Exception handling

- provides a way to transfer control and information from a point in the execution to an exception handler
- a handler can be invoked by a throw expression in the handler's try block, or thrown in a function called from the try block
- *try block – function try block – exception specification – throw expression – handler – exception declaration*

```
…
try
{
   fun();
}
catch (const E& e) {
   …
}
…
```

```
void fun()
try
{
   if (disaster) throw E{};
}
catch (const E& e) {
   …
   throw; // rethrows e
}
```

```
C::C()                // ctor
try : member(…)
{
…
}
catch (const E& e) {
   …
   // e re-thrown implicitly!
}
```

- a throw expression has type **void**
  - initializes a temporary object – the *exception object*
  - in a handler, a simple throw expression rethrows the caught exception object
  - in a function try block of a constructor or a destructor the caught exception object is always implicitly rethrown

Handling an exception

- handlers of a try block are tried in order of appearance
  - makes it possible to write handlers that can never be executed:

```
try {
   …
}
catch (const exception& e) {      // catches also subtype logic_error
   …
}
catch (const logic_error& e) {
   …                                // dead code
}
```

- if no match is found among the handlers of at try block, the search continues in a dynamically surrounding try block
  - if no handler is found, the program calls `terminate()` which in turn calls `abort()`
- `...` (ellipsis) in a handler's exception declaration specifies a match for any exception
  - must be the last handler for a try block, if used

```
catch (...) {  // "catch-all handler"
   …
}
```

- an exception is considered handled upon entry to a handler
  - the stack will be unwound at that point

Exception specifications

Two forms, one new and one deprecated.

- *noexcept specification*

```
void fun() noexcept(false);

void fun() noexcept(true);

void fun() noexcept;            // equivalent to noexcept(true)
```

  - the expression supplied to **noexcept** shall be a *constant expression* convertible to **bool**
  - if a **noexcept**(**true**) specification is violated the program will call std::terminate()
- *dynamic exception specification* – deprecated in C++11 – don't use
  - dynamic exception specifications are handled dynamically – no static checks – code size may increase

```
void fun() throw (range_error, length_error);

void fun() throw ();       // does not throw…
```

  - if violated the program will call std::terminate()

```
void fun() throw (range_error, length_error, bad_exception);
```

  - any exception not in the exception specification will be replaced with std::bad_exception

Special functions used by the exception handling mechanism

```
[[noreturn]] void terminate() noexcept;
```

- called when exception handling must be abandoned for other error handling techniques; calls a *terminate handler function*
- the default terminate handler function can be replaced by calling `set_terminate(`*my_terminate_handler*`)`
- the attribute `[[noreturn]]` specifies that a function does not return

```
[[noreturn]] void abort() noexcept;
```

- is not directly associated with exception handling but called by the default terminate handler function.
- terminates the program without executing destructors for object of automatic, thread or static storage duration

There is more…

Standard exceptions

				e.g thrown by/when:
exception				
	logic_error			
		domain_error		*illegal functions values*
		invalid_argument		`bitset` *constructor*
		length_error		*object length is exceeded*
		out_of_range		`at()`
		future_error		*functions in the thread library*
	runtime_error			
		range_error		*certain computations*
		overflow_error		`bitset::to_long()`
		underflow_error		*certain computations*
		system_error		*system related functions*
			ios_base::failure	`ios_base::clear()`
	bad_typeid			**typeid**
	bad_cast			**dynamic_cast**
	bad_weak_pointer			`std::shared_ptr` *constructors*
	bad_exception			*exception specification*
	bad_function_call			`std::function::operator()`
	bad_alloc			**new**
		bad_array_new_length		**new**[]

Class exception

Base class for all standard exceptions.

```
class exception
{
public:
   exception() noexcept;

   exception(const exception&) noexcept;

   virtual ~exception() noexcept;

   exception& operator=(const exception&) noexcept;

   virtual const char* what() const noexcept;
};
```

- `what()` returns an implementation-dependent message
  - subclasses carry their own messages, supplied when thrown

Class logic_error

Example of a typical direct subclass to exception.

```
class logic_error : public exception
{
public:
   explicit logic_error(const string& what_arg) noexcept
      : msg_{ what_arg } {}

   explicit logic_error(const char* what_arg) noexcept
      : msg_{ what_arg } {}

   virtual const char* what() const noexcept { return str_.data(); }

private:
   string  msg_;
};
```

- copy constructor and copy assignment operator are compiler generated
- logic_error, runtime_error, and their subclasses are suitable to derive your own exception classes from

Class length_error

Example of a typical second level subclass to exception – a “concrete” exception class.

```
class length_error : public logic_error
{
public:
   explicit length_error(const string& what_arg) noexcept
      : logic_error{ what_arg } {}

   explicit length_error(const char* what_arg) noexcept
      : logic_error{ what_arg } {}
};
```

- all functionality is inherited from `logic_error`, only two constructors need to be defined
- the standard exception hierarchy can easily be extended with user defined exceptions like length_error and alike

Streams and exceptions

When an I/O failure occur, a stream by default silently “freezes”.

- the stream position locks on the faulty position
- the operation fails – returns **false** if it’s a bool returning operation
  - all following reads will also fail
- the occurrence of the failure may not be obvious

As an alternative, a stream can be set to throw an exception, e.g.

```
cin.exceptions(ios::eofbit);

clog.exceptions(ios::badbit | ios::failbit);
```

The exception thrown will be `ios::failure`, a subtype to `exception::runtime_error::system_error`

```
try
{
   cin >> x;
}
catch (const ios::failure& e)
{
   cout << e.what() << '\n';       // will tell about the cause to the read failure
}
```


Runtime assertions

```
#include <cassert>

void print(int* p)
{
   assert(p != nullptr);
   cout << *p;
}
```

- if the expression is false, a message is printed and abort() is called
- the message shall include
  - the expression whose assertion failed,
  - the name of the source file, and
  - the line number where it happened, usually

    `Assertion failed:` *expression*`, file` *filename*`, line` *line number*

- assert is designed to capture programming errors, not user or running errors
  - generally disabled after a program exits its debugging phase
- assertion checks are disabled if the macro NDEBUG is defined when <cassert> is included

```
#define NDEBUG 1
#include <cassert>

g++ -DNDEBUG
```


Static assertions

```
static_assert(sizeof(long) >= 8, "64-bit code generation required");
```

- a constant expression that can be contextually converted to **bool**, and a string literal
  - if the expression is **true** the declaration has no effect
  - if the expression is **false** the resulting diagnostic message shall include the string literal
- type traits for inquiring about type properties and type relations is a new interesting possibility in C++11

```
static_assert(std::numeric_limits<T>::is_integer, "T must be an integer!");
```

  - T is supposed to be a template type parameter

```
static_assert(std::is_same<std::result_of(fun())>::type, short, "Error!");
```


Recommendations for error handling and exceptions (1)

- design and write error-safe code
  - give the strongest safety guarantee which is reasonable in each case
  - *The Basic Guarantee:* ensure that errors always leave your program in a valid state
  - *The Strong Guarantee:* prefer to additionally guarantee that the final state is either the original state or the intended target state
  - *The No-Fail Guarantee:* prefer to additionally guarantee that the operation can never fail
- prefer to use exceptions over error codes to report errors
  - exceptions can’t be silently ignored
  - exceptions propagate automatically
  - exception handling removes error handling and recovery from the main line of control
  - exception handling is better than the alternatives to report errors from constructors and destructors
- use assertions to document assumptions internal to a module
  - don’t use runtime assertions to report run-time errors

Recommendations for error handling and exceptions (2)

- be careful with exception specifications
  - when violated they terminate your program
  - can causes the compiler to inject additional run-time overhead in the form of implicit try/catch blocks to enforce via run-time checking that a function only emit listed exceptions
  - in general one can *not* write useful exception specifications for template functions
  - writing exception specifications for virtual functions forces overridings to have compatible specifications
- throw by value – catch by reference
  - don’t throw by pointer (copying Java syntax)
  - catch by reference (usually to **`const`**) to avoid copying and destruction
  - catch by reference to preserve polymorphism
  - when rethrowing an exception `E`, prefer **`throw`** instead of **`throw`** `E`
- exceptions are well suited for communicating errors between independently developed program parts
  - e.g. library components should report errors by throwing exceptions
  - use other error handling techniques when appropriate, e.g. for dealing with local errors

