Part IV

The Standard Library

This part describes the C++ standard library. The aim is to provide an understanding of how to use the library, to demonstrate generally useful design and programming techniques, and to show how to extend the library in the ways in which it was intended to be extended.

Chapters

"... I am just now beginning to discover the difficulty of expressing one's ideas on paper. As long as it consists solely of description it is pretty easy; but where reasoning comes into play, to make a proper connection, a clearness & a moderate fluency, is to me, as I have said, a difficulty of which I had no idea ..."

– Charles Darwin

30

Standard-Library Overview

Many secrets of art and nature
are thought by the unlearned to be magical.
– Roger Bacon

- Introduction
  Standard-Library Facilities; Design Constraints; Description Style
- Headers
- Language Support
  `initializer_list` Support; Range-`for` Support
- Error Handling
  Exceptions; Assertions; `system_error`
- Advice

30.1 Introduction

The standard library is the set of components specified by the ISO C++ standard and shipped with identical behavior (modulo performance) by every C++ implementation. For portability and long-term maintainability, I strongly recommend using the standard library whenever feasible. Maybe you can design and implement a better alternative for your application, but:

- How easy will it be for some future maintainer to learn that alternative design?
- How likely is the alternative to be available on a yet unknown platform ten years from now?
- How likely is the alternative to be useful for future applications?
- How likely is it that your alternative will be interoperable with code written using the standard library?
- How likely is it that you can spend as much effort optimizing and testing your alternative as was done for the standard library?

And, of course, if you use an alternative, you (or your organization) will be responsible for the maintenance and evolution of the alternative ‘‘forever.’’ In general: try not to reinvent the wheel.

The standard library is rather large: its specification in the ISO C++ standard is 785 dense pages. And that is without describing the ISO C standard library, which is a part of the C++ standard library (another 139 pages). To compare, the C++ language specification is 398 pages. Here, I summarize, relying heavily on tables, and give a few examples. Details can be found elsewhere, including online copies of the standard, complete online documentation of implementations, and (if you like to read code) open source implementations. Rely on the references to the standard for complete details.

The standard-library chapters are not intended to be read in their order of presentation. Each chapter and typically each major subsection can be read in isolation. Rely on cross-references and the index if you encounter something unknown.

30.1.1 Standard-Library Facilities

What ought to be in the standard C++ library? One ideal is for a programmer to be able to find every interesting, significant, and reasonably general class, function, template, etc., in a library. However, the question here is not ''What ought to be in *some* library?'' but ''What ought to be in the *standard* library?'' ''Everything!'' is a reasonable first approximation to an answer to the former question but not to the latter. A standard library is something that every implementer must supply so that every programmer can rely on it.

The C++ standard library provides:

- Support for language features, such as memory management (§11.2), the range-**for** statement (§9.5.1), and run-time type information (§22.2)
- Information about implementation-defined aspects of the language, such as the largest finite **float** value (§40.2)
- Primitive operations that cannot be easily or efficiently implemented in the language itself, such as **is_polymorphic**, **is_scalar**, and **is_nothrow_constructible** (§35.4.1)
- Facilities for low-level (''lock-free'') concurrent programming (§41.3)
- Support for thread-based concurrency (§5.3, §42.2)
- Minimal support for task-based concurrency, such as **future** and **async()** (§42.4)
- Functions that most programmers cannot easily implement optimally and portably, such as **uninitialized_fill()** (§32.5) and **memmove()** (§43.5)
- Minimal support for (optional) reclamation of unused memory (garbage collection), such as **declare_reachable()** (§34.5)
- Nonprimitive foundational facilities that a programmer can rely on for portability, such as **list**s (§31.4), **map**s (§31.4.3), **sort()** (§32.6), and I/O streams (Chapter 38)
- Frameworks for extending the facilities it provides, such as conventions and support facilities that allow a user to provide I/O of a user-defined type in the style of I/O for built-in types (Chapter 38) and the STL (Chapter 31)

A few facilities are provided by the standard library simply because it is conventional and useful to do so. Examples are the standard mathematical functions, such as **sqrt()** (§40.3), random number generators (§40.7), **complex** arithmetic (§40.4), and regular expressions (Chapter 37).

The standard library aims to be the common foundation for other libraries. In particular, combinations of its facilities allow the standard library to play three supporting roles:

- A foundation for portability
- A set of compact and efficient components that can be used as the foundation for performance-sensitive libraries and applications
- A set of components enabling intra-library communications

The design of the library is primarily determined by these three roles. These roles are closely related. For example, portability is commonly an important design criterion for a specialized library, and common container types such as lists and maps are essential for convenient communication between separately developed libraries.

The last role is especially important from a design perspective because it helps limit the scope of the standard library and places constraints on its facilities. For example, string and list facilities are provided in the standard library. If they were not, separately developed libraries could communicate only by using built-in types. However, advanced linear algebra and graphics facilities are not provided. Such facilities are obviously widely useful, but they are rarely directly involved in communication between separately developed libraries.

Unless a facility is somehow needed to support these roles, it can be left to some library outside the standard. For good and bad, leaving something out of the standard library opens the opportunity for different libraries to offer competing realizations of an idea. Once a library proves itself widely useful in a variety of computing environments and application domains, it becomes a candidate for the standard library. The regular expression library (Chapter 37) is an example of this.

A reduced standard library is available for freestanding implementations, that is, implementations running with minimal or no operating system support (§6.1.1).

30.1.2 Design Constraints

The roles of a standard library impose several constraints on its design. The facilities offered by the C++ standard library are designed to be:

- Valuable and affordable to essentially every student and professional programmer, including the builders of other libraries.
- Used directly or indirectly by every programmer for everything within the library's scope.
- Efficient enough to provide genuine alternatives to hand-coded functions, classes, and templates in the implementation of further libraries.
- Either policy free or with an option to supply policies as arguments.
- Primitive in the mathematical sense. That is, a component that serves two weakly related roles will almost certainly suffer overhead compared to individual components designed to perform only a single role.
- Convenient, efficient, and reasonably safe for common uses.
- Complete in what they do. The standard library may leave major functions to other libraries, but if it takes on a task, it must provide enough functionality so that individual users or implementers need not replace it to get the basic job done.
- Easy to use with built-in types and operations.
- Type safe by default, and therefore in principle checkable at run time.
- Supportive of commonly accepted programming styles.
- Extensible to deal with user-defined types in ways similar to the way built-in types and standard-library types are handled.

For example, building the comparison criteria into a sort function is unacceptable because the same data can be sorted according to different criteria. This is why the C standard-library `qsort()` takes a comparison function as an argument rather than relying on something fixed, say, the `<` operator (§12.5). On the other hand, the overhead imposed by a function call for each comparison compromises `qsort()` as a building block for further library building. For almost every data type, it is easy to do a comparison without imposing the overhead of a function call.

Is that overhead serious? In most cases, probably not. However, the function call overhead can dominate the execution time for some algorithms and cause users to seek alternatives. The technique described in §25.2.3 of supplying comparison criteria through a template argument solves that problem for `sort()` and many other standard-library algorithms. The sort example illustrates the tension between efficiency and generality. It is also an example of how such tensions can be resolved. A standard library is not merely required to perform its tasks. It must also perform them so efficiently that users are not tempted to supply their own alternatives to what the standard offers. Otherwise, implementers of more advanced features are forced to bypass the standard library in order to remain competitive. This would add a burden to the library developer and seriously complicate the lives of users wanting to stay platform-independent or to use several separately developed libraries.

The requirements of ‘‘primitiveness’’ and ‘‘convenience of common uses’’ can conflict. The former requirement precludes exclusively optimizing the standard library for common cases. However, components serving common, but nonprimitive, needs can be included in the standard library in addition to the primitive facilities, rather than as replacements. The cult of orthogonality must not prevent us from making life convenient for the novice and the casual user. Nor should it cause us to leave the default behavior of a component obscure or dangerous.

30.1.3 Description Style

A full description of even a simple standard-library operation, such as a constructor or an algorithm, can take pages. Consequently, I use an extremely abbreviated style of presentation. Sets of related operations are typically presented in tables:

Some Operations	
`p=op(b,e,x)`	`op` does something to the range [`b`:`e`) and `x`, returning `p`
`foo(x)`	`foo` does something to `x` but returns no result
`bar(b,e,x)`	Does `x` have something to do with [`b`:`e`)?

I try to be mnemonic when choosing identifiers, so `b` and `e` will be iterators specifying a range, `p` a pointer or an iterator, and `x` some value, all depending on context. In this notation, only the commentary distinguishes no result from a Boolean result, so you can confuse those if you try hard enough. For an operation returning a Boolean, the explanation usually ends with a question mark. Where an algorithm follows the usual pattern of returning the end of an input sequence to indicate ‘‘failure,’’ ‘‘not found,’’ etc. (§4.5.1, §33.1.1), I do not mention that explicitly.

Usually, such an abbreviated description is accompanied with a reference to the ISO C++ standard, some further explanation, and examples.

30.2 Headers

The facilities of the standard library are defined in the `std` namespace and presented as a set of headers. The headers identify the major parts of the library. Thus, listing them gives an overview of the library.

The rest of this subsection is a list of headers grouped by function, accompanied by brief explanations and annotated by references to where they are discussed. The grouping is chosen to match the organization of the standard.

A standard header with a name starting with the letter `c` is equivalent to a header in the C standard library. For every header `<X.h>` defining part of the C standard library in the global namespace and also in namespace `std`, there is a header `<cX>` defining the same names. Ideally, the names from a `<cX>` header do not pollute the global namespace (§15.2.4), but unfortunately (due to complexities of maintaining multilanguage, multi-operating-system environments) most do.

Containers		
`<vector>`	One-dimensional resizable array	§31.4.2
`<deque>`	Double-ended queue	§31.4.2
`<forward_list>`	Singly-linked list	§31.4.2
`<list>`	Doubly-linked list	§31.4.2
`<map>`	Associative array	§31.4.3
`<set>`	Set	§31.4.3
`<unordered_map>`	Hashed associative array	§31.4.3.2
`<unordered_set>`	Hashed set	§31.4.3.2
`<queue>`	Queue	§31.5.2
`<stack>`	Stack	§31.5.1
`<array>`	One-dimensional fixed-size array	§34.2.1
`<bitset>`	Array of `bool`	§34.2.2

The associative containers `multimap` and `multiset` can be found in `<map>` and `<set>`, respectively. The `priority_queue` (§31.5.3) is declared in `<queue>`.

General Utilities		
`<utility>`	Operators and pairs	§35.5, §34.2.4.1
`<tuple>`	Tuples	§34.2.4.2
`<type_traits>`	Type traits	§35.4.1
`<typeindex>`	Use a `type_info` as a key or a hash code	§35.5.4
`<functional>`	Function objects	§33.4
`<memory>`	Resource management pointers	§34.3
`<scoped_allocator>`	Scoped allocators	§34.4.4
`<ratio>`	Compile-time rational arithmetic	§35.3
`<chrono>`	Time utilities	§35.2
`<ctime>`	C-style date and time	§43.6
`<iterator>`	Iterators and iterator support	§33.1

Iterators provide the mechanism to make standard algorithms generic (§3.4.2, §33.1.4).

Algorithms		
<algorithm>	General algorithms	§32.2
<cstdlib>	bsearch(), qsort()	§43.7

A typical general algorithm can be applied to any sequence (§3.4.2, §32.2) of any type of element. The C standard library functions bsearch() and qsort() apply to built-in arrays with elements of types without user-defined copy constructors and destructors only (§12.5).

Diagnostics		
<exception>	Exception class	§30.4.1.1
<stdexcept>	Standard exceptions	§30.4.1.1
<cassert>	Assert macro	§30.4.2
<cerrno>	C-style error handling	§13.1.2
<system_error>	System error support	§30.4.3

Assertions using exceptions are described in §13.4.

Strings and Characters		
<string>	String of T	Chapter 36
<cctype>	Character classification	§36.2.1
<cwctype>	Wide-character classification	§36.2.1
<cstring>	C-style string functions	§43.4
<cwchar>	C-style wide-character string functions	§36.2.1
<cstdlib>	C-style allocation functions	§43.5
<cuchar>	C-style multibyte characters	
<regex>	Regular expression matching	Chapter 37

The <cstring> header declares the strlen(), strcpy(), etc., family of functions. The <cstdlib> declares atof() and atoi() which convert C-style strings to numeric values.

Input/Output		
<iosfwd>	Forward declarations of I/O facilities	§38.1
<iostream>	Standard iostream objects and operations	§38.1
<ios>	iostream bases	§38.4.4
<streambuf>	Stream buffers	§38.6
<istream>	Input stream template	§38.4.1
<ostream>	Output stream template	§38.4.2
<iomanip>	Manipulators	§38.4.5.2
<sstream>	Streams to/from strings	§38.2.2
<cctype>	Character classification functions	§36.2.1
<fstream>	Streams to/from files	§38.2.1
<cstdio>	printf() family of I/O	§43.3
<cwchar>	printf()-style I/O of wide characters	§43.3

Manipulators are objects used to manipulate the state of a stream (§38.4.5.2).

Localization		
<locale>	Represent cultural differences	Chapter 39
<clocale>	Represent cultural differences C-style	
<codecvt>	Code conversion facets	§39.4.6

A locale localizes differences such as the output format for dates, the symbol used to represent currency, and string collation criteria that vary among different natural languages and cultures.

Language Support		
<limits>	Numeric limits	§40.2
<climits>	C-style numeric scalar-limit macros	§40.2
<cfloat>	C-style numeric floating-point limit macros	§40.2
<cstdint>	Standard integer type names	§43.7
<new>	Dynamic memory management	§11.2.3
<typeinfo>	Run-time type identification support	§22.5
<exception>	Exception-handling support	§30.4.1.1
<initializer_list>	initializer_list	§30.3.1
<cstddef>	C library language support	§10.3.1
<cstdarg>	Variable-length function argument lists	§12.2.4
<csetjmp>	C-style stack unwinding	
<cstdlib>	Program termination	§15.4.3
<ctime>	System clock	§43.6
<csignal>	C-style signal handling	

The <cstddef> header defines the type of values returned by sizeof(), size_t, the type of the result of pointer subtraction and of array subscripts, ptrdiff_t (§10.3.1), and the infamous NULL macro (§7.2.2).

C-style stack unwinding (using setjmp and longjmp from <csetjmp>) is incompatible with the use of destructors and with exception handling (Chapter 13, §30.4) and is best avoided. C-style stack unwinding and signals are not discussed in this book.

Numerics		
<complex>	Complex numbers and operations	§40.4
<valarray>	Numeric vectors and operations	§40.5
<numeric>	Generalized numeric operations	§40.6
<cmath>	Standard mathematical functions	§40.3
<cstdlib>	C-style random numbers	§40.7
<random>	Random number generators	§40.7

For historical reasons, abs() and div() are found in <cstdlib> rather than in <cmath> with the rest of the mathematical functions (§40.3).

Concurrency		
<atomic>	Atomic types and operations	§41.3
<condition_variable>	Waiting for an action	§42.3.4
<future>	Asynchronous task	§42.4.4
<mutex>	Mutual exclusion classes	§42.3.1
<thread>	Threads	§42.2

C provides standard-library facilities of varying relevance to C++ programmers. The C++ standard library provides access to all such facilities:

C Compatibility		
<cinttypes>	Aliases for common integer types	§43.7
<cstdbool>	C bool	
<ccomplex>	<complex>	
<cfenv>	Floating-point environment	
<cstdalign>	C alignment	
<ctgmath>	C "type generic math": <complex> and <cmath>	

The <cstdbool> header will not define macros bool, true, or false. The <cstdalign> header will not define a macro alignas. The .h equivalents to <cstdbool>, <ccomplex>, <calign>, and <ctgmath> approximate C++ facilities for C. Avoid them if you can.

The <cfenv> header provides types (such as fenv_t and fexcept_t), floating-point status flags, and control modes describing an implementation's floating-point environment.

A user or a library implementer is not allowed to add or subtract declarations from the standard headers. Nor is it acceptable to try to change the contents of a header by defining macros to change the meaning of declarations in a header (§15.2.3). Any program or implementation that plays such games does not conform to the standard, and programs that rely on such tricks are not portable. Even if they work today, the next release of any part of an implementation may break them. Avoid such trickery.

For a standard-library facility to be used, its header must be included. Writing out the relevant declarations yourself is *not* a standards-conforming alternative. The reason is that some implementations optimize compilation based on standard header inclusion, and others provide optimized implementations of standard-library facilities triggered by the headers. In general, implementers use standard headers in ways programmers cannot predict and shouldn't have to know about.

A programmer can, however, specialize utility templates, such as swap() (§35.5.2), for non-standard-library, user-defined types.

30.3 Language Support

A small but essential part of the standard library is language support, that is, facilities that must be present for a program to run because language features depend on them.

Library Supported Language Features		
<new>	new and delete	§11.2
<typeinfo>	typeid() and type_info	§22.5
<iterator>	Range-for	§30.3.2
<initializer_list>	initializer_list	§30.3.1

30.3.1 initializer_list Support

A {}-list is converted into an object of type std::initializer_list<X> according to the rules described in §11.3. In <initializer_list>, we find initializer_list:

```
template<typename T>
class initializer_list {      // §iso.18.9
public:
      using value_type = T;
      using reference = const T&;          // note const: initializer_list elements are immutable
      using const_reference = const T&;
      using size_type = size_t;
      using iterator = const T*;
      using const_iterator = const T*;

      initializer_list() noexcept;

      size_t size() const noexcept;        // number of elements
      const T*  begin() const noexcept;    // first element
      const T* end() const noexcept;       // one-past-last element
};

template<typename T>
      const T* begin(initializer_list<T> lst) noexcept { return lst.begin(); }
template<typename T>
      const T* end(initializer_list<T> lst) noexcept { return lst.end(); }
```

Unfortunately, initializer_list does not offer a subscript operator. If you want to use [] rather than *, subscript a pointer:

```
void f(initializer_list<int> lst)
{
      for(int i=0; i<lst.size(); ++i)
            cout << lst[i] << '\n';        // error

      const int* p = lst.begin();
      for(int i=0; i<lst.size(); ++i)
            cout << p[i] << '\n';          // OK
}
```

Naturally, an initializer_list can also be used by a range-for. For example:

```
void f2(initializer_list<int> lst)
{
        for (auto x : lst)
                cout << x << '\n';
}
```

30.3.2 Range-for Support

A range-for statement is mapped to a for-statement using an iterator as described in §9.5.1.

In <iterator>, the standard library provides std::begin() and std::end() functions for built-in arrays and for every type that provides member begin() and end(); see §33.3.

All standard-library containers (e.g., vector and unordered_map) and strings support iteration using range-for; container adaptors (such as stack and priority_queue) do not. The container headers, such as <vector>, include<initializer_list>, so the user rarely has to do so directly.

30.4 Error Handling

The standard library consists of components developed over a period of almost 40 years. Thus, their style and approaches to error handling are not consistent:

- C-style libraries consist of functions, many of which set errno to indicate that an error happened; see §13.1.2 and §40.3.
- Many algorithms operating on a sequence of elements return an iterator to the one-past-the-last element to indicate ‘‘not found’’ or ‘‘failure’’; see §33.1.1.
- The I/O streams library relies on a state in each stream to reflect errors and may (if the user requests it) throw exceptions to indicate errors; see §38.3.
- Some standard-library components, such as vector, string, and bitset, throw exceptions to indicate errors.

The standard library is designed so that all facilities obey ‘‘the basic guarantee’’ (§13.2); that is, even if an exception is thrown, no resource (such as memory) is leaked and no invariant for a standard-library class is broken.

30.4.1 Exceptions

Some standard-library facilities report errors by throwing exceptions:

Standard-Library Exceptions (continues)	
bitset	Throws invalid_argument, out_of_range, overflow_error
iostream	Throws ios_base::failure if exceptions are enabled
regex	Throws regex_error
string	Throws length_error, out_of_range
vector	Throws out_of_range

Standard-Library Exceptions (continued)	
new T	Throws bad_alloc if it cannot allocate memory for a T
dynamic_cast<T>(r)	Throws bad_cast if it cannot convert the reference r to a T
typeid()	Throws bad_typeid if it cannot deliver a type_info
thread	Throws system_error
call_once()	Throws system_error
mutex	Throws system_error
condition_variable	Throws system_error
async()	Throws system_error
packaged_task	Throws system_error
future and promise	Throws future_error

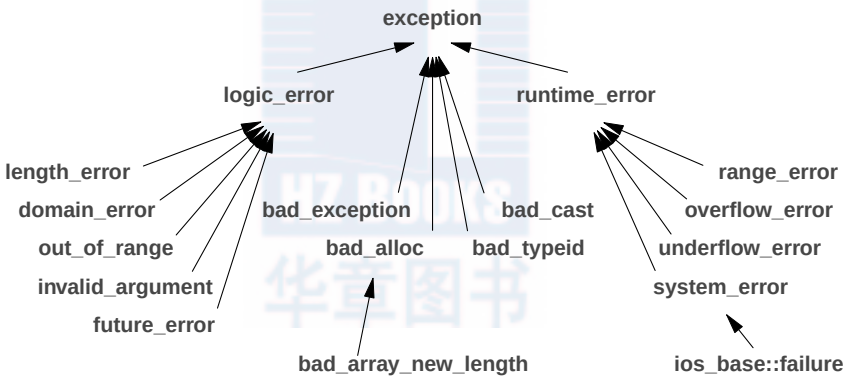
These exceptions may be encountered in any code that directly or indirectly uses these facilities. In addition, any operation that manipulates an object that may throw an exception must be assumed to throw (that exception) unless care has been taken to avoid that. For example, a packaged_task will throw an exception if the function it is required to execute throws.

Unless you know that no facility is used in a way that could throw an exception, it is a good idea to always catch one of the root classes of the standard-library exception hierarchy (such as exception) as well as any exception (...) somewhere (§13.5.2.3), for example, in main().

30.4.1.1 The Standard exception Hierarchy

Do not throw built-in types, such as int and C-style strings. Instead, throw objects of types specifically defined to be used as exceptions.

This hierarchy of standard exception classes provides a classification of exceptions:



This hierarchy attempts to provide a framework for exceptions beyond the ones defined by the standard library. Logic errors are errors that in principle could be caught either before the program starts executing or by tests of arguments to functions and constructors. Run-time errors are all other errors. The system_error is described in §30.4.3.3.

The standard-library exception hierarchy is rooted in class `exception`:

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

The `what()` function can be used to obtain a string that is supposed to indicate something about the error that caused the exception.

A programmer can define an exception by deriving from a standard-library exception like this:

```
struct My_error : runtime_error {
        My_error(int x) :runtime_error{"My_error"}, interesting_value{x} { }
        int interesting_value;
};
```

Not all exceptions are part of the standard-library `exception` hierarchy. However, all exceptions thrown by the standard library are from the `exception` hierarchy.

Unless you know that no facility is used in a way that could throw an exception, it is a good idea to somewhere catch all exceptions. For example:

```
int main()
try {
        // ...
}
catch (My_error& me) {          // a My_error happened
        // we can use me.interesting_value and me.what()
}
catch (runtime_error& re) {     // a runtine_error happened
        // we can use re.what()
}
catch (exception& e) {          // some standard-library exception happened
        // we can use e.what()
}
catch (...) {                   // Some unmentioned exception happened
        // we can do local cleanup
}
```

As for function arguments, we use references to avoid slicing (§17.5.1.4).

30.4.1.2 Exception Propagation

In `<exception>`, the standard library provides facilities for making propagation of exceptions accessible to programmers:

Exception Propagation (§iso.18.8.5)	
exception_ptr	Unspecified type used to point to exceptions
ep=current_exception()	ep is an exception_ptr to the current exception, or to no exception if there is no currently active exception; noexcept
rethrow_exception(ep)	Re-throw the exception pointed to by ep; ep's contained pointer must not be nullptr; noreturn (§12.1.7)
ep=make_exception_ptr(e)	ep is an exception_ptr to exception e; noexcept

An exception_ptr can point to any exception, not just exceptions from the exception hierarchy. Think of exception_ptr as a smart pointer (like shared_ptr) that keeps its exception alive for as long as an exception_ptr points to it. That way, we can pass an exception_pointer to an exception out of a function that caught it and re-throw elsewhere. In particular, an exception_ptr can be used to implement a re-throw of an exception in a different thread from the one in which the exception was caught. This is what promise and future (§42.4) rely on. Use of rethrow_exception() on an exception_ptr (from different threads) does not introduce a data race.

The make_exception_ptr() could be implemented as:

```
template<typename E>
exception_ptr make_exception_ptr(E e) noexcept;
try {
        throw e;
}
catch(...) {
        return current_exception();
}
```

A nested_exception is class that stores an exception_ptr obtained from a call of current_exception():

nested_exception (§iso.18.8.6)	
nested_exception ne {};	Default constructor: ne holds an exception_ptr to the current_exception(); noexcept
nested_exception ne {ne2};	Copy constructor: both ne and ne2 hold an exception_ptr to the stored exception
ne2=ne	Copy assignment: both ne and ne2 hold an exception_ptr to the stored exception
ne.~nested_exception()	Destructor; virtual
ne.rethrow_nested()	Rethrow ne's stored exception; terminate() if no exception is stored in ne; noreturn
ep=ne.nested_ptr()	ep is an exception_ptr pointing to ne's stored exception; noexcept
throw_with_nested(e)	Throw an exception of type derived from nested_exception and e's type; e must not be derived from nested_exception; noreturn
rethrow_if_nested(e)	dynamic_cast<const nested_exception&>(e).rethrow_nested(); e's type must be derived from nested_exception

The intended use of `nested_exception` is as a base class for a class used by an exception handler to pass some information about the local context of an error together with a `exception_ptr` to the exception that caused it to be called. For example:

```
struct My_error : runtime_error {
        My_error(const string&);
        // ...
};

void my_code()
{
        try {
                // ...
        }
        catch (...) {
                My_error err {"something went wrong in my_code()"};
                // ...
                throw_with_nested(err);
        }
}
```

Now `My_error` information is passed along (rethrown) together with a `nested_exception` holding an `exception_ptr` to the exception caught.

Further up the call chain, we might want to look at the nested exception:

```
void user()
{
        try {
                my_code();
        }
        catch(My_error& err) {

         // ... clear up My_error problems ...

                try {
                        rethrow_if_nested(err);    // re-throw the nested exception, if any
                }
                catch (Some_error& err2) {
                        // ... clear up Some_error problems ...
                }
        }
}
```

This assumes that we know that `some_error` might be nested with `My_error`.

An exception cannot propagate out of a `noexcept` function (§13.5.1.1).

30.4.1.3 terminate()

In `<exception>`, the standard library provides facilities for dealing with unexpected exceptions:

terminate (§iso.18.8.3, §iso.18.8.4)	
h=get_terminate()	h is the current terminate handler; noexcept
h2=set_terminate(h)	h becomes the current terminate handler; h2 is the previous terminate handler; noexcept
terminate()	Terminate the program; noreturn; noexcept
uncaught_exception()	Has an exception been thrown on the current thread and not yet been caught? noexcept

Avoid using these functions, except very occasionally set_terminate() and terminate(). A call of terminate() terminates a program by calling a terminate handler set by a call of set_terminate(). The – almost always correct – default is to immediately terminate the program. For fundamental operating system reasons, it is implementation-defined whether destructors for local objects are invoked when terminate() is called. If terminate() is invoked as the result of a noexcept violation, the system is allowed (important) optimizations that imply that the stack may even be partially unwound (§iso.15.5.1).

It is sometimes claimed that uncaught_exception() can be useful for writing destructors that behave differently depending on whether a function is exited normally or by an exception. However, uncaught_exception() is also true during stack unwinding (§13.5.1) after the initial exception has been caught. I consider uncaught_exception() too subtle for practical use.

30.4.2 Assertions

The standard provides:

Assertions (§iso.7)	
static_assert(e,s)	Evaluate e at compile time; give s as a compiler error message if !e
assert(e)	If the macro NDBUG is not defined, evaluate e at run time and if !e, write a message to cerr and abort(); if NDBUG is defined, do nothing

For example:

```
template<typename T>
void draw_all(vector<T*>& v)
{
        static_assert(Is_base_of<Shape,T>(),"non-Shape type for draw_all()");

        for (auto p : v) {
                assert(p!=nullptr);
                // ...
        }
}
```

The assert() is a macro found in <cassert>. The error message produced by assert() is implementation-defined but should contain the source file name (__FILE__), and the source line number (__LINE__) containing the assert().

Asserts are (as they should be) used more frequently in production code than in small illustrative textbook examples.

The name of the function (__func__) may also be included in the message. It can be a serious mistake to assume that the assert() is evaluated when it is not. For example, given a usual compiler setup, assert(p!=nullptr) will catch an error during debugging, but not in the final shipped product.

For a way to manage assertions, see §13.4.

30.4.3 system_error

In <system_error>, the standard library provides a framework for reporting errors from the operating system and lower-level system components. For example, we may write a function to check a file name and then open a file like this:

```
ostream& open_file(const string& path)
{
    auto dn = split_into_directory_and_name(path);          // split into {path,name}

    error_code err {does_directory_exist(dn.first)};        // ask "the system" about the path
    if (err) {    // err!=0 means error

        // ... see if anything can be done ...

        if (cannot_handle_err)
            throw system_error(err);
    }

    // ...
    return ofstream{path};
}
```

Assuming that "the system" doesn't know about C++ exceptions, we have no choice about whether to deal with error codes or not; the only questions are "where?" and "how?" In <system_error>, the standard library provides facilities for classifying error codes, for mapping system-specific error codes into more portable ones, and for mapping error codes into exceptions:

System Error Types	
error_code	Holds a value identifying an error and the category of that error; system-specific (§30.4.3.1)
error_category	A base class for types used to identify the source and encoding of a particular kind (category) of error code (§30.4.3.2)
system_error	A runtime_error exception containing an error_code (§30.4.3.3)
error_condition	Holds a value identifying an error and the category of that error; potentially portable (§30.4.3.4)
errc	enum class with enumerators for error codes from <cerrno> (§40.3); basically POSIX error codes
future_errc	enum class with enumerators for error codes from <future> (§42.4.4)
io_errc	enum class with enumerators for error codes from <ios> (§38.4.4)

30.4.3.1 Error Codes

When an error ‘‘bubbles up’’ from a lower level as an error code, we must handle the error it represents or turn it into an exception. But first we must classify it: different systems use different error codes for the same problem, and different systems simply have different kinds of errors.

	error_code (§iso.19.5.2)
error_code ec {};	Default constructor: ec={0,&generic_category}; noexcept
error_code ec {n,cat};	ec={n,cat}; cat is an error_category and n is an int representing an error in cat; noexcept
error_code ec {n};	ec={n,&generic_category}; n represents an error; n is a value of type EE for which is_error_code_enum<EE>::value==true; noexcept
ec.assign(n,cat)	ec={n,cat}; cat is an error_category; n represents an error; n is a value of type EE for which is_error_code_enum<EE>::value==true; noexcept
ec=n	ec={n,&generic_category}: ec=make_error_code(n); n represets an error; n is a value of type EE for which is_error_code_enum<EE>::value==true; noexcept
ec.clear()	ec={0,&generic_category()}; noexcept
n=ec.value()	n is ec’s stored value; noexcept
cat=ec.category()	cat is a reference to ec’s stored category; noexcept
s=ec.message()	s is a string representing ec potentially used as an error message: ec.category().message(ec.value())
bool b {ec};	Convert ec to bool; b is true if ec represents an error; that is, b==false means ‘‘no error’’; explicit
ec==ec2	Either or both of ec and ec2 can be an error_code; to compare equal ec and ec2 must have equivalent category()s and equivalent value()s; if ec and ec2 are of the same type, equivalence is defined by ==; if not, equivalence is defined by category().equivalent().
ec!=ec2	!(ec==ec2)
ec<ec2	An order ec.category()<ec2.category() || (ec.category()==ec2.category() && ec.value()<ec2.value())
e=ec.default_error_condition()	e is a reference to an error_condition: e=ec.category().default_error_condition(ec.value())
os<<ec	Write ec.name() to the ostream os
ec=make_error_code(e)	e is an errc; ec=error_code(static_cast<int>(e),&generic_category())

For a type representing the simple idea of an error code, error_code provides a lot of members. It is basically as simple map from an integer to a pointer to an error_category:

```
class error_code {
public:
    // representation: {value,category} of type {int,const error_category*}
};
```

An `error_category` is an interface to an object of a class derived from `error_category`. Therefore, an `error_category` is passed by reference and stored as a pointer. Each separate `error_category` is represented by a unique object.

Consider again the `open_file()` example:

```
ostream& open_file(const string& path)
{
    auto dn = split_into_directory_and_name(path);          // split into {path,name}

    if (error_code err {does_directory_exist(dn.first)}) {  // ask "the system" about the path
        if (err==errc::permission_denied) {
            // ...
        }
        else if (err==errc::not_a_directory) {
            // ...
        }
        throw system_error(err); // can't do anything locally
    }

    // ...
    return ofstream{path};
}
```

The `errc` error codes are described in §30.4.3.6. Note that I used an if-then-else chain rather than the more obvious `switch`-statement. The reason is that == is defined in terms of equivalence, taking both the error `category()` and the error `value()` into account.

The operations on `error_code`s are system-specific. In some cases, `error_code`s can be mapped into `error_condition`s (§30.4.3.4) using the mechanisms described in §30.4.3.5. An `error_condition` is extracted from an `error_code` using `default_error_condition()`. An `error_condition` typically contains less information than an `error_code`, so it is usually a good idea to keep the `error_code` available and only extract its `error_condition` when needed.

Manipulating `error_code`s does not change the value of `errno` (§13.1.2, §40.3). The standard library leaves the error states provided by other libraries unchanged.

30.4.3.2 Error Categories

An `error_category` represents a classification of errors. Specific errors are represented by a class derived from class `error_category`:

```
class error_category {
public:
    // ... interface to specific categories derived from error_category ...
};
```

error_category (§iso.19.5.1.1)	
cat.~error_category()	Destructor; virtual; noexcept
s=cat.name()	s is the name of cat; s is a C-style string; virtual; noexcept
ec=cat.default_error_condition(n)	ec is the error_condition for n in cat; virtual; noexcept
cat.equivalent(n,ec)	Is ec.category()==cat and ec.value()==n? ec is an error_condition; virtual; noexcept
cat.equivalent(ec,n)	Is ec.category()==cat and ec.value()==n? ec is an error_code; virtual; noexcept
s=cat.message(n)	s is a string describing the error n in cat; virtual
cat==cat2	Is cat the same category as cat2? noexcept
cat!=cat2	!(cat==cat2); noexcept
cat<cat2	Is cat<cat2 in an order based on error_category addresses: std::less<const error_category*>()(cat, cat2)? noexcept

Because error_category is designed to be used as a base class, no copy or move operations are provided. Access an error_category through pointers or references.

There are four named standard-library categories:

Standard-library Error Categories (§iso.19.5.1.1)	
ec=generic_category()	ec.name()=="generic"; ec is a reference to an error_category
ec=system_category()	ec.name()=="system" ec is a reference to an error_category; represents system errors: if ec corresponds to a POSIX error then ec.value() equals that error's errno
ec=future_category()	ec.name()=="future"; ec is a reference to an error_category; represents errors from <future>
iostream_category()	ec.name()=="iostream"; ec is a reference to an error_category; represents errors from the iostream library

These categories are necessary because a simple integer error code can have different meanings in different contexts (categorys). For example, 1 means "operation not permitted" (EPERM) in POSIX, is a generic code (state) for all errors as an iostream error, and means "future already retrieved" (future_already_retrieved) as a future error.

30.4.3.3 Exception system_error

A system_error is used to report errors that ultimately originate in the parts of the standard library that deal with the operating system. It passes along an error_code and optionally an error-message string:

```
class system_error : public runtime_error {
public:
    // ...
};
```

Exception Class system_error (§iso.19.5.6)	
system_error se {ec,s};	se holds {ec,s}; ec is an error_code; s is a string or a C-style string intended as part of an error message
system_error se {ec};	se holds {ec}; ec is an error_code
system_error se {n,cat,s};	se holds {error_code{n,cat},s}; cat is an error_category and n is an int representing an error in cat; s is a string or a C-style string intended as part of an error message
system_error se {n,cat};	se holds error_code{n,cat}; cat is an error_category and n is an int representing an error in cat
ec=se.code()	ec is a reference to se's error_code; noexcept
p=se.what()	p is a C-style string version of se's error string; noexcept

Code catching a system_error has its error_code available. For example:

```
try {
    // something
}
catch (system_error& err) {
    cout << "caught system_error " << err.what() <<'\n';      // error message

    auto ec = err.code();
    cout << "category: " << ec.category().what() <<'\n';
    cout << "value: " << ec.value() <<'\n';
    cout << "message: " << ec.message() <<'\n';
}
```

Naturally, system_errors can be used by code that is not part of the standard library. A system-specific error_code is passed, rather than a potentially portable error_condition (§30.4.3.4). To get an error_condition from an error_code use default_error_condition() (§30.4.3.1).

30.4.3.4 Potentially Portable Error Conditions

Potentially portable error codes (error_conditions) are represented almost identically to the system-specific error_codes:

```
class error_condition {    // potentially portable (§iso.19.5.3)
public:
    // like error_code but
    // no output operator (<<) and
    // no default_error_condition()
};
```

The general idea is that each system has a set of specific (''native'') codes that are mapped into the potentially portable ones for the convenience of programmers of programs (often libraries) that need to work on multiple platforms.

30.4.3.5 Mapping Error Codes

Making an error_category with a set of error_codes and at least one error_condition starts with defining an enumeration with the desired error_code values. For example:

```
enum class future_errc {
        broken_promise = 1,
        future_already_retrieved,
        promise_already_satisfied,
        no_state
};
```

The meaning of these values is completely category-specific. The integer values of these enumerators are implementation-defined.

The future error category is part of the standard, so you can find it in your standard library. The details are likely to differ from what I describe.

Next, we need to define a suitable category for our error codes:

```
class future_cat : error_category {        // to be returned from future_category()
public:
        const char* name() const noexcept override { return "future"; }

        string message(int ec) const override;
};

const error_category& future_category() noexcept
{
        static future_cat obj;
        return &obj;
}
```

The mapping from integer values to error message() strings is a bit tedious. We have to invent a set of messages that are likely to be meaningful to a programmer. Here, I'm not trying to be clever:

```
string future_error::message(int ec) const
{
        switch (ec) {
        default:                                return "bad future_error code";
        future_errc::broken_promise:            return "future_error: broken promise";
        future_errc::future_already_retrieved:  return "future_error: future already retrieved";
        future_errc::promise_already_satisfied: return "future_error: promise already satisfied";
        future_errc::no_state:                  return "future_error: no state";
        }
}
```

We can now make an error_code out of a future_errc:

```
error_code make_error_code(future_errc e) noexcept
{
        return error_code{int(e),future_category()};
}
```

For the error_code constructor and assignment that take a single error value, it is required that the argument be of the appropriate type for the error_category. For example, an argument intended to become the value() of an error_code of future_category() must be a future_errc. In particular, we can't just use any int. For example:

```
error_code ec1 {7};                          // error
error_code ec2 {future_errc::no_state};      // OK

ec1 = 9;                                     // error
ec2 = future_errc::promise_already_satisfied;  // OK
ec2 = errc::broken_pipe;                     // error: wrong error category
```

To help the implementer of error_code, we specialize the trait is_error_code_enum for our enumeration:

```
template<>
struct is_error_code_enum<future_errc> : public true_type { };
```

The standard already provides the general template:

```
template<typename>
struct is_error_code_enum : public false_type { };
```

This states that anything we don't deem an error code value isn't. For error_condition to work for our category, we must repeat what we did for error_code. For example:

```
error_condition make_error_condition(future_errc e) noexcept;

template<>
struct is_error_condition_enum<future_errc> : public true_type { };
```

For a more interesting design, we could use a separate enum for the error_condition and have make_error_condition() implement a mapping from future_errc to that.

30.4.3.6 errc Error Codes

Standard error_codes for the system_category() are defined by enum class errc with values equivalent to the POSIX-derived contents of <cerrno>:

enum class errc Enumerators (§iso.19.5) (continues)	
address_family_not_supported	EAFNOSUPPORT
address_in_use	EADDRINUSE
address_not_available	EADDRNOTAVAIL
already_connected	EISCONN
argument_list_too_long	E2BIG
argument_out_of_domain	EDOM
bad_address	EFAULT
bad_file_descriptor	EBADF
bad_message	EBADMSG

enum class errc Enumerators (§iso.19.5) (continued, continues)	
broken_pipe	EPIPE
connection_aborted	ECONNABORTED
connection_already_in_progress	EALREADY
connection_refused	ECONNREFUSED
connection_reset	ECONNRESET
cross_device_link	EXDEV
destination_address_required	EDESTADDRREQ
device_or_resource_busy	EBUSY
directory_not_empty	ENOTEMPTY
executable_format_error	ENOEXEC
file_exists	EEXIST
file_too_large	EFBIG
filename_too_long	ENAMETOOLONG
function_not_supported	ENOSYS
host_unreachable	EHOSTUNREACH
identifier_removed	EIDRM
illegal_byte_sequence	EILSEQ
inappropriate_io_control_operation	ENOTTY
interrupted	EINTR
invalid_argument	EINVAL
invalid_seek	ESPIPE
io_error	EIO
is_a_directory	EISDIR
message_size	EMSGSIZE
network_down	ENETDOWN
network_reset	ENETRESET
network_unreachable	ENETUNREACH
no_buffer_space	ENOBUFS
no_child_process	ECHILD
no_link	ENOLINK
no_lock_available	ENOLCK
no_message	ENOMSG
no_message_available	ENODATA
no_protocol_option	ENOPROTOOPT
no_space_on_device	ENOSPC
no_stream_resources	ENOSR
no_such_device	ENODEV
no_such_device_or_address	ENXIO
no_such_file_or_directory	ENOENT
no_such_process	ESRCH
not_a_directory	ENOTDIR

enum class errc **Enumerators (§iso.19.5) (continued)**	
not_a_socket	ENOTSOCK
not_a_stream	ENOSTR
not_connected	ENOTCONN
not_enough_memory	ENOMEM
not_supported	ENOTSUP
operation_canceled	ECANCELED
operation_in_progress	EINPROGRESS
operation_not_permitted	EPERM
operation_not_supported	EOPNOTSUPP
operation_would_block	EWOULDBLOCK
owner_dead	EOWNERDEAD
permission_denied	EACCES
protocol_error	EPROTO
protocol_not_supported	EPROTONOSUPPORT
read_only_file_system	EROFS
resource_deadlock_would_occur	EDEADLK
resource_unavailable_try_again	EAGAIN
result_out_of_range	ERANGE
state_not_recoverable	ENOTRECOVERABLE
stream_timeout	ETIME
text_file_busy	ETXTBSY
timed_out	ETIMEDOUT
too_many_files_open	EMFILE
too_many_files_open_in_system	ENFILE
too_many_links	EMLINK
too_many_symbolic_link_levels	ELOOP
value_too_large	EOVERFLOW
wrong_protocol_type	EPROTOTYPE

These codes are valid for the "system" category: system_category(). For systems supporting POSIX-like facilities, they are also valid for the "generic" category: generic_category().

The POSIX macros are integers whereas the errc enumerators are of type errc. For example:

```
void problem(errc e)
{
    if (e==EPIPE) {              // error: no conversion of errc to int
        // ...
    }

    if (e==broken_pipe) {        // error: broken_pipe not in scope
        // ...
    }
```

```
        if (e==errc::broken_pipe) {      // OK
            // ...
        }
    }
```

30.4.3.7 future_errc Error Codes

Standard error_codes for the future_category() are defined by enum class future_errc:

enum class future_errc Enumerators (§iso.30.6.1)	
broken_promise	1
future_already_retrieved	2
promise_already_satisfied	3
no_state	4

These codes are valid for the "future" category: future_category().

30.4.3.8 io_errc Error Codes

Standard error_codes for the iostream_category() are defined by enum class io_errc:

enum class io_errc Enumerator (§iso.27.5.1)	
stream	1

This code is valid for the "iostream" category: iostream_category().

30.5 Advice

[1] Use standard-library facilities to maintain portability; §30.1, §30.1.1.
[2] Use standard-library facilities to minimize maintenance costs; §30.1.
[3] Use standard-library facilities as a base for more extensive and more specialized libraries; §30.1.1.
[4] Use standard-library facilities as a model for flexible, widely usable software; §30.1.1.
[5] The standard-library facilities are defined in namespace std and found in standard-library headers; §30.2.
[6] A C standard-library header X.h is presented as a C++ standard-library header in <cX>; §30.2.
[7] Do not try to use a standard-library facility without #including its header; §30.2.
[8] To use a range-for on a built-in array, #include<iterator>; §30.3.2.
[9] Prefer exception-based error handling over return-code-based error handling; §30.4.
[10] Always catch exception& (for standard-library and language support exceptions) and ... (for unexpected exceptions); §30.4.1.
[11] The standard-library exception hierarchy can be (but does not have to be) used for a user's own exceptions; §30.4.1.1.
[12] Call terminate() in case of serious trouble; §30.4.1.3.

[13] Use `static_assert()` and `assert()` extensively; §30.4.2.
[14] Do not assume that `assert()` is always evaluated; §30.4.2.
[15] If you can't use exceptions, consider `<system_error>`; §30.4.3.