4.3. Error and Exception Handling

The C++ standard library is heterogeneous. It contains software from diverse sources that have different styles of design and implementation. Error and exception handling is a typical example of these differences. Parts of the library, such as string classes, support detailed error handling, checking for every possible problem that might occur and throwing an exception if there is an error. Other parts, such as the STL and valarrays, prefer speed over safety, so they rarely check for logical errors and throw exceptions only if runtime errors occur.

4.3.1. Standard Exception Classes

All exceptions thrown by the language or the library are derived from the base class `exception`, defined in `<exception>`. This class is the root of several standard exception classes, which form a hierarchy, as shown in Figure 4.1. These standard exception classes can be divided into three groups:

1. Language support
2. Logic errors
3. Runtime errors

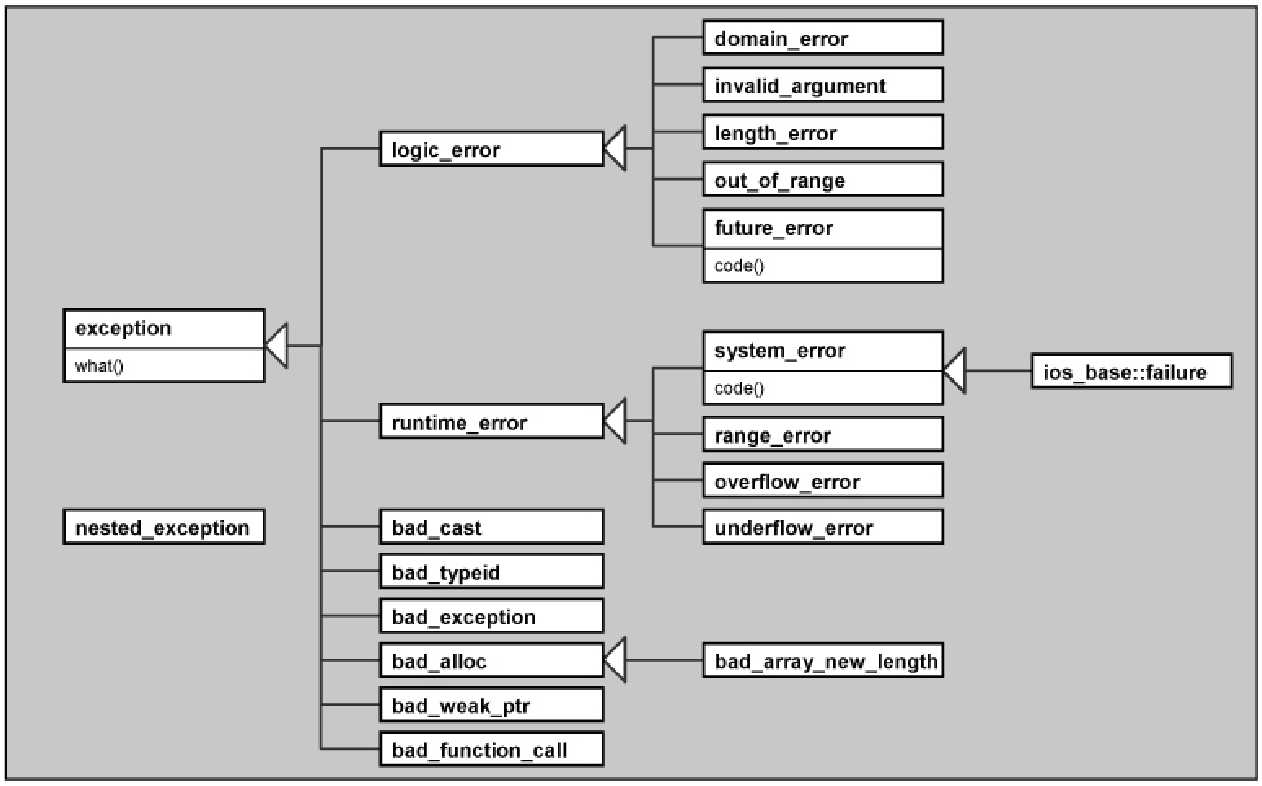

Figure 4.1. Hierarchy of Standard Exceptions

Logic errors usually can be avoided because the reason is inside the scope of a program, such as a precondition violation. Runtime exceptions are caused by a reason that is outside the scope of the program, such as not enough resources.

Exception Classes for Language Support

Exceptions for language support are used by language features. So in a way they are part of the core language rather than the library. These exceptions are thrown when the following operations fail:

- An exception of class `bad_cast`, defined in `<typeinfo>`, is thrown by the `dynamic_cast` operator if a type conversion on a reference fails at runtime.
- An exception of class `bad_typeid`, defined in `<typeinfo>`, is thrown by the `typeid` operator for runtime type identification. If the argument to `typeid` is zero or the null pointer, this exception gets thrown.
- An exception of class `bad_exception`, defined in `<exception>`, is used to handle unexpected exceptions. It can be thrown by the function `unexpected()`, which is called if a function throws an exception that is not listed in an exception specification Note, however, that the use of exception specifications is deprecated since C++11 (see Section 3.1.7, page 24).

These exceptions might also be thrown by library functions. For example, `bad_cast` might be thrown by `use_facet<>` if a facet is not available in a locale (see Section 16.2.2, page 864).

Exception Classes for Logic Errors

Exception classes for logic errors are usually derived from class `logic_error`. Logic errors are errors that, at least in theory, could be avoided by the program; for example, by performing additional tests of function arguments. Examples of such errors are a violation of logical preconditions or a class invariant. The C++ standard library provides the following classes for logic errors:

- An exception of class `invalid_argument` is used to report invalid arguments, such as when a bitset (array of bits) is initialized with a `char` other than `'0'` or `'1'`.
- An exception of class `length_error` is used to report an attempt to do something that exceeds a maximum allowable size, such as appending too many characters to a string.
- An exception of class `out_of_range` is used to report that an argument value is not in the expected range, such as when a wrong index is used in an array-like collection or string.
- An exception of class `domain_error` is used to report a domain error.
- Since C++11, an exception of class `future_error` is used to report logical errors when using asynchronous system calls (see Chapter 18). Note that runtime errors in this domain are raised via class `system_error`.

In general, classes for logic errors are defined in `<stdexcept>`. However, class `future_error` is defined in `<future>`.

Exception Classes for Runtime Errors

Exceptions derived from `runtime_error` are provided to report events that are beyond the scope of a program and are not easily avoidable. The C++ standard library provides the following classes for runtime errors:

- An exception of class `range_error` is used to report a range error in internal computations. In the C++ standard library, the exception can occur since C++11 in conversions between wide strings and byte strings (see Section 16.4.4, page 901).
- An exception of class `overflow_error` is used to report an arithmetic overflow. In the C++ standard library the exception can occur if a bitset is converted into an integral value (see Section 12.5.1, page 652).
- An exception of class `underflow_error` is used to report an arithmetic underflow.
- Since C++11, an exception of class `system_error` is used to report errors caused by the underlying operating system. In the C++ standard library, this exception can be thrown in the context of concurrency, such as class `thread`, classes to control data races, and `async()` (see Chapter 18).
- An exception of class `bad_alloc`, defined in `<new>`, is thrown whenever the global operator `new` fails, except when the `nothrow` version of `new` is used. This is probably the most important runtime exception because it might occur at any time in any nontrivial program.

  Since C++11, `bad_array_new_length`, derived from `bad_alloc`, will be thrown by `new` if the size passed to `new` is less than zero or such that the size of the allocated object would exceed the implementation-defined limit (that is, if it's a logic error rather than a runtime error).
- An exception of class `bad_weak_ptr`, definedin `<memory>`, is thrown whenever the creation of a weak pointer out of a shared pointer fails. See Section 5.2.2, page 89, for details.
- An exception of class `bad_function_call`, defined in `<functional>`, is thrown whenever a `function` wrapper object gets invoked but has no target. See Section 5.4.4, page 133, for details.

In addition, for the I/O part of the library, a special exception class called `ios_base::failure` is provided in `<ios>`. An exception of this class may be thrown when a stream changes its state due to an error or end-of-file. Since C++11, this class is derived from `system_error`; before C++11, it was directly derived from class `exception`. The exact behavior of this exception class is described in Section 15.4.4, page 762.

Conceptually, `bad_alloc` can be considered a system error. However, for historical reasons and because of its importance, implementations are encouraged to throw a `bad_alloc` rather than a `system_error` exception if an error represents an out-of-memory condition.

In general, classes for runtime errors are defined in `<stdexcept>`. Class `system_error`, however, is defined in `<system_error>`.

Exceptions Thrown by the Standard Library

As the previous description shows, almost all exception classes can be thrown by the C++ standard library. In particular, `bad_alloc` exceptions can be thrown whenever storage is allocated.

In addition, because library features might use code provided by the application programmer, functions might throw any exception indirectly.

Any implementation of the standard library might also offer additional exception classes either as siblings or as derived classes. However, the use of these nonstandard classes makes code nonportable because you could not use another implementation of the standard library without breaking your code. So, you should always use only the standard exception classes.

Header Files for Exception Classes

The exception classes are defined in many different header files. Thus, to be able to deal with all exceptions the library might throw, you have to include:

Click here to view code image

```
#include <exception>      // for classes exception and bad_exception
#include <stdexcept>      // for most logic and runtime error classes
#include <system_error>   // for system errors (since C++11)
#include <new>            // for out-of-memory exceptions
#include <ios>            // for I/O exceptions
#include <future>         // for errors with async() and futures (since C++11)
#include <typeinfo>       // for bad_cast and bad_typeid
```

4.3.2. Members of Exception Classes

To handle an exception in a `catch` clause, you may use the interface provided by the exception classes. For all classes, `what()` is provided; for some classes, `code()` also is provided.

The Member `what()`

For all standard exception classes, only one member can be used to get additional information besides the type itself: the virtual member function `what()`, which returns a null-terminated byte string:

```
namespace std {
    class exception {
      public:
        virtual const char* what() const noexcept;
        ...
    };
}
```

The content of the string returned by `what()` is implementation defined. Note that the string might be a null-terminated multibyte string that is suitable to convert and display as `wstring` (see Section 13.2.1, page 664). The C-string returned by `what()` is valid until the exception object from which it is obtained gets destroyed or a new value is assigned to the exception object.

Error Codes versus Error Conditions

For the exception classes `system_error` and `future_error`, there is an additional member to get details about the exception. However, before going into details, we have to introduce the difference between error codes and error conditions:

- **Error codes** are light-weight objects that encapsulate error code values that might be implementation-specific. However, some error codes also are standardized.
- **Error conditions** are objects that provide portable abstractions of error descriptions.

Depending on the context, for exceptions the C++ standard library sometimes specifies error code and sometimes error conditions. In fact:

- Class `std::errc` provides *error conditions* for `std::system_error` exceptions corresponding to standard system error numbers defined in `<cerrno>` or `<errno.h>`.
- Class `std::io_errc` provides an *error code* for `std::ios_base::failure` exceptions thrown by stream classes since C++11 (see Section 15.4.4, page 762).
- Class `std::future_errc` provides *error codes* for `std::future_error` exceptions thrown by the concurrency library (see Chapter 18).

Tables 4.1 and 4.2 list the error condition values that are specified by the C++ standard library for `system_error` exceptions. These are *scoped enumerators* (see Section 3.1.13, page 32), so the prefix `std::errc::` has to be used. The values of these conditions are required to have the corresponding `errno` value defined in `<cerrno>` or `<errno.h>` This is *not* the error code; the error codes usually will be implementation-specific.

Table 4.1. Error Conditions of `system_error`s, Part 1

Error Condition	Enum Value
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
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_available	ENODATA
no_message	ENOMSG
no_protocol_option	ENOPROTOOPT
no_space_on_device	ENOSPC
no_stream_resources	ENOSR
no_such_device_or_address	ENXIO
no_such_device	ENODEV
no_such_file_or_directory	ENOENT
no_such_process	ESRCH
not_a_directory	ENOTDIR
not_a_socket	ENOTSOCK

Table 4.2. Error Conditions of `system_error`s, Part 2

Error Condition	Enum Value
`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_in_system`	`ENFILE`
`too_many_files_open`	`EMFILE`
`too_many_links`	`EMLINK`
`too_many_symbolic_link_levels`	`ELOOP`
`value_too_large`	`EOVERFLOW`
`wrong_protocol_type`	`EPROTOTYPE`

Table 4.3 lists the error code values that are specified by the C++ standard library for exceptions of type `future_errc`. These are *scoped enumerators* (see Section 3.1.13, page 32), so the prefix `std::future_errc::` has to be used.[1]

[1] Note that in the C++11 standard, the error codes of future errors are defined explicitly with `future_errc::broken_promise` having the value `0`. But because error code `0` usually stands for "no error," this was a design mistake. The fix is that all future error code values are now defined to be implementation-specific.

Table 4.3. Error Codes of ***`future_error`*****s**

Error Code	Meaning
`broken_promise`	shared state abandoned
`future_already_retrieved`	`get_future()` already called
`promise_already_satisfied`	Shared state already has a value/exception or already invoked
`no_state`	No shared state

The only error code specified for `ios_base::failure` exceptions is `std::io_errc::stream`.

Dealing with Error Codes and Error Conditions

For error codes and error conditions, two different types are provided by the C++ standard library: class `std::error_code` and class `std::error_condition`. This might lead to the impression that dealing with errors is pretty complicated. However, the library is designed so that you can always compare error codes with error conditions using both the objects or enumeration values. For example, for any error object `ec` of type `std::error_code` or `std::error_condition` the following is possible:

Click here to view code image

```
   if (ec == std::errc::invalid_argument) {   // check for specific error condition
```

```
        …
    }
    if (ec == std::future_errc::no_state) {     // check for specific error code
        …
    }
```

Thus, when dealing with errors only to check for specific error codes or conditions, the difference between codes and conditions doesn't matter.

To be able to deal with error codes and error conditions, class `std::system_error` , including its derived class `std::ios_base::failure` , and class `std::_future_error` provide the additional nonvirtual member function `code()` returning an object of class `std::error_code` :[2]

```
namespace std {
    class system_error : public runtime_error {
      public:
        virtual const char* what() const noexcept;
        const error_code& code() const noexcept;
        …
    };

    class future_error : public logic_error {
      public:
        virtual const char* what() const noexcept;
        const error_code& code() const noexcept;
        …
    };
}
```

[2] Strictly speaking, these declarations are in different header files, and `what()` is not declared as virtual here but derives its virtuality from its base class.

Class `error_code` then provides member functions to get some details of the error:

Click here to view code image

```
namespace std {
    class error_code {
      public:
        const error_category& category() const noexcept;
        int                   value() const noexcept;
        string                message() const;
        explicit operator bool() const noexcept;
        error_condition       default_error_condition() const noexcept;
        …
    };
}
```

This interface is driven by the following design:

- Different libraries might use the same integral values for different error codes. So, each error has a category and a value. Only inside a category is each value distinct and has a clear specified meaning.
- `message()` yields a corresponding message, which usually is part of what `what()` yields in general for all exceptions, although this is not required.
- `operator bool()` yields whether an error code is set (`0` is the value that stands for "no error"). When exceptions are caught, this operator usually should yield `true` .
- `default_error_condition()` returns the corresponding `error_condition` , again providing `category()` , `value()` , `message()` , and `operator bool()` :

Click here to view code image

```
namespace std {
    class error_condition {
      public:
        const error_category& category() const noexcept;
        int                   value() const noexcept;
        string                message() const;
        explicit operator bool() const noexcept;
        …
    };
}
```

Class `std::error_category` provides the following interface:

Click here to view code image

```
namespace std {
    class error_category {
      public:
        virtual const char*      name() const noexcept = 0;
        virtual string           message (int ev) const = 0;
        virtual error_condition default_error_condition (int ev)
                                                     const noexcept;
        bool operator == (const error_category& rhs) const noexcept;
        bool operator != (const error_category& rhs) const noexcept;
        ...
    };
}
```

Here, `name()` yields the name of the category. `message()` and `default_error_condition()` return the message and the default error condition according to the passed value (this is what the corresponding `error_code` member functions call). Operators `==` and `!=` allow you to compare error categories.

The following category names are defined by the C++ standard library:

- `"iostream"` for I/O stream exceptions of type `ios_base::failure`
- `"generic"` for system exceptions of type `system_error`, where the value corresponds to a POSIX errno value
- `"system"` for system exceptions of type `system_error`, where the value does not correspond to a POSIX errno value
- `"future"` for exceptions of type `future_error`

For each category, global functions are provided that return the category:[3]

```
const error_category& generic_category() noexcept;     // in
<system_error>
const error_category& system_category() noexcept;      // in <system_error>
const error_category& iostream_category();             // in <ios>
const error_category& future_category() noexcept;      // in <future>
```

[3] It's probably an oversight that `iostream_category()` is not declared with `noexcept`.

Thus, for an error code object `e`, you can also call the following to find out whether it is an I/O failure:

```
if (e.code().category() == std::iostream_category())
```

The following code demonstrates how to use a generic function to process (here, print) different exceptions:

Click here to view code image

```
// util/exception.hpp

#include <exception>
#include <system_error>
#include <future>
#include <iostream>

template <typename T>
void processCodeException (const T& e)
{
    using namespace std;
    auto c = e.code();
    cerr << "- category:     " << c.category().name() << endl;
    cerr << "- value:        " << c.value() << endl;
    cerr << "- msg:          " << c.message() << endl;
    cerr << "- def category: "
         << c.default_error_condition().category().name() << endl;
    cerr << "- def value:    "
         << c.default_error_condition().value() << endl;
    cerr << "- def msg:      "
         << c.default_error_condition().message() << endl;
}

void processException()
{
    using namespace std;
    try {
        throw;  // rethrow exception to deal with it here
    }
    catch (const ios_base::failure& e) {
        cerr << "I/O EXCEPTION: " << e.what() << endl;
        processCodeException(e);
    }
```

```
        catch (const system_error& e) {
            cerr << "SYSTEM EXCEPTION: " << e.what() << endl;
            processCodeException(e);
        }
        catch (const future_error& e) {
            cerr << "FUTURE EXCEPTION: " << e.what() << endl;
            processCodeException(e);
        }
        catch (const bad_alloc& e) {
            cerr << "BAD ALLOC EXCEPTION: " << e.what() << endl;
        }
        catch (const exception& e) {
            cerr << "EXCEPTION: " << e.what() << endl;
        }
        catch (...) {
            cerr << "EXCEPTION (unknown)" << endl;
        }
    }
```

This allows to handle exceptions as follows:

```
    try {
        …
    }
    catch (...) {
        processException();
    }
```

Other Members

The remaining members of the standard exception classes create, copy, assign, and destroy exception objects.

Note that besides `what()` and `code()`, for any of the standard exception classes, no additional member is provided that describes the kind of exception. For example, there is no portable way to find out the context of an exception or the faulty index of a range error. Thus, a portable evaluation of an exception could print only the message returned from `what()`:

```
    try {
        …
    }
    catch (const std::exception& error) {
        //print implementation-defined error message
        std::cerr << error.what() << std::endl;
        …
    }
```

The only other possible evaluation might be an interpretation of the exact type of the exception. For example, when a `bad_alloc` exception is thrown, a program might try to get more memory.

4.3.3. Passing Exceptions with Class `exception_ptr`

Since C++11, the C++ standard library provides the ability to store exceptions into objects of type `exception_ptr` to process them later or in other contexts:

[Click here to view code image]

```
    #include <exception>

    std::exception_ptr eptr;      //object to hold exceptions (or nullptr)

    void foo ()
    {
        try {
            throw  ...;
        }
        catch (...) {
            eptr = std::current_exception();     //save exception for later
    processing
        }
    }

    void bar ()
    {
        if (eptr != nullptr) {
            std::rethrow_exception(eptr);        //process saved exception
        }
    }
```

`current_exception()` returns an `exception_ptr` object that refers to the currently handled exception. The value returned by `current_exception()` is valid as long as an `exception_ptr` refers to it.

`rethrow_exception()` rethrows the stored exception so that `bar()` behaves as the initial exception thrown in `foo()` would have occured inside `bar()` .

This feature is especially useful to pass exception between threads (see Section 18.2.1, page 964).

4.3.4. Throwing Standard Exceptions

You can throw standard exceptions inside your own library or program. All logic error and runtime error standard exception classes that provide the `what()` interface have only a constructor for `std::string` and (since C++11) for `const char*` . The value passed here will become the description returned by `what()` . For example, the class `logic_error` is defined as follows:

Click here to view code image

```
namespace std {
    class logic_error : public exception {
      public:
        explicit logic_error (const string& whatString);
        explicit logic_error (const char*   whatString);    // since C++11
        ...
    };
}
```

Class `std::system_error` provides the ability to create an exception object by passing an error code, a `what()` string, and an optional category:

Click here to view code image

```
namespace std {
    class system_error : public runtime_error {
      public:
        system_error (error_code ec, const string& what_arg);
        system_error (error_code ec, const char* what_arg);
        system_error (error_code ec);
        system_error (int ev, const error_category& ecat,
                      const string& what_arg);
        system_error (int ev, const error_category& ecat,
                      const char* what_arg);
        ...
    };
}
```

To provide an `error_code` object, `make_error_code()` convenience functions are provided that take only the error code value.

Class `std::ios_base::failure` provides constructors taking a `what()` string and (since C++11) an optional `error_code` object. Class `std::future_error` provides only a constructor taking a single `error_code` object.

Thus, throwing a standard exception is pretty easy:

Click here to view code image

```
throw std::out_of_range ("out_of_range (somewhere, somehow)");

throw
  std::system_error (std::make_error_code(std::errc::invalid_argument),
                     "argument ... is not valid");
```

You can't throw exceptions of the base class `exception` and any exception class that is provided for language support (`bad_cast` , `bad_typeid` , `bad_exception`).

4.3.5. Deriving from Standard Exception Classes

Another possibility for using the standard exception classes in your code is to define a special exception class derived directly or indirectly from class `exception` . To do this, you must ensure that the `what()` mechanism or `code()` mechanism works, which is possible because `what()` is virtual. For an example, see class `Stack` in Section 12.1.3, page 635.