginning or the end of the data structure. Such insertions always take constant time and cause a single call to the copy constructorof T, where T is the type of the inserted object.

If an element is inserted into the middle of the deque, then in the worst case the time taken is linear in the minimum of the distance from the insertion point to the beginning of the deque and the distance from the insertion point to the end of the deque.

The insert and push member functions invalidate all the iterators and references to the deque.

Notes on Erase Member Functions

The erase and pop invalidate all the iterators and references to the deque. The number of calls to the destructor (of the erased type T) is the same as the number of elements erased, but the number of calls to the assignment operator of T is equal to the minimum of the number of elements before the erased elements and the number of elements after the erased elements.

Template class list<T>

Files

```
#include <list.h>
```

Declaration

```
template <class T>
class list;
```

Description

This class implements the sequence abstraction as a linked list. All lists are “doubly-linked” and may be traversed in either direction.

Lists should be used in preference to other sequence abstractions when there are frequent insertions and deletions in the middle of sequences. As with all STL containers, storage management is handled automatically.

Unlike vectors or deques, lists are not random-access data structures. For this reason, some STL generic algorithms such as sort, random_shuffle, etc., cannot operate on lists. The list class provides its own sort member function.

Besides sort, lists also include some other special member functions for splicing two lists, reversing lists, making all list elements unique and for merging two lists. All of these special member functions are discussed on "Special Operations list" on page 208.

The topics in this section are:

Typedef Declarations list

Description The following typedef's are defined in the class `list<T>`.

iterator

Description The type iterator is a bidirectional iterator referring to T.

Definition

```
typedef iterator;
```

const_iterator

Description The type const_iterator is a constant bidirectional iterator referring to const T. It is guaranteed that there is a constructor for const_iterator out of iterator.

Definition

```
typedef const_iterator;
```

pointer

Description The type T* (pointer to T).

Definition

```
typedef Allocator<T>::pointer pointer;
```

reference

Description The type T& that can be used for storing into T objects.

Definition

```
typedef Allocator<T>::reference reference;
```

const_reference

Description The type const T& for const references that can be used for storing into T objects.

Definition

```
typedef Allocator<T>::const_reference
      const_reference;
```

size_type

Description size_type is an unsigned integral type that can represent the size of any list instance.

Definition

```
typedef size_type;
```

difference_type

Description A signed integral type that can represent the difference between any two list::iterator objects.

Definition

```
typedef difference_type;
```

value_type

Description The type `T` of values the list holds.

Definition

```
typedef T value_type;
```

reverse_iterator

Description Non-constant reverse bidirectional iterator type.

Definition

```
typedef reverse_iterator;
```

const_reverse_iterator

Description Constant reverse bidirectional iterator type.

Definition

```
typedef const_reverse_iterator;
```

Constructors, Destructors and Related Functions list

Default Constructor

Description The default constructor. Constructs an empty list.

Prototype

```
list();
```

Overloaded and Copy Constructors

Description Constructs a list of size n, and initializes all its elements with value.

Prototype

```
explicit list(size_type n, const T& value = T());
```

Prototype

```
list(const list<T>& x);
```

Remarks The list copy constructor. Constructs a list and initializes it with copies of the elements of list x .

Prototype

```
list(const T* first, const T* last);
```

Remarks Constructs a list of sizelast-first and initializes it with copies of elements in the range [first,last).

Assignment Operator =

Description The list assignment operator. Replaces the contents of the current list with a copy of the parameter list x.

Prototype

```
list<T>& operator=(const list<T>& x);
```

Destructor

Description The list destructor. Returns all allocated storage back to the free store.

Prototype

```
~list();
```

list::swap

Description Swaps the contents of the current list with those of the input list x. The current list replaces x and vice versa.

Prototype

```
void swap(list<T>& x);
```

Comparison Operations list

Equality Operator ==

Description

Equality operation on lists. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator==(const list<T>& x,
      const list<T>& y);
```

Less Than Operator <

Description

Returns true if x islexicographically less than y, false otherwise . Takes linear time

Prototype

```
bool operator<(const list<T>& x, const list<T>& y);
```

Element Access Member Functions list

list::begin

Description

Returns an iterator (const_iterator for constant list) that can be used to begin traversing through the list.

Prototype

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

list::end

Description

Returns an iterator (const_iterator for constant list) that can be used in a comparison for ending traversal through the list.

Prototype

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

list::rbegin

Description Returns a reverse_iterator (const_reverse_iterator for constant lists), that can be used to begin traversing the list in the reverse of the normal order.

Prototype

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

list::rend

Description Returns a reverse_iterator (const_reverse_iterator for constant lists), that can be used in a comparison for ending reverse-direction traversal through the list.

Prototype

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

list::size

Description Returns the number of elements currently stored in the list.

Prototype

```
size_type size() const;
```

list::max_size

Description Returns the maximum possible size of the list.

Prototype

```
size_type max_size() const;
```

list::empty

Description Returns true if the list contains no elements (i.e., if begin() == end()), false otherwise

Prototype

```
bool empty() const;
```

list::front

Description

Returns the first element of the list; i.e., the element pointed to by the iterator begin() .

Prototype

```
reference front();
const_reference front() const;
```

list::back

Description

Returns the last element of the list; i.e., the element pointed to by the iterator(end()-1) .

Prototype

```
reference back();
const_reference back() const;
```

Insert Member Functions list

list::push

Description

Inserts the element x at the beginning of the list .

Prototype

```
void push_front(const T& x);
```

list::push_back

Description

Inserts the element x at the end of the list.

Prototype

```
void push_back(const T& x);
```

list::insert

Description

Inserts one or more elements into the list.

Prototype
```
iterator insert(iterator position, const T& x);
```

Remarks Inserts the element x at the position in the list pointed to by the iterator position. The iterator returned points to the position that contains the inserted element.

Prototype
```
void insert(iterator position, size_type n,
      const T& x = T());
```

Remarks Inserts n copies of the element x starting at the position pointed to by the iterator `position`

Prototype
```
void insert(iterator position, const T* first,
      const T* last);
```

Remarks Inserts elements into the list before the position pointed to by the iterator position. Copies of elements in the range [first,last) are inserted into the list.

Erase Member Functions list

list::pop_front

Description Erases the first element of the list.

Prototype
```
void pop_front();
```

list::pop_back

Description Erases the last element of the list.

Prototype
```
void pop_back();
```

list::erase

Description Erases one or more elements of the list pointed to by the iterator position.

Prototype
```
void erase(iterator position);
```

Remarks Erases the element of the list pointed to by the iterator position.

Prototype
```
void erase(iterator first, iterator last);
```

Remarks The iterators first and last are assumed to point into the list, and all elements in the range [first, last) are erased from the list.

Special Operations list

list::splice

Description These member functions inserts the contents of list x before the iterator position, and x becomes empty.

Prototype
```
void splice(iterator position, list<T>& x);
```

Remarks This member function inserts the contents of list x before the iterator position, and x becomes empty. The operation takes constant time. Essentially, the contents of x are transferred into the current list.

Prototype
```
void splice(iterator position, list<T>& x,
      iterator x_elem);
```

Remarks Inserts the element pointed to by iterator x_elem from list x before position, and removes the element from x. It takes constant time. x_elem is assumed to be a valid iterator of the list x. The function basically transfers a single element from the list x into the current list.

Prototype

```
void splice(iterator position, list<T>& x,
      iterator first, iterator last);
```

Remarks

Inserts the elements in the range [first, last) before the iterator position, and removes the elements from list x. The operation takes linear time. The range [first,last) is assumed to be a valid range in x.

list::remove

Description

This function removes all elements in the list that are equal tovalue, using `T::operator==`. The relative order of other elements is not affected. The entire list is traversed exactly once.

Prototype

```
void remove(const T& value);
```

list::unique

Description

This function erases all but the first element from every consecutive group of equal elements in the list.Exactly size()-1 applications of T::operator== are done. This function is most useful when the list is sorted, so that all elements that are equal appear in consecutive positions. In that case, each element in theresulting list is unique.

Prototype

```
void unique();
```

list::merge

Description

This function merges the argument list x into the current list. It is assumed that both lists are sorted according to the operator< of type T. The merge is stable; i.e., for equal elements in the two lists, the elements from the current list always precede the elements from the argument list x. x becomes empty after the merge. At most size() + x.size() - 1 comparisons are done.

Prototype

```
void merge(list<T>& x);
```

list::reverse

Description Reverses the order of the elements in the list. It takes linear time.

Prototype

```
void reverse();
```

list::sort

Description Sorts the list according to the operator< of type T. The sort is stable; i.e., the relative order of equal elements is preserved. Approximately NlogN comparisons are done, where N is equal to size().

Prototype

```
void sort();
```

List Class Notes

Notes on Insert Member Functions

List insert operations do not affect the validity of iterators and references to other elements of the list. Insertion of a single element of type T into a list takes constant time and makes only one call to the copy constructor of T. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor of T is exactly equal to the number of elements inserted.

Notes on Erase Member Functions

erase invalidates only the iterators and references to the erased elements. Erasing a single element of type T is a constant time operation, with a single call to the destructor of T. Erasing a range in a list takes time linear in the size of the range, and the number of calls to the destructor of type T is exactly equal to the size of the range.

Template class set<T>

Files

```
#include <set.h>
```

Declaration

```
template <class Key, class Compare = less<Key> >
class set;
```

Description

A set<Key, Compare> stores unique elements of type Key, and allows for the retrieval of the elements themselves. All elements in the set are ordered by the ordering relation Compare, which induces a total ordering on the elements.

As with all STL containers, the set container only allocates storage and provides a minimal set of operations (such as insert, erase, find, count, etc.). The set does not itself provide operations for union, intersection, difference etc. These operations are handled by generic algorithms in STL.

The topics in this section are:

- "Typedef Declarations set" on page 211
- "Constructors, Destructors and Related Functions set" on page 214
- "Comparison Operations set" on page 216
- "Element Access Member Functions set" on page 216
- "Insert Member Functions set" on page 218
- "Erase Member Functions set" on page 219
- "Special Operations set" on page 220

Typedef Declarations set

Description

The following typedef's are defined in the class `set`.

key_type

Description

The type of the keys with which the set is instantiated.

Definition

```
typedef Key key_type;
```

value_type

Description

value_type represents the type of the values stored in the set. This is the same as key_type.

Definition

```
typedef Key value_type;
```

value_type represents the type of the values stored in the set. This is the same as key_type.

pointer

Description

The type Key* (pointer to Key).

Definition

```
typedef Allocator<Key>::pointer pointer;
```

The type Key* (pointer to Key).

reference

Description

The type Key& that can be used for storing into Key objects.

Definition

```
typedef Allocator<Key>::reference reference;
```

const_reference

Description

The type Const Key& for const references that can be used for storing into Key objects.

Definition

```
typedef Allocator<Key>::const_reference
      const_reference;
```

The type Key& (const Key& for const references) that can be used for storing into Key objects.

compare_key

Description

The comparison object type, Compare, with which the set is instantiated. This type is used to order the keys in the set.

Definition

```
typedef Compare key_compare;
```

value_compare

Description

This is the ordering relation that is used to order the values stored in the set. Its type is the same as key_compare, since the type of a value stored in a set is the same as the type of the key.

Definition

```
typedef Compare value_compare;
```

iterator

Description

The type iterator is a constant bidirectional iterator referring to const value_type.

Definition

```
typedef iterator;
```

const_iterator

Description

The type const_iterator is the same type as iterator.

Definition

```
typedef const_iterator;
```

size_type

Description

size_type is an unsigned integral type that can represent the size of any set instance.

Definition

```
typedef size_type;
```

difference_type

Description A signed integral type that can represent the difference between any two set::iterator objects.

Definition

```
typedef difference_type;
```

reverse_iterator

Description Non-constant reverse bidirectional iterator type.

Definition

```
typedef reverse_iterator;
```

const_reverse_iterator

Description Constant reverse bidirectional iterator type.

Definition

```
typedef const_reverse_iterator;
```

Constructors, Destructors and Related Functions set

Default Constructor

Description The default constructor. Constructs an empty setusing the relation comp to order the elements.

Prototype

```
set(const Compare& comp = Compare());
```

Overloaded and Copy Constructors

Prototype

```
set(const set<Key, Compare>& x);
```

Remarks The set copy constructor. Constructs a set and initializes it with copies of the elements of set x.

Prototype

```
set(const value_type* first,
    const value_type* last,
    const Compare& comp = Compare());
```

Remarks Constructs an empty set and initializes it with copies of elements in the range [first,last). The ordering relation comp is used to order the elements of the set.

Assignment Operator =

Description The set assignment operator. Replaces the contents of the current set witha copy of the parameter set x.

Prototype

```
set<Key, Compare>& operator=(const set<Key,
      Compare>& x);
```

set::swap

Description Swaps the contents of the current set with those of the input set x. The current set replaces x and vice versa.

Prototype

```
void swap(set<Key, Compare>& x);
```

Destructor

Description The set destructor. Returns all allocated storage back to the free store.

Prototype

```
~set();
```

Comparison Operations set

Equality Operator ==

Description

Equality operation on sets. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator==(const set<Key, Compare>& x,
      const set<Key, Compare>& y);
```

Less Than Operator <

Description

Returns true if x islexicographically less than y, false otherwise . Takes linear time.

Prototype

```
bool operator<(const set<Key, Compare>& x,
      const set<Key, Compare>& y);
```

Element Access Member Functions set

set::key_compare

Description

This function returns the comparison object of the set. The comparison object is an object of class Compare, which represents the ordering relation used to construct the set.

Prototype

```
key_compare key_comp() const;
```

set::value_comp

Description

Returns an object of type value_compare constructed out of the comparison object. For sets, this is simply an object of type Compare.

Prototype
```
value_compare value_comp() const;
```

set::begin

Description Returns the iterator that can be used to begin traversing through all locations in the set.

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

set::end

Description Returns an iterator that can be used in a comparison for ending traversal through the set.

Prototype
```
iterator end() const;
```

set::rbegin

Description Returns a reverse_iterator, that can be used to begin traversing all locations in the set in the reverse of the normal order.

Prototype
```
reverse_iterator rbegin();
```

set::rend

Description Returns a reverse_iterator, that can be used in a comparison for ending reverse-direction traversal through all locations in the set.

Prototype
```
reverse_iterator rend();
```

set::empty

Description Returns true if the set is empty, false otherwise.

Prototype
```
bool empty() const;
```

set::size

Description Returns the number of elements in the set.

Prototype

```
size_type size() const;
```

set::max_size

Description Returns the maximum possible size of the set. The maximum size is simply the total number of elements of type Key that can be represented in the memory model used.

Prototype

```
size_type max_size() const;
```

Insert Member Functions set

set::insert

Description Inserts one or more elements into the set.

Prototype

```
iterator insert(iterator position,
      const value_type& x);
```

Remarks Inserts the element x into the set if x is not already present in the set. The iterator position is a hint, indicating where the insert function should start to search to do the insert. The search is necessary since sets are ordered containers.

The insertion takes O(log N) time, where N is the number of elements in the set, but is amortized constant if x is inserted right after the iterator position.

Prototype

```
pair<iterator, bool> insert(const value_type& x);
```

Remarks Inserts the element x into the set if x is not already present in the set. The returned value is a pair, whose bool component indicates whether the insertion has taken place, and whose iterator component points to the just inserted element in the set, if the insertion takes place, otherwise to the element x already present.

The insertion takes O(log N) time, where N is the number of elements in the set.

Prototype

```
void insert(const value_type* first,
      const value_type* last);
```

Remarks Copies of elements in the range [first,last) are inserted into the set. This insert member function allows elements from other containers to be inserted into the set.

In general, the time taken for this insertion is `Nlog(size() + N)`, where `N` is the distance from first to last, and linear if the range `[first, last)` is sorted according to the set ordering relation `value_comp()`.

Erase Member Functions set

set::erase

Description Erases one or more set elements.

Prototype

```
void erase(iterator position);
```

Remarks Erases the set element pointed toby the iterator position. The time taken is amortized constant.

Prototype

```
size_type erase(const key_type& x);
```

Remarks Erases all the set elements with key equal to x (i.e., removes all x's from the set). Returns the number of erased elements, which is 1 if x is present in the set, and 0 otherwise (since sets do not store duplicate elements). In general, this function takes time proportional to log (size()) + count(x), where count(k) is a set member function that returns the number of elements with key equal to k.

Prototype

```
void erase(iterator first, iterator last);
```

Remarks

The iterators first and last are assumed to point into the set, and all elements in the range [first,last) are erased from the set. The time taken is log(size())+N, where N is the distance from first to last.

Special Operations set

set::find

Description

Searches for the element x in the set. If x is found, the function returns the iterator pointing to it. Otherwise, end() is returned. The function takes O(log N) time, where N is the number of elements in the set.

Prototype

```
iterator find(const key_type& x) const;
```

set::count

Description

Returns the number of elements in the set that are equal to x. If x is present in the set, this number is always 1; otherwise, it is 0. The function takes O(log(size())) time.

Prototype

```
size_type count(const key_type& x) const;
```

set::lower_bound

Description

Returns an iterator pointing to the first set element whose key is not less than x. Since set elements are not repeated, the returned iterator points to x itself if x is present in the set. If x is not present in the set, end() is returned. The function takes O(log N) time, where N is the number of elements in the set.

Prototype

```
iterator lower_bound(const key_type& x) const;
```

set::upper_bound

Description

The upper_bound function returns an iterator to the first set element whose key is greater than x. If no such element is found, end() is returned. The function takes O(log N) time, where N is the number of elements in the set.

Prototype

```
iterator upper_bound(const key_type& x) const;
```

set::equal_range

Description

This function returns the pair(lower_bound(x), upper_bound(x)) . The function takes O(log N) time, where N is the number of elements in the set.

Prototype

```
pair<iterator,iterator> equal_range(
  const key_type& x) const;
```

Template class multiset<Key>

Files

```
#include <multiset.h>
```

Declaration

```
template <class Key, class Compare = less<Key> >
class multiset;
```

Description

A multiset is an associative container that can store multiple copies of the same key.As with regular sets, all elements in the multiset are ordered by the ordering relation Compare, which induces a total ordering on the elements.

Multisets are necessary because we sometimes need to store elements, all of which are alike in most ways but which differ only in certain known characteristics. For example, a set of cars sorted by the make of the car would be a multiset, since there could be several cars in the set with the same manufacturer, but different in other aspects, such as engine capacity, price, etc.

The interface of the multiset class is exactly the same as that of the regular set class. The only difference is that multisets possibly contain multiple values of the same key value. As a result, some of the member functions also have slightly different semantics.

The topcs in this section are:

Typedef Declarations multiset

Description The following typedef's are defined in the class `multiset`.

key_type

Description The type of the keys with which the multiset is instantiated.

Definition

```
typedef Key key_type;
```

value_type

Description value_type represents the type of the values stored in the multiset. This is the same as key_type.

Definition

```
typedef Key value_type;
```

pointer

Description The type Key* (pointer to Key).

Definition typedef Allocator<Key>::pointer pointer

reference

Description The type Key& that can be used for storing into Key objects.

Definition

```
typedef Allocator<Key>::reference reference;
```

const_reference

Description The type const Key& for const references that can be used for storing into Key objects.

Definition

```
typedef Allocator<Key>::const_reference
      const_reference;
```

key_compare

Description The comparison object type, Compare, with which the multiset is instantiated. This type is used to order the keys in the multiset.

Definition

```
typedef Compare key_compare;
```

value_compare

Description This is the ordering relation that is used to order the values stored in the multiset.Its type is the same as key_compare, since the type of a value stored in a multiset is the same as the type of the key.

Definition

```
typedef Compare value_compare;
```

iterator

Description The type iterator is a constant bidirectional iterator referring to const value_type.

Definition

```
typedef iterator;
```

const_iterator

Description The type const_iterator is the same type as iterator.

Definition

```
typedef const_iterator;
```

size_type

Description size_type is an unsigned integral type that can represent the size of any multiset instance.

Definition

```
typedef size_type;
```

difference_type

Description A signed integral type that can represent the difference between any two multiset::iterator objects.

Definition

```
typedef difference_type;
```

reverse_iterator

Description Non-constant reverse bidirectional iterator types.

Definition

```
typedef reverse_iterator;
```

const_reverse_iterator

Description Constant reverse bidirectional iterator types.

Definition

```
typedef const_reverse_iterator;
```

Constructors, Destructors and Related Functions multiset

Default Constructor

Description The default constructor. Constructs an empty multiset using the relation comp to order the elements.

Prototype

```
multiset(const Compare& comp = Compare());
```

Overloaded and Copy Constructors

Prototype

```
multiset(const multiset<Key, Compare>& x);
```

Remarks The multiset copy constructor. Constructs a multiset and initializes it withcopies of theelements of multiset x.

Prototype

```
multiset(const value_type* first,
      const value_type* last,
      const Compare& comp = Compare());
```

Remarks Constructs an empty multiset and initializes it with copies of elements in the range [first,last).The ordering relation comp is used to order the elements of the multiset.

Assignment Operator =

Description The multiset assignment operator. Replaces the contents of the current multiset with a copy ofthe parameter multiset x.

Prototype

```
multiset<Key, Compare>& operator=
      (const multiset<Key, Compare>& x);
```

multiset::swap

Description

Swaps the contents of the current multiset with those of the input multiset x. The current multiset replaces x and vice versa.

Prototype

```
void swap(multiset<Key, Compare>& x);
```

Destructor

Description

The multiset destructor. Returns all allocated storage back to the free store.

Prototype

```
~multiset();
```

Comparison Operations multiset

Equality Operator ==

Description

Equality operation on multisets. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator==(const multiset<Key, Compare>& x,
      const multiset<Key, Compare>& y);
```

Less Than Operator <

Description

Returns true if x islexicographically less than y, false otherwise . Takes linear time.

Prototype

```
bool operator<(const multiset<Key, Compare>& x,
      const multiset<Key, Compare>& y);
```

Element Access Member Functions multiset

multiset::key_comp

Description

This function returns the comparison object of the multiset. The comparison object is an object of class Compare, which represents the ordering relation used to construct the multiset.

Prototype

```
key_compare key_comp() const;
```

multiset::value_comp

Description

Returns an object of type value_compare constructed out of the comparison object. For multisets, this is simply an object of type Compare.

Prototype

```
value_compare value_comp() const;
```

multiset::begin

Description

Returns the iterator that can be used to begin traversing through all locations in the multiset.

Prototype

```
iterator begin() const;
```

multiset::end

Description

Returns an iterator that can be used in a comparison for ending traversal through the multiset.

Prototype

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

multiset::rbegin

Description Returns a reverse_iterator, that can be used to begin traversing all locations in the multiset in the reverse of the normal order.

Prototype
```
reverse_iterator rbegin();
```

multiset::rend

Description Returns a reverse_iterator, that can be used in a comparison for ending reverse-direction traversal through all locations in the multiset.

Prototype
```
reverse_iterator rend();
```

multiset::empty

Description Returns true if the multiset is empty, false otherwise.

Prototype
```
bool empty() const;
```

multiset::size

Description Returns the number of elements in the multiset.

Prototype
```
size_type size() const;
```

multiset::max_size

Description Returns the maximum possible size of the multiset. The maximum size is simply the total number of elements of type Key that can be represented in the memory model used.

Prototype
```
size_type max_size() const;
```

Insert Member Functions multiset

multiset::insert

Description Inserts one more elements into the multiset.

Prototype

```
iterator insert(iterator position,
      const value_type& x);
```

Remarks Inserts the element x into the multiset if x is not already present in the multiset. The iterator position is a hint, indicating where the insert function should start to search to do the insert. This insertion takes O(log N) time in general, where N is the number of elements in the multiset, but is amortized constant if x is inserted right after the iterator position.

Prototype

```
iterator insert(const value_type& x);
```

Remarks Inserts the elementx into the multiset and returns the iterator pointing to the newly inserted element.The function takes O(log N) time, where N is the number of elements in the multiset.

Prototype

```
void insert(const value_type* first,
      const value_type* last);
```

Remarks Copies of elements in the range [first,last) are inserted into the multiset.This insert member function allows elements from other containers to be inserted into the multiset.

In general, the function takes O(Nlog(size()+N)) time, where N is the distance from first to last, and O(N) time if the range [first,last) is sorted according to the multiset ordering relationvalue_comp() .

Erase Member Functions multiset

multiset::erase

Description Erases one or more multiset elements.

Prototype `void erase(iterator position);`

Remarks Erases the multiset element pointed to by the iterator position. The time taken is amortized constant.

Prototype `size_type erase(const key_type& x);`

Remarks Erases the multiset element with key equal to x (i.e., removes all x's from the multiset). Returns the number of erased elements. In general, this function takes time proportional to log (size()) + count(x), where count(k) is a multiset member function that returns the number of elements with key equal to k.

Prototype `void erase(iterator first, iterator last);`

Remarks The iterators first and last are assumed to point into the multiset, and all elements in the range [first,last) are erased from the multiset. The time taken is log(size())+N, where N is the distance from first to last.

Special Operations multiset

multiset::find

Description Searches for the element x in the multiset. If x is found, the function returns the iterator pointing to it. Otherwise, end() is returned. The function takes O(log N) time, where N is the number of elements in the multiset.

Prototype

```
iterator find(const key_type& x) const;
```

multiset::count

Description

Returns the number of elements in the multiset that are equal to x. The function takes O(log N) time, where N is the number of elements in the multiset.

Prototype

```
size_type count(const key_type& x) const;
```

multiset::lower_bound

Description

Returns an iterator pointing to the first multiset element whose key is not less than x. If no such element is found, end() is returned. The function takes O(log N) time, where N is the number of elements in the multiset.

Prototype

```
iterator lower_bound(const key_type& x) const;
```

multiset::upper_bound

Description

The upper_bound function returns an iterator to the first multiset element whose key is greater than x. If no such element is found, end() is returned. The function takes O(log N) time, where N is the number of elements in the multiset.

Prototype

```
iterator upper_bound(const key_type& x) const;
```

multiset::equal_range

Description

This function returns the pair(lower_bound(x), upper_bound(x)) . The function takes O(log N) time, where N is the number of elements in the multiset.

Prototype

```
pair<iterator,iterator> equal_range(
    const key_type& x) const;
```

Template class map<Key, T>

Files

```
#include <map.h>
```

Declaration

```
template <class Key, class T, class Compare =
less<Key> > class map;
```

Description

A map is an associative container that supports unique keys of a given type Key, and provides for fast retrieval of values of another type T based on the stored keys. As in all other STL associative containers, the ordering relation Compare is used to order the elements of the map.

Maps are necessary because we often need to associate elements of one type with values of another. For example, consider a telephone directory, which contains associations of names (string types) and phone numbers (integers). A map<string, long> can be used to provide for fast retrieval of a phone-number that corresponds to a given name.

Elements are stored in maps as pairs in which each Key has an associated value of type T. Since maps store only unique keys, each map contains is at most one <Key, T> pair for each Key value. It is not possible to associate a single Key with more than one value.

The topics in this section are:

- “Typedef Declarations map” on page 233
- “Constructors, Destructors and Related Functions map” on page 235
- “Comparison Operations map” on page 237
- “Element Access Member Functions map” on page 237
- “Insert Member Functions map” on page 240
- “Erase Member Functions map” on page 241
- “Special Operations map” on page 241

Typedef Declarations map

Description The following typedef's are defined in the class `map`.

key_type

Description key_type represents the type of the keys in the map.

Definition
```
typedef Key key_type;
```

value_type

Description value_type represents the type of the values stored in the map. Since maps store pairs of values, value_type is a pair which associates every key (of type Key) with a value of type T.

Definition
```
typedef pair<const Key, T> value_type;
```

key_compare

Description This is the comparison object type, Compare, with which the map is instantiated. It is used to order keys in the map.

Definition
```
typedef Compare key_compare;
```

value_compare

Description A class for comparing objects of map::value_type (i.e., objects of type pair<const Key,T>), by comparing their keys using map::key_compare.

Definition
```
class value_compare;
```

iterator

Description iterator is a bidirectional iterator referring to value_type. It is guaranteed that there is a constructor for const_iterator out of iterator.

Definition

```
typedef iterator;
```

const_iterator

Description Const_iterator is a constant bidirectional iterator referring to const value_type. It is guaranteed that there is a constructor for const_iterator out of iterator.

Definition

```
typedef const_iterator;
```

pointer

Description The type value_type*.(i.e., pair<const Key,T>*).

Definition

```
typedef Allocator<value_type>::pointer pointer;
```

reference

Description The type pair<const Key,T>& that can be used for storing into map::value_type objects.

Definition

```
typedef Allocator<value_type>::reference reference;
```

const_reference

Description The type const Key,T>& (const pair<const Key,T>& for const references that can be used for storing into map::value_type objects.

Definition

```
typedef Allocator<value_type>::const_reference
  const_reference;
```

size_type

Description size_type is an unsigned integral type that can represent the size of any map instance.

Definition
```
typedef size_type;
```

difference_type

Description A signed integral type that can represent the difference between any two map::iterator objects.

Definition
```
typedef difference_type;
```

reverse_iterator

Description Non-constant reverse bidirectional iterator types.

Definition
```
typedef reverse_iterator;
```

const_reverse_iterator

Description Constant reverse bidirectional iterator types.

Definition
```
typedef const_reverse_iterator;
```

Constructors, Destructors and Related Functions map

Default Constructor

Description The default constructor. Constructs an empty map using the relation comp to order the elements.

Prototype

```
map(const Compare& comp = Compare());
```

Overloaded and Copy Constructors

Prototype

```
map(const map<Key, T, Compare>& x);
```

Remarks

The map copy constructor. Constructs a map and initializes it with copiesof the elements of map x.

Prototype

```
map(const value_type* first,
      const value_type* last,
      const Compare& comp = Compare());
```

Remarks

Constructs an empty map and initializes it with copies of elements in the range [first,last). The orderingrelation comp is used to order the elements of the map.

Assignment Operator =

Description

The map assignment operator. Replaces the contents of the current map with a copy of the parameter map x.

Prototype

```
map<Key, T, Compare>& operator=
    (const map<Key, Compare>& x);
```

map::swap

Description

Swaps the contents of the current map with those of theinput map x. The current map replaces x and vice versa.

Prototype

```
void swap(map<Key, T, Compare>& x);
```

Destructor

Description

The map destructor. Returns all allocated storage back to the free store.

Prototype

```
~map();
```

Comparison Operations map

Equality Operator ==

Description

Equality operation on maps. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator==(const map<Key, Compare>& x,
      const map<Key, Compare>& y);
```

Less Than Operator <

Description

Returns true if x islexicographically less than y, false otherwise . Takes linear time.

Prototype

```
bool operator<(const map<Key, Compare>& x,
      const map<Key, Compare>& y);
```

Element Access Member Functions map

map::key_comp

Description

This function returns the comparison object of the map. The comparison object is an object of class Compare, which represents the ordering relation used to construct the map.

Prototype

```
key_compare key_comp() const;
```

map::value_comp

Description

Returns an object of type value_compare constructed out of the comparison object. For maps, value_compare is a class that can be used to compare values stored as pairs in the map.

Prototype
```
value_compare value_comp() const;
```

map::begin

Description Returns an iterator (const_iterator for constant map) that can be used to begin traversing through all locations in the map.

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

map::end

Description Returns an iterator (const_iterator for constant map) that can be used in a comparison for ending traversal through the map.

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

map::rbegin

Description Returns a reverse_iterator (const_reverse_iterator for constant maps), that can be used to begin traversing all locations in the map in the reverse of the normal order.

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

map::rend

Description Returns a reverse_iterator (const_reverse_iterator for constant maps), that can be used in a comparison for ending reverse-direction traversal through all locations in the map.

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

map::empty

Description Returns true if the map is empty, false otherwise.

Prototype

```
bool empty() const;
```

map::size

Description Returns the number of elements in the map.

Prototype

```
size_type size() const;
```

map::max_size

Description Returns the maximum possible size of the map. The maximum size is simply the total number of elements of type Key that can be represented in the memory model used.

Prototype

```
size_type max_size() const;
```

Sub Operator []

Description For a map<Key, T, Compare>, this operator returns the element of type T that is associated with the key Key.

Prototype

```
reference operator[](const key_type& x);
```

Remarks The map subscripting operator is different from the subscripting operator of vectors and deques in that if the map contains no element of type T associated with Key x, then the pair (x, T()) is inserted into the map.

Insert Member Functions map

map::insert

Description

```
Inserts one or more elements or values into the
map.
```

Prototype

```
iterator insert(iterator position,
      const value_type& x);
```

Remarks

Inserts the valuex into the map ifx is not already present in the map. The iterator position is a hint, indicating where theinsert function should start to search to do the insert. This insertion takes O(log N) time in general, where N is the number of elements in the set, but is amortized constant ifx is inserted right afterthe iterator position.

Prototype

```
pair<iterator, bool> insert(const value_type& x);
```

Remarks

Inserts the valuex into the map ifx is not already present in the map. The returned value is apair, whosebool component indicates whether the insertion has taken place, and whose iterator component points to the just inserted value in the map, if the insertion takes place, otherwise to the value x already present.

NOTE: If Notice, that x is a pair of type pair< Key, T>. The function takes O(log N) time, where N is the number of elements in the set.

Prototype

```
void insert(const value_type* first,
      const value_type* last);
```

Remarks

Copies of elements in the range [first,last) are inserted into the map. This insert member function allows elements from other containers to be inserted into the map. In general, this insertion takes O(Nlog(size()+N)) time, where N is the distancefrom first to last, and O(N) time if the range [first,last) is sorted according to the map ordering relation value_comp() .

Erase Member Functions map

map::erase

Description Erases one or more map elements.

Prototype

```
void erase(iterator position);
```

Remarks Erases the map element pointed to by the iterator position. The time taken is amortized constant.

Prototype

```
size_type erase(const key_type& x);
```

Remarks Erases the map element with key equal to x (i.e., removes all pairs whose first element is x from the map). Returns the number of erased elements, which is 1 if x is present in the map, and 0 otherwise. In general, this function takes time proportional to log (size()) + count(x) , where count(k) is a map member function that returns the number of elements with key equal to k.

Prototype

```
void erase(iterator first, iterator last);
```

Remarks The iterators first and last are assumed to point into the map, and all elements in the range [first,last) are erased from the map.The time taken is log(size())+N, where N is the distance from first to last.

Special Operations map

map::find

Description Searches the map for an element with Key equal to x. If such an element is found, the function returns the iterator (const_iterator for constant maps) pointing to it.Otherwise, end() is returned. The function takes O(log N) time, where N is the number of elements in the map.

Prototype

```
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
```

map::count

Description

Returns the number of elements in the map with Key equal to x. If an element with Key equal to x has been inserted into the map then this number is always 1; otherwise, it is 0.

Prototype

```
size_type count(const key_type& x) const;
```

map::lower_bound

Description

Returns an iterator (const_iterator for constant maps) pointing to the first map element whose key is not less than x. If the map contains an element with key not less than x, then the returned iterator points to this element. If such an element is not present in the map, end() is returned. The function takes O(log N) time, where N is the number of elements in the map.

Prototype

```
iterator lower_bound(const key_type& x);
const_iterator lower_bound(
      const key_type& x) const;
```

map::upper_bound

Description

The upper_bound function returns an iterator (const_iterator for constant maps) to the first map element whose key is greater than x. If no such element is found, end() is returned. The function takes O(log N) time, where N is the number of elements in the map.

Prototype

```
iterator upper_bound(const key_type& x)
    const_iterator upper_bound(
      const key_type& x) const;
```

map::equal_range

Description

This function returns the pair(lower_bound(x), upper_bound(x)) . The function takes O(log N) time, where N is the number of elements in the map.

Prototype

```
pair<iterator, iterator> equal_range
      (const key_type& x);
pair<const_iterator, const_iterator> equal_range(
const key_type& x) const;
```

Template class multimap<Key, T>

Files

```
#include <multimap.h>
```

Declaration

```
template <class Key, class T,
  class Compare = less<Key> >class multimap;
```

It is assumed that the operators operator== and an operator< are defined on the type Key.

Description

A multimap is an associative container that stores allows users to store multiple keys of a given type Key, and to efficiently retrieve values of another type T based on the stored Key. As in all other STL associative containers, the ordering relation Compare is used to order the elements of the map.

Multimaps are necessary because we often need to associate more than one object of type T with each Key.

For example, consider a telephone directory organized by last names. Here we might need to associate different telephone numbers (of type integer) with all names ending in Smith. A multimap<name, integer> can be used to hold this information (where name is an appropriately defined type). Corresponding to each name, there might be several telephone numbers, allowing us to easily determine the telephone numbers of different people all of whose last names are Smith.

Elements are stored in multimaps as pairs in which each Key has an associated value of type T. Since multimaps allow multiple keys, it is possible to associate a single Key with more than one value (as was done in the example above).

The topics in this section are:

- "Typedef Declarations multimap" on page 244
- "Constructors, Destructors and Related Functions multimap" on page 247
- "Comparison Operations multimap" on page 248
- "Element Access Member Functions multimap" on page 249
- "Insert Member Functions multimap" on page 251
- "Erase Member Functions multimap" on page 252
- "Special Operations multimap" on page 253

Typedef Declarations multimap

Description The following typedef's are defined in the class `multimap`.

key_type

Description key_type represents the type of the keys in the multimap.

Definition

```
typedef Key key_type;
```

value_type

Description value_type represents the type of the values stored in the multimap. Since multimaps store pairs of values, value_type is a pair which associates every key (of type Key) with a value of type T.

Definition

```
typedef pair<const Key, T> value_type;
```

key_compare

Description This is the comparison object type, Compare, with which the map is instantiated. It is used to order keys in the map.

Definition

```
typedef Compare key_compare;
```

value_compare

Description A class for comparing objects of multimap::value_type (i.e., objects of type pair<const Key,T>), by comparing their keys using multiap::key_compare.

Definition

```
class value_compare;
```

iterator

Description Iterator is a bidirectional iterator referring to value_type. It is guaranteed that there is a constructor for const_iterator out of iterator.

Definition

```
typedef iterator,;
```

const_iterator

Description Const_iterator is a constant bidirectional iterator referring to const value_type. It is guaranteed that there is a constructor for const_iterator out of iterator.

Definition

```
typedef const_iterator;
```

value_type*

Description The type value_type*(i.e., pair<const Key,T>*).

Definition

```
typedef Allocator<value_type>::pointer pointer;
```

reference

Description The type pair<const Key,T>& that can be used for storing into map::value_type objects.

Definition
```
typedef Allocator<value_type>::reference reference;
```

const_reference

Description The type const pair<const Key,T>& for const references that can be used for storing into map::value_type objects.

Definition
```
typedef Allocator<value_type>::const_reference
      const_reference;
```

size_type

Description size_type is an unsigned integral type that can represent the size of any map instance.

Definition
```
typedef size_type;
```

difference_type

Description A signed integral type that can represent the difference between any two map::iterator objects.

Definition
```
typedef difference_type;
```

reverse_iterator

Description Non-constant reverse bidirectional iterator type.

Definition
```
typedef reverse_iterator;
```

name

Description Constant reverse bidirectional iterator type.

Definition

```
typedef const_reverse_iterator;
```

Constructors, Destructors and Related Functions multimap

Default Constructor

Description The default constructor. Constructs an empty multimap using the relationcomp to order the elements.

Prototype

```
multimap(const Compare& comp = Compare());
```

Overloaded and Copy Constructors

Prototype

```
multimap(const multimap<Key, T, Compare>& x);
```

Remarks The multimap copy constructor. Constructs a multimap and initializes it with copies of the elements of multimap x.

Prototype

```
multimap(const value_type* first, const
value_type* last, const Compare& comp = Compare());
```

Remarks Constructs an empty multimap and initializes it with copies of elements in the range [first,last).The ordering relation comp is used to order the elements of the multimap.

Assignment Operator =

Description The multimap assignment operator. Replaces the contents of the current multimap with a copy of the parameter multimap x.

Prototype

```
multimap<Key, T, Compare>& operator=
(const multimap<Key, Compare>& x);
```

multimap::swap

Description

Swaps the contents of the current multimap with those of the input multimap x. The current multimap replaces x and vice versa.

Prototype

```
void swap(multimap<Key, T, Compare>& x);
```

Destructor

Description

The multimap destructor. Returns all allocated storage back to the free store.

Prototype

```
~multimap();
```

Comparison Operations multimap

Equality Operator ==

Description

Equality operation on multimaps. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator==(const multimap<Key, Compare>& x,
      const multimap<Key, Compare>& y);
```

Less Than Operator <

Description

Returns true if x is lexicographically less than y, false otherwise. Takes linear time.

Prototype

```
bool operator<(const multimap<Key, Compare>& x,
      const multimap<Key, Compare>& y);
```

Element Access Member Functions multimap

multimap::key_comp

Description

This function returns the comparison object of the multimap. The comparison object is an object of class Compare, which represents the ordering relation used to construct the multimap.

Prototype

```
key_compare key_comp() const;
```

multimap::value_comp

Description

Returns an object of type value_compare constructed out of the comparison object. For multimaps, value_compare is a class that can be used to compare values stored as pairs in the multimap.

Prototype

```
value_compare value_comp() const;
```

multimap::begin

Description

Returns an iterator (const_iterator for constant multimap) that can be used to begin traversing through all locations in the multimap.

Prototype

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

multimap::end

Description

Returns an iterator (const_iterator for constant multimap) that can be used in a comparison for ending traversal through the multimap.

Prototype

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

multimap::rbegin

Description

Returns a reverse_iterator (const_reverse_iterator for constant multimaps), that can be used to begin traversing all locations in the vector in the reverse of the normal order.

Prototype

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

multimap::rend

Description

Returns a reverse_iterator (const_reverse_iterator for constant multimaps), that can be used in a comparison for ending reverse-direction traversal through all locations in the multimap.

Prototype

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

multimap::empty

Description

Returns true if the multimap is empty, false otherwise.

Prototype

```
bool empty() const;
```

multimap::size

Description

Returns the number of elements in the multimap.

Prototype

```
size_type size() const;
```

multimap::max_size

Description

Returns the maximum possible size of the multimap. The maximum size is simply the total number of elements of type Key that can be represented in the memory model used.

Prototype
```
size_type max_size() const;
```

Insert Member Functions multimap

multimap::Insert

Description Inserts a value or elements into a multimap object.

Prototype
```
iterator insert(iterator position,
      const value_type& x);
```

Remarks Inserts the value `x` into the multimap if `x` is not already present in the multimap. The iterator position is a hint, indicating where the insert function should start to search to do the insert. The insertion takes O(log N) time in general, where N is the number of elements in the map, but is amortized constant if `x` is inserted right after the iterator position.

Prototype
```
iterator insert(const value_type& x);
```

Remarks Inserts the value `x` into the multimap and returns the iterator pointing to the newly inserted value.

NOTE: The value x is a pair of the form pair<const Key, T>. The insertion takes O(log N) time, where N is the number of elements in the map.

```
void insert(const value_type* first,
      const value_type* last);
```

Remarks Copies of elements in the range [first,last) are inserted into the multimap. This insert member function allows elements from other containers to be inserted into the multimap. In general, the time taken for this insertion is O(Nlog(size()+N)), where N is the distance from first to last, and O(N) if the range [first,last) is sorted according to the multimap ordering relation value_comp().

Erase Member Functions multimap

multimap::erase

Description Erases one or more elements or values from the multimap object.

Prototype
```
void erase(iterator position);
```

Remarks Erases the multimap element pointed to by the iterator position. The time taken is amortized constant.

Prototype
```
size_type erase(const key_type& x);
```

Remarks Erases the multimap element with key equal to x (i.e., removes all pairs whose first element is x from the multimap). Returns the number of erased elements.

NOTE: There could be more than one multimap element with key equal to x, since multimaps allow multiple keys.

In general, this function takes time proportional to log(size()) + count(x), where count(k) is a multimap member function that returns the number of elements with key equal to k.

Prototype
```
void erase(iterator first, iterator last);
```

Remarks The iterators first and last are assumed to point into the multimap, and all elements in the range [first,last) are erased from the multimap. The time taken is O(log(size()) + N), where N is the distance from first to last.

Special Operations multimap

multimap::find

Description

Searches the multimap for an element with Key equal to x. Ifsuch an element is found, the function returns the iterator (const_iterator for constant multimaps) pointing to it.Otherwise, end() is returned. The function takes O(log N) time, where N is the number of elements in the multimap.

Prototype

```
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
```

multimap::count

Description

Returns the number of elements in the multimap with key equal to x.

Prototype

```
size_type count(const key_type& x) const;
```

multimap::lower_bound

Description

Returns an iterator (const_iterator for constant multimaps) pointing to the first multimap element whose key is not less than x. If the multimap contains an element with Key not less than x, then the returned iterator points to this single element. If such an element is not present in the multimap, end() is returned. The function takes O(log N) time, where N is the number of elements in the multimap.

Prototype

```
iterator lower_bound(const key_type& x) const;
const_iterator lower_bound(
      const key_type& x) const;
```

multimap::upper_bound

Description

The upper_bound function returns an iterator (const_iterator for constant multimaps) to the first multimap element whose key is greater than x. If no such element is found, end() is returned. The function takes O(log N) time, where N is the number of ele ments in the multimap.

Prototype

```
iterator upper_bound(const key_type& x) const;
const_iterator upper_bound(
      const key_type& x) const;
```

multimap::equal_range

Description

This function returns the pair(lower_bound(x), upper_bound(x)) . The function takes O(log N) time, where N is the number of elements in the multimap.

Prototype

```
pair<iterator,iterator> equal_range(
const key_type& x) const;
pair<const_iterator,const_iterator> equal_range(
const key_type& x) const;
```

Template class stack

Files

```
#include <stack.h>
```

Declaration

```
template <class Container> class stack;
```

NOTE: It is assumed that the operators operator== and an operator< are defined for objects of type Container.

Description

A stack is a data structure that allows the following operations: insertion at one end, deletion from the same end, retrieving the value at the end, and testing for emptiness. Thus, stacks provide a "last-in/first-out" service. The element deleted or retrieved is always the last one inserted.

STL provides a stack container adapter, which can be used to instantiate a stack with any container that supports the following operations: back, push_back, and pop_back. In particular, vectors, lists and deques can be used to instantiate stacks.

The topics in this section are:

- "Public Member Functions stack" on page 255
- "Comparison Operations stack" on page 256

Public Member Functions stack

stack::empty

Description Returns true if the stack is empty, false otherwise.

Prototype

```
bool empty() const;
```

stack::size

Description Returns the current size of the stack (i.e the number of elements the stack currently holds).

Prototype

```
size_type size() const;
```

stack::top

Description Returns the element at the top of the stack. The stack remains unchanged.

Prototype

```
value_type& top() const;,
const value_type& top() const;
```

stack::push

Description Inserts the value `x`at the top of the stack.

Prototype

```
void push(const value_type& x);
```

stack::pop

Description Removes the element at the top of the stack.

Prototype

```
void pop();
```

Comparison Operations stack

Equality Operator ==

Description Equality operation on stacks. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator== (const stack<container>& x,
      const stack<container>& y);
```

Less Than Operator <

Description Returns true if x islexicographically less than y, false otherwise . Takes linear time.

Prototype

```
bool operator< (const stack<container>& x,
      const stack<container>& y);
```

Template class queue

Files

```
#include <queue.h>
```

Declaration

```
template <class Container> class queue;
```

It is assumed that the operators operator== and an operator< are defined for objects of type Container.

Description

A queue is a data structure in which elements are inserted at one end and removed from the opposite end. The order of removal is the same as the order of insertion.

STL provides a queue container adapter, which can be used to instantiate a queue with any container that supports the following operations: empty, size, front, back, push_back, and pop_front. In particular, lists and deques can be used to instantiate queues:

queue< list<int> >, declares a queue of integers with an underlying list implementation

queue< deque<float> >, declares a queue of floats with an underlying deque implementation.

NOTE: The vectors cannot be used to instantiate queues, since they do not provide a pop_front function. This function is not provided for vectors, since it would be highly inefficient for long vectors.

The topics in this section are:

- “Public Member Functions queue” on page 257
- “Comparison Operations queue” on page 259

Public Member Functions queue

queue::empty

Description

Returns true if the queue is empty, false otherwise.

Prototype

```
bool empty() const;
```

queue::size

Description

Returns the current size of the queue (i.e the number of elements the queue currently holds).

Prototype

```
size_type size() const;
```

queue::front

Description

Returns the element at the front of the queue. The queue remains unchanged.

Prototype

```
value_type& front() const;
const value_type& front() const;
```

queue::back

Description

Returns the element at the end of the queue. This is the element that was last inserted into the queue. The queue remains unchanged.

Prototype

```
value_type& back() const;
const value_type& back() const;
```

queue::push

Description

Adds the element x at the end of the queue.

Prototype

```
void push(const value_type& x);
```

queue::pop

Description

Removes the element at the front of the queue.

Prototype

```
void pop()
```

Comparison Operations queue

Equality Operator ==

Description

Equality operation on queues. Returns true if the sequences of elements in x and y are element-wise equal (using T::operator==). Takes linear time.

Prototype

```
bool operator== (const queue<container>& x,
      const queue<container>& y);
```

Less Than Operator <

Description

Returns true if x is lexicographically less than y, false otherwise. Takes linear time.

Prototype

```
bool operator< (const queue<container>& x,
      const queue<container>& y);
```

Template class priority_queue

Files

```
#include <queue.h>
```

Declaration

```
template <class Container> class priority_queue;
```

It is assumed that the operators operator== and operator< are defined for objects of type Container.

Description

A priority queue is a container in which the element immediately available for retrieval is the largest of those in the container, for some particular way of ordering the elements. The order of removal is the same as the order of insertion.

STL provides a priority_queue container adapter, which can be used to instantiate a priority_queue with any container that supports the following operations: empty, size, front, push_back, and pop_back.

In particular, vectors and deques can be used to instantiate priority_queues.

NOTE: Since priority_queues involve an ordering on their elements, a comparison function object comp needs to be supplied to instantiate a priority queue. For example:

priority_queue< vector<int>, less<int> >, declares a priority_queue of integers with a vector implementation and using the built-in < operation for integers to compare the objects.

priority_queue< deque<float>, greater<float> >, declares a priority_queue of floats with a deque implementation, using the > operation on floats for comparisons.

NOTE: Since > is used instead of <, the element available for retrieval at any time is actually the smallest element rather than the largest.

The topics in this section are:

- “Constructors priority_queue” on page 260
- “Public Member Functions priority_queue” on page 261
- “Comparison Operations priority_queue” on page 262

Constructors priority_queue

Default Constructor

Description The default constructor. Constructs a priority_queue using a comparison function object of type Compare.

Prototype

```
priority_queue (const Compare& x = Compare());
```

Overloaded and Copy Constructors

Prototype

```
priority_queue (const value_type* first,
      const value_type* last,
      const Compare& x = Compare());
```

Remarks Constructs a priority queue whose elements are copies of elements in the range [first,last). The default comparison function used is Compare.

Public Member Functions priority_queue

priority_queue::empty

Description Returns true if the priority_queue is empty, false otherwise.

Prototype

```
bool empty() const;
```

priority_queue::size

Description Returns the current size of the priority_queue (i.e the number of elements the priority_queue currently holds).

Prototype

```
size_type size() const;
```

priority_queue::top

Description Returns the element at the top of the priority_queue. The priority_queue remains unchanged.

Prototype

```
value_type& top() const;
const value_type& top() const;
```

priority_queue::push

Description

Adds the element x to the priority_queue.

Prototype

```
void push(const value_type& x);
```

priority_queue::pop

Description

Removes the element at the top of the priority_queue.

Prototype

void pop();

Comparison Operations priority_queue

Equality and comparison operations are not provided for priority queues.

9

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.

Overview of Iterators

This chapter is a reference guide to the requirements that must be satisfied by a class or a built-in type to be used as an iterator. The –iterators of a particular category include: stream iterator classes, iterator adaptors (reverse iterators and insert iterators).

The principle sections in this chapter are:

The following terminology is used in the statement of iterator requirements.

Value type

Iterators are objects that have `operator*` returning a value of some class or built-in `type T` called the `value type` of the `iterator`.

Distance type

For every iterator type for which equality is defined, there is a corresponding signed integral type called the `distance type` of the `iterator`.

Past-the-end values

Just as a regular pointer to an array guarantees that there is a valid pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding container. These values are called `past-the-end` values.

Dereferenceable values.

Values of the iterator for which `operator*` is defined are called dereferenceable. STL components never assume that past-the-end values are dereferenceable.

Singular values.

Iterators might also have singular values that are not associated with any container. For example, after the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` should always be assumed to have a singular value of a pointer. Results of most expressions are undefined for singular values. The only exception is an assignment of a non-singular value to an iterator that holds a singular value. In this case the singular value is overwritten the same way as any other value. Dereferenceable and past-the-end values are always non-singular.

Reachability

An iterator `j` is called reachable from an iterator `i` if and only if there is a finite sequence of applications of `operator++` to `i` that makes `i == j`. If `i` is reachable from `j`, they refer to the same container.

Ranges

Most of the library's algorithmic templates that operate on containers have interfaces that use ranges. A range is a pair of iterators that serve as beginning and end markers for a computation. A range `[i, i)` is an empty range; in general, a range `[i, j)` refers to the positions in a container starting with the one referred to by `i` up to but not including the one pointed to by `j`. Range `[i, j)` is valid if and only if `j` is reachable from `i`. The result of the application of the algorithms in the library to invalid ranges is undefined.

Mutable versus constant

Iterators can be mutable or constant depending on whether the result of `operator*` behaves as a reference or as a reference to a constant. Constant iterators do not satisfy the requirements for output iterators.

NOTE: For all iterator operations that are required in each category, the computing time requirement is `constant time` (amortized). For this reason, we will not mention computing times separately in any of the following sections on requirements.

Iterator Requirements

This section discusses the requirements for all five types of iterators. The topics in this section are:

- "Forward Iterator Requirements" on page 269
- "Random Access Iterator Requirements" on page 271
- "Bidirectional Iterator Requirements" on page 270

In this and the following four requirements sections, for each iterator type `X` we will assume

- `a` and `b` denote values of type `X`,
- `n` denotes a value of the distance type for `X`,
- `r` denotes a value of `X&`,
- `t` denotes a value of value type `T`, and
- `u`, `tmp`, and `m` denote identifiers.

Input Iterator Requirements

A class or a built-in type `X` satisfies the requirements of an input iterator for the value type `T` if and only if the expressions described below are valid.

`X(a)`

Remarks The copy constructor, which makes `X(a) == a`. A destructor is assumed.

```
X u(a);
X u = a;
```

Remarks Either of these results in `u == a.`

`a == b`

Remarks The return type must be convertible to `bool`, and == must be an equivalence relation.

`a != b`

Remarks The return type must be convertible to `bool`, and the result must be the same as `!(a == b)`.

`*a`

Remarks The return type must be convertible to `T`. It is assumed that `a` is dereferenceable. If `a == b`, then it must be the case that `*a == *b`.
`++r`

Remarks The return type must be convertible to `const X&`. It is assumed that `r` is dereferenceable. The result is that `r` is either dereferenceable or `r` is the past-the-end value of the container, and `&r == &++r`.
`r++`

Remarks The return type must be convertible to `const X&`. The result must be the same as that of `{ X tmp = r; ++r; return tmp;}`
`*++r`
`*r++`

Remarks The return type must be convertible to `T`.

NOTE: For input iterators, `a == b` does not imply `++a == ++b`. The main consequence is that algorithms on input iterators should be `single pass` algorithms; i.e., they should never attempt to copy the value of an iterator and use it to pass through the same position twice. Furthermore, value `type T` is not required to be an lvalue type, so algorithms on input iterators should not attempt to assign through them. (Forward iterators remove these restrictions.)

Output Iterator Requirements

A class or a built-in type `X` satisfies the requirements of an output iterator for the value type `T` if and only if the expressions described below are valid.
`X(a);`

Remarks `*a = t` is equivalent to `*X(a) = t`. Further, a destructor is assumed in this case.
`X u(a);`
`X u = a;`

Remarks The result is that `u` is a copy of `a`.

NOTE: However that equality and inequality are not necessarily defined, and algorithms should not attempt to use output iterators to pass through a position twice (i.e., should be single-pass).

`*a = t`

Remarks `t` is assigned through the iterator to the position to which `a` refers.The result of this operation is not used.

`++r`

Remarks The return type must be convertible to `const X&.` It is assumed that `r` is dereferenceable on the left hand side of an assignment. The result is that `r` is either dereferenceable on the left hand side of an assignment or `r` is the past-the-end value of the container, and `&r == &++r.`

`r++`

Remarks The return type must be convertible to `const X&`. The result must be the same as that of `{ X tmp = r; ++r; return tmp;}`

`*++r`

`*r++`

Remarks The return type must be convertible to T.

NOTE: The only valid use of `operator*` on output iterators is on the left hand side of an assignment statement. As with input iterators, algorithms that use output iterators should be single-pass. Equality and inequality operators might not be defined. Algorithms that use output iterators can be used with ostreams as the destination for placing data via the `ostream_iterator` class as well as with insert iterators and insert pointers.

Forward Iterator Requirements

A class or a built-in type `X` satisfies the requirements of a forward iterator for the value type `T` if and only if the expressions described below are valid.

`X u;`

Remarks The resulting value of `u` might be singular. A destructor is assumed.

`X();`

Remarks `X()` might be singular.

`X(a);`

Remarks The result is required to satisfy `a == X(a)`.

`X u(a);`
`X u = a;`

Remarks The result is required to satisfy `u == a`.

`a == b`

Remarks The return type must be convertible to `bool`, and == must be an equivalence relation.

`a != b`

Remarks The return type must be convertible to `bool`, and the result must be the same as `!(a == b)`.

`r = a`

Remarks The return type is `X&` and the result must satisfy `r == a`.

`*a`

Remarks The return type must be convertible to `T`. It is assumed that `a` is dereferenceable. If `a == b`, then it must be the case also that `*a == *b`. If `X` is mutable, `*a = t` is valid.

`++r`

Remarks The return type must be convertible to `X&`. It is assumed that `r` is dereferenceable, and the result is that `r` is either dereferenceable or

is the past-the-end value, and `&r == &++r`. Moreover, `r == s` and `r` is dereferenceable implies `++r == ++s`.
`r++`

Remarks The return type must be convertible to `const X&`. The result must be the same as that of `{ X tmp = r; ++r; return tmp;}`
`*++r`
`*r++`

Remarks The return type must be convertible to `T`.

NOTE: The condition that a == b implies ++a == ++b (which is not true for input or output iterators) and the removal of the restrictions on the number of the assignments through the iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators.

Bidirectional Iterator Requirements

A class or a built-in type `X` satisfies the requirements of a bidirectional iterator for the value type `T` if and only if the expressions described below are valid, in addition to the requirements that are described in the previous section, forward iterators.
`--r`

Remarks The return type is `X&`. It is assumed that there exists `s` such that `r == ++s`, and the result is `r` refers to the same position as `s`, is dereferenceable and `&r == &--r`. Both of the following properties must hold: `--(++r) == r`, and if `--r == --s` implies that `r == s`.
`r--`

Remarks The return type must be convertible to `const X&`.
`*r--`

Remarks The return type must be convertible to `T`.

Random Access Iterator Requirements

A class or a built-in type `X` satisfies the requirements of a random access iterator for the value type `T` if and only if the expressions described below are valid, in addition to the requirements of a bidirectional iterator type.

`r += n`

Remarks The return type must be `X&`. The result must be the same as would be computed by

```
{ Distance m = n;
  if (m >= 0)
    while (m--) ++r;
  else
    while (m++) --r;
return r;}
```

but is computed in constant time.

`a + n`

`n + a`

Remarks The return type must be `X`. The result must be the same as would be computed by

```
{ X tmp = a; return tmp += n;}.
```

`r -= n`

Remarks The return type must be `X&`. The result must be the same as would be computed by `r += -n`.

`a - n`

Remarks The return type must be `X`. The result must be the same as would be computed by `{ X tmp = a; return tmp -= n;}`.

`b - a`

Remarks The return type must be `Distance`. It is assumed that there exists a value `n` of the type `Distance` such that `a + n == b`; the result returned is `n`.

`a[n]`

Remarks The return type must be convertible to `T`.

`a < b`

Remarks The return type must be convertible to `bool`, and `<` must be a total ordering relation.

`a > b`

Remarks The return type must be convertible to `bool`, and `>` must be a total ordering relation opposite to `<`.

`a >= b`

Remarks The return type is convertible to `bool`, and the result must be the same as that of `!(a < b)`.

`a <= b`

Remarks The return type is convertible to `bool`, and the result must be the same as that of `!(b < a)`.

Stream Iterators

The library provides `stream iterators`, defined by template classes, to allow algorithms to work directly with input/output streams. The `istream_iterator` class defines input iterator types and the `ostream_iterator` class defines output iterator types. For example, the following code fragment:

```
istream_iterator<int> end_of_stream;
  partial_sum_copy(istream_iterator<int>(cin),
  end_of_stream,ostream_iterator<int>(cout,“\n”));
```

reads a file containing integers from the input stream `cin`, and prints the partial sums to `cout`, separated by newline characters.

The two stream iterators are:

- “Template class istream_iterator” on page 273
- “Template class ostream_iterator” on page 275

The `istream_iterators` are used to read values from the input stream for which they are constructed. The `ostream_iterators`, are used to write values into the output stream for which they are constructed.

Template class istream_iterator

Files `#include <iterator.h>`

Description An `istream_iterator<T>` reads (using `operator>>`) successive elements of type `T` from the input stream for which it was constructed. Each time `++` is used on a constructed `istream_iterator<T>` object, the iterator reads and stores a value of `T`. The end of stream value is reached when operator `void*()` on the stream returns `false`. In this case, the iterator becomes equal to the `end-of-stream` iterator value. This end-of-stream value can only be constructed using the constructor with no arguments: `istream_iterator<T>()`. Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a `non-end-of-stream iterator.` Two `non-end-of-stream iterators` are equal when they are constructed from the same stream.

One can only use `istream_iterators` to read values; it is impossible to store anything into a position referred to by an `istream_iterator` value.

The main peculiarity of istream iterators is that fact that ++ operators are not equality-preserving; that is, `i == j` does not guarantee that `++i == ++j`. Every time `++` is used a new value is read from the associated istream. The practical consequence of this fact is that istream iterators can only be used with single-pass algorithms.

Prototype

```
template <class T, class Distance = ptrdiff_t>
class istream_iterator :
input_iterator<T,Distance>;
```

The topics in this section are:

- “Constructor istream_iterator” on page 274
- “Public Member Functions istream_iterator” on page 274

- "Comparison Operations istream_iterator" on page 275

Constructor istream_iterator

Default Constructor

Description Constructs the end-of-stream iterator value.

NOTE: The two end-of-stream iterators are always equal.

Prototype

```
istream_iterator();
```

Overloaded and Copy Constructors

Prototype

```
istream_iterator(istream& s);
```

Remarks Constructs an istream_iterator<T> object that reads values from the input stream s.

```
istream_iterator(
  const istream_iterator<T, Distance>& x);
```

Remarks Copy constructor.

Destructor

Prototype

```
~istream_iterator();
```

Public Member Functions istream_iterator

Dereferencing Operator *

Description Dereferencing operator. By returning a reference to `const T`, it ensures that it cannot be used to write values to the input stream for which the iterator is constructed.

Prototype
```
const T& operator*() const;
```

Incrementation Operator ++

Description Incrementation operators.

Prototype
```
istream_iterator<T, Distance>& operator++();
```

Remarks This operator reads and stores a value of `T` each time it is called.

Prototype
```
istream_iterator <T, Distance> operator++(int);
```

Remarks This operator reads and stores x values of T each time it is called.

Comparison Operations istream_iterator

Equality Operator ==

Description Equality operator. Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

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

Template class ostream_iterator

Files
```
#include <iterator.h>
```

Description An `ostream_iterator<T>` object writes (using `operator<<`) successive elements onto the output stream for which it was constructed. If it is constructed with `char*` as a constructor argument, then this `delimiter` string is written to the stream after each `T` value is written.

Prototype

```
template <class T>
class ostream_iterator : public output_iterator;
```

Remarks It is not possible to read a value of the output iterator. It can only be used to `write` values out to an output stream for which it is constructed.

The topics in this section are:

Constructor ostream_iterator

Default Constructor

Description Constructs an iterator that can be used to write to the output stream `s`.

Prototype

```
ostream_iterator(ostream& s);
```

Overloaded and Copy Constructors

Description Constructs an iterator that can be used to write to the output stream `s`. The character string `delimiter` is written out after every value (of type `T`) written to `s`.

Prototype

```
ostream_iterator(ostream& s, const char* delim);
ostream_iterator(const ostream_iterator<T>& x);
```

Remarks Copy constructor.

Destructor

Prototype

```
~ostream_iterator ();
```

Public Member Functions ostream_iterator

Dereferencing Operator *

Description Dereferencing operator. An assignment `*o = t` through an output iterator `o` causes `t` to be written to the output stream and the stream pointer advanced in preparation for the next write.

Prototype

```
ostream_iterator<T>& operator*();
```

Assignment Operator =

Description Assignment operator. Replaces the current iterator with a copy of the iterator `x`.

Prototype

```
ostream_iterator<T>& operator=(
  const ostream_iterator<T>& x);
```

Incrementation Operator ++

Description These operators are present to allow ostream iterators to be used with algorithms that both assign through an output iterator and advance the iterator; they actually do nothing, since assignments through the iterator advance the stream pointer also.

Prototype

```
ostream_iterator <T>& operator++();
ostream_iterator <T> operator++(int x);
```

Template class istreambuf_iterator

Declaration

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

```
class istreambuf_iterator;
```

Description

The template class `istreambuf_iterator` reads successive characters from the `streambuf` for which it was constructed. `operator*` provides access to the current input character, if any. Each time `operator++` is evaluated, the iterator advanced to the next input character. If the end of stream is reached (`streambuf::sgetc ()` returns `traits::eof()`), the iterator becomes equal to the `end of stream iterator` value. The default constructor `istreambuf_iterator ()` and the constructor `istreambuf_iterator(0)` both construct an end of stream iterator object suitable for use as an end-of-range.

The result of `operator*()` on an end of stream is undefined. For any other iterator value a `char_type` is returned. It is impossible to assign a character via an input iterator. In input iterators, ++ operators are not equality preserving, that is, `i == j` does not guarantee at all that `++i == ++j`. Every time ++ is evaluated a new value is used. A practical consequence of this fact is that an `istreambuf_iterator` object can be used only for `one-pass algorithms.` Two end of stream iterators are always equal. An end of stream iterator is not equal to a non-end of stream iterator.

The topics in this section are:

Typedef Declarations istreambuf_iterator

Description

The following typedef's are defined in the class `istreambuf_iterator`.

char_type

Prototype

```
typedef charT char_type;
```

traits_type

Prototype

```
typedef traits traits_type;
```

streambuf

Prototype

```
typedef basic_streambuf<charT, traits> streambuf;
```

istream

Prototype

```
typedef basic_istream<charT, traits istream;
```

Placeholder proxy istreambuf_iterator

Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (operator++). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

Constructor istreambuf_iterator

Default Constructor

Description

Constructs an end-of-stream iterator.

Prototype

```
istreambuf_iterator ();
```

Overloaded and Copy Constructor

Description

Prototype

```
istreambuf_iterator
  (basic_istream<charT, traits>& s);
```

Remarks This constructs the `istream_iterator` pointing to the `basic_streambuf` object `*(s.rdbuf())`.

```
istreambuf_iterator (const proxy& p);
```

Remarks This constructs the `istreambuf_iterator` pointing to the `basic_streambuf` object related to the `proxy` object `p`.

Operators istreambuf_iterator

Dereferencing Operator *

Description This operator extracts one character pointed to by the streambuf *sbuf_.

Prototype

```
charT operator* ();
```

Incrementation Operator ++

Description These operators advances the iterator.

Prototype

```
istreambuf_iterator<charT, traits>&
istreambuf_iterator<charT, traits>::operator++ ();
```

Remarks This operator advances the iterator and returns the result.

Prototype

```
proxy istreambuf_iterator<charT,
  traits>::operator++ (int);
```

Remarks This operator advances the iterator and returns the proxy object keeping the character pointed to by the previous iterator.

istream_iterator::equal

Description This function returns true if and only if both iterators are either at end-of-stream, or are the end-of-stream value, regardless of what streambuf they iterator over.

Prototype

```
bool equal (istreambuf_iterator<charT, traits>& b);
```

istream_iterator::iterator_category

Description

Returns the category of the iterator s.

Prototype

```
input_iterator_tag iterator_category (
  const istreambuf_iterator& s);
```

Equality Operator ==

Description

operator== returns `a.equal (b)`.

Prototype

```
template<class charT, class traits> bool operator==
  (istreambuf_iterator<charT, traits>& a,
    istreambuf_iterator<charT, traits>& b);
```

Not Equal Operator !=

Description

This returns `!a.equal(b)`.

Prototype

```
template<class charT, class traits>bool operator!=
  (istreambuf_iterator<charT, traits>& a,
    istreambuf_iterator<charT, traits>& b);
```

Template class ostreambuf_iterator

Declaration

```
template <class charT, class traits>
class ostreambuf_iterator;
```

Description

The template class `ostreambuf_iterator` writes successive characters onto the output stream from which it was constructed. It is not possible to get a value out of the output iterator. Two output iterators are equal if they are constructed with the same output streambuf.

The topics in this section are:

- "Typedef Declarations ostreambuf_iterator" on page 282
- "Constructor ostreambuf_iterator" on page 282
- "Operators ostreambuf_iterator" on page 283

Typedef Declarations ostreambuf_iterator

Description The following typedef's are defined in the class `ostreambuf_iterator`.

char_type

Prototype
```
typedef charT char_type;
```

traits_type

Prototype
```
typedef traits traits_type;
```

streambuf

Prototype
```
typedef basic_streambuf<charT, traits> streambuf;
```

ostream

Prototype
```
typedef basic_ostream<charT, traits> ostream;
```

Constructor ostreambuf_iterator

Default Constructor

Description Constructs an iterator with `sbuf_` set to `0`.

Prototype
```
ostreambuf_iterator ();
```

Overloaded and Copy Constructors

Prototype

```
ostreambuf_iterator (ostream& s);
```

Remarks This constructs the ostream_iterator pointing to the `basic_streambuf` object `*(s.rdbuf())`.

Prototype

```
ostreambuf_iterator (streambuf* s);
```

Remarks This constructs the `ostreambuf_iterator` pointing to the `basic_streambuf` object `s`.

Operators ostreambuf_iterator

Dereferecing Operator *

Description This operator returns `*this`.

Prototype

```
ostreambuf_iterator<charT, traits>& operator* ();
```

ostreambuf_iterator::equal

Description This function returns true if `sbuf_ == b.sbuf_`.

Prototype

```
bool equal (ostreambuf_iterator<charT, traits>& b);
```

ostreambuf_iterator::iterator_category

Description Returns `output_iterator_tag()`.

Prototype

```
output_iterator_tag iterator_category (
  const ostreambuf_iterator& s);
```

Equality Operator ==

Description `operator==` returns `a.equal (b)`.

Prototype

```
template <class charT, class traits> bool
  operator==(ostreambuf_iterator<charT,traits>& a,
    ostreambuf_iterator<charT, traits> & b);
```

Not Equal Operator !=

Description This returns `!a.equal(b)`.

Prototype

```
template <class charT, class traits> bool
  operator!=(ostreambuf_iterator<charT,traits>& a,
    ostreambuf_iterator<charT, traits>& b);
```

Template class reverse_bidirectional_iterator

Bidirectional iterators and random access iterators have a corresponding reverse iterator adaptor. These adaptors produce iterators that can be used for traversing through a data structure in the opposite of the normal direction. This section describes `reverse_bidirectional_iterator` and the following section describes `reverse_iterator` (for reversing a random access iterator).

The `reverse_bidirectional_iterator` adaptor takes a bidirectional iterator and produces a new bidirectional iterator for traversal in the opposite direction.

Declaration A template class for reverse bidirectional iterators.

Prototype

```
template<class BidirectionalIterator,class T,
class Reference = T&, class Distance = ptrdiff_t>
class reverse_bidirectional_iterator : public
  bidirectional_iterator<T, Distance>
```

The topics in this section are:

- "Constructor reverse_bidirectional_iterator" on page 285
- "Public Member Functions reverse_bidirectional_iterator" on page 285

Constructor reverse_bidirectional_iterator

Default Constructor

Prototype

```
reverse_bidirectional_iterator();
```

Overloaded Constructors

Prototype

```
explicit reverse_bidirectional_iterator
  (BidirectionalIterator x);
```

Remarks This constructor initializes the value of `current` with `x`.

Public Member Functions reverse_bidirectional_iterator

reverse_bidirectional_iterator::base

Prototype

```
BidirectionalIterator base();
```

Dereferencing Operator *

Prototype

```
Dereference operator *();
```

Incrementation Operator ++

Prototype

```
reverse_bidirectional_iterator
    <BidirectionalIterator, T,
    Reference, Distance>& operator ++();
```

Prototype

```
reverse_bidirectional_iterator
    <BidirectionalIterator, T,
    Reference, Distance> operator ++(int);
```

Decrementation Operator --

Prototype

```
reverse_bidirectional_iterator
    <BidirectionalIterator, T,
    Reference, Distance>& operator --();
```

Prototype

```
reverse_bidirectional_iterator
    <BidirectionalIterator, T,
    Reference, Distance> operator --(int);
```

Equality Operator ==

Description Return a `true` if the `reverse_bidirectional_iterator x` is equal to `reverse_bidirectional_iterator y`.

Prototype

```
template <class BidirectionalIterator,
  class T,class Distance> bool operator ==
    (const reverse_bidirectional_iterator
    <BidirectionalIterator,
    T, Reference, Distance>& x,
      const reverse_bidirectional_iterator
        <BidirectionalIterator,
        T, Reference, Distance>& y);
```

Template class reverse_iterator

The `reverse_iterator` adaptor takes a random access iterator and produces a new random access iterator for traversal in the opposite direction.

Description A template class for reverse iterators.

Prototype

```
template <class RandomAccessIterator, class T,
  class Reference = T&, class Distance = ptrdiff_t>
class reverse_iterator : public
    random_access_iterator <T, Distance>;
```

The topics in this section are:

- "Constructor reverse_iterator" on page 287
- "Public Member Functions reverse_iterator" on page 287

Constructor reverse_iterator

Default Constructor

Prototype

```
reverse_iterator();
```

Overloaded and Copy Constructors

Description This constructor initializes the value of `current` with `x`.

Prototype

```
explicit reverse_iterator(RandomAccessIterator x);
```

Public Member Functions reverse_iterator

reverse_iterator::base

Prototype

```
RandomAccessIterator base();
```

Dereferencing Operator *

Prototype

```
Reference operator *();
```

Incrementation Operator ++

Prototype

```
reverse_iterator<RandomAccessIterator, T,
  Reference, Distance>& operator ++();
```

Prototype
```
reverse_iterator<RandomAccessIterator, T,
  Reference, Distance> operator ++(int);
```

Decrementation Operator --

Prototype
```
reverse_iterator<RandomAccessIterator, T,
  Reference, Distance>& operator --();
```

Prototype
```
reverse_iterator<RandomAccessIterator, T,
  Reference, Distance> operator --(int);
```

Add Operator +

Prototype
```
reverse_iterator<RandomAccessIterator, T,
  Reference, Distance> operator+
    (Distance n) const;
```

Add & Assign Operator +=

Prototype
```
reverse_iterator<RandomAccessIterator, T,
  Reference, Distance>& operator +=
    (Distance n);
```

Minus Operator -

Prototype
```
reverse_iterator<RandomAccessIterator, T,
  Reference, Distance> operator -
    (Distance n) const;
```

Minus & Assign Operator -=

Prototype
```
reverse_iterator<RandomAccessIterator, T,
  Reference, Distance>& operator -=
    (Distance n);
```

Subset Operator []

Prototype

```
Reference operator[] (Distance n);
```

Equality Operator ==

Prototype

```
template <class RandomAccessIterator, class T,
  class Reference, class Distance>
    bool operator == (
    const reverse_iterator<RandomAccessIterator,T,
      Reference, Distance>& x,
    const reverse_iterator<RandomAccessIterator, T,
      Reference, Distance>& y);
```

Less Than Operator <

Prototype

```
template <class RandomAccessIterator,
  class T, class Reference, class Distance>
    bool operator < (const reverse_iterator
    <RandomAccessIterator,T,Reference,
      Distance>& x, const reverse_iterator
      <RandomAccessIterator,T,Reference,
      Distance>& y);
```

Minus Operator -

Prototype

```
template <class RandomAccessIterator, class T,
  class Reference, class Distance>
  Distance operator - (const reverse_iterator
  <RandomAccessIterator,T,Reference,Distance>& x,
    const reverse_iterator
  <RandomAccessIterator,T,Reference,Distance>& y);
```

Add Operator +

Prototype

```
template <class RandomAccessIterator, class T,
  class Reference, class Distance>
```

```
Reverse_iterator<RandomAccessIterator,T,
Reference, Distance> operator + (
  Distance n, const reverse_iterator
  <RandomAccessIterator,T,Reference,
  Distance>& x);
```

Template class back_insert_iterator

Insert iterators are provided in the STL to deal with the problem of insertion similar to writing in an array. These iterator are iterator adaptors. There are three types of insert iterator adaptors. They are `back_insert_iterator`, `front_insert_iterator` and `insert_iterator`. In this section, we shall study the interface of these three iterator adaptors.

Description A template class for back insert iterators.

Prototype

```
template <class Container>
  class back_insert_iterator : public
      output_iterator;
```

The topics in this section are:

Constructor back_insert_iterator

Copy Constructor

Prototype

```
explicit back_insert_iterator(Container& x);
```

Public Member Functions back_insert_iterator

Assignment Operator =

Prototype

```
back_insert_iterator<Container>&
    operator=(const typename
    Container::value_type& value);
```

Dereferencing Operator *

Prototype

```
back_insert_iterator<Container>& operator*();
```

Incrementation Operator ++

Prototype

```
back_insert_iterator<Container>& operator++();
```

Prototype

```
back_insert_iterator<Container> operator++(int);
```

back_insert_iterator::back_inserter

Prototype

```
template <class Container>
  back_insert_iterator<Container>
  back_inserter(Container& x);
```

Template class front_insert_iterator

Description A template class for front insert iterators.

Prototype

```
template <class Container>
  class front_insert_iterator : public
    output_iterator;
```

The topics in this section are:

- "Public Member Functions front_insert_iterator" on page 292

Constructor front_insert_iterator

Copy Constructor

Prototype

```
explicit front_insert_iterator(Container& x);
```

Public Member Functions front_insert_iterator

Assignment Operator =

Prototype

```
front_insert_iterator<Container>& operator =
  (const typename Container::value_type& value);
```

Dereferencing Operator *

Prototype

```
front_insert_iterator<Container>& operator*();
```

Incrementation Operator ++

Prototype

```
front_insert_iterator<Container>& operator++();
```

Prototype

```
front_insert_iterator<Container> operator++(int);
```

front_insert_iterator::front_inserter

Prototype

```
template <class Container>
  front_insert_iterator<Container>
  front_inserter(Container& x);
```

Template class insert_iterator

Description A template class for insert iterators

Prototype

```
template <class Container>
class insert_iterator : public output_iterator;
```

The topics in this section are:

- "Constructor insert_iterator" on page 293
- "Public Member Function insert_iterator" on page 293

Constructor insert_iterator

Copy Constructor

Prototype

```
insert_iterator(Container& x,
  typename Container::iterator i);
```

Public Member Function insert_iterator

Assignment Operator =

Prototype

```
insert_iterator<Container>&
  operator = (const typename
    Container::value_type& value);
```

Dereferencing Operator *

Prototype

```
insert_iterator<Container>& operator*();
```

Incrementation Operator ++

Prototype

```
insert_iterator<Container>& operator++();
```

Prototype

```
insert_iterator<Container> operator++(int);
```

insert_iterator::inserter

Prototype

```
template <class Container, class Iterator>
```

```
insert_iterator<Container>
  inserter (Container& x, Iterator i);
```

10

Algorithms Library

This chapter discusses the algorithms library. These algorithms cover sequences, sorting, and numerics.

Overview of the Algorithms Library

The algorithms library contains 32 distinct algorithms, divided into four main categories:

- "Non Mutating Sequence Algorithms" on page 297
- "Mutating Sequence Algorithms" on page 303
- "Sorting and Related Algorithms" on page 318
- "Generalized Numeric Algorithms" on page 341

All of the library algorithms are generic, in the sense that they can operate on a variety of data structures. The algorithms are not directly parameterized in terms of data structures. Instead, they are parameterized by iterator types. This allows the algorithms to work with user-defined data structures, as long as these data structures have iterator types satisfying the assumptions of the algorithms.

The remaining topics in this overview are:

- "In-place and Copying Versions" on page 295
- "Algorithms with Predicate Parameters" on page 296
- "Binary Predicates" on page 296

In-place and Copying Versions

Both in-place and copying versions are provided for certain algorithms. The decision whether to include a copying version is based on complexity considerations.

For example, `sort_copy` is not provided, since the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`. On the other hand,

`replace_copy` is provided, since the cost of copying is greater than the cost of replacing a value in a container.

Whenever, a copying version is provided for algorithm, it is called algorithm_copy.

Finally, algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`, for copying algorithms).

Algorithms with Predicate Parameters

Several algorithms accept function objects as parameters. These function objects are applied to the result of dereferencing the iterators accepted by the algorithm, with the requirement that the resulting value be testable as `true`. A unary `Predicate` class is used in the definition of such algorithms.

In other words, if an algorithm takes a Predicate `pred` as its argument, and `first` as its iterator argument, it should work correctly in the construct

```
if (pred(*first)) {......}
```

The function object `pred` is assumed not to apply any non-constant function through the dereferenced iterator.

Binary Predicates

A `BinaryPredicate` class is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and a type `T` (where `T` is part of the function signature), returns a value testable as `true`.

In other words, if an algorithm takes

```
BinaryPredicate binary_pred
```

as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct

```
if (binary_pred(*first1, *first2)) {...}
```

`BinaryPredicate` always takes the first iterator type as its first argument. That is, in those cases when `T value` is part of the function signature, `binary_pred` can be used as follows:

```
if (binary_pred(first, value)) {....}
```

It is expected that `binary_pred` will not apply any non-constant function through the dereferenced iterators.

Non Mutating Sequence Algorithms

Non-mutating sequence algorithms are those that do not directly modify the containers they operate on. The algorithms in this category are:

- "for_each" on page 297
- "find_if" on page 298
- "adjacent_find" on page 299
- "count" on page 299
- "count_if" on page 300
- "mismatch" on page 300
- "equal" on page 301
- "search" on page 302

Each of these algorithms, except `for_each`, has two versions: a "normal" version, which uses `operator<` or `operator==` for comparisons, and a "predicate" version, which uses an appropriate function object for comparisons.

for_each

Description The `for_each` algorithm applies a specified function to each element of the input container.

Prototype

```
template <class InputIterator, class Function>
  Function for_each(InputIterator first,
  InputIterator last, Function f);
```

Remarks

The function f is applied to the result of dereferencing every iterator in the range `[first, last)`. It is assumed that the function `f` does not apply any non-constant function through the dereferenced iterator. `f` is applied exactly `last-first` times. If `f` returns a result, the result is ignored.

Complexity

Time complexity is linear. If `n` is the size of `[first, last)`, then exactly `n` applications of `f` are made. Space complexity is constant.

find

Description

The first version of the algorithm traverses the iterators `(first, last]` and returns the first iterator `i` such that `*i == value`. In either case, if such an iterator is not found then the iterator `last` is returned.

Prototype

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
  InputIterator last, const T& value);
```

find_if

Description

The second version returns the first iterator `i` such that `pred(*i) == true`. In either case, if such an iterator is not found then the iterator `last` is returned.

Prototype

```
template <class InputIterator, class Predicate>
  InputIterator find_if(InputIterator first,
  InputIterator last, Predicate pred);
```

Complexity

Time complexity is linear. The number of applications of `operator !=` (or `pred`, for `find_if)` is the size of the range `[first, last)`. Space complexity is constant.

adjacent_find

Description

The `adjacent_find` algorithm returns an iterator `i` referring to the first consecutive duplicate element in the range `[first, last)`, or `last` if there is no such element. By consecutive duplicate, it is meant that an element is equal to the element immediately following it in the range.

Prototype

```
template <class InputIterator>
  InputIterator adjacent_find(InputIterator first,
  InputIterator last);
template <class InputIterator,
  class BinaryPredicate>
  InputIterator adjacent_find(InputIterator first,
  InputIterator last, BinaryPredicate binary_pred);
```

Remarks

Comparisons are done using `operator==` in the first version of the algorithm and a function object `binary_pred` in the second version.

Complexity

Time complexity is linear. The number of comparisons done is the size of the range `[first, i)`. Space complexity is constant.

count

Description

The `count` algorithm adds the number of elements in the range `[first, last)` that are equal to `value`, and places the result into the reference argument `n`.

Prototype

```
template <class InputIterator, class T, class Size>
  void count(InputIterator first,
    InputIterator last, const T& value, Size& n);
```

Remarks

`Count` must store the result into a reference argument since it cannot deduce the size type from the built-in iterator types, such as `int*`.

Complexity Time complexity is linear. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

count_if

Description The `count_if` algorithm adds to `n` the number of iterators in the range `[first, last)` for which the condition `pred(*i) == true` is satisfied.

Prototype

```
template <class InputIterator, class Predicate,
  class Size> void count_if(InputIterator first,
    InputIterator last, Predicate pred, Size& n);
```

Complexity Time complexity is linear. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

mismatch

Description `mismatch` compares corresponding pairs of elements from two ranges, and returns the first mismatched pair.

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 binary_pred);
```

Remarks

The algorithm finds the first position at which the values in the range `[first1, last1)` disagree with the values in the range starting at `first2`. It returns a pair of iterators `i` and `j` which satisfy the following conditions:

- i points into the range `[first, last)`
- `j` points into the range beginning at `first2`
- `i` and `j` are both equidistant from the beginning of their corresponding ranges.
- `*i != *j`, or `binary_pred(i, j) == false`, depending on the version of `mismatch` invoked. In the first version, checks for equality are made with `operator==` and in the second version they are made with the function object `binary_pred`.

Complexity

Time complexity is linear. The number of equality operations or applications of the binary predicate is the size of the range `[first1, i)`.

equal

Description

`equal` returns true if the ranges `[first1, last1)` and the range of size `first1-last1` beginning at `first2` contain the same elements in the same order, false otherwise.

Prototype

```
template <class InputIterator1,
class InputIterator2> bool equal(
  InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2);
```

Prototype

```
template <class InputIterator1,
  class InputIterator2,
  class BinaryPredicate> bool equal(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2,
    BinaryPredicate binary_pred);
```

Remarks In the first version of `equal`, checks for element equality are made with `operator==`, and in the second version they are made with the function object `pred`.

Complexity Time complexity is linear. The number of equality operations is the size of the range `[first1,i)`, where `i` is the iterator referring to the first element in the range `[first1, last1)` that does not match the corresponding element in the range beginning at `first2`. Space complexity is constant.

search

Description `search` checks whether the second range `[first2, last2)` is contained in the first range `[first1, last1)`. If so, the iterator `i` in `[first1, last1)` that represents the start of the second range in the first is returned. Otherwise, the past-the-end location of the first range, (i.e. `last1`), is returned.

Prototype

```
template <class ForwardIterator1,
class ForwardIterator2>
  void search(ForwardIterator1 first1,
  Forward Iterator1 last1,
  ForwardIterator2 first2,
  ForwardIterator2 last2);
```

Prototype

```
template <class ForwardIterator1,
class ForwardIterator2,
class BinaryPredicate>
  void search(ForwardIterator1 first1,
  ForwardIterator1 last1,
  ForwardIterator2 first2,
  ForwardIterator2 last2,
  BinaryPredicate binary_pred);
```

Remarks In the first version of the algorithm, checks for element equality are made with `operator==`, while in the second they are made with the function object `binary_pred`.

Complexity

Time complexity is quadratic. If `n` is the size of the range `[first1, last1)`, and `m` is the size of the range `[first2, last2)`, then the number of applications of `operator!=` or `binary_pred` is `(n-m)*m`, which is less than or equal to `n*n/4`. If `m > n`, the time taken is 0 (i.e. no match can be found in this case).

The implementation does not use the Knuth-Morris-Pratt algorithm. The KMP algorithm guarantees linear time, but tends to be slower in most practical cases than the naive algorithm with worst case quadratic behavior. The worst case is extremely unlikely. Space complexity is constant.

Mutating Sequence Algorithms

Mutating sequence algorithms typically modify the containers they operate on.

Table 10.1 The algorithms in this category are:

copy	copy_backward
swap	iter_swap
swap_ranges	transform
replace	replace_if
replace_copy	replace_copy_if
fill	fill_n
generate	generate_n
remove	remove_if
remove_copy	remove_copy_if
unique	unique_copy
reverse	reverse_copy
rotate	rotate_copy
random_shuffle	partition
stable_partition	

copy

Description

`copy` copies elements from the sequence `[first, last)` to the sequence of size `last - first` beginning at the iterator `result`, and returns the past-the-end iterator, `result + last - first`. For each non-negative integer `n < (last-first),` the operation `*(result + n) = *(first + n)` is performed. The result of `copy` is undefined if `result` is in the range `[first, last).`

Prototype

```
template<class InputIterator,
class OutputIterator>
  void copy(InputIterator first,
  InputIterator last,
  OutputIterator result);
```

Complexity

Time complexity is linear for both copy algorithms. At most `n` assignments are performed, where `n` is the size of the range `[first, last)`. Space complexity is constant.

copy_backward

Description

`copy_backward` copies all of the values in the range `[first, last)` to the range of size `last - first` starting at the iterator `result,` and returns the iterator that contains the last element copied (i.e. the beginning of the sequence). For `copy_backward`, the source and destination ranges may overlap if `result >= last`.

Prototype

```
template <class BidirectionalIterator1,
class BidirectionalIterator2>
  void copy_backward(InputIterator first,
  InputIterator last, OutputIterator result);
```

Complexity

Time complexity is linear for both copy algorithms. At most `n` assignments are performed, where `n` is the size of the range `[first, last)`. Space complexity is constant.

swap

Description

The `swap` function exchanges two elements.

Prototype

```
template <class T>
  void swap(T &a, T &b);
```

Complexity

The time required is constant for `swap` and `iter_swap`. For `swap_ranges`, the time complexity is linear. The number of swaps performed is the size of the range `[first, last)`. Space complexity is constant for all the swap algorithms.

iter_swap

Description

The function `iter_swap` exchanges values pointed to by two iterators.

Prototype

```
template <class ForwardIterator1, ForwardIterator2>
  void iter_swap(ForwardIterator1 a,
  ForwardIterator2 b);
```

Complexity

The time required is constant for `swap` and `iter_swap`. For `swap_ranges`, the time complexity is linear. The number of swaps performed is the size of the range `[first, last)`. Space complexity is constant for all the swap algorithms.

swap_ranges

Description

The function `swap_ranges` exchanges the elements in the range `[first, last)` with those in the range of size `last - first` beginning at `first2`. `swap_ranges` returns the past-the-end iterator, `first2+last-first`.

Prototype

```
template <class ForwardIterator1,
class ForwardIterator2>
  ForwardIterator swap_ranges(
  ForwardIterator1 first1,
```

```
  ForwardIterator1 last1,
  ForwardIterator2 first2);
```

Complexity The time required is constant for `swap` and `iter_swap`. For `swap_ranges`, the time complexity is linear. The number of swaps performed is the size of the range `[first, last)`. Space complexity is constant for all the swap algorithms.

transform

Description The first version of `transform` generates an output sequence of elements by applying a `unary` function `op` to each element of the input sequence `[first, last)`.

The second version of `transform` accepts the input sequence `[first1, last1)` and the sequence of length `last1 - first1` starting at `first2`, and generates an output by applying a `binary` operation `binary_op` to each corresponding pair of elements from the input sequences.

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);
```

Remarks

For both versions of `transform`, the resulting sequence is placed starting at the position `result`, and the past-the-end iterator is returned.

Complexity

Time complexity is linear. The number of applications of `op` is the size of the range `[first, last)` and the number of applications of `binary_op` is the size of the range `[first1, last1)`. Space complexity is constant.

replace

Description

The `replace` algorithm modifies the range `[first, last)` so that all elements equal to `old_value` are replaced by `new_value`, while other values remain unchanged.

Prototype

```
template <class ForwardIterator, class T>
  void replace(ForwardIterator first,
    ForwardIterator last, const T& old_value,
    const T& new_value);
```

Complexity

Time complexity is linear for all replace algorithms. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

replace_if

Description

`replace_if` modifies the range `[first, last)` so that all elements that satisfy the predicate `pred` are replaced by `new_value`, while other values remain unchanged.

Prototype

```
template <class ForwardIterator,
class Predicate, class T>
  void replace_if(
    ForwardIterator first, ForwardITerator last,
    Predicate pred, const T& new_value);
```

Complexity

Time complexity is linear for all replace algorithms. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

replace_copy

Description

`replace_copy` is similar to `replace`, except that the original sequence is not modified. Rather, the altered sequence is placed in the range of size `last - first` beginning at `result`.

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);
```

Complexity

Time complexity is linear for all replace algorithms. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

replace_copy_if

Description

`replace_copy_if` is similar to `replace_if`, except that the original sequence is not modified. Rather, the altered sequence is placed in the range of size `last - first` beginning at `result`.

Prototype

```
template <class InputIterator,
class OutputIterator,
class Predicate, class T>
  OutputIterator replace_copy_if(
    InputIterator first, InputIterator last,
    OutputIterator result, Predicate pred,
    const T& new_value):
```

Complexity Time complexity is linear for all replace algorithms. The number of equality operations performed, or of applications of the predicate `pred`, is the size of the range `[first, last)`. Space complexity is constant.

fill

Description `fill` assigns `value` through all the iterators in the range `[first, last)`.

Prototype

```
template <class ForwardIterator, class T>
  void fill(ForwardIterator first, ForwardIterator
    last, const T& value);
```

Complexity Time complexity is linear. The number of assignments for both versions of the algorithm is the size of the range `[first, last)`. Space complexity is constant.

fill_n

Description `fill_n` assigns `value` through all the iterators in the range `[first, first + n)`.

Prototype

```
template <class ForwardIterator,
class Size, class T>
  void fill_n(ForwardIterator first,
    Size n, const T& value);
```

Complexity Time complexity is linear. The number of assignments for both versions of the algorithm is the size of the range `[first, last)`. Space complexity is constant.

generate

Description `generate` fills the range `[first, last)` with the sequence generated by `last - first` successive calls to the function object `gen`.

Prototype

```
template <class ForwardIterator, class Generator>
  void generate(ForwardIterator first,
    ForwardIterator last, Generator gen);
```

Complexity

Time complexity is linear. The number of assignments for `generate` is the size of the range `[first, last)`, while for `generate_n` the number of assignments is `n`. Space complexity is constant.

generate_n

Description

`generate_n` fills the range of size `n` beginning at `first` with the sequence generated by `n` successive calls to `gen`.

Prototype

```
template <class ForwardIterator, class Size,
class Generator>
  void generate_n(ForwardIterator first,
    Size n, Generator gen);
```

Complexity

Time complexity is linear. The number of assignments for `generate` is the size of the range `[first, last)`, while for `generate_n` the number of assignments is `n`. Space complexity is constant.

remove

Description

The function `remove` removes those elements from the range `[first, last)` that are equal to `value`, and returns the location `i` that is the past-the-end iterator for the resulting range of values that are not equal to `value`.

Prototype

```
template <class ForwardIterator, class T>
  ForwardIterator remove(ForwardIterator first,
  ForwardIterator last, const T& value);
```

It is important to note that neither `remove` nor `remove_if` alters the size of the original container: the algorithms operate by copying (with assignments) the final generated elements into the range `[first, i)`. No calls are made to the `insert` or `erase` member functions of the containers operated on by the algorithms.

All versions of `remove` are stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

Complexity

Time complexity is linear for all versions of `remove`. The number of assignments is the number of elements not removed, at most the size of the range `[first, last)`. Space complexity is constant for all the remove algorithms.

remove_if

Description

The function `remove_if` removes those elements from the range `[first, last)` which satisfy the predicate `pred`, and returns the location `i` that is the past-the-end iterator for the resulting range of values that are not equal to `value`.

Prototype

```
template <class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ForwardIterator first,
  ForwardIterator last, Predicate pred);
```

Remarks

It is important to note that neither `remove` nor `remove_if` alters the size of the original container: the algorithms operate by copying (with assignments) the final generated elements into the range `[first, i)`. No calls are made to the `insert` or `erase` member functions of the containers operated on by the algorithms.

All versions of `remove` are stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

Complexity

Time complexity is linear for all versions of `remove`. The number of assignments is the number of elements not removed, at most the size of the range `[first, last)`. Space complexity is constant for all the remove algorithms.

remove_copy

Description

`remove_copy` is similar to `remove`, except that the final resulting sequences are copied into the range beginning at `result`.

Prototype

```
template <class InputIterator,
class OutputIterator, class T>
  OutputIterator remove_copy(
    InputIterator first, InputIterator last,
    OutputIterator result, const T& value);
```

Remarks

All versions of `remove` are stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

Complexity

Time complexity is linear for all versions of `remove`. The number of assignments is the number of elements not removed, at most the size of the range `[first, last)`. Space complexity is constant for all the remove algorithms.

remove_copy_if

Description

`remove_copy_if` is similar to `remove_if`, except that the final resulting sequences are copied into the range beginning at `result`.

Prototype

```
template <class InputIterator,
class OutputIterator, class Predicate>
  OutputIterator remove_copy_if(
    InputIterator first, InputIterator last,
    OutputIterator result, Predicate pred);
```

Remarks

All versions of `remove` are stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

Complexity

Time complexity is linear for all versions of `remove`. The number of assignments is the number of elements not removed, at most the size of the range `[first, last)`. Space complexity is constant for all the remove algorithms.

unique

Description

`unique` eliminates consecutive duplicates from the range `[first, last)`. An element is considered to be a consecutive duplicate if it is equal to an element in the location to its immediate right in the range.

In the first version of `unique`, checks for equality are made using `operator==`, while in the second they are made with the function object `binary_pred`.

Prototype

```
template <class ForwardIterator>
ForwardIterator unique(
  ForwardIterator first, ForwardIterator last);
```

Prototype

```
template <class ForwardIterator,
class BinaryPredicate>
  ForwardIterator unique(
    Forwarditerator first, ForwardIterator last,
    BinaryPredicate binary_pred);
```

Remarks

All versions of `unique` return the end of the resulting range.

The `unique` algorithms are typically applied to a sorted range, since in this case all duplicates are consecutive duplicates.

Complexity

Time complexity is linear for all versions of unique. Exactly `last - first` applications of the corresponding predicates are done. Space complexity is constant for `unique`, and linear for `unique_copy`.

unique_copy

Description

`unique_copy` is similar to `unique`, except that the resulting sequence is copied into the range starting at `result`.

Prototype

```
template <class inputIterator,
class OutputIterator>
  OutputIterator unique_copy(
    InputIterator first, InputIterator last,
    OutputIterator result);
```

Prototype

```
template <class ForwardIterator,
class BinaryPredicate>
  OutputIterator unique_copy(
    InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryPredicate binary_pred);
```

Remarks All versions of `unique` return the end of the resulting range.

The `unique` algorithms are typically applied to a sorted range, since in this case all duplicates are consecutive duplicates.

Complexity Time complexity is linear for all versions of unique. Exactly `last - first` applications of the corresponding predicates are done. Space complexity is constant for `unique`, and linear for `unique_copy`.

reverse

Description The `reverse` algorithm reverses the order of elements in the range `[first, last)`.

Prototype

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first,
  BidirectionalIterator last);
```

Complexity Time complexity is linear. For `reverse`, exactly `[n/2]` element exchanges are performed, where `n` is the size `[first, last)`. For `reverse_copy`, exactly `last-first` assignments are done. Space complexity is constant.

reverse_copy

Description

The `reverse_copy` algorithm reverses the sequence `[first, last)` and copies the resulting sequence into the range beginning `result`.

Prototype

```
template <class BidirectionalIterator,
class OutputIterator>
  OutputIterator reverse_copy(
    BidirectionalIterator first,
    BidirectionalIterator last,
    OutputIterator result);
```

Complexity

Time complexity is linear. For `reverse`, exactly `[n/2]` element exchanges are performed, where `n` is the size `[first, last)`. For `reverse_copy`, exactly `last-first` assignments are done. Space complexity is constant.

rotate

Description

The `rotate` algorithm shifts elements in a sequence leftward as follows: for each non-negative integer `i < (last - first)`, `rotate` places the element from the position `first + i` into position `first + (i + (middle -first)) % (last - first)`.

After the `rotate` operation, an element originally at location `i` in the sequence `[first, last)` is finally placed at location `(i+n-m) mod n`, where `m` is the size of the range `[first,middle)`, and `n` is the size of the range `[first, last)`.

Prototype

```
template <class ForwardIterator>
  void rotate(ForwardIterator first,
    ForwardIterator middle, ForwardIterator last);
```

Complexity

Time complexity is linear.

For `rotate`, exactly `2*[n/2]+[m/2]+[(n-m)/2]` element exchanges are performed, where `m` is the size of the range `[first,` `middle)` and `n` is the size `[first, last)`. Space complexity is constant.

rotate_copy

Description

`rotate_copy` is similar to `rotate`, except that it copies the elements of the resulting sequence into a range of size `last-first` starting at the location `result`.

Prototype

```
template <class ForwardIterator,
class OutputIterator>
  void rotate_copy(ForwardIterator first,
    ForwardIterator middle,ForwardIterator last,
    OutputIterator result);
```

Complexity

Time complexity is linear.

For `rotate_copy`, exactly `n` assignment operations are performed, where `n` is the size `[first, last)`. Space complexity is constant.

random_shuffle

Description

`random_shuffle` shuffles the elements in the range `[first, last)` with uniform distribution. `random_shuffle` can take a particular random number generating function object `rand` such that `rand` returns a randomly chosen `double` in the interval `[0, 1)`.

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);
```

Complexity

Time complexity is linear. The algorithm performs exactly (`last - first) - 1` swaps. Space complexity is constant.

partition

Description

The `partition` algorithm places all elements in the range `[first, last)` that satisfy `pred` before all elements that do not satisfy it.

Prototype

```
template <class BidirectionalIterator,
class Predicate>
  void partition(BidirectionalIterator first,
    BidirectionalIterator last, Predicate pred);
```

Remarks

Both algorithms return an iterator `i` such that for any iterator `j` in the range `[first,i), pred(*j) == true`, and for any iterator `k` in the range `[i, last), pred(*k) == false`.

For `partition`, exactly `[n/2]` element exchanges are performed, where `n` is the size `[first, last)`. Exactly `last - first` applications of the predicate are performed.

Complexity

Time complexity is linear for both versions of the algorithm. Space complexity is constant.

stable_partition

Prototype

```
template <class BidirectionalIterator,
class Predicate>
  void stable_partition(
    BidirectionalIterator first,
    BidirectionalIterator last, Predicate pred);
```

Description

In `stable_partition`, the relative positions of the elements in both groups are preserved. `partition` does not guarantee this.

Remarks

Both algorithms return an iterator `i` such that for any iterator `j` in the range `[first,i), pred(*j) == true`, and for any iterator `k` in the range `[i, last), pred(*k) == false`.

If the available memory for a buffer is smaller than the range `[first, last)`, the `stable_partition` function requires `O(n log n)` time and performs `n log n` swaps, where `n` is the size of `[first, last)`. If there is enough available memory for the buffer to contain all the elements in the range `[first, last)`, then the `stable_partition` function requires linear time, performing `n + m` assignment operations and applying the predicate exactly `n` times, where `m` is the size of the range `[i, last)` and `n` is the size of `[first, last)`.

Complexity

Time complexity is linear for both versions of the algorithm. For `stable_partition`, the time and space complexity varies with the available memory.

Sorting and Related Algorithms

There are several distinct sets of algorithms related to sorting. Each individual set contains a collection of related algorithms. The sets are:

- "Sorting" on page 319
- "Binary Searching" on page 323
- "Merging" on page 327
- "Set Operations on Sorted Structures" on page 328
- "Heap Operations" on page 333
- "Finding Min and Max" on page 336
- "Lexicographical Comparison" on page 338
- "Permutation Generators" on page 339

All of the algorithms have two versions: one that uses `operator<` for comparisons and another that uses a function object of type `Compare`.

`Compare` is used as a function object which accepts two arguments, returns `true` if the first argument is less than the second, and returns `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) == true` defaults to `*i < *j == true`. For the algorithms to work correctly, `comp` has to induce a total ordering on the values.

A sequence is sorted with respect to a comparator `comp` if for any iterator `i` pointing to the sequence and any non-negative integer `n` such that `i+n` is a valid iterator pointing to an element of the sequence, `comp(*(i+n), *i) == false`.

In the descriptions of the functions that deal with ordering relationships, we frequently use a notion of equality to describe concepts such as stability. The equality to which we refer is not necessarily an `operator==`, but an equality relation induced by the total ordering. That is, two elements `a` and `b` are considered equal if and only if `!(a < b) && (!b < a)`.

Sorting

Four different sorting algorithms are provided by the library: `sort`, `stable_sort`, `partial_sort` and `partial_sort_copy`.

`sort` uses the quicksort algorithm, which is generally the fastest for a randomly shuffled sequence. It should be used as the default sorting algorithm. However, in the worst case, `sort` might take quadratic time. This worst case occurs if the original input to the algorithm is already sorted.

If worst case behavior is absolutely critical, then `stable_sort` should be used instead of `sort`.

sort

Description

`sort` sorts the elements in the range `[first, last)`. It uses a fast quicksort algorithm. If worst case behavior is important `stable_sort` or `partial_sort` should be used.

Prototype

```
template <class RandomAccessIterator>
  void sort(RandomAccessIterator first,
    RandomAccessIterator last);
template <class RandomAccessIterator,
class Compare>
  void sort(RandomAccessIterator first,
    RandomAccessIterator last, Compare comp);
```

Complexity

Sort does approximately `NlogN` (where `N` is `(last-first)`) comparisons on the average. Space complexity is constant.

stable_sort

Description

`stable_sort` sorts the elements in the range `[first, last)`, and ensures that the relative order of the equal elements is preserved.

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);
```

Remarks

`stable_sort` performs at most `NlogN*logN` (where N is `last-first`) comparisons. If adequate memory is available, then the number of comparisons is `NlogN`. Space complexity for `stable_sort` is variable, at most `O(N)`.

Complexity

Sort does approximately `NlogN` (where `N` is `(last-first)`) comparisons on the average. Space complexity is constant.

partial_sort

Description

`partial_sort` sorts only a subsequence of the input range. It places the first `middle-first` sorted elements from the range `[first, last)` into the range `[first,middle)`. The rest of the elements (i.e. those in the range `[middle,last)`) are placed in an undefined order.

Prototype

```
template <class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last);
```

Prototype

```
template <class RandomAccessIterator,
class Compare>
  void partial_sort(RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last, Compare comp);
```

Remarks

`partial_sort` does approximately `(last-first) * log(middle -first)` comparisons. Space complexity is constant.

Complexity

Sort does approximately `NlogN` (where `N` is `(last-first)`) comparisons on the average. Space complexity is constant.

partial_sort_copy

Description

`partial_sort_copy` is similar to `partial_sort`. The algorithm places the first `min(last-first, result_last - result_first)` sorted elements from the range `[first, last)` into the sequence beginning at `result_first`.

Prototype

```
template <class InputIterator,
class RandomAccessIterator>
```

```
  RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last);
```

Prototype

```
template <class InputIterator,
class RandomAccessIterator, class Compare>
  RandomAccessIterator partial_sort_copy(
    InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last,
    Compare comp);
```

Remarks

`partial_sort_copy` does approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons. Space complexity is proportional to the length of the sequence copied.

Complexity

Sort does approximately `NlogN` (where `N` is `(last-first)`) comparisons on the average. Space complexity is constant.

nth_element

Description

The `nth_element` algorithm is a restricted form of the `sort` algorithm. It places an element of a sequence in the location where it would be if the sequence were sorted.

Prototype

```
template <class RandomAccessIterator>
  void nth_element(
    RandomAccessIterator first,
    RandomAccessIterator nth,
    RandomAccessIterator last);
```

Prototype

```
template <class RandomAccessIterator,
class Compare>
  void nth_element(
    RandomAccessIterator first,
    RandomAccessIterator nth,
```

```
    RandomAccessIterator last, Compare comp);
```

Remarks

In the first version of the algorithm, element comparisons are done using `operator<`, while in the second version they are done using the function object `comp`.

After a call to `nth_element`, for any iterator `i` in the range `[first,nth)` and any iterator `j` in the range `[nth,last)` it holds that `!(*i > *j) or comp(*i, *j) == false`. That is, the algorithm partitions the elements of the sequence according to size: elements to the left of the nth element are all less than those to its right.

Complexity

The algorithm takes `O(N)` time on the average, where `N` is the size of the range `[first, last)`. Space complexity is constant.

Binary Searching

All of the algorithms in this section are versions of binary search.

Although binary search is typically efficient (i.e., performs in logarithmic time) only for random access data structures (such as vectors, deques, etc.), the algorithms here have been written so as to also work on non-random access data structures such as lists. For all non-random access data structures, the total time taken is linear in the size of the container, but the number of comparisons is only logarithmic in the size of the container.

binary_search

Description

The `binary_search` function returns true if `value` is in the range `[first, last)` and false otherwise.

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);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity For random access iterators only, the number of comparisons will be

- at most `log(N)+1`, for `lower_bound` and `upper_bound`
- at most `log(N)+2`, for `binary_search`
- at most `2*log(N)+1`, for `equal_range`

where `N` is the size of the range `[first, last)`.

For all iterators other than random access, time complexity is linear, since the algorithm has to sequentially traverse through the data structure. Space complexity is constant for all the binary search algorithms.

lower_bound

Description The `lower_bound` functions return an iterator `i` referring to the first position in the sorted sequence in the range `[first, last)` into which `value` may be inserted while maintaining the sorted ordering.

Prototype

```
template <class ForwardIterator, class T>
  ForwardIterator lower_bound(
    ForwardIterator first,
    ForwardIterator last, const T& value);
```

Prototype

```
template <class ForwardIterator, class T,
class Compare>
  ForwardIterator lower_bound(
    ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
```

Remarks

In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity

For random access iterators only, the number of comparisons will be

- at most `log(N)+1`, for `lower_bound` and `upper_bound`
- at most `log(N)+2`, for `binary_search`
- at most `2*log(N)+1`, for `equal_range`

where `N` is the size of the range `[first, last)`.

For all iterators other than random access, time complexity is linear, since the algorithm has to sequentially traverse through the data structure. Space complexity is constant for all the binary search algorithms.

upper_bound

Description

The `upper_bound` functions return an iterator `i` referring to the last position in the sorted sequence in the range `[first, last)` into which `value` may be inserted while maintaining the sorted ordering.

Prototype

```
template <class ForwardIterator, class T>
  ForwardIterator upper_bound(
    ForwardIterator first,
    ForwardIterator last, const T& value);
```

Prototype

```
template <class ForwardIterator, class T,
class Compare>
  ForwardIterator upper_bound(
    ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
```

Remarks

In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity

For random access iterators only, the number of comparisons will be

- at most `log(N)+1`, for `lower_bound` and `upper_bound`
- at most `log(N)+2`, for `binary_search`
- at most `2*log(N)+1`, for `equal_range`

where `N` is the size of the range `[first, last)`.

For all iterators other than random access, time complexity is linear, since the algorithm has to sequentially traverse through the data structure. Space complexity is constant for all the binary search algorithms.

equal_range

Description

The `equal_range` functions return a pair of iterators `i` and `j` referring to the first and last positions in the sorted sequence in the range `[first, last)` into which `value` may be inserted while maintaining the sorted ordering.

Prototype

```
template <class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
  equal_range (ForwardIterator first,
    ForwardIterator last,const T& value);
```

Prototype

```
template <class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
  equal_range(ForwardIterator first,
    ForwardIterator last,
    const T& value, Compare comp);
```

Remarks

In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity

For random access iterators only, the number of comparisons will be

- at most `log(N)+1`, for `lower_bound` and `upper_bound`
- at most `log(N)+2`, for `binary_search`

- at most `2*log(N)+1`, for `equal_range`

where `N` is the size of the range `[first, last)`.

For all iterators other than random access, time complexity is linear, since the algorithm has to sequentially traverse through the data structure. Space complexity is constant for all the binary search algorithms.

Merging

The merge algorithms join two sorted ranges.

merge

Description

`merge` merges two sorted ranges `[first1, last1)` and `[first2, last2)` into the range `[result,result + (last1 - first1) + (last2 - first2))`. The merge is stable, that is, for equal elements in the two ranges, the elements from the first range always precede the elements from the second. `merge` returns `result + (last1 - first1) + (last2 - first2)`. The result of `merge` is undefined if the resulting range overlaps with either of the original ranges.

Prototype

```
template <class InputIterator1,
class InputIterator2,
class OutputIterator>
  OutputIterator merge(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator last2,
    OutputIterator result);
```

Prototype

```
template <class InputIterator1,
class InputIterator2,
class OutputIterator, class Compare>
  OutputIterator merge(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator last2,
    OutputIterator result, Compare comp);
```

Complexity

For `merge`, at most `(last1 - first1) + (last2 - first2) - 1` comparisons are performed.

inplace_merge

Description

`inplace_merge` merges two sorted consecutive ranges `[first,middle)` and `[middle,last)` putting the result of the merge into the range `[first, last)`. The merge is `stable`, that is, for equal elements in the two ranges, the elements from the first range always precede the elements from the second.

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);
```

Complexity

For `inplace_merge`, at most `last - first` comparisons are performed. If no additional memory is available, the number of assignments can be equal to `NlogN` where `N` is equal to `last - first`.

Set Operations on Sorted Structures

The library provides five different types of set operations:

- “includes” on page 329
- “set_union” on page 330
- “set_intersection” on page 330
- “set_difference” on page 331

• “set_symmetric_difference” on page 332

The operations work on sorted structures, such as all STL associative containers.

The algorithms even work with multisets containing multiple copies of equal elements. The semantics of the operations have been generalized to multisets in a standard way, by defining union to contain the maximum number of occurrences of every element, intersection to contain the minimum, and so on.

includes

Description

`includes` checks if the second sequence is a subset of the first sequence (both ranges are assumed to be sorted). The algorithm returns `true` if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`, and `false` otherwise.

Prototype

```
template <class InputIterator1,
class InputIterator2>
  bool includes(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2);
```

Prototype

```
template <class InputIterator1,
class InputIterator2, class Compare>
  bool includes(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    Compare comp);
```

Complexity

TAll of the set operations functions require linear time. In all cases, at most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. Space complexity is constant.

set_union

Description

`set_union` constructs a sorted union of the elements from the two ranges. `set_union` is stable, that is, if an element is present in both ranges, the one from the first range is copied.

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);
```

Complexity

The union, intersection, difference and symmetric difference algorithms all return the end of the constructed range. The result of each of these algorithms is undefined if the resulting range overlaps with either of the original ranges.

TAll of the set operations functions require linear time. In all cases, at most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. Space complexity is constant.

set_intersection

Description

`set_intersection` constructs a sorted intersection of the elements from the two ranges. `set_intersection` is guaranteed to be `stable`, that is, if an element is present in both ranges, the one from the first range is copied into the intersection.

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);
```

Remarks

The union, intersection, difference and symmetric difference algorithms all return the end of the constructed range. The result of each of these algorithms is undefined if the resulting range overlaps with either of the original ranges.

Complexity

TAll of the set operations functions require linear time. In all cases, at most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. Space complexity is constant.

set_difference

Description

`set_difference` constructs a sorted difference of the elements from the two ranges. This difference contains elements that are present in the first set but not in the second.

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);
```

Remarks

The union, intersection, difference and symmetric difference algorithms all return the end of the constructed range. The result of each of these algorithms is undefined if the resulting range overlaps with either of the original ranges.

Complexity

TAll of the set operations functions require linear time. In all cases, at most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. Space complexity is constant.

set_symmetric_difference

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,
class InputIterator2,
class OutputIterator,
class Compare>
  OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);
```

Description

`set_symmetric_difference` constructs a sorted symmetric difference of the elements from the two ranges (i.e. a combination of the set of elements that are in the first range but not in the second and vice-versa).

Remarks

The union, intersection, difference and symmetric difference algorithms all return the end of the constructed range. The result of each of these algorithms is undefined if the resulting range overlaps with either of the original ranges.

Complexity

TAll of the set operations functions require linear time. In all cases, at most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed. Space complexity is constant.

Heap Operations

A heap represents a particular organization of a random access data structure (such as a vector or a deque). Given a range `[first, last)`, where `first` and `last` are random access iterators, we say that the elements in the range represent a heap if two key properties are satisfied:

- The value pointed to by the iterator `first` is the largest element in the range
- The value pointed to by the iterator `first` may be removed by `pop_heap`, or a new element added by `push_heap`, in logarithmic time. Both `pop_heap` and `push_heap` return valid heaps.

These properties allow heaps to be used as priority queues.

In addition to `pop_heap` and `push_heap` there are two more heap algorithms: `make_heap` and `sort_heap`.

push_heap

Description

`push_heap` assumes the range `[first1, last - 1)` is a valid heap and properly places the value in the location `last - 1` into the resulting heap `[first, last)`.

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);
```

Remarks

In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity

Tn all of the complexity descriptions below, the number `N` represents the size of the range `[first, last)`.

The `push_heap` and `pop_heap` functions require logarithmic time. The `push_heap` functions require at most `logN` time, and the `pop_heap` functions require at most `2*logN` comparisons.

Space complexity is constant for all heap algorithms.

pop_heap

Description

`pop_heap` assumes the range `[first, last)` is a valid heap, then swaps the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into 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);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity Tn all of the complexity descriptions below, the number `N` represents the size of the range `[first, last)`.

The `push_heap` and `pop_heap` functions require logarithmic time. The `push_heap` functions require at most `logN` time, and the `pop_heap` functions require at most `2*logN` comparisons.

Space complexity is constant for all heap algorithms.

make_heap

Description The `make_heap` functions construct a heap in the range `[first, last)` using the elements in the range `[first, last)`.

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);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity Tn all of the complexity descriptions below, the number `N` represents the size of the range `[first, last)`.

The `make_heap` functions require linear time and require at most `3*N` comparisons.

Space complexity is constant for all heap algorithms.

sort_heap

Description The `sort_heap` functions sort the elements that are stored in the heap represented in the range `[first, last)`.

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);
```

Remarks In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity Tn all of the complexity descriptions below, the number `N` represents the size of the range `[first, last)`.

The `sort_heap` functions require `NlogN` time and `NlogN` comparisons.

Space complexity is constant for all heap algorithms.

Finding Min and Max

The min and max algorithms identifiy the larger or smaller of two elements, or of elements in a range.

min

Description The `min` function is passed two elements, and returns the one that is smaller.

Prototype

```
template <class T>
  T min(const T &a, const T &b);
```

Prototype

```
template <class T, class Compare>
  T min(const T &a, const T &b, Compare comp);
```

Remarks

In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity

min and `max` take constant time. Space complexity is constant for all `min` and `max` algorithms.

max

Description

The `max` function is passed two elements, and returns the one that is larger.

Prototype

```
template <class T>
  T max(const T &a, const T &b);
```

Prototype

```
template <class T, class Compare>
  T max(const T &a, const T &b, Compare comp);
```

Remarks

In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity

min and `max` take constant time. Space complexity is constant for all `min` and `max` algorithms.

min_element

Description

The `min_element` function returns an iterator `i` referring to the minimum of the elements in the range `[first, last)`.

Prototype

```
template <class InputIterator>
  InputIterator min_element(
  InputIterator first, InputIterator last);
```

Prototype

```
template <class InputIterator, class Compare>
  InputIterator min_element(
    InputIterator first, InputIterator last,
    Compare comp);
```

Remarks

In the first function of each pair of functions, element comparisons are done using `operator<`, while in the second they are done using the function object `comp`.

Complexity

T`min_element` takes linear time, with the number of element comparisons being the size of the range `[first, last)`. Space complexity is constant for all `min` and `max` algorithms.

max_element

Description

The `max_element` functions returns an iterator `i` referring to the maximum of the elements in the range `[first, last)`.

Prototype

```
template <class InputIterator>
InputIterator max_element(
  InputIterator first, InputIterator last);
```

Prototype

```
template <class InputIterator, class Compare>
  InputIterator max_element(
    InputIterator first, InputIterator last,
    Compare comp);
```

Complexity

T`max_element` takes linear time, with the number of element comparisons being the size of the range `[first, last)`. Space complexity is constant for all `min` and `max` algorithms.

Lexicographical Comparison

The lexicographical comparison algorithm compares two sequences of elements.

lexicographical_compare

Description

The lexicographical comparison of two sequences `[first1, last1)` and `[first2, last2)` is defined as follows: traverse the sequences, comparing corresponding pairs of elements `e1` and `e2`. If `e1 < e2`, stop and return `true`. If `e2 < e1`, stop and return `false`. Otherwise, continue to the next corresponding pair of elements.

If the first sequence is exhausted but the second is not, then return `true`, otherwise return `false`.

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);
```

Remarks

Comparisons of `e1` and `e2` are done with `operator<` in the first version of the algorithm, and with the function object `comp` in the second version.

Complexity

Time complexity is linear. The number of comparisons done is at most `i`, where `i` is the smallest index at which a disagreement occurs. Space complexity is constant.

Permutation Generators

The library provides two permutation generation algorithms: `next_permutation` and `prev_permutation`. As with all sorting related operations, there are two versions of each algorithm: one that uses `operator<` for comparisons, and one that uses a function object `comp`.

next_permutation

Description

`next_permutation` takes a sequence defined by the range `[first, last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`. If such a permutation exists, the algorithm returns `true`. Otherwise, it transforms the sequence into the smallest permutation (i.e., the one sorted in an ascending order), and returns `false`.

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);
```

Complexity

Time complexity is linear for both algorithms. The number of comparisons done is at most `n` where `n` is half the range `[first, last)`. Space complexity is constant.

prev_permutation

Description

`prev_permutation` takes a sequence defined by the range `[first, last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the largest permutation, (i.e., the descending sorted one), and returns `false`.

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);
```

Complexity Time complexity is linear for both algorithms. The number of comparisons done is at most `n` where `n` is half the range `[first, last)`. Space complexity is constant.

Generalized Numeric Algorithms

The library provides four algorithms for numeric processing:

- "accumulate" on page 341
- "inner_product" on page 342
- "partial_sum" on page 343
- "adjacent_difference" on page 344

See also "Generalized Numeric Algorithms" on page 341.

accumulate

Description `accumulate` is similar to the APL reduction operator and the Common Lisp reduce function, but avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.

Prototype

```
template <class InputIterator,class T>
  T accumulate(InputIterator first,
    InputIterator last, T init);
```

Prototype

```
template <class InputIterator, class T,
class BinaryOperation>
  T accumulate(
    InputIterator first, InputIterator last,
    T init, BinaryOperation binary_op);
```

Remarks

Accumulation is done by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first, last)` in order. `binary_op` is assumed not to cause any side effects.

Complexity

Time complexity is linear. The number of applications of `operator+` or `binary_op` done is `n` where `n` is the range `[first, last)`. Space complexity is constant.

inner_product

Description

The inner_product algorithm computes its result by initializing the accumulator acc with initial value init and then modifying it as follows:

```
acc = acc + (*i1) * (*i2)
```

or

```
acc = binary_op1(acc, binary_op2(*i1, *i2))
```

for every iterator `i1` in the range

```
[first1, last1)
```

and `i2` in the range

```
[first2, first2 + (last1 - first1))
```

in order.

`binary_op1` and `binary_op2` are assumed to cause no side effects.

Prototype

```
template <class InputIterator1,
class InputIterator2, class T>
  T inner_product(
```

```
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, T init);
```

Prototype

```
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);
```

Complexity

Time complexity is linear. The number of applications of `operator+` or `binary_op1`, and the number of applications of `operator*` or `binary_op2` is `n` where `n` is the size of the range `[first, last)`. Space complexity is constant.

partial_sum

Description

The `partial_sum` algorithm computes partial sums of the input sequence and stores the result in the output sequence.

Formally, `partial_sum` assigns to every iterator `i` in the range

```
    [result, result + (last - first))
```

a value correspondingly equal to

```
    ((...(*first + *(first + 1)) +...)
      + (*first + (i - result)))
```

or

```
    binary_op(binary_op
    (., binary_op(*first + *(first + 1)) +...)
      + (*first + (i - result))).
```

`binary_op` is expected not to have any side effects.

Prototype

```
template <class InputIterator,
class OutputIterator>
  InputIterator partial_sum(
```

```
    InputIterator first, InputIterator last,
    OutputIterator result);
```

Prototype

```
template <class InputIterator,
class OutputIterator, class BinaryOperation>
  InputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);
```

Remarks

The `partial_sum` algorithm returns an iterator referring to the past-the-end location of the result sequence.

NOTE: The `result` may be equal to `first`; i.e., it is possible for the algorithm to work “in-place”, meaning that the algorithm can generate the partial sums and replace the original sequence with them.

Complexity

Time complexity is linear. The number of applications of `operator+` or `binary_op` is one less than `n`, where `n` is the size of the range `[first, last)`. Space complexity is constant.

adjacent_difference

Description

`adjacent_difference` assigns to every element referred to by iterator `i` in the range

```
  [result + 1, result + (last - first))
```

a value correspondingly equal to

```
    *(first + (i - result)) - *(first
      + (i - result) - 1)
```

or

```
    binary_op(*(first + (i - result)),
      *(first + (i - result) - 1)).
```

`binary_op` is expected not to have any side effects.

Prototype

```
template <class InputIterator,
class OutputIterator>
InputIterator adjacent_difference(
InputIterator first, InputIterator last,
OutputIterator result);
```

Prototype

```
template <class InputIterator, class OutputIterator
class BinaryOperation>
  InputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);
```

Remarks

`result` may be equal to `first`; i.e., the algorithm can work "in-place".

Complexity

Time complexity is linear. The number of applications of `operator-` or `binary_op` is one less than `n` where `n` is the range `[first, last)`. Space complexity is constant.

11

Numerics Library

This chapter is a reference guide to the ANSI/ISO standard Numeric classes which are used to perform the semi-numerical operations.

Overview of the Numerics Library

This library contains the classes for complex number types, numeric arrays, generalized numeric algorithms, and facilities included from the ISO C library.

The sections in this chapter are:

- “Numeric Type Requirements” on page 347
- “Template Class complex” on page 348
- “Numeric Arrays” on page 354
- “Generalized Numeric Operations” on page 390

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 with types that satisfy the following requirements:

- `T` is not an abstract class, i.e it has no pure virtual member functions;
- `T` is not a reference type;
- `T` is not cv-qualified;
- If `T` is a class, it has a public default constructor;

- If `T` is a class, it has a public copy constructor with the signature `T::T(const T&);`
- If `T` is a class, it has a public destructor;
- If `T` is a class, it has a public assignment operator whose signature is either
  `T& T::operator=(const T&) or T& T:operator=(T);`

If `T` is a class, its assignment operator, copy and default constructors, and destructor must correspond to each other in the following sense: Initialization of raw storage using the default constructor, followed by assignment, is semantically 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.

In addition, many member and related functions of `valarray<T>` can be successfully instantiated and will exhibit well-defined behavior if and only if `T` satisfies additional requirements specified for each such member or related function.

Template Class complex

The header `<complex>` defines a template class, and numerous functions for representing and manipulating complex numbers.

Files `#include <complex.h>`

The topics in this section are:

Constructor complex

Constructor

Description This function constructs an object of class complex.

Prototype

```
template<class T> complex(T re=T(), T im = T());
```

Member Operators complex

Add & Assign Operator +=

Description This function adds the complex value `rhs` to the complex value `*this` and stores the sum in `*this`.

Prototype

```
template<class T> complex<T>& operator+=
  (const complex<T>& rhs);
```

Minus and Assign Operator -=

Description This function subtracts the complex value `rhs` from the complex value `*this` and stores the difference in `*this`. It returns `*this`.

Prototype

```
template<class T> complex<T>& operator-=
  (const complex<T>&rhs);
```

Multiply and Assign Operator *=

Description This function multiplies the complex value `rhs` from the complex value `*this` and stores the product in `*this`. It returns `*this`.

Prototype

```
template<class T> complex<T>&
  operator*=(const complex<T>& rhs);
```

Divide and Assign Operator /=

Description This function divides the complex value `rhs` from the complex value `*this` and stores the quotient in `*this`. It returns `*this`.

Prototype

```
template<class T> complex<T>&
  operator/=(const complex<T>& rhs);
```

Non-member Operators complex

Add Operator +

Description This function acts as the unary operator. It returns `lhs`.

Prototype

```
template<class T> complex<T>
  operator+(const complex<T>& lhs);
```

Prototype

```
template<class T> complex<T> operator+(
  const complex<T>& lhs,
  const complex<T>& rhs);
template<class T> complex<T>
  operator+(const complex<T>& lhs, T rhs);
```

Prototype

```
template<class T> complex<T>
  operator+(T lhs,const complex<T>& rhs);
```

Remarks These function adds `lhs` to `rhs` and returns the result.

Minus Operator -

Description This function acts as the unary operator. It returns `lhs`.

Prototype

```
template<class T> complex<T>
  operator-(const complex<T>& lhs);
```

Prototype

```
template<class T> complex<T>
  operator-(const complex<T>& lhs,
  const complex<T>& rhs);
```

Prototype

```
template<class T> complex<T>
  operator-(const complex<T>& lhs, T rhs);
```

Prototype

```
template<class T> complex<T>
  operator-(T lhs, const complex<T>& rhs);
```

Remarks These function subtracts `rhs` from `lhs` and returns the result.

Multiply Operator *

Description These function multiplies `rhs` with `lhs` and returns the result.

Prototype

```
template<class T> complex<T>
  operator*(const complex<T>& lhs,
  const complex<T>& rhs);
```

Prototype

```
template<class T> complex<T>
  operator*(const complex<T>& lhs, T rhs);
```

Prototype

```
template<class T> complex<T>
  operator*(T lhs, const complex<T>& rhs);
```

Divide Operator /

Description These function divides `rhs` from `lhs` and returns the result.

Prototype

```
template<class T> complex<T> operator/(const
  complex<T>& lhs, const complex<T>& rhs);
```

Prototype

```
template<class T> complex<T> operator/(const
  complex<T>& lhs, T rhs);
```

Prototype

```
template<class T> complex<T> operator/(T lhs,
  const complex<T>& rhs);
```

Equality Operator

Description

These function compare the real and imaginary parts of `rhs` and `lhs` and returns the result. In case of `T` arguments, the imaginary part is assumed to be 0.

Prototype

```
template<class T> complex<T> operator==(const
  complex<T>& lhs, const complex<T>& rhs);
```

Prototype

```
template<class T> complex<T> operator==(const
  complex<T>& lhs, T rhs);
```

Prototype

```
template<class T> complex<T> operator==(T lhs,
  const complex<T>& rhs);
```

Not Equal Operator !=

Description

These function compare the real and imaginary parts of `rhs` and `lhs` and returns the result. In case of `T` arguments, the imaginary part is assumed to be 0.

Prototype

```
template<class T> complex<T> operator!=(const
  complex<T>& lhs, const complex<T>& rhs);
```

Prototype

```
template<class T> complex<T> operator!=(const
  complex<T>& lhs, T rhs);
```

Prototype

```
template<class T> complex<T> operator!=(T lhs,
  const complex<T>& rhs);
```

Extractor Operator >>

Description

This function extracts a complex number `x` from the input stream `is`. If bad input is encountered, calls `is.setstate(ios::failbit)`. It returns `is`.

Prototype

```
template<class T> istream& operator>>(istream& is,
  complex<T>& x);
```

Inserter Operator <<

Description

This function inserts a complex number `x` into the output stream `os`. If bad input is encountered, calls `is.setstate(ios::failbit)`. It returns `is`.

Prototype

```
template<class T> istream& operator<<(ostream& os,
  complex<T>& x);
```

Value Operations complex

complex::real

Description

This function returns the real part of `x`.

Prototype

```
template<class T> T real(const complex<T>& x);
```

complex::imag

Description

This function returns the imaginary part of `x`.

Prototype

```
template<class T> T imag(const complex<T>& x);
```

complex::arg

Description

This function returns the argument of the complex number `x`.

Prototype
```
template<class T> T arg(const complex<T>& x);
```

complex::norm

Description This function returns the squared magnitude of `x`.

Prototype
```
template<class T> T norm(const complex<T>& x);
```

complex::conj

Description This function returns the conjugate of the complex number `x`.

Prototype
```
template<class T> T conj(const complex<T>& x);
```

complex::polar

Description This function returns the complex value corresponding to a complex number whose magnitude is `rho` and whose phase angle is `theta`.

Prototype
```
template<class T> T polar(T rho, const t& theta);
```

Transcendentals complex

This contains the trigonometric functions like `acos`, `asin`, `atan`, `atan2` etc. These functions return complex value corresponding to the mathematical function computed for complex arguments.

Numeric Arrays

The header `<valarray>` defines five template classes: `valarray`, `slice_array`, `gslice_array`, `mask_array` and `indirect_`array, two classes: `slice` and `gslice`, and a series of related function signatures for representing and manipulating arrays of values.

Files
```
#include <valarray.h>
```

The array classes are:

Template Class valarray

Description

The template class `valarray<T>` is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices, together with the powerful subsetting capabilities provided by the generalized subscript operators.

The topics in this section are:

Constructors valarray

Default Constructor

Description This constructs an object of class `valarray<T>`, which has zero length until it is passed into a library function as a modifiable lvalue or through a non-constant this pointer.

Prototype `valarray();`

Overloaded Constructors

Prototype `valarray (size_t);`

Remarks This constructor sets the array length equal to the value of the argument. The elements of the array are constructed using the default constructor for the instantiating type `T`.

`valarray (const T&, size_t);`

Remarks This constructor sets the array equal to the value of the second argument and initializes all the elements with the value of the first argument.

`valarray (const T*, size_t);`

Remarks The array created by this constructor has a length equal to the second argument. The values of the elements of the array are initialized with the first n values pointed to by the first argument. It is necessary that the value of the second argument is greater than the number of vales pointed to by the first argument.

Copy Constructor

Description Copy constructor creates a distinct array rather than a alias.

Prototype `valarray (const valarray<T>&);`

Conversion Constructors

Description These are conversion constructors which convert one of the four reference templates to `valarray`.

Prototype

```
valarray (const slice_array<T>&);
valarray (const gslice_array<T>&);
valarray (const mask_array<T>&);
valarray (const indirect_array<T>&);
```

Destructor

Description Destructor for `valarray`.

Prototype

```
~valarray ();
```

Assignment Operators valarray

Assignment Operator =

Description The assignment operator modifies the length of the `*this` array to be equal to that of the argument array. Each element of `*this` array is then assigned the value of the corresponding element of the argument array. Assignment is the usual way to change the length of an array after initialization. Assignment results in a distinct array rather than an alias.

Prototype

```
valarray<T>& operator= (const valarray<T>&);
```

Overloaded Assignment Operators

Prototype

```
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>&);
```

Remarks These operators allow the results of a generalized subscripting operation to be assigned directly to a `valarray`.

Element Access valarray

Subscript Operator []

Description For const valarray objects, the subscript operator returns the value of the corresponding element of the array. For non-const valarray objects, a reference to the corresponding element of the array is returned.

Prototype

```
T operator[] (size_t) const;
T& operator[] (size_t);
```

Subset Operations valarray

Subscript Operator []

Description Each of these operations returns a subset of the array. The `const`-qualified versions return this subset as a new `valarray`. The non-`const` versions return a class template object which has reference semantics to the original array.

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>&);
```

Unary Operators valarray

Description Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `&T` or which may be unambiguously converted to type `T`. Each of these operators returns an array whose length is equal to the length of the array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

Add Operator +

Prototype

```
valarray<T> operator+() const;
```

Minus Operator -

Prototype

```
valarray<T> operator-() const;
```

Complement Operator ~

Prototype

```
valarray<T> operator~() const;
```

Not Operator !

Prototype

```
valarray<T> operator!() const;
```

Computed Assignment Type valarray<T>

Description Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array. The array is then returned by reference. If the array and the argument array do not have the same length, the behavior is undefined.   The appearance of an array on the left hand side of a computed assignment does not invalidate references or pointers.

Multiply & Assign Operator *=

Prototype
```
valarray<T>& operator*= (const valarray<T>&);
```

Divide and Assign Operator /=

Prototype
```
valarray<T>& operator/= (const valarray<T>&);
```

Remainder & Assign Operator %=

Prototype
```
valarray<T>& operator%= (const valarray<T>&);
```

Add & Assign Operator +=

Prototype
```
valarray<T>& operator+= (const valarray<T>&);
```

Minus and Assign Operator -=

Prototype
```
valarray<T>& operator-= (const valarray<T>&);
```

XOR and Assign Operator ^=

Prototype
```
valarray<T>& operator^= (const valarray<T>&);
```

And & Assign Operator &=

Prototype
```
valarray<T>& operator&= (const valarray<T>&);
```

Or & Assign Operator |=

Prototype
```
valarray<T>& operator|= (const valarray<T>&);
```

Not Equal Operator !=

Prototype

```
valarray<T>& operator!= (const valarray<T>&);
```

Right Shift & Assign Operator >>=

Prototype

```
valarray<T>& operator>>=(const valarray<T>&);
```

Computed Assignment Type<T>

Description

Each of these operators may only be instantiated for a type T to which the indicated operator can be applied. Each of these operators applies the indicated operation to each element of the array and the scalar argument. The array is then returned by reference. The appearance of an array on the left hand side of a computed assignment does not invalidate references or pointers to the elements of the array.

Multiply & Assign Operator *=

Prototype

```
valarray<T>& operator*= (const T&);
```

Divide & Assign Operator /=

Prototype

```
valarray<T>& operator/= (const T&);
```

Remainder & Assign Operator %=

Prototype

```
valarray<T>& operator%= (const T&);
```

Add & Assign Operator +=

Prototype

```
valarray<T>& operator+= (const T&);
```

Minus & Assign Operator -=

Prototype

```
valarray<T>& operator-= (const T&);
```

XOR & Assign Operator ^=

Prototype

```
valarray<T>& operator^= (const T&);
```

And & Assign Operator &=

Prototype

```
valarray<T>& operator&= (const T&);
```

Not Equal Operator !=

Prototype

```
valarray<T>& operator|= (const T&);
```

Right Shift & Assign Operator >>=

Prototype

```
valarray<T>& operator>>=(const T&);
```

Non-Member Binary Operators valarray

Description

Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously converted to type `T`. Each of these operators returns an array whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays. If the argument arrays do not have the same length, the behavior is undefined.

Multiply Operator *

Prototype

```
template<class T> valarray<T> operator*
```

```
  (const valarray<T>&, const valarray<T>&);
```

Divide Operator /

Prototype
```
template<class T> valarray<T> operator/
  (const valarray<T>&, const valarray<T>&);
```

Remainder Operator %

Prototype
```
template<class T> valarray<T> operator%
  (const valarray<T>&, const valarray<T>&);
```

Add Operator +

Prototype
```
template<class T> valarray<T> operator+
  (const valarray<T>&, const valarray<T>&);
```

Minus Operator -

Prototype
```
template<class T> valarray<T> operator-
  (const valarray<T>&, const valarray<T>&);
```

Bitwise XOR Operator ^

Prototype
```
template<class T> valarray<T> operator^
  (const valarray<T>&, const valarray<T>&);
```

Bitwise And Operator &

Prototype
```
template<class T> valarray<T> operator&
  (const valarray<T>&, const valarray<T>&);
```

Bitwise Or Operator

Prototype
```
template<class T> valarray<T> operator|
```

```
  (const valarray<T>&, const valarray<T>&);
```

Left Shift Operator <<

Prototype

```
template<class T> valarray<T>operator<<
  (const valarray<T>&, const valarray<T>&);
```

Right Shift Operator >>

Prototype

```
template<class T> valarray<T> operator>>
  (const valarray<T>&, const valarray<T>&);
```

Logical And Operator &&

Prototype

```
template<class T> valarray<T> operator&&
  (const valarray<T>&, const valarray<T>&);
```

Logical Or Operator ||

Prototype

```
template<class T> valarray<T> operator||
  (const valarray<T>&, const valarray<T>&);
```

Overloaded Binary Operators valarray

Description

Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously converted to type `T`. Each of these operators returns an array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the scalar argument.

Multiply Operator *

Prototype

```
template<class T> valarray<T> operator*
```

```
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator*
  (const T&, const valarray<T>&);
```

Divide Operator /

Prototype
```
template<class T> valarray<T> operator/
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator/
  (const T&, const valarray<T>&);
```

Remainder Operator %

Prototype
```
template<class T> valarray<T >operator%
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator%
  (const T&, const valarray<T>&);
```

Add Operator +

Prototype
```
template<class T> valarray<T> operator+
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator+
  (const T&, const valarray<T>&);
```

Minus Operator -

Prototype
```
template<class T> valarray<T> operator-
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator-
  (const T&, const valarray<T>&);
```

Bitwise XOR Operator ^

Prototype
```
template<class T> valarray<T> operator^
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator^
  (const T&, const valarray<T>&);
```

Bitwise And Operator &

Prototype
```
template<class T> valarray<T> operator&
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator&
  (const T&, const valarray<T>&);
```

Bitwise Or Operator |

Prototype
```
template<class T> valarray<T> operator|
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator|
  (const T&, const valarray<T>&);
```

Left Shift Operator <<

Prototype
```
template<class T> valarray<T> operator<<
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> operator<<
  (const T&, const valarray<T>&);
```

Right Shift Operator >>

Prototype
```
template<class T> valarray<T> operator>>
  (const valarray<T>&, const T&);
```

Prototype

```
template<class T> valarray<T> operator>>
  (const T&, const valarray<T>&);
```

Logical And Operator &&

Prototype

```
template<class T> valarray<T> operator&&
  (const valarray<T>&, const T&);
```

Prototype

```
template<class T> valarray<T> operator&&
  (const T&, const valarray<T>&);
```

Logical Or Operator ||

Prototype

```
template<class T> valarray<T> operator||
  (const valarray<T>&, const T&);
```

Prototype

```
template<class T> valarray<T> operator||
  (const T&, const valarray<T>&);
```

Comparison Operators valarray

Description

Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously converted to type bool. Each of these operators returns a bool array whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays. If the two array arguments do not have the same length, the behavior is undefined.

Equality Operator ==

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>&);
```

Less Than Operator <

Prototype

```
template<class T> valarray<bool> operator<
  (const valarray<T>&, const valarray<T>&);
```

Greater Than Operator >

Prototype

```
template<class T> valarray<bool> operator>
  (const valarray<T>&, const valarray<T>&);
```

Not Equal Operator !=

Prototype

```
template<class T> valarray<bool> operator!=
  (const valarray<T>&, const valarray<T>&);
```

Greater Than or Equal Operator >=

Prototype

```
template<class T> valarray<bool> operator>=
  (const valarray<T>&, const valarray<T>&);
```

Overloaded Comparison Operators valarray

Description

Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously converted to type bool. Each of these operators returns a bool array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array and the scalar argument.

Equality Operator ==

Prototype
```
template<class T> valarray<bool> operator==
  (const valarray&, const T&);
```

Prototype
```
template<class T> valarray<bool> operator==
  (const T&, const valarray&);
```

Not Equal Operator !=

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

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

Less Than Operator <

Prototype
```
template<class T> valarray<bool> operator<
  (const valarray&, const T&);
```

Prototype
```
template<class T> valarray<bool> operator<
  (const T&, const valarray&);
```

Greater Than Operator >

Prototype
```
template<class T> valarray<bool> operator>
  (const valarray&, const T&);
```

Prototype
```
template<class T> valarray<bool> operator>
  (const T&, const valarray&);
```

Less Than or Equal Operator <=

Prototype
```
template<class T> valarray<bool> operator<=
  (const valarray&, const T&);
```

Prototype

```
template<class T> valarray<bool> operator<=
  (const T&, const valarray&);
```

Greater Than or Equal Operator >=

Prototype

```
template<class T> valarray<bool> operator>=
  (const valarray&, const T&);
```

Prototype

```
template<class T> valarray<bool> operator>=
  (const T&, const valarray&);
```

Member Functions valarray

valarray::length

Description

This function returns the number of elements in the array.

Prototype

```
size_t length() const;
```

Pointer Conversion

Description

A non-constant array may be converted to a pointer to the instantiating type. A constant array may be converted to a pointer to the instantiating type, qualified by const. It is guaranteed that &a[0] == (`T*`)a for any non-constant valarray<T> a. The pointer returned for a non-constant array (whether or not it points to a type qualified by const) is valid for the same duration as a reference returned by the size_t subscript operator. The pointer returned for a constant array is valid for the lifetime of the array.

Prototype

```
operator T*();
operator const T*() const;
```

valarray::sum

Description

This function may only be instantiated for a type `T` to which operator+= can be applied. This function returns the sum of all the elements of the array. If the array has length 0, the behavior is undefined. If the array has length 1, sum returns the value of element 0. Otherwise, the returned value is calculated by applying operator+= to a copy of an element of the array and all other elements of the array in an unspecified order.

Prototype

```
T sum() const;
```

valarray::fill

Description

This function assigns the value of the argument to all the elements of the array. The length of the array is not changed, nor are any pointers or references to the elements of the array invalidated.

Prototype

```
void fill (const T&);
```

valarray::shift

Prototype

```
valarray<T> shift(int) const;
```

Description

This function returns an array whose length is identical to the array, but whose element values are shifted the number of places indicated by the argument. A positive argument value results in a left shift, a negative value in a right shift, and a zero value in no shift.

valarray::cshift

Description

This function returns an array whose length is identical to the array, but whose element values are shifted in a circular fashion the number of places indicated by the argument. A positive argument value results in a left shift, a negative value in a right shift, and a zero value in no shift.

Prototype
```
valarray<T> cshift(int) const;
```

valarray::apply

Description These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.

Prototype
```
valarray<T> apply(T func(T)) const;
```

Prototype
```
valarray<T> apply(T func(const T&)) const;
```

valarray::free

Description This function sets the length of an array to zero.

Prototype
```
void free();
```

Min And Max Functions valarray

valarray::min

Description This function may only be instantiated for a type `T` to which operator> and operator< may be applied and for which operator> and operator< return a value which is of type bool or which can be unambiguously converted to type bool. This function returns the minimum (`a.min()`) value found in the argument array a. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator>` and `operator<`, in a manner analogous to the application of `operator+=` for the sum function.

Prototype
```
template<class T> T min(const valarray<T>& a);
```

valarray::max

Description

This function may only be instantiated for a type `T` to which operator> and operator< may be applied and for which operator> and operator< return a value which is of type bool or which can be unambiguously converted to type bool. These functions return the maximum (`a.max()`) value found in the argument array a. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator>` and `operator<`, in a manner analogous to the application of `operator+=` for the sum function.

Prototype

```
template<class T> T max(const valarray<T>& a);
```

Transcendentals valarray

Description

Each of these functions may only be instantiated for a type `T` to which a unique function with the indicated name can be applied. This function must return a value which is of type `T` or which can be unambiguously converted to type `T`.

valarray::abs

Prototype

```
template<class T> valarray<T> abs
  (const valarray<T>&);
```

valarray::acos

Prototype

```
template<class T> valarray<T> acos
  (const valarray<T>&);
```

valarray::asin

Prototype

```
template<class T> valarray<T> asin
  (const valarray<T>&);
```

valarray::atan

Prototype

```
template<class T> valarray<T> atan
  (const valarray<T>&);
```

valarray::atan2

Prototype

```
template<class T> valarray<T> atan2
  (const valarray<T>&, const valarray<T>&);
```

Prototype

```
template<class T> valarray<T> atan2
  (const valarray<T>&, const T&);
```

Prototype

```
template<class T> valarray<T> atan2
  (const T&, const valarray<T>&);
```

valarray::cos

Prototype

```
template<class T> valarray<T> cos
  (const valarray<T>&);
```

valarray::cosh

Prototype

```
template<class T> valarray<T> cosh
  (const valarray<T>&);
```

valarray::exp

Prototype

```
template<class T> valarray<T> exp
  (const valarray<T>&);
```

valarray::log

Prototype

```
template<class T> valarray<T> log
  (const valarray<T>&);
```

valarray::log10

Prototype
```
template<class T> valarray<T> log10
  (const valarray<T>&);
```

valarray::pow

Prototype
```
template<class T> valarray<T> pow
  (const valarray<T>&, const valarray<T>&);
```

Prototype
```
template<class T> valarray<T> pow
  (const valarray<T>&, const T&);
```

Prototype
```
template<class T> valarray<T> pow
  (const T&, const valarray<T>&);
```

valarray::sin

Prototype
```
template<class T> valarray<T> sin
  (const valarray<T>&);
```

valarray::sinh

Prototype
```
template<class T> valarray<T> sinh
  (const valarray<T>&);
```

valarray::sqrt

Prototype
```
template<class T> valarray<T> sqrt
  (const valarray<T>&);
```

valarray::tan

Prototype
```
template<class T> valarray<T> tan
  (const valarray<T>&);
```

valarray::tanh

Prototype

```
template<class T> valarray<T> tanh
  (const valarray<T>&);
```

Class slice

Description

The slice class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.

Prototype

```
class slice {
  public:
    slice();
    slice(size_t, size_t, size_t);

    size_t start() const;
    size_t length() const;
    size_t stride() const;
};
```

The topics in this section are:

- “Constructors slice” on page 376
- “Access Functions slice” on page 377

Constructors slice

Default Constructor

Description

The default constructor for slice creates a slice which specifies no elements. A default constructor is provided only to permit the declaration of arrays of slices.

Prototype

```
slice();
```

Overloaded and Copy Constructors

Description

The constructor with arguments for a slice takes a start, length, and stride parameter.

Prototype

```
slice(size_t start, size_t length, size_t stride);
slice(const slice&);
```

Access Functions slice

Description

These functions return the start, length, or stride specified by a `slice` object.

slice::start

Prototype

```
size_t start() const;
```

slice::length

Prototype

```
size_t length() const;
```

slice::stride

Prototype

```
size_t stride() const;
```

Template class slice_array

Description

The `slice_array` template is a helper template used by the slice subscript operator `slice_array<T> valarray<T>::operator[](slice)`; It has reference semantics to a subset of an array specified by a `slice` object.

The topics in this section are:

- "Constructors slice_array" on page 378

- “Assignment Operators slice_array” on page 378
- “Computed Assignment slice_array” on page 378
- “Public Member Function slice_array” on page 380—the `fill` function

Constructors slice_array

Default and Copy Constructor

Description The `slice_array` template has no public constructors. These constructors are declared to be private. These constructors need not be defined.

Prototype

```
slice_array();
```

Prototype

```
slice_array(const slice_array&);
```

Assignment Operators slice_array

Assignment Operator =

Description The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `slice_array` object refers.

Prototype

```
void operator=(const valarray<T>&) const;
slice_array& operator=(const slice_array&);
```

Computed Assignment slice_array

Description These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `slice_array` object refers.

Multiply & Assign Operator *=

Prototype
```
void operator*= (const valarray<T>&) const;
```

Divide & Assign Operator /=

Prototype
```
void operator/= (const valarray<T>&) const;
```

Remainder & Assign Operator %=

Prototype
```
void operator%= (const valarray<T>&) const;
```

Add & Assign Operator +=

Prototype
```
void operator+= (const valarray<T>&) const;
```

Minus & Assign Operator -=

Prototype
```
void operator-= (const valarray<T>&) const;
```

XOR & Assign Operator ^=

Prototype
```
void operator^= (const valarray<T>&) const;
```

And & Assign Operator &=

Prototype
```
void operator&= (const valarray<T>&) const;
```

Or & Assign Operator |=

Prototype
```
void operator|= (const valarray<T>&) const;
```

Left Shift & Assign Operator <<=

Prototype

```
void operator<<=(const valarray<T>&) const;
```

Right shift & Assign Operator >>=

Prototype

```
void operator>>=(const valarray<T>&) const;
```

Public Member Function slice_array

slice_array::fill

Description

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the slice_array object refers.

Prototype

```
void fill(const T&);
```

Class gslice

Description

This class represents a generalized slice out of an array. A gslice is defined by a starting offset (s), a set of lengths (l_j), and a set of strides (d_j). The number of lengths must equal the number of strides. A gslice represents a mapping from a set of indices (i_j), equal in number to the number of strides, to a single index k. It is useful for building multidimensional array classes using the valarray template, which is one-dimensional. The set of one-dimensional index values specified by a gslice are $k = s + \Sigma i_j d_j$ where the multidimensional indices i_j range in value from 0 to l_{ij}-1.It is possible to have degenerate generalized slices in which an address is repeated. If a degenerate slice is used as the argument to the non-const version of `operator[](const gslice&)`, the resulting behavior is undefined.

The topics in this section are:

- “Constructors gslice” on page 381

- “Access Functions gslice” on page 381

Constructors gslice

Default Constructor

Description The default constructor creates a gslice which specifies no elements.

Prototype

```
gslice ();
```

Overloaded Constructors

Description The constructor with arguments builds a gslice based on a specification of start, lengths, and strides, as explained in the previous section.

Prototype

```
gslice (size_t start,
    const valarray<size_t>& lengths,
    const valarray<size_t>& strides);
gslice (const gslice&);
```

Access Functions gslice

Description These access functions return the representation of the start, lengths, or strides specified for the gslice.

gslice::start

Prototype

```
size_t start() const;
```

gslice::length

Prototype

```
valarray<size_t> length() const;
```

gslice::stride

Prototype `valarray<size_t> stride() const;`

Template Class gslice_array

Description This template is a helper template used by the slice subscript operator `gslice_array<T> valarray<T>::operator[](const gslice&);` It has reference semantics to a subset of an array specified by a gslice object. Thus, the expression `a[gslice(1, length, stride)] = b` has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.

The topics in this section are:

- “Constructors gslice_array” on page 382
- “Assignment gslice_array” on page 383
- “Computed Assignment gslice_array” on page 383
- “Public Member Function gslice_array” on page 384—the `fill` function

Constructors gslice_array

Default and Copy Constructors

Description The `gslice_array` template has no public constructors. It declares the constructor to be private.

Prototype `gslice_array();`

Prototype `gslice_array(const gslice_array&);`

Assignment gslice_array

Assignment Operator =

Description

The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `gslice_array` refers.

Prototype

```
void operator=(const valarray<T>&) const;
gslice_array& operator=(const gslice_array&);
```

Computed Assignment gslice_array

Description

These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `gslice_array` object refers.

Multiply & Assign Operator *=

Prototype

```
void operator*= (const valarray<T>&) const;
```

Divide & Assign Operator /=

Prototype

```
void operator/= (const valarray<T>&) const;
```

Remainder & Assign Operator %=

Prototype

```
void operator%= (const valarray<T>&) const;
```

Add & Assign Operator +=

Prototype

```
void operator+= (const valarray<T>&) const;
```

Minus & Assign Operator -=

Prototype

```
void operator-= (const valarray<T>&) const;
```

XOR & Assign Operator ^=

Prototype

```
void operator^= (const valarray<T>&) const;
```

And & Assign Operator &=

Prototype

```
void operator&= (const valarray<T>&) const;
```

Or & Assign Operator |=

Prototype

```
void operator|= (const valarray<T>&) const;
```

Left Shift & Assign Operator <<=

Prototype

```
void operator<<=(const valarray<T>&) const;
```

Right Shift & Assign Operator >>=

Prototype

```
void operator>>=(const valarray<T>&) const;
```

Public Member Function gslice_array

gslice_array::fill

Description

This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `gslice_array` object refers.

Prototype

```
void fill(const T&);
```

Template Class mask_array

Description

This template is a helper template used by the mask subscript `operator: mask_array<T> valarray<T>::operator[](const valarray<bool>&)`. It has reference semantics to a subset of an array specified by a `boolean mask`. Thus, the expression `a[mask] = b;` has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is true.

The topics in this section are:

- "Constructors mask_array" on page 385
- "Assignment mask_array" on page 385
- "Computed Assignment mask_array" on page 386
- "Public Member Function mask_array" on page 387—the `fill` function

Constructors mask_array

Constructors

Description

The `mask_array` template has no public constructors. It declares the above constructors to be private. These constructors need not be defined.

Prototype

```
mask_array();
```

Prototype

```
mask_array(const mask_array&);
```

Assignment mask_array

Assignment Operator =

Description

The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

Prototype

```
void operator=(const valarray<T>&) const;
mask_array& operator=(const mask_array&);
```

Computed Assignment mask_array

Description

These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the mask object refers.

Multiply & Assign Operator *=

Prototype

```
void operator*= (const valarray<T>&) const;
```

Divide & Assign Operator /=

Prototype

```
void operator/= (const valarray<T>&) const;
```

Remainder & Assign Operator %=

Prototype

```
void operator%= (const valarray<T>&) const;
```

Add & Assign Operator +=

Prototype

```
void operator+= (const valarray<T>&) const;
```

Minus & Assign Operator -=

Prototype

```
void operator-= (const valarray<T>&) const;
```

XOR & Assign Operator ^=

Prototype

```
void operator^= (const valarray<T>&) const;
```

And & Assign Operator &=

Prototype

```
void operator&= (const valarray<T>&) const;
```

Or & Assign Operator |=

Prototype

```
void operator|= (const valarray<T>&) const;
```

Left Shift & Assign Operator <<=

Prototype

```
void operator<<=(const valarray<T>&) const;
```

Right Shift & Assign Operator >>=

Prototype

```
void operator>>=(const valarray<T>&) const;
```

Public Member Function mask_array

mask_array::fill

Prototype

```
void fill(const T&);
```

Description This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the mask_array object refers.

Template Class indirect_array

Description This template is a helper template used by the indirect subscript operator indirect_array<T> valarray<T>::opera-

`tor[](const valarray<int>&)`. It has reference semantics to a subset of an array specified by an indirect_array. Thus the expression `a[indirect] = b`; has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in indirect.

The topics in this section are:

- "Constructors indirect_array" on page 388
- "Assignment indirect_array" on page 388
- "Computed Assignment indirect_array" on page 389
- "Public Member Function indirect_array" on page 390—the `fill` function

Constructors indirect_array

Default and Copy Constructors

Description The `indirect_array` template has no public constructors. The constructors listed above are private. These constructors need not be defined.

Prototype `indirect_array();`

Prototype `indirect_array(const indirect_array&);`

Assignment indirect_array

Assignment Operator =

Description The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers. If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

Prototype `void operator=(const valarray<T>&) const;`

```
indirect_array& operator=(const indirect_array&);
```

Computed Assignment indirect_array

Description

These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `indirect_array` object refers. If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

Multiply & Assign Operator *=

Prototype

```
void operator*= (const valarray<T>&) const;
```

Divide & Assign Operator /=

Prototype

```
void operator/= (const valarray<T>&) const;
```

Remainder & Assign Operator %=

Prototype

```
void operator%= (const valarray<T>&) const;
```

Add & Assign Operator +=

Prototype

```
void operator+= (const valarray<T>&) const;
```

Minus & Assign Operator -=

Prototype

```
void operator-= (const valarray<T>&) const;
```

XOR & Assign Operator ^=

Prototype

```
void operator^= (const valarray<T>&) const;
```

And & Assign Operator &=

Prototype

```
void operator&= (const valarray<T>&) const;
```

Or & Assign Operator |=

Prototype

```
void operator|= (const valarray<T>&) const;
```

Left Shift & Assign Operator <<=

Prototype

```
void operator<<=(const valarray<T>&) const;
```

Right Shift & Assign Operator >>=

Prototype

```
void operator>>=(const valarray<T>&) const;
```

Public Member Function indirect_array

indirect_array::fill

Description

This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `indirect_array` object refers.

Prototype

```
void fill(const T&);
```

Generalized Numeric Operations

The classes are:

- "Template Class accumulate" on page 391
- "Template Class inner_product" on page 391
- "Template Class partial_sum" on page 392
- "Template Class adjacent_difference" on page 393

See also “Generalized Numeric Algorithms” on page 341.

Template Class accumulate

Description

Initializes the accumulator `acc` with the initial value init and then modifies it with `acc = acc + *i or acc = binary_op(acc, *i)` for every iterator i in the range `[first, last)` in order.This function requires that `binary_op` shall not cause side effects.

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);
```

Template Class inner_product

Description

Computes its result by initializing the accumulator acc with the initial value init and then modifying it with `acc = acc + (*i1) * (*i2)` or `acc = binary_op1(acc, binary_op2(*i1, *i2))` for every iterator i1 in the range `[first, last)` and iterator `i2` in the range `[first2, first2 + (last - first))` in order. This function requires that `binary_op1` and `binary_op2` shall not cause side effects.

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);
```

Template Class partial_sum

Description

Assigns to every iterator i in the range `[result, result + (last - first))` a value correspondingly equal to `((...(*first + *(first + 1)) +...) + *(first + (i - result)))` or `binary_op(binary_op(..., binary_op(*first, *(first + 1)),...), *(first + (i - result)))`. This function returns `result + (last - first)`. The complexity of this function is exactly `(last - first) - 1` applications of `binary_op`. This function requires `binary_op` is expected not to have any side effects. The result may be equal to `first`.

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);
```

Template Class adjacent_difference

Description

Assigns to every element referred to by iterator `i` in the range `[result + 1, result + (last - first))` a value correspondingly equal to `*(first + (i - result)) - *(first + (i - result) - 1)` or `binary_op(*(first + (i - result)), *(first + (i - result) - 1))`. Result gets the value of `*first`. This function requires binary_op shall not have any side effects. Result may be equal to `first`. This function returns `result + (last - first)`. The complexity of this function is exactly `(last - first) - 1` applications of `binary_op`.

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);
```

12

27.1 Input and Output Library

A listing of the set of components that C++ programs may use to perform input/output operations.

Overview of Input and Output Library

The sections in this chapter are:

Input and Output Library Summary

This library includes the headers.

Table 12.1 Input/Output Library Summary

Include	Purpose
<iosfwd>	Forward declarations
<iostream>	Standard iostream objects
<ios>	Iostream base classes
<streambuf>	Stream buffers
<istream>	Formatting and manipulators
<ostream>	
<iomanip>	
<sstream>	String streams

Include	Purpose
`<cstdlib>`	
`<fstream>`	`File Streams`
`<cstdio>`	
`<cwchar>`	

27.1 Iostreams requirements

No requirements library has been defined.

Topics in this section are:

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` - A 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 pre-

viously 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. I is used to represent the number of characters transferred in and input/output operation or for the size of the input/output buffers.

13

27.2 Forward Declarations

The header `<iosfwd>` is used for forward declarations of template classes.

Header <iosfwd>

Prototype

```
namespace std {
  template<class charT> class basic_ios;
  template<class charT> class basic_istream;
  template<class charT> class basic_ostream;

typedef basic_ios<char> ios;
typedef basic_ios<wchar> wios;

typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> istream;

typedef basic_ostream<char> ostream;
typedef basic_ostream<char> wostream;
}
```

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`.

14

27.3 Standard 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>`.

Header <iostream>

Description 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;
extern wostream wclog;
}
```

Additional topics are:

- “27.3.1 Narrow stream objects” on page 402
- “27.3.2 Wide stream objects” on page 403

27.3.1 Narrow stream objects

Description Narrow stream objects provide unbuffered input and output associated with standard input and output declared in <cstdio>.

istream cin

Description 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

Description 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

Description 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

Description 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

Description Narrow stream objects provide unbuffered input and ouput associated with standard input and output declared in <cstdio>.

istream win

Description An unbuffered input stream.

Prototype

```
wistream win;
```

Remarks The object `win` controls input from an unbuffered stream buffer associated with `stdin declared in <cstdio>`. After cin is initialized `win.tie()` returns `wout`.

Return An wistream object;

ostream wout

Description An unbuffered output stream.

Prototype `wostream wout;`

Remarks The object `cout` controls output to an unbuffered stream buffer associated with `stdout declared in <cstdio>`.

Return An wostream object;

wostream werr

Description Controls `output` to an unbuffered stream.

Prototype `wostream werr;`

Remarks The object `werr` controls output to an unbuffered stream buffer associated with `stderr` declared in `<cstdio>`. After werr is initialized, werr.flags() and unitbuf is nonzero.

Return An `ostream` object;

wostream wlog

Description Controls output to a stream buffer.

Prototype `wostream wlog;`

Remarks The object `wlog` controls output to a stream buffer associated with cerr declared in `<cstdio>`.

Return An `ostream` object

15

27.4 Iostreams Base Classes

The include header `<ios>` contains the basic class definitions, types, and enumerations necessary for input and output stream reading writing and other manipulations.

Overview of iostream base classes

The sections in this chapter are:

Header <ios>

Description

The header file <ios> provides for implementation of stream objects for standard input and output.

Prototype

```
Header <ios>

typedef OFF_T streamoff;
typedef SZ_T streamsize;

class ios_base;
template <class charT, class traits =
ios_traits<charT> >
```

```
class basic_ios

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)
```

27.4.1 Typedef Declarations

Description The following typedef's are defined in the class `ios_base`.

Definition `typedef OFF_T wstreamoff;`

Definition `typedef POS_T wstreampos;`

Definition `typedef SZ_T streamsize;`

27.4.3 Class ios_base

Description A base template 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 411
- “27.4.3.1.1 Class ios_base::failure” on page 412
- “27.4.3.1.1.1 failure” on page 412
- “27.4.3.1.2 Type ios_base::fmtflags” on page 413
- “27.4.3.1.3 Type ios_base::iostate” on page 414
- “27.4.3.1.4 Type ios_base::openmode” on page 414
- “27.4.3.1.5 Type ios_base::seekdir” on page 415
- “27.4.3.1.6 Class ios_base::Init” on page 415
- “Class ios_base::Init Constructor” on page 416
- “27.4.3.2 ios_base fmtflags state functions” on page 416
- “27.4.3.3 ios_base locale functions” on page 424
- “27.4.3.4 ios_base storage function” on page 424
- “27.4.3.5 ios_base Constructor” on page 426

Prototype

```
namespace std{
class ios_base{
public:
  class failure;

  typedef T1 fmtflags;
  static const formatflags boolalpha;
  static const formatflags dec;
```

```
static const formatflags fixed;
static const formatflags hex;
static const formatflags internal;
static const formatflags left;
static const formatflags oct;
static const formatflags right;
static const formatflags scientific;
static const formatflags showbase;
static const formatflags showpos;
static const formatflags skipws;
static const formatflags unitbuf;
static const formatflags uppercase;
static const formatflags adjustfield;
static const formatflags basefield;
static const formatflags 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;

iostate exceptions() const;
void exceptions (iostate except);
```

```
  fmtflags flags() const;
  fmtflags flags (fmtflags f);
  fmtflags setf (fmtflags f);
  fmtflags setf (fmtflags f, fmtflags mask);
  void unsetf (fmtflags mask);

  streamsize precision () const;
  streamsize precision (streamsize prec);
  streamsize width () const;
  streamsize width (streamsize w);

  locale imbue (const locale &loc);
  locale getloc () const;

  static int xalloc ();
  long& iword (int index);
  void* & pword (int index);
  void register_callback(event_callback, int);

  ~ios_base();

  enum event{ erase_event, imbue_event,
    copyfmt_event};
  typedef void(*event_callback)(event, ios_base&,
    int index);
  void register_callback(event_call_back fn,
    int index);

  protected:
  ios_base ();
  };
}
```

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 Class ios_base::failure

Description 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

Description Construct a class failure.

Prototype

```
explicit failure(const string& msg);
```

Remarks The function `failure() construct a class failure initializing with exception(msg).`

failure::what

Description To return the exception message.

Prototype

```
const char *what() const;
```

Remarks The function `what()` is use to deliver the msg.str().

Returns Returns the message with which the exception was created.

27.4.3.1.2 Type ios_base::fmtflags

An enumeration used to set various formatting flags for reading and writing of streams.

Table 15.1 Format Flags Enumerations

Flag	Effects when set
`boolalpha`	insert and extract bool type in alphabetic form
`dec`	decimal output
`fixed`	when set show floating point numbers in normal manner by default that is six decimal places
`hex`	hexadecimal output
`left`	left justified
`oct`	octal output
`right`	right justified
`scientific`	show scientific notation for floating point numbers
`showbase`	show the bases numeric values
`showpoint`	show the decimal point and trailing zeros
`showpos`	show 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
Constants	**Allowable values**
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

Listing 15.1 Example of `ios` format flags usage

```
see basic_ios::setf() and basic_ios::unsetf()
```

27.4.3.1.3 Type ios_base::iostate

An enumeration that is used to define the various states of a stream.

Table 15.2 Enumeration iostate

Flags	Usage
`badbit`	iostate improper read/write
`failbit`	iostate failure
`eofbit`	end of file bit set note: see variance from AT&T Standards

Listing 15.2 Example of ios iostate flags usage:

```
See basic_ios::setstate() and basic_ios::rdstate()
```

27.4.3.1.4 Type ios_base::openmode

An enumeration that is used to specify various file opening modes.

Table 15.3 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 file

Mode	Definition
`out`	Start the write at the beginning of the file
`trunc`	Start the read or write at the beginning of the file

27.4.3.1.5 Type ios_base::seekdir

An enumeration to position a pointer to a specific place in a file stream.

Table 15.4 Enumeration seekdir

Enumeration	Position
`beg`	Begging of stream
`cur`	Current position of stream
`end`	End of stream

Listing 15.3 Example of ios seekdir usage:

See: `streambuf::pubseekoff`

27.4.3.1.6 Class ios_base::Init

Description 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 ios_base::Init Constructor

Default Constructor

Description 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

Description To set the state of the ios_base format flags.

ios_base::flags

Description 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.

Returns The `fmtflags` type before alterations.

NOTE: See `ios` enumerators for a list of `fmtflags`.

See Also: `setiosflags()` and `resetiosflags()`

Listing 15.4 Example of flags() usage:

```
#include <iostream>

  // showf() displays flag settings
void showf();

main()
{
  showf(); // show format flags

  cout << "press enter to continue" << endl;
  cin.get();

  cout.setf(ios::right|ios::showpoint|ios::fixed);
  showf();
  return 0;
}

// showf() displays flag settings
void showf()
{
  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
```

ios_base::setf

Description Set the stream format flags.

Prototype

```
fmtflags setf(fmtflags)
fmtflags setf(fmtflags, fmtflags)
```

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.

Returns `type basic_ios::fmtflags`

Listing 15.5 Example of setf() usage:

```
#include <iostream>

main()
{
  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:
+10.0100
10.0100
```

ios_base::unsetf

Description 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.

Returns There is no return.

Listing 15.6 Example of unsetf() usage:

```
#include <iostream>
```

```
main()
{
  double d = 10.01;

  cout.setf(ios::showpos | ios::showpoint);
  cout << d << endl;

  cout.unsetf(ios::showpoint);
  cout << d << endl;
  return 0;
}
```

```
Result:
+10.0100
+10.01
```

ios_base::precision

Description 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.

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.

Returns The current value set.

See Also `setprecision()`

Listing 15.7 Example of precision() usage:

```
#include <iostream>

extern double pi;

main()
{
  double TenPi = 10*pi;

  cout.precision(5);
  cout.setf(0, 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
```

ios_base::width

Description 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.

Returns The previous width setting is returned.

Listing 15.8 Example of ios_base::width() usage:

```
#include <iostream>

 main()
{
  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

Description Sets the locale for input output operations.

ios_base::imbue

Description 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()`.

ios_base::getloc

Description 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

Description To allocate storage pointers.

ios_base::xalloc

Description Allocation function.

Prototype `static int xalloc()`

Return `index++`.

ios_base::iword

Description 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 `irray[idx]`

ios_base::pword

Description 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].`

ios_base::register_callback

Description 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 `imbue()`, `copyfmt()` or `~ios_base()` the function `fn` is called with argument `index`. Function 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.

27.4.3.5 ios_base Constructor

Default Constructor

Description 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. It 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

Description

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 428
- "27.4.4.2 Member Functions" on page 429
- "27.4.4.3 basic_ios iostate flags functions" on page 434

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

Description 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 and 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 15.5 Conditions after init()

Member	Postcondition Value
rdbuf()	sb
tie()	zero
rdstate()	goodbit if stream buffer is not a null pointer otherwise badbit.
exceptions()	goodbit
flags()	skipws | dec
width()	zero
precision()	six
fill()	the space character
getloc()	locale::classic()
index	undefined
iarray	a null pointer
parray	a null pointer

27.4.4.2 Member Functions

basic_ios::tie

Description To tie an `ostream` to the calling stream.

Prototype

```
basic_ostream<charT, traits>* tie() const;
basic_ostream<charT, traits>*
 tie(basic_ostream<charT, traits>* tiestr);
```

Remarks

Any stream can have an `ostream` tied to it to ensure that the `ostream` is flushed before any operation. The standard input and output objects `cin` and `cout` are tied to ensure that `cout` is flushed before any `cin` operation. The function `tie()` is overloaded the parameterless version returns the current ostream that is tied if any. The `tie()` function with an argument ties the new object to the `ostream` and returns a pointer if any from the first. The postcondition of `tie()` function that takes the argument `tiestr` is that `tiestr` is equal to `tie()`;

Returns

A pointer to type `ostream` that is or previously was tied, or zero if there was none.

Listing 15.9 Example of basic_ios::tie() usage:

```
The file MW Reference contains
Metrowerks CodeWarrior "Software at Work"
```

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

char inFile[] = "MW Reference";

void main()
{
  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();
}
```

```
Result:
The streams are not tied
The streams are tied

The file MW Reference now contains
Metrowerks CodeWarrior "Software at Work"
Registered Trademark
```

basic_ios::rdbuf

Description 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 postcondition of `sb` is equal to `rdbuf()`.

Returns A pointer to `basic_streambuf` object.

Listing 15.10 Example of basic_ios::rdbuf() usage:

```
#include <iostream>

struct address {
  int number;
  char street[40];
} addbook;

main()
{
  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
```

basic_ios::imbue

Description Stores a value representing the locale.

Prototype `locale imbue(const locale& rhs);`

Remarks The function `imbue()` calls `ios_base::imbue()` and `rdbuf->pubimbue()`.

Returns The current locale.

basic_ios::fill

Description 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.

Returns The current character being used as a filler.

See Also `manipulator setfill()`

Listing 15.11 Example of basic_ios::fill() usage:

```
#include <iostream>

main()
{
  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 *
```

basic_ios::copyfmt

Description 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.

Returns The `this` pointer.

27.4.4.3 basic_ios iostate flags functions

Description To set flags pertaining to the state of the input and output streams.

basic_ios::operator bool

Description A `bool` operator.

Prototype `operator bool() const;`

Returns `!fail()`

basic_ios::operator !

Description A `bool not` operator.

Prototype `bool operator ! ();`

Return `fail().`

basic_ios::rdstate

Description To retrieve the state of the current formatting flags.

Prototype `iostate rdstate() const`

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 `ios::setstate()` to reset the flags at a later date.

Returns Type `iostate` used in `ios::setstate()`

See Also `ios::setstate()`

Listing 15.12 Example of basic_ios::rdstate() usage:

```
The file MW Reference contains:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char * inFile = "MW Reference";

void status(ifstream &in);
```

```
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
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
Fatal I/O Error
```

basic_ios::clear

Description 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. Postcondition of clear is the argument is equal to rdstate().

NOTE: If `rdstate()` and `exceptions() != 0` an exception is thrown.

Returns No value is returned.

Listing 15.13 Example of basic_ios::clear() usage:

```
The file MW Reference contains:
ABCDEFGH
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char * inFile = "MW Reference";

void status(ifstream &in);

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
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
GoodBit set
Non-Fatal I/O Error
```

basic_ios::setstate

Description 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.

Returns No Return

Listing 15.14 Example of basic_ios::setstate() usage:

```
See ios::rdstate()
```

basic_ios::good

Description To test for `goodbit` being set.

Prototype `bool good() const;`

Remarks Use the function `good()` to test for the setting of the `goodbit` flag.

Returns True if `rdstate() == 0`.

Listing 15.15 Example of basic_ios::good() usage:

```
See basic_ios::bad()
```

basic_ios::eof

Description To test for the end of the file.

Prototype `bool eof() const`

Remarks Use the `eof()` function to test for an end of a file is set in a stream being processed. This end of file bit is not set by stream opening.

NOTE: Variation from AT&T
The `eofbit` in streams is only set by operations whose specifications explicitly say that they do so.

Returns True if `eofbit` is set in `rdstate()`.

Listing 15.16 Example of basic_ios::eof() usage:

```
MW Reference is simply a one line text document
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

const char* TheText = "MW Reference";

main()
{
  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
```

basic_ios::fail

Description To test for stream reading failure from any cause.

Prototype `bool fail() const`

Remarks The member function `fail()` is used to test for failure of a stream for any cause. The function `fail()` replaces `eof()` as the reliable test for the end of file. It will also test for `failbit` and `badbit` unlike `eof()`, therefore it is more reliable of a test than `eof()`.

Returns True if `failbit` or `badbit` is set in `rdstate().`

Listing 15.17 Example of basic_ios::fail() usage:

```
MW Reference file for input contains.
float 33.33 double 3.16e+10 Integer 789 character C
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

main()
{
  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
```

basic_ios::bad

Description 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.

Returns True if `badbit` is set in `rdstate()`.

See Also `basic_ios::fail()`

Listing 15.18 Example of basic_ios::bad() usage:

```
The File MW Reference contains:
abcdefghijklmnopqrstuvwxyz
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char * inFile = "MW Reference";
```

```
void status(ifstream &in);

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
```

basic_ios::exceptions

Description 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 An iostate mask is returned by the non argument exceptions().

27.4.5 ios_base manipulators

Description To provide an in line input and output formatting mechanism.

The topics in this section are:

- "27.4.5.1 fmtflags manipulators" on page 445
- "27.4.5.2 adjustfield manipulators" on page 447
- "27.4.5.3 basefield manipulators" on page 447
- "27.4.5.4 floatfield manipulators" on page 448

27.4.5.1 fmtflags manipulators

Description To provide an in line input and output numerical formatting mechanism.

Table 15.6 `Prototype` of ios_base manipulators

Manipulator	Definition
ios_base& boolalpha(ios_base&)	insert and extract bool type in alphabetic format
ios_base& noboolalpha (ios_base&)	unsets insert and extract bool type in alphabetic format
ios_base& showbase(ios_base& b)	set the number base to parameter b
ios_base& noshowbase (ios_base&)	remove show base
ios_base& `showpoint(`ios_base&`)`	show decimal point
ios_base& noshowpoint(ios_base&)	do not show decimal point
ios_base& showpos(ios_base&)	show the positive sign
ios_base& noshowpos(ios_base&)	do not show positive sign
ios_base& skipws(ios_base&)	input only skip white spaces
ios_base& noskipws(ios_base&)	input only no skip white spaces
ios_base& uppercase(ios_base&)	show scientific in uppercase
ios_base& nouppercase (ios_base&)	do not show scientific in uppercase

Manipulator	Definition
ios_base& unitbuf (ios_base::unitbuf)	set the unitbuf flag
ios_base& nounitbuf (ios_base::unitbuf)	unset the unitbuf flag

Remarks Manipulators are used in the stream to alter the formatting of the stream.

Returns A reference to an object of type `ios_base` is returned to the stream. (The `this` pointer.)

27.4.5.2 adjustfield manipulators

Description To provide an in line input and output orientation formatting mechanism.

Table 15.7 Adjust filed manipulators

Manipulator	Definition
ios_base& internal(ios_base&)	fill between indicator and value
ios_base& left(ios_base&)	left justify in a field
ios_base& right(ios_base&)	right justify in a field

Remarks Manipulators are used in the stream to alter the formatting of the stream.

Returns A reference to an object of type `ios_base` is returned to the stream. (The `this pointer`.)

27.4.5.3 basefield manipulators

Description To provide an in line input and output numerical formatting mechanism.

Table 15.8 basefield manipulators

Manipulator	Definition
ios_base& dec(ios_base&)	format output data as a decimal
ios_base& oct(ios_base&)	format output data as octal
ios_base& hex(ios_base&)	format output data as hexadecimal

Remarks Manipulators are used in the stream to alter the formatting of the stream.

Returns A reference to an object of type `ios_base` is returned to the stream. (The `this pointer`.)

27.4.5.4 floatfield manipulators

Description To provide an in line input and output numerical formatting mechanism.

Table 15.9 floatfield manipulators

Manipulator	Definition
ios_base& fixed(ios_base&)	format in fixed point notation
ios_base& scientific(ios_base&)	use scientific notation

Remarks Manipulators are used in the stream to alter the formatting of the stream.

Returns A reference to an object of type `ios_base` is returned to the stream. (The `this pointer`.)

Listing 15.19 Example of manipulator usage:

```
#include <iostream>
#include <iomanip>
```

```
main()
{
  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 :" << oct
      << 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

Description 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.

Returns A reference to `ostream`. (Usually the `this pointer`.)

See Also `<iomanip>` for manipulators with parameters

Listing 15.20 Example of overloaded manipulator usage:

```
#include <iostream>
ostream &rJus(ostream &stream);

main()
{
  cout << "align right " << rJus << "for column";
  return 0;
}

ostream &rJus(ostream &stream)
{
  stream.width(30);
  stream.setf(ios::right);
  return stream;
}
```

```
Result:
align right                    for column
```

16

27.5 Stream Buffers

The header `<streambuf>` defines types that control input and output to character sequences.

Overview of Stream Buffers

The sections in this chapter are:

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
- The sequences can be association 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:

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 pubseekoff(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);

  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* epptr() 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 showmanyc();
  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

Description 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

Description The public member functions allow access to member functions from derived classes.

27.5.2.2.1 Locales

Locales are used for encapsulation and manipulation of information to a particular locale.

basic_streambuf::pubimbue

Description 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

Description 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.

27.5.2.2.2 Buffer Management and Positioning

Functions used to manipulate the buffer and the input and output positioning pointers.

basic_streambuf::pubsetbuf

Description 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 an 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 16.1 Example of basic_streambuf::pubsetbuf() usage:

```
#include <iostream>
#include <sstream>

const int size = 100;
char temp[size] = "\0";

main()
{
  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

Description Determine 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()`

Listing 16.2 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";

void main()
{
  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();
```

```
}
```

```
Result:
The File now reads:
Metrowerks CodeWarrior "Software at Work"
Registered Trademark
```

basic_streambuf::pubseekpos

Description Determine and move to a desired offset.

Prototype

```
pos_type pubseekoff(pos_type sp,
  ios_base::openmode which = ios::in | ios::out);
```

Remarks The function `pubseekpos()` is use 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(), offset()`

Listing 16.3 Example of streambuf::pubseekpos() usage:

```
The file MW Reference contains:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
#include <iostream.h>
#include <fstream>
#include <stdlib.h>

main()
{
  ifstream in("MW Reference");
  if(!in.is_open())
```

```
    {cout << "could not open file"; exit(1);}

  streampos spEnd, spStart, aCheck;
  spEnd = spStart = 5;

  aCheck = in.rdbuf()->pubseekpos(spStart,ios::in);
  cout << "The offset at the start of the reading"
      << " in bytes is "
      << aCheck.offset() << 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);
  cout << "\nThe final position's offset"
      << " in bytes now is "
      << aCheck.offset() << endl;

  in.close();
  return 0;
}
```

```
Result:
The offfset for the start of the reading in bytes is 5
FGHIJKLMNO
The final position's offset in bytes now is 15
```

basic_streambuf::pubsync

Description 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.

Returns Zero if successful or `EOF` if not via `sync()`.

Listing 16.4 Example of streambuf::pubsync() usage:

```
#include <iostream>

struct address {
  int number;
  char street[40];
}addbook;

main()
{
  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

Description To test for availability of input stream.

Prototype `streamsize in_avail();`

Return I a read is permitted returns size of stream as a type `streamsize`.

basic_streambuf::snextc

Description 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()`.

Listing 16.5 Example of streambuf::snextc() usage:

```
#include <iostream>
#include <sstream>

const int size = 100;

main()
{
  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

Description 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 16.6 Example of streambuf::sbumpc() usage:

```
#include <iostream>
#include <sstream>

const int size = 100;
```

```
string buf = "Metrowerks CodeWarrior --Software at Work--";

main()
{
  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

Description 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()`.

Listing 16.7 Example of streambuf::sgetc() usage:

```
See streambuf::sbumpc()
```

basic_streambuf::sgetn

Description 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 16.8 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

Description 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 16.9 Example of streambuf::sputbackc() usage:

```
#include <iostream>
#include <sstream>

string buffer = "ABCDEF";

main()
{
  stringbuf strbuf(buffer);
  char ch;

  ch = strbuf.sgetc(); // extract first character
  cout << ch; // show it

    //get the next character
  ch = strbuf.snextc();

  // 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 != EOF)
  {
    ch = strbuf.snextc();
    cout << ch;
  }

  return 0;
}
```

```
Result:
AxBC
xBCDEF
```

basic_streambuf::sungetc

Description 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 16.10 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

Description 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 16.11 Example of streambuf::sputc() usage:

```
#include <iostream>
#include <sstream>

main()
{
  stringbuf strbuf;
  strbuf.sputc('A');

  char ch;
  ch = strbuf.sgetc();
  cout << ch;

  return 0;
```

```
}
```

```
Result:
A
```

basic_streambuf::sputn

Description 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.

basic_streambuf::eback

Description Retrieve the beginning pointer for stream input.

Prototype

```
char_type* eback() const;
```

Return The beginning pointer.

basic_streambuf::gptr

Description Retrieve the next pointer for stream input.

Prototype
```
char_type* gptr() const;
```

Return The next pointer.

basic_streambuf::egptr

Description Retrieve the end pointer for stream input.

Prototype
```
char_type* egptr() const;
```

Return The end pointer.

basic_streambuf::gbump

Description Advance 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

Description 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

Protec5ted member functions for stream output sequences.

basic_streambuf::pbase

Description To retrieve the beginning pointer for stream output.

Prototype `char_type* pbase() const;`

Return The `beginning` pointer.

basic_streambuf::pptr

Description To retrieve the next pointer for stream output.

Prototype `char_type* pptr() const;`

Return The `next` pointer.

basic_streambuf::epptr

Description To retrieve the end pointer for stream output.

Prototype `char_type* epptr() const;`

Return The `end` pointer.

basic_streambuf::pbump

Description 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

Description To set the values for the beginning, next and end pointers.

Prototype `void setp(char_type* pbeg, char_type* pend);`

Remarks After the call to `setp()`, `pbeg` equals `pbase()`, `pbeg` equals `pptr()` and `pend` equals `epptr()`.

27.5.2.4 basic_streambuf Virtual Functions

Description The virtual functions in `basic_streambuf` class are to be overloaded in any derived class.

27.5.2.4.1 Locales

Description To get and set the stream locale. These functions should be overridden in derived classes.

basic_streambuf::imbue

Description 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

Description 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

Description 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

Description 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

Description 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

Description Virtual functions for extracting information from an input stream buffer. These functions should be overridden in derived classes.

basic_streambuf::showmanc

Description Shows how many characters in an input stream

Prototype

```
virtual int showmanyc();
```

Remarks If the function `showmanyc()` returns a positive value then calls to `underflow()` will succeed. If `showmanyc()` 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

Description 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

Description 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

Description 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

Description 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

Description 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

Description 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.

17

27.6 Formatting And Manipulators

This chapter discusses formatting and manipulators in the input/output library.

Overview of Formatting and Manipulators

There are three headers—<istream>, <ostream>, and <iomanip>—that contain stream formatting and manipulator routines and implementations.

The sections in this chapter are:

Headers

This section lists the header for istream, ostream, and iomanip.

Header <istream>

Prototype

```
#include <ios>
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 <ios>
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 <istream> controls input from a stream buffer.

The topics in this section are:

27.6.1.1 Template class basic_istream

Description

A class that defines several functions for stream input mechanisms from a controlled stream buffer.

Prototype

```
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_steambuf<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(
  steamsize 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);
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.

27.6.1.1.1 basic_istream Constructors

constructor

Description Creates and 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

Description Destroy the `basic_istream` object.

Prototype

```
virtual ~basic_istream()
```

Remarks The `basic_istream` destructor removes from memory the `basic_istream` object.

Listing 17.1 Example of basic_istream() usage:

```
/* text file contains
"Ask the teacher anything you want to know" */
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

main()
{
  ofstream out("text file", 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 basic_istream prefix and suffix

NOTE: If ipfx and isfx has been replaced with the class sentry in newer versions of the working draft standards

basic_istream::ipfx

Description Prepares for formatted or unformatted input.

Prototype `bool ipfx(bool noskipws = false);`

Remarks If the preparation is not completed `setstate(falibit)` is set.

Returns if `good()` returns `true` else returns `false`.

basic_istream::isfx

Description isfx() has no effects.

Prototype `void isfx();`

Remarks The function isfx is called when an istream terminates, it has no effect and does not return any value.

27.6.1.1.2 Class basic_istream::sentry

Description A class for exception safe prefix and suffix operations.

Prototype

```
namespace std {
template<class charT, class traits =
      char_traits<charT> >
class basic_istream<chartT, 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

Description 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

Description 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 >>

Description Extractors that provide formatted arithmetic input operation.

Prototype

```
basic_istream<charT, traits>& operator >>(
    bool & n);
```

Prototype

```
basic_istream<charT, traits>& operator >>(
    short &n);
```

Remarks: Extracts a short integer value and stores it in `n`.

Prototype

```
basic_istream<charT, traits>& operator >>(
    unsigned short & n);
```

Prototype
```
basic_istream<charT, traits>& operator >>(
    int & n);
```

Prototype
```
basic_istream<charT, traits>& operator >>(
    unsigned int &n);
```

Prototype
```
basic_istream<charT, traits>& operator >>(
    long & n);
```

Prototype
```
basic_istream<charT, traits>& operator >>(
    unsigned long & n);
```

Prototype
```
basic_istream<charT, traits>& operator >>(
    float & f);
```

Prototype
```
basic_istream<charT, traits>& operator >>(
    double& f);
```

Prototype
```
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 17.1 States and stdio equivalents

state	stdio equivalent
(flags() & basefield) == oct	%o
(flags() & basefield) == hex	%x
(flags() & basefield) != 0	%x
(flags() & basefield) == 0	%i
Otherwise	
signed integral type	%d
unsigned integral type	%u

27.6.1.2.3 basic_istream extractor operator >>

Description

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)`.

Prototype

```
basic_istream<charT, traits>& operator >>(
  basic_ios<charT, traits>& (*pf)(
  basic_ios<charT,traits>&))
```

Remarks

Calls `pf(*this)` then returns `*this`.

Prototype

```
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().

Prototype

```
basic_istream<charT, traits>& operator >>(
    char_type& c);
```

Remarks: Extracts a single character and stores it in c if possible otherwise call setstate(failbit).

Prototype

```
basic_istream<charT, traits>& operator >>(
    void*& p);
```

Remarks

Converts a pointer to void and stores it in `p`.

Prototype

```
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.

Returns The `this pointer is returned`.

See Also `basic_ostream::operator <<`

Listing 17.2 Example of basic_istream:: extractor usage:

```
The MW Reference input file should read
float 33.33 double 3.16e+10 Integer 789 character C
```

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

char ioFile[81] = "MW Reference";

main()
{
```

```
  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;
  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:

Description 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.

Returns The `this pointer is returned`.

Listing 17.3 Example of basic_istream overloaded extractor usage:

```
#include <iostream>
#include <string.h>
#include <iomanip>
#include <stdlib.h>

class phonebook {
  friend ostream &operator<<(ostream &stream,
    phonebook o);
  friend istream &operator>>(istream &stream,
    phonebook &o);

  private:
```

```
  char name[80];
  int areacode;
  int exchange;
  int num;

  public:
  void putname() {cout << num;}
  phonebook() {}; // default constructor
  phonebook(char *n, int a, int p, int nm)
    {strcpy(name, n); areacode = a;
      exchange = p; num = nm;}
};

main()
{
  phonebook a;

  cin >> a;
  cout << a;

  return 0;
}

ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.exchange << "-";
  cout << setfill('0') << setw(4) << o.num << "\n";
  return stream;
}

istream &operator>>(istream &stream, phonebook &o)
{
  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

Description 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.

Returns An `int` type count of the bytes read.

Listing 17.4 Example of basic_istream::gcount() usage:

```
#include <iostream>
#include <iostream.h>
#include <fstream>

const SIZE = 4;

struct stArray {
  int index;
  double dNum;
};

main()
{
  ofstream fOut("test");
  if(!fOut.is_open())
    {cout << "can't open out file"; return 1;}

  stArray arr;
  short i;

  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

Description 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()`.

Prototype `basic_istream<charT, traits>& get(char_type& c);`

Remarks Extracts a character and assigns it to `c` if possible else calls `set-state(failbit)`.

Prototype

```
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 `set-state(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`.

Prototype

```
basic_istream<charT, traits>& get(
  basic_steambuf<char_type, traits>& sb,
  char_type delim = traits::newline());
```

Remarks Extracts a characters and assigns them to the `basic_streambuf` object `sb` if possible else calls `setstate(failbit).` Extraction stops if...

- an insertion fails
- `end-of-file` is encountered.
- an exception is thrown
- the next the next available character `c == delim` (in which case `c` is not extracted.)

Returns 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 17.5 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.h>
#include <fstream.h>
#include <stdlib.h>

main()
{
  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];
```

```
main()
{
  cout << " Enter your name: ";
  cin.get(buf, size);
  cout << buf;

  return 0;
}
```

```
Result:
Enter your name: Metrowerks CodeWarrior <enter>
Metrowerks CodeWarrior
```

basic_istream::getline

Description

To obtain a delimiter terminated character sequence from an input stream.

Prototype

```
basic_istream<charT, traits>& getline(
  char_type* s,
  streamsize 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.

In any case it stores a null char into the next successive location of the array.

Returns The `this` pointer is returned.

See Also `basic_ostream::flush()`

Listing 17.6 Example of basic_istream::getline() usage:

```
#include <iostream>

const int size = 120;
main()
{
  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'

main()
{
  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

Description 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();`

Returns The `this pointer` is returned.

Listing 17.7 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 <stdlib.h>

char inFile[] = "MW Reference";
char bslash = '/';

main()
{
  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.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

Description 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.

Returns If `good()` is false returns `traits::eof()` else returns the value of the next character in the stream.

Listing 17.8 Example of basic_istream::peek() usage:

```
See basic_istream::ignore()
```

basic_istream::read

Description 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.

Returns The `this pointer` is returned.

See Also `write()`

Listing 17.9 Example of basic_istream::read() usage:

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <stdlib.h>
#include <string.h>

struct stock {
  char name[80];
  double price;
  long trades;
};

char *Exchange = "BBSE";
char *Company = "Big Bucks Inc.";

 main()
{
  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_istream::readsome

Description Extracts characters and stores them in an array.

Prototype `streamsize readsome(charT_type* s, streamsize n);`

Remarks The function `readsome` extracts and stores characters storing them in the buffer pointed to by `s` until the following conditions are met.

- end-of-file is encountered (in which case setstate(eofbit) is called.)
- No characters are extracted.
- A limit of characters is extracted either n or the size of the buffer.

Returns The number of characters extracted.

Listing 17.10 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 <stdlib.h>

const short Size = 81;

main()
{
  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))
  {
    Paragraph << Buffer;
    in.readsome(Buffer, 5);
  }

  cout << Paragraph.str();

  in.close();
  return 0;
}
```

```
Result:
Metrowerks CodeWarrior
```

```
Software at Work
Registered Trademark
```

basic_istream::putback

Description 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.

Returns The `this pointer` is returned.

See Also `sungetc()`

Listing 17.11 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 = '/';

main()
{
  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

Description 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.

Returns The `this` pointer is returned.

See Also `putback()`, `ignore()`

Listing 17.12 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 <stdlib.h>

char inFile[] = "MW Reference";
char bslash = '/';

main()
{
  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

Description 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.

Returns If `rdbuf() is Null` returns `-1` otherwise returns zero.

Listing 17.13 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 <stdlib.h>

char inFile[] = "MW Reference";
```

```
main()
{
  ifstream in(inFile);
  if(!in.is_open())
    {cout << "could not open file"; exit(1);}

  char str[10];
  if(in.sync() != EOF);  // actually does no good
  while (in.good())
  {
    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

Description 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)`.

Returns The current offset as a `pos_type` if successful else returns `-1`.

See Also `basic_streambuf::pubseekoff()`

Listing 17.14 Example of basic_istream::tellg() usage:

```
See basic_istream::seekg()
```

basic_istream::seekg

Description To move to a variable position in a stream.

Prototype `basic_istream<charT, traits>& seekg(pos_type);`

Prototype
```
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.

Returns The `this pointer` is returned.

See Also `basic_streambuf::pubseekoff()` and `pubseekpos()`.

Listing 17.15 Example of basic_istream::seekg() usage:

```
The file MW Reference contains:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

main()
{
  ifstream in("MW Reference");
```

```
  if(!in.is_open())
  {cout << "could not open file"; exit(1);}

  streampos spEnd, spStart, aCheck;
  spEnd = spStart = 5;

  in.seekg(spStart);
  aCheck = in.tellg();
  cout << "The offfset at the start of the reading in bytes is "
    << aCheck.offset() << 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.offset() << endl;
  streamoff gSet = 0;
  in.seekg(gSet, ios::beg);

  aCheck = in.tellg();
  cout << "The final position's offset in bytes now is "
  << aCheck.offset() << endl;

  in.close();
  return 0;
}
```

```
Result:
The offfset 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

Description 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.

Returns The `this pointer`.

Listing 17.16 Example of basic_istream:: manipulator ws usage:

```
The File MWRef (where the number of blanks (and/or tabs)
is unknown) contains:
    a     b   c
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

main()
{
  char * inFileName = "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 << "|";
  while (1)
```

```
  {
    in >> ws >> ch;  // ignore white spaces
    // in >> ch; // does not skip white spaces
    if (in.good())
      cout << ch;
    else break;
  }
  cout << "|" << endl;
  in.close();
  return(0);
}
```

```
Result:
|abc|
Using the other input statement the result is:
|      a      b   c|
```

27.6.1.4.1 basic_iostream Constructor

Constructor

Description

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

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 charTchar_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)
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 a independent class, as a base class for the `basic_ofstream` class or a user derived classes.

27.6.2.2 basic_ostream Constructor

Description 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 17.17 Example of basic_ostream() usage:

```
The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"

#include <iostream>
#include <fstream>
#include <stdlib.h>

char inFile[] = "MW Reference";

void main()
{
  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();
}
```

```
Result:
The File now reads:
Metrowerks CodeWarrior "Software at Work"
Registered Trademark
```

27.6.2.3 basic_ostream prefix and suffix functions

NOTE: opfx and osfx have been replaced with the class sentry in newer versions of the working draft standards

basic_ostream::opfx

Description Prepares for a stream formatted and unformatted output.

Prototype

```
bool opfx();
```

Remarks If tie() is not a null pointer calls tie()->flush().

Returns good().

basic_ostream::osfx

Description Cleans up for closing a stream opened for output.

Prototype

```
void osfx();
```

Remarks If not unit buffered calls flush().

27.6.2.3 Class basic_ostream::sentry

Description A class for exception safe prefix and suffix operations.

Prototype

```
namespace std {
template<class charT, class traits =
      char_traits<charT> >
class basic_ostream<chartT, 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

Description 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

Description 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

Description

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 <<

Description

To provide formatted insertion of types into a stream.

Prototype

```
basic_ostream<charT, traits>& operator<<
  (short n)
```

Prototype

```
basic_ostream<charT, traits>& operator<<
  (unsigned short n)
```

Prototype

```
basic_ostream<charT, traits>& operator<<
  (int n)
```

Prototype

```
basic_ostream<charT, traits>& operator<<
(unsigned int n)
```

Prototype

```
basic_ostream<charT, traits>& operator<<
  (long n)
```

Prototype

```
basic_ostream<charT, traits>& operator<<
  (unsigned long n)
```

Prototype
```
basic_ostream<charT, traits>& operator<<
  (float f)
```

Prototype
```
basic_ostream<charT, traits>& operator<<
  (double f)
```

Prototype
```
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

Returns The `this pointer` is returned

Table 17.2 Output states and stdio equivalents.

Output State	stdio equivalent
Integers	
(flags() & basefield) == oct	%o
(flags() & basefield) == hex	%x
(flags() & basefield) != 0	%x
Otherwise	
signed integral type	%d
unsigned integral type	%u
Floating Point Numbers	
(flags() & floatfield) == fixed	%f
(flags() & floatfield) == scientific (flags() & uppercase) != 0	%e %E
Otherwise	
(flags() & uppercase) != 0	%g %G
An integral type other than a char type	

Output State	stdio equivalent
(flags() & showpos) != 0	+
(flags() & showbase) != 0	#
A floating point type	
(flags() & showpos) != 0	+
(flags() & showpoint) != 0	#

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() and fixed) !=0` or if `precision() >0` the conversion specification is the value of `precision()`.

For any conversion padding behaves in the following manner.

Table 17.3 Conversion state and stcio equivalents.

State	Justification	stdio equivalent
(flags()& adjustfield) == left	left	space padding
(flags() & adjustfield) == internal	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.

Returns 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).`

Prototype
```
basic_ostream<charT, traits>& operator<<
  (basic_ostream<charT, traits>&
    (*pf)(basic_ios<charT, traits>&));
```

Remarks Calls `pf(*this)` return `*this`.

Prototype
```
basic_ostream<charT, traits>& operator<<
  (const char_type *s)
```

Prototype
```
basic_ostream<charT, traits>& operator<<
  (char_type c)
```

Prototype
```
basic_ostream<charT, traits>& operator<<
  (bool n)
```

Remarks Behaves depending on how the boolalpha flag is set.

Prototype
```
basic_ostream<charT, traits>& operator<<
  (void p)
```

Remarks Converts the pointer to `void p` as if the specifier was `%p` and returns `*this.`

Prototype
```
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 17.18 Example of basic_ostream inserter usage:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char oFile[81] = "MW Reference";

main()
{
  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

Description To provide specialized output mechanisms for an object.

Prototype

```
Overloading inserter prototype
basic_ostream &oerator<<(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.

Returns The `this pointer`.

Listing 17.19 Example of overloaded inserter usage:

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

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) {}
};

main()
{
  phonebook a("Sales", 800, 377, 5416);
  phonebook b("Voice", 512, 305, 0400);
  phonebook c("Fax",   512, 873, 4901);

  cout << a << b << c;

  return 0;
}

ostream &operator<<(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) 305-0256
Fax (512) 873-4901
```

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

Description To return the offset of the put pointer in an output stream.

Prototype `pos_type tellp();`

Returns If `fail()` returns `-1` else returns `rdbuf()->pubseekoff(0, cur, out)`.

See Also basic_istream::tellg(), seekp(0).

Listing 17.20 Example of basic_ostream::tellp() usage.

```
see basic_ostream::seekp().
```

basic_ostream::seekp

Description Randomly move to a position in an output stream.

Prototype `basic_ostream<charT, traits>& seekp(pos_type);`

Prototype
```
basic_ostream<charT, traits>& seekp
    (off_type, iosbase::seekdir);
```

Remarks The function `seekp` is overloaded to take a single argument of a `pos_type pos` that calls `rdbuf()->pubseekpos(pos)`. It is also overloaded to take two arguments an `off_type off` and ios_base::seekdir type dir that calls rdbuf()->pubseekoff(off, dir).

Returns The this pointer.

See Also basic_istream seekg(), tellp()

Listing 17.21 Example of basic_ostream::seekp() usage.

```
#include <iostream>
#include <sstream>
#include <string>

string motto = "Metrowerks CodeWarrior - Software at Work";

main()
{
  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

Description 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)`.

Returns The `this pointer`.

Listing 17.22 Example of basic_ostream::put() usage:

```
#include <iostream>
```

```
main()
{
  char *str = "Metrowerks CodeWarrior Software at Work";
  while(*str)
  {
    cout.put(*str++);
  }
  return 0;
}
```

```
Result:
Metrowerks CodeWarrior Software at Work
```

basic_ostream::write

Description 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 is can be used to write an object by casting that object as a `unsigned char` pointer. If the operation fails calls `setstate(badbit)`.

Returns A reference to `ostream`. (The `this pointer`.)

See Also `read()`

Listing 17.23 Example of basic_ostream::write() usage:

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <stdlib.h>
#include <string.h>
```

```
struct stock {
  char name[80];
  double price;
  long trades;
};

char *Exchange = "BBSE";
char *Company = "Big Bucks Inc.";

main()
{
  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

Description 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)`.

Returns The `this pointer`.

Listing 17.24 Example of basic_ostream::flush() usage:

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

class stopwatch {
private:
  double begin, set, end;
public:
  stopwatch();
```

```
  ~stopwatch();
  void start();
  void stop();
};

stopwatch::stopwatch()
{
  begin = (double) clock() / CLOCKS_PER_SEC;
  end   = 0.0;
  start();
  cout << "begin the timer: ";
}

stopwatch::~stopwatch()
{
  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()
{
  set = double(clock()/CLOCKS_PER_SEC);
}

void stopwatch::stop()
{
  end = double(clock()/CLOCKS_PER_SEC);
}

void time_delay(unsigned short t);

main()
{
  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)
{
  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 >
The Object lasted: 4.78 seconds
```

27.6.2.6 Standard basic_ostream manipulators

Description To provide an inline formatting mechanism.

basic_ostream:: endl

Description 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.

Returns A reference to `basic_ostream`. (The `this pointer`.)

See Also `ostream:: operators`

basic_ostream::ends

Description 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.

Returns 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 17.25 Example of basic_ostream:: ends usage:

```
#include <iostream>
#include <sstream>

main()
{
  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

Description 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.

Returns A reference to `ostream`. (The `this pointer`.)

See Also `ostream::flush()`

Listing 17.26 Example of basic_ostream:: flush usage:

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

class stopwatch {
private:
  double begin, set, end;
public:
  stopwatch();
  ~stopwatch();
  void start();
```

```
  void stop();
};

stopwatch::stopwatch()
{
  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()
{
  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()
{
  set = double(clock()/CLOCKS_PER_SEC);
}

void stopwatch::stop()
{
  end = double(clock()/CLOCKS_PER_SEC);
}

void time_delay(unsigned short t);

main()
{
  stopwatch watch; // create object and initialize
```

```
  time_delay(5);
  return 0; // destructor called at return
}
    //time delay function
void time_delay(unsigned short t)
{
  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: 5.42 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

Description To create a specific use instance of a template by replacing the parameterized elements with pre-defined types.

resetiosflags

Description 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.

Returns A smanip type, that is an implementation defined type.

See Also `ios_base::setf(), ios_base::unsetf()`

Listing 17.27 Example of resetiosflags() usage:

```
#include <iostream>
#include <iomanip>

main()
{
  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

Description 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.

Returns A smanip type, that is an implementation defined type.

See Also `ios_base::setf(), ios_base::unsetf()`

Listing 17.28 Example of setiosflags() usage:

```
See resetiosflags()
```

:setbase

Description 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.

Returns A smanip type, that is an implementation defined type.

See Also `ios_base::setf()`

Listing 17.29 Example of <omanip>::setbase usage:

```
#include <iostream>
#include <iomanip>

main()
```

```
{
  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

Description 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`.

Returns A smanip type, that is an implementation defined type.

See Also `basic_ios::fill`

Listing 17.30 Example of basic_ios::setfill() usage:

```
#include <iostream>
#include <iomanip>

main()
{
  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

Description 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.

Returns A smanip type, that is an implementation defined type.

See Also ios_base::setf(), ios_base::precision()

Listing 17.31 Example of <omanip>::setprecision() usage:

```
#include <iostream>
#include <iomanip>

main()
{
  cout << "Original: " << 321.123456 <<   endl;
  cout << "Precision set: " << setprecision(8)
    << 321.123456 <<   endl;
  return 0;
}
```

```
Result:
Original:  321.123
Precision set:  321.12346
```

setw

Description 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.

Returns A pointer to `ostream`

See Also `ios_base::width()`

Listing 17.32 Example of <omanip>::setw() usage:

```
#include <iostream>
#include <iomanip>

main()
{
  cout << setw(8)
    << setfill('*')
    << "Hi!" << endl;
  return 0;
}
```

```
Result:
Hi!*****
```

Overloaded Manipulator

Description 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;
}
```

Prototype Overloaded output manipulator for `int` type.

```
ostream &omanip_name(ostream &stream, type param)
{
 // body of code
 return stream;
```

```
}
```

Prototype 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.

Returns A pointer to stream object.

Listing 17.33 Example of overloaded manipulator usage:

```
#include <iostream>
#include <iomanip>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

char buffer[80];
char *Password = "Metrowerks";

char *StrUpr(char * str);

imanip<char *> verify(char *check);
istream &verify_implement(istream &stream, char *check);

main()
{
  cin >> verify(StrUpr(Password));
  cout << "Log in was Completed ! \n";

  return 0;
}

imanip<char *> verify(char *check)
{
```

```
  return imanip <char *> (verify_implement, check);
}

istream &verify_implement(istream &stream, char *check)
{
  short attempts = 3;

  do {
    cout << "Enter password: ";
    stream >> buffer;

    StrUpr(buffer);
    if (! strcmp(check, buffer)) return stream;
    cout << "\a\a";
    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++ = toupper(*p);
  return str;
}
```

```
Result:
Enter password: <codewarrior>
Enter password: <mw>
Enter password: <metrowerks>
Log in was Completed !
```

18

27.7 String-Based Streams

This chapter discusses string-based streams in the standard C++ library.

Overview

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 sections in this chapter are:

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_strngbuf<wchar>wstringbuf;

  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` is use 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 551
- "27.7.1.2 Member functions" on page 553
- "27.7.1.3 Overridden virtual functions" on page 554

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` constuctors:

- `explicit basic_stringbuf(ios_base::openmode);`
- `explicit basic_stringbuf(const basic_string, ios_base::openmode);`

Constructor

Description To create a string buffer for characters for input/output.

Prototype

```
explicit basic_stringbuf(ios_base::openmode which =
        ios_base::in | ios_base:out);
```

Prototype

```
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 18.1 Example of basic_stringbuf::basic_stringbuf() usage:

```
#include <iostream>
#include <sstream>

const int size = 100;

main()
{
  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

Description To return the `basic_string` object stored in the buffer.

Prototype `basic_string<char_type> str() const;`

Remarks The function `str()` freezes the buffer then returns a `basic_string` object.

Returns If successful a `basic_string` object.

Prototype `void str(const basic_string<char_type>&s);`

Remarks The function `str()` assigns the value of the `basic_string` object to the argument `'s'` if successful.

Listing 18.2 **Example of basic_stringbuf::str() usage:**

```
#include <iostream>
#include <sstream>

string buf;
```

```
char CW[] = "Metrowerks CodeWarrior";
char AW[] = " - Software at work";

main()
{
  stringbuf strbuf(buf, ios::out);

  int size;
  size = strlen(CW);
  strbuf.sputn(CW, size);
  size = strlen(AW);
  strbuf.sputn(AW, size);

  cout << strbuf.str();

  return 0;
}
```

```
Result
Metrowerks CodeWarrior - Software at work
```

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

Description 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.

Returns 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

Description 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.

Returns 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

Description 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.

Returns The function returns `traits::eof()` for failure or some unspecified result to indicate success.

See Also `basic_streambuf::overflow()`

basic_stringbuf::seekoff

Description 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` overrides the basic_streambuf virtual function.

Returns A `pos_type` value, which is an invalid stream position.

See Also `basic_streambuf::seekoff()`

basic_stringbuf::seekpos

Description 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.

Returns 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 use 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 558
- "27.7.2.2 Member functions" on page 559

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_strng<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

Description The `basic_istringstream` constructors create a `basic_stringstream` object and initialize the `basic_streambuf` object.

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 18.3 Example of basic_istringsteam::basic_istringsteam() usage

```
#include <iostream>
#include <string>
#include <sstream>
```

```
const short Size = 100;

string sBuffer = "3 12.3 line";

main()
{
  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

Description 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.

Returns 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 18.4 Example of basic_istringstream::rdbuf() usage.

```
#include <iostream>
#include <sstream>

string buf = "Metrowerks CodeWarrior - Software at work";
char words[50];
main()
{
  istringstream ist(buf);
  istream in(ist.rdbuf());
  in.seekg(25);

  in.get(words,50);
  cout << words;

  return 0;
}
```

```
Result
Software at work
```

basic_istringstream::str

Description To return the `basic_string` object stored in the buffer.

Prototype `basic_strng<charT> str() const;`
`void str(const basic_string<charT> &s);`

Remarks The function `str()` freezes the buffer then returns a `basic_string` object.

Returns If successful a `basic_string` object.

See Also `basic_streambuf::str()`, `basic_ostringstream.str()`

`basic_stringstream::str()`

Listing 18.5 Example of basic_istringstream::str() usage.

```
#include <iostream>
#include <sstream>

string buf = "Metrowerks CodeWarrior - Software at Work";

main()
{
  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_strng<charT> str() const;
  void str(const basic_string<charT> &s);

  private:
  basic_stringbuf<charT,traits> sb; exposition only
  };
}
```

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

Description

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 18.6 Example of basic_ostringsteam::basic_ostringsteam() usage

```
The file MW Reference contains
Metrowerks CodeWarrior - Software at Work
Registered Trademark
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>

const short Size = 100;

main()
{
```

```
  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_ostringstream` has two member functions:

- `rdbuf()`
- `str()`

basic_ostringstream::rdbuf

Description 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.

Returns 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 18.7 example of basic_ostringsteam::rdbuf() usage

```
#include <iostream>
#include <sstream>
#include <string>

string motto = "Metrowerks CodeWarrior - Software at Work";

main()
{
  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_ostringstream::str

Description To return the `basic_string` object stored in the buffer.

Prototype

```
basic_strng<charT> str() const;
void str(const basic_string<charT> &s);
```

Remarks The function `str()` freezes the buffer then returns a `basic_string` object.

Returns If successful a `basic_string` object.

See Also `basic_streambuf::str()`, `basic_istringstream.str()`

`basic_stringstream::str()`

Listing 18.8 Example of basic_ostringstream::str() usage.

```
#include <iostream>
#include <sstream>

main()
{
  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_strng<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::open-mode)`
- `explicit basic_stringstream (const basic_string, ios_base::openmode)`

Constructor

Description

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 18.9 Example of basic_stringstream::basic_stringstream() usage

```
#include <iostream>
#include <sstream>

char buf[50] = "ABCD 22 33.33";
char words[50];

main()
{
  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

Description 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.

Returns 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 18.10 Example of basic_stringstream::rdbuf() usage

```
#include <iostream>
#include <sstream>

string buf = "Metrowerks CodeWarrior - Software at work";
char words[50];
main()
{
  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

Description To return the `basic_string` object stored in the buffer.

Prototype

```
basic_strng<charT> str() const;
void str(const basic_string<charT> &s);
```

Remarks The function `str()` freezes the buffer then returns a `basic_string` object.

Returns If successful a `basic_string` object.

See Also basic_streambuf::str(), basic_ostringstream.str()

basic_istringstream::str()

Listing 18.11 Example of basic_stringstream::str() usage

```
#include <iostream>
#include <sstream>

string buf = "Metrowerks CodeWarrior - Software at Work";
char words[50];

main()
{
  stringstream iost(buf, ios::in);

  cout << iost.str();

  return 0;
}
```

```
Result
Metrowerks CodeWarrior - Software at Work
```

19

27.8 File Based Streams

Association of stream buffers with files for file reading and writing.

Overview of File Based Streams

The sections in this chapter are:

Header <fstream>

Description

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;
```

```
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

Description 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 576
- "27.8.1.3 Member functions" on page 576
- "27.8.1.4 Overridden virtual functions" on page 579

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 showmanyc();
  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.

27.8.1.2 basic_filebuf Constructors

Default Constructor

Description 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

Description To remove the `basic_filebuf` object from memory.

Prototype `virtual ~basic_filebuf();`

Listing 19.1 **For example of basic_filebuf::basic_filebuf() usage:**

```
See basic_filebuf::open().
```

27.8.1.3 Member functions

basic_filebuf::is_open

Description 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.

Returns True if stream is available and open.

Listing 19.2 For example of basic_filebuf::is_open() usage

```
See: basic_filebuf::basic_filebuf
```

basic_filebuf::open

Description 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.

WARNING! 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 19.1 Legal basic_filebuf file opening modes

input only	
ios::in	ios::in | ios::binary
output only	
ios::out | ios::trunc	ios::out | ios::trunc | ios::binary
ios::out | ios::app	ios::out | ios::app | ios::binary
input and output	
ios::in | ios::out	ios::in | ios::out | ios::binary

input only

ios::in | ios::out | ios::trunc | ios::binary	ios::in | ios::out | ios::app
ios::in | ios::out | ios::trunc	ios::in | ios::out | ios::app | ios::binary

Returns If successful the `this pointer` is returned, if `is_open()` equals true then a `null pointer` is returned.

Listing 19.3 Example of filebuf::open() usage:

```
The file MW Reference before operation
contains.Metrowerks CodeWarrior "Software at Work”
#include <iostream>
```

```
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char inFile[] = "MW Reference";

main(){
  filebuf in;
  in.open(inFile, ios::out | ios::app);
  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
```

basic_filebuf::close

Description 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.

Returns The `this pointer` with success otherwise a `null pointer.`

Listing 19.4 For example of basic_filebuf::close() usage

```
See basic_filebuf::open()
```

27.8.1.4 Overridden virtual functions

basic_filebuf::showmanyc

Description Overrides basic_streambuf::showmanyc().

Prototype `virtual int showmanyc();`

Remarks Behaves the same as basic_sreambuf::showmanyc().

basic_filebuf::underflow

Description Overrides basic_streambu::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

Description 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.

Returns `traits::eof()` if failure and returns either the character put back or `traits::not_eof(c)` for success.

basic_filebuf::overflow

Description 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.

Returns traits::eof() with failure.

basic_filebuf::seekoff

Description 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 `fseek(file, off, whence).`

Returns Seekoff function returns a newly formed `pos_type` object which contains all information needed to determine the current position if successful. An invalid stream position if it fails.

basic_filebuf::seekpos

Description 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.

Returns Seekpos function returns a newly formed `pos_type` object which contains all information needed to determine the current position if successful. An invalid stream position if it fails.

basic_filebuf::setbuf

Description 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

Description Overrides basic_streambuf::sync

Prototype

```
virtual int sync();
```

Remarks Description undefined in standard at the time of writing.

basic_filebuf::imbue

Description Overrides basic_streambuf::imbue

Prototype virtual void imbue(const locale& loc);

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 583
- "27.8.1.7 Member functions" on page 585

Synopsis

```
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

Description 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.he 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 19.5 Example of basic_ifstream::basic_ifstream() constructor usage:

```
The MW Reference file contains:
Metrowerks CodeWarrior "Software at Work"
#include <iostream>
#include <fstream>
#include <stdlib.h>

char inFile[] = "MW Reference";

main()
{
  ifstream in(inFile, ios::in);
    if(!in.is_open())
      {cout << "can't open input file"; exit(1);}

  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

Description 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.

Returns A pointer to type `basic_filebuf.`

Listing 19.6 Example of basic_ifstream::rdbuf() usage:

```
The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char inFile[] = "MW Reference";

void main()
{
  ifstream inOut(inFile, ios::in | ios::out);
  if(!inOut.is_open())
    {cout << "Could not open file"; exit(1);}

  ostream Out(inOut.rdbuf());

  char str[] = "\n\tRegistered Trademark";
```

```
  inOut.rdbuf()->pubseekoff(0, ios::end);

  Out << str;

  inOut.close();
}
```

```
Result:
The File now reads:
Metrowerks CodeWarrior "Software at Work"
  Registered Trademark
```

basic_ifstream::is_open

Description 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.

Returns True if file is open.

Listing 19.7 For example of basic_ifstream::is_open() usage

```
See basic_ifstream::basic_ifstream()
```

basic_ifstream::open

Description 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.

Returns No return

Table 19.2 17.4.1.1.4Legal basic_ifstream file opening modes1

Opening Mode	
input only	
ios::in	ios::in | ios::binary
input and output	
ios::in | ios::out	ios::in | ios::out | ios::binary
ios::in | ios::out | ios::trunc | ios::binary	ios::in | ios::out | ios::app
ios::in | ios::out | ios::trunc	ios::in | ios::out | ios::app | ios::binary

WARNING! 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 19.8 Example of basic_ifstream::open() usage:

```
The MW Reference file contains:
Metrowerks CodeWarrior "Software at Work"
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char inFile[] = "MW Reference";
```

```
main()
{
  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

Description 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 and exception.

Returns: No return.

Listing 19.9 Example of basic_ifstream::close() usage:

```
See basic_ifstream::basic_ifstream()
```

27.8.1.8 Template class basic_ofstream

Description

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 590
- "27.8.1.10 Member functions" on page 591

Synopsis

```
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

Description 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 19.10 Example of basic_ofstream::ofstream() usage:

```
Before the operation MW Reference
may or may not exist.
```

```
#include <iostream>
#include <fstream>
```

```
#include <stdlib.h>

char outFile[] = "MW Reference";

main()
{
  ofstream out(outFile);
  if(!out.is_open())
    {cout << "file not opened"; exit(1);

  out << "This is a limited reference which "
    << "contains a description\n"
    << "of the ANSI Working Draft Standard "
    << "C++ library bundled\n"
    << "with Metrowerks C++. ";

  out.close();
  return 0;
}
```

```
Result:
The File MW Reference reads after the operation:
This is a limited reference which contains a description
of the ANSI Working Draft Standard C++ library bundled
with Metrowerks C++.
```

27.8.1.10 Member functions

basic_ofstream::rdbuf

Description 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.

Returns A pointer to `filebuf` type.

Listing 19.11 Example of basic_ofstream::rdbuf() usage:

```
The file MW Reference before the operation contains:
This is a limited reference which contains a
description of the ANSI Working Draft Standard
C++ library bundled with Metrowerks C++.
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char outFile[] = "MW Reference";

main()
{
  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 a limited reference which contains a
description of the ANSI Working Draft Standard
C++ library bundled with Metrowerks C++.

The file MW Reference after operation contains:
This is a limited reference which contains a
description of the ANSI Working Draft Standard
C++ library bundled with Metrowerks C++.
And so it goes...
```

basic_ofstream::is_open

Description 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.

Returns `True` if file stream is open and available for output.

Listing 19.12 For example of basic_ofstream::is_open() usage

```
See basic_ofstream::ofstream()
```

basic_ofstream::open

Description 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.

Returns No return

Table 19.3 Legal basic_ofstream file opening modes.

Opening Mode	
output only	
ios::out | ios::trunc	ios::out | ios::trunc | ios::binary
ios::out | ios::app	ios::out | ios::app | ios::binary
input and output	
ios::in | ios::out	ios::in | ios::out | ios::binary
ios::in | ios::out | ios::trunc | ios::binary	ios::in | ios::out | ios::app
ios::in | ios::out | ios::trunc	ios::in | ios::out | ios::app | ios::binary

WARNING! 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 19.13 Example of basic_ofstream::open() usage:

```
Before operation, the file MW Reference contained:
Chapter One
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char outFile[] = "MW Reference";

main()
{
  ofstream out;
  out.open(outFile, ios::out | ios::app);
    if(!out.is_open())
      {cout << "file not opened"; exit(1);}

  out << "\nThis is a limited reference which contains\n"
    << "a description of the ANSI Working Draft Standard\n"
    << "C++ library bundled with Metrowerks C++.\n";

  out.close();
  return 0;
}
```

```
Result:
After the operation MW Reference contained
Chapter One
This is a limited reference which contains
a description of the ANSI Working Draft Standard
C++ library bundled with Metrowerks C++.
```

---

basic_ofstream::close

Description 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.

Returns No return.

Listing 19.14 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 597
- “27.8.1.13 Member Functions” on page 599

Synopsis

```
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_ofstream 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

Description 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 19.15 Example of basic_fstream:: basic_fstream() usage

```
The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char inFile[] = "MW Reference";

void main()
{
  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();
}
```

```
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

Description 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 of an `fstream` stream you may need to manipulate the base buffer. The function `rdbuf()` returns a pointer to this buffer for manipulation.

Returns A pointer to type `basic_filebuf`.

Listing 19.16 Example of basic_fstream::rdbuf() usage

```
The MW Reference file contains originally
Metrowerks CodeWarrior "Software at Work"
```

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

char inFile[] = "MW Reference";

main()
{
  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
```

basic_fstream::is_open

Description 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.

Returns True if a file is available and open.

For an example, see “Example of basic_fstream:: basic_fstream() usage” on page 598.

basic_fstream::open

Description 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.

WARNING! 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 19.4 Legal file opening modes

<table>
<tr><td colspan="2">input only</td></tr>
<tr><td>ios::in</td><td>ios::in | ios::binary</td></tr>
<tr><td colspan="2">output only</td></tr>
<tr><td>ios::out | ios::trunc</td><td>ios::out | ios::trunc | ios::binary</td></tr>
<tr><td>ios::out | ios::app</td><td>ios::out | ios::app | ios::binary</td></tr>
<tr><td colspan="2">input and output</td></tr>
<tr><td>ios::in | ios::out</td><td>ios::in | ios::out | ios::binary</td></tr>
<tr><td>ios::in | ios::out | ios::trunc | ios::binary</td><td>ios::in | ios::out | ios::app</td></tr>
<tr><td>ios::in | ios::out | ios::trunc</td><td>ios::in | ios::out | ios::app | ios::binary</td></tr>
</table>

Returns No return.

For an example, see “Example of basic_fstream::rdbuf() usage” on page 599.

basic_fstream::close

Description 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.

Returns No return.

For an example, see "Example of basic_fstream:: basic_fstream() usage" on page 598.

20

C Library files

The header `<cstdio>` contains the C++ implementation of the Standard C Headers.

27.8.2 C Library files

<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	

Index

S

T

U

V

W

X

CodeWarrior

MSL C++ Reference

Credits

writing lead:	Ron Liechty
other writers:	Portions copyright © 1995 by Modena Software Inc
engineering:	Michael Marcotty, Benjamin Koznik , Andreas Hommel
frontline warriors:	Tech Support and all CodeWarriors

Guide to CodeWarrior Documentation

If you need information about...	See this
Installing updates to CodeWarrior	*QuickStart Guide*
Getting started using CodeWarrior	*QuickStart Guide*; *Tutorials* (Apple Guide)
Using CodeWarrior IDE (Integrated Development Environment)	*IDE User's Guide*
Debugging	*Debugger Manual*
Important last-minute information on new features and changes	*Release Notes* folder
Creating Macintosh and Power Macintosh software	*Targeting Mac OS*; *Mac OS* folder
Creating Microsoft Win32/x86 software	*Targeting Win32*; *Win32/x86* folder
Creating Java software	Targeting Java Sun Java Documentation folder
Creating Magic Cap software	*Targeting Magic Cap*; *Magic Cap* folder
Using ToolServer with the CodeWarrior editor	*IDE User's Guide*
Controlling CodeWarrior through AppleScript	*IDE User's Guide*
Using CodeWarrior to program in MPW	*Command Line Tools Manual*
C, C++, or 68K assembly-language programming	*C, C++, and Assembly Reference*; *MSL C Reference*; *MSL C++ Reference*
Pascal or Object Pascal programming	*Pascal Language Manual*; *Pascal Library Reference*
Fixing compiler and linker errors	*Errors Reference*
Fixing memory bugs	*ZoneRanger Manual*
Speeding up your programs	*Profiler Manual*
PowerPlant	*The PowerPlant Book;* PowerPlant Advanced Topics; PowerPlant reference documents
Creating a PowerPlant visual interface	*Constructor Manual*
Creating a Java visual interface	Constructor for Java Manual
Learning how to program for the Mac OS	*Discover Programming for Macintosh*
Learning how to program in Java	Discover Programming for Java
Contacting Metrowerks about registration, sales, and licensing	*Quick Start Guide*
Contacting Metrowerks about problems and suggestions using CodeWarrior software	*email Report Forms* in the *Release Notes* folder
Sample programs and examples	*CodeWarrior Examples* folder; *The PowerPlant Book*; *PowerPlant Advanced Topics;* *Tutorials* (Apple Guide)
Problems other CodeWarrior users have solved	*Internet newsgroup [docs]* folder

Revision code 1013rdl