Chapter 1

Safe Features

ch-safe
sec-safe-cpp11 Intro text should be here.

Chapter 1 Safe Features

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

Asking if an Expression Cannot `throw`

w-(noexcept operator)

The **noexcept** operator provides a standard programmatic means of querying, at *compile time*, whether a given expression — typically involving a function call — can be relied upon never to emit a C++ exception.

Description

description

Some operations can choose significantly more efficient algorithms (e.g., by an order of magnitude) if they can identify that the expressions they will use will never emit exceptions.[1] The ability to query an expression for this property facilitates **generic programming**, especially with respect to *move* operations.

C++11 introduces a compile-time operator, **noexcept**, that, when applied to an arbitrary expression, evaluates to **true** if and only if *no* **potentially evaluated** subexpression within that expression *is allowed to* emit a C++ exception:

```
static_assert(noexcept(0), "");  // OK, 0 doesn't throw.
```

The **noexcept** operator is intentionally conservative in that the compiler must consider any potentially evaluated subexpression of its operand. Consider a **ternary operator** with a compile-time constant as its **conditional expression**:

```
static_assert(noexcept(1 ? throw : 0), "");  // Error, throw throws.
```

Clearly, this expression will always throw, so it is unsurprising that **noexcept** informs us of that. Perhaps counterintuitively, however, the similar expression that would never evaluate the **throw** at run time is also identified by the **noexcept** operator as potentially throwing:

```
static_assert(noexcept(0 ? throw : 0), "");  // Error, throw throws.
```

This inspection of subexpressions does not, thankfully, extend to **unevaluated operands** within the expression, such as those that are arguments of the **sizeof** operator:

```
static_assert(noexcept(sizeof(throw, 1)), "");  // OK
```

In other words, if *any* individual potentially evaluated subexpression is capable of throwing, then the entire expression *must* be reported as potentially throwing.

Operator produced exceptions

r-produced-exceptions

In addition to functions (see *Introducing* **noexcept** *exception specifications for functions* on page 8), certain operators in classical C++ have edge cases that might throw. Consider the familiar **new** operator[2] used to allocate dynamic memory. The **contract** for **new** states that it will either allocate and return a pointer to the requested integral number of bytes or throw an

[1] Note that the **noexcept** operator does not appertain to operating-system-level signals such as a floating-point exception, segmentation fault, and so on.

[2] Note that when we say "the **new** operator," we are taking about the C++ language construct that first calls the underlying function known as global `::operator new` and then invokes the appropriate constructor. The terminology is analogous to other operators such as the infix `+` operator, which in turn calls the appropriate, overloaded `operator+` function, passing it the infix operator's `lhs` and `rhs` arguments.

`std::bad_alloc` exception defined in **<new>**. There is an overload of the underlying global **operator new** — and also the corresponding **operator new[]** — that takes an argument of type `std::nothrow_t` defined in **<new>**. This overload of **new** ignores the value of its `std::nothrow_t` argument but, instead of throwing on allocation failure, returns a *null* address value:

```
#include <new>  // std::nothrow
char* cp1 = new               char[1000LL*1000*1000*1000];  // might throw
char* cp2 = new(std::nothrow) char[1000LL*1000*1000*1000];  // will not throw

static_assert(noexcept(new               int[1000]), "");   // Error, can throw
static_assert(noexcept(new(std::nothrow) int[1000]), "");   // OK, cannot throw
```

Invoking a **dynamic_cast** on a reference (but not a pointer) to a **polymorphic type** will result in a runtime exception of type `std::bad_cast` if the referent is not of a class type that is publicly and unambiguously derived from the target type of the cast operation:

```
struct B
{
   virtual ~B() {}
};

struct BB
{
   virtual ~BB() {}
};

struct D1 : B     { };  // one base class
struct D2 : B, BB { };  // two base classes

D1 x;
D2 y;

B& bx = x;  // reference to B base class
B& by = y;  // another reference to B base class

D1& d1x = dynamic_cast<D1&>(bx);    // OK
D1& d1y = dynamic_cast<D1&>(by);    // throws std::bad_cast
D2& d2x = dynamic_cast<D2&>(bx);    // throws std::bad_cast
D2& d2y = dynamic_cast<D2&>(by);    // OK

BB& b1 = dynamic_cast<BB&>(bx);     // throws std::bad_cast
BB& b2 = dynamic_cast<BB&>(by);     // OK

// dynamic_cast to a pointer never throws
B*  bp   = 0;
D1* dp   = dynamic_cast<D1*>( bp);  // dp == nullptr
D1* d1xp = dynamic_cast<D1*>(&bx);  // OK
D1* d1yp = dynamic_cast<D1*>(&by);  // d1yp == nullptr
D2* d2xp = dynamic_cast<D2*>(&bx);  // d2xp == nullptr
```

```
D2* d2yp = dynamic_cast<D2*>(&by);  // OK

BB* b1p = dynamic_cast<BB*>(&bx);   // b1p == nullptr
BB* b2p = dynamic_cast<BB*>(&by);   // OK
```

Observe that for `b2`, although `by` is a reference to `B&` that has no apparent relationship with class `BB`, the object that it refers to, `y`, is of class `D2` that is derived from *both* classes `B` and `BB`, which are both polymorphic types, so the dynamic cast succeeds at run time, where other strictly compile-time casts would fail.

Because it is possible for a **`dynamic_cast`** to throw `std::bad_cast`, which is defined in `<typeinfo>`, on a reference but not a pointer, the **`noexcept`** operator discriminates between these two kinds of expressions:

```
static_assert( noexcept(dynamic_cast<D1*>(bp)), "");  // OK, never throws
static_assert(!noexcept(dynamic_cast<D1&>(bx)), "");  // OK, can throw

static_assert( noexcept(dynamic_cast<D2*>(bp)), "");  // OK, never throws
static_assert(!noexcept(dynamic_cast<D2&>(bx)), "");  // OK, can throw
```

In the example above, when the dynamic cast fails on a pointer, a *null* pointer value is returned. When the dynamic cast fails on a reference, however, the only other option would be to return a *null* reference, which is not allowed by the language, so throwing an exception is the only reasonable way of indicating a failure to the caller.

Runtime type identification (**RTTI**) exhibits similar behavior with respect to reference types. The **`typeid`** operator returns a **`const`** *lvalue* reference to an `std::type_info` object. If **`typeid`** queries a reference, it returns a reference to the **`type_info`** of the referenced object, and if the reference is to a polymorphic class, it returns a reference to the **`type_info`** of the complete object that is queried rather than to the base-class subobject. A special rule allows for dereferencing a null pointer, which otherwise has undefined behavior, as the target of a **`typeid`** query. As such a dereference would involve a runtime query of the **vtable** if the declared type of the pointer points to a polymorphic class, invoking **`typeid`** on a null pointer throws a `std::bad_cast` exception. Note that this is true even if the pointer refers to a nonpolymorphic class. Hence, invoking **`noexcept`** on the **`typeid`** operator will return **`false`** if the target is a pointer and **`true`** otherwise:

```
#include <typeinfo>  // typeid, std::typeinfo, std::bad_typeid

class B { virtual ~B() { } };
class C { };

B* bp = 0;
C* cp = 0;

static_assert(noexcept(typeid( cp)), "");  // OK, returns valid type_info
static_assert(noexcept(typeid( bp)), "");  // OK, a null pointer is a valid type.
static_assert(noexcept(typeid(*cp)), "");  // OK, never a need to evaluate *cp
static_assert(noexcept(typeid(*bp)), "");  // Error, can throw std::bad_typeid
```

fications-for-functions

Deprecated, dynamic exception specifications for functions

Classic C++ provided what has now been renamed **dynamic exception specifications**, which could be used to decorate a function with the types of exception objects that a function was permitted to throw:

```
int f();                    // Function f may throw anything.
int g() throw(const char*); // Function g may throw a string literal.
int h() throw();            // Function h is not allowed to throw anything.
```

The **noexcept** operator looks *only* at the function declaration:

```
static_assert(noexcept(f()) == false, "");  // may throw anything
static_assert(noexcept(g()) == false, "");  // may throw a const char*
static_assert(noexcept(h()),  "");  // may not throw anything
```

Providing a *dynamic* exception specification does not prevent a function from *attempting* to throw or rethrow a caught exception object. When an exception is thrown from within a function, the runtime system automatically checks to see if that function has an associated dynamic exception specification and, if so, looks up the type of the thrown exception. If the type of the thrown exception is listed, the exception is allowed to propagate outside of the function body; otherwise, `[[noreturn]] void std::unexpected()` is invoked, which calls `std::terminate` unless a user-supplied handler exits the program first. `[[noreturn]]` is an attribute that indicates the function will not return but may throw; see Section 1.1."**??**" on page **??** and Section 1.1."**??**" on page **??**:

```
void f0()                   { throw 5;   }  // throws int
void f1()                   { throw 5.0; }  // throws double

void f2() throw(int)        { throw 5;   }  // throws int
void f3() throw(int)        { throw 5.0; }  // calls std::unexpected()

void f4() throw(double)     { throw 5;   }  // calls std::unexpected()
void f5() throw(double)     { throw 5.0; }  // throws double

void f6() throw(int, double) { throw 5;   }  // throws int
void f7() throw(int, double) { throw 5.0; }  // throws double

void f8() throw()           { throw 5;   }  // calls std::unexpected()
void f9() throw()           { throw 5.0; }  // calls std::unexpected()
```

The **noexcept** operator is unconcerned with the type of exceptions that might be thrown, reporting back only as to whether an exception of *any* type may escape the body of the function:

```
static_assert(noexcept(f0()) == false, "");  // doesn't say it doesn't throw
static_assert(noexcept(f1()) == false, "");  //   "      "   "    "      "

static_assert(noexcept(f2()) == false, "");  // f2 may throw an int.
static_assert(noexcept(f3()) == false, "");  // f3  "    "    "   "

static_assert(noexcept(f4()) == false, "");  // f4 may throw a double.
```

```
static_assert(noexcept(f5()) == false, "");  // f5  "    "   "    "

static_assert(noexcept(f6()) == false, "");  // f6 may throw int or double.
static_assert(noexcept(f7()) == false, "");  // f7  "    "     "    "    "

static_assert(noexcept(f8()), "");   // f8 may not throw.
static_assert(noexcept(f9()), "");   // f9  "    "     "
```

There are, however, practical drawbacks to dynamic exception specifications.

1. *Brittle* — These classic, fine-grained exception specifications attempt to provide excessively detailed information that is not programmatically useful and is subject to frequent changes due to otherwise inconsequential updates to the implementation.

2. *Expensive* — When an exception is thrown, a *dynamic*-exception list must be searched at run time to determine if that specific exception type is allowed.

3. *Disruptive* — When an exception reaches a *dynamic*-exception specification, the stack must be unwound, whether or not the exception is permitted by that specification, losing useful stack-trace information if the program is about to terminate.

These deficiencies proved, over time, to be insurmountable, and dynamic exception specifications other than **`throw()`** were largely unused in practice.

As of C++11, dynamic exception specifications are officially deprecated[3] in favor of the more streamlined **`noexcept`** specifier (see Section 3.1."**??**" on page **??**), which we introduce briefly in the next section.

Introducing `noexcept` exception specifications for functions

cations-for-functions

C++11 introduces an alternative exception-specification mechanism for arbitrary functions, member functions, and lambda expressions (see ."**??**" on page **??** [**AUs: there is no feature called lambda expressions; what did you intend?**]):

```
void f() noexcept(expr);  // expr is a boolean constant expression.
void f() noexcept;        // same as void f() noexcept(true)
```

Instead of specifying a *list* of exceptions that may be thrown, whether *any* exception may be thrown is specified. As with C++03, no annotation is the equivalent of saying anything might be thrown:

```
#include <exception>  // std::bad_exception
// old (C++03)                          // modern (C++11) equivalent

void f0();                              void g0();
void f1() throw();                      void g1() noexcept;
void f2() throw(std::bad_exception);    void g2() noexcept(false);
void f3() throw(int, double);           void g3() noexcept(false);
```

[3] C++17 removes all dynamic exception specifications other than **`throw()`**, which becomes a synonym for **`noexcept`** before it too is removed by C++20.

```
static_assert(noexcept(f0()), "");  // Error, f0() defaults to throwing.
static_assert(noexcept(g0()), "");  // Error, g0()     "     "     "
static_assert(noexcept(f1()), "");  // OK,    f1() claims not to throw.
static_assert(noexcept(g1()), "");  // OK,    g1()    "    "   "   "
static_assert(noexcept(f2()), "");  // Error, f2() claims it can throw.
static_assert(noexcept(g2()), "");  // Error, g2()    "    "  "    "
static_assert(noexcept(f3()), "");  // Error, f3()    "    "  "    "
static_assert(noexcept(g3()), "");  // Error, g3()    "    "  "    "
```

The principle advantage of this approach is analogous to that of the Unix return-status convention of returning 0 on success and a non-zero value otherwise. This convention leverages the realization that there can be many ways to fail, but typically only one way to succeed. By consistently returning 0 on success, we enable an easy and efficient, *uniform* way to discriminate programmatically between typically two primary code paths corresponding, respectively, to success and failure:

```
void func(/*...*/)
{
    int status = doSomething(/*...*/);

    if (0 != status)  // Quickly check that it didn't fail.
    {
        // failure branch: handle, return, abort, etc.
    }

    // All good; Continue on the good path.

    // ...
}
```

Much like a return status, generic libraries might need to choose from among just two algorithms having substantially different performance characteristics based solely on whether a given operation can be depended upon never to throw. By distilling the details of dynamic exception specifications down to a simple binary **noexcept** specification, we sidestep most of the brittleness while preserving the ability to query programmatically for the essential information:

```
template <typename T>
void doSomething(T t)
{
    if (noexcept(t.someFunction()))
    {
        // Use a faster algorithm that assumes no exception will be thrown.
    }
    else
    {
        // Use a slower algorithm that can handle a thrown exception.
    }
}
```

Notice that the primary, compile-time branch in the example above depends on *only* whether it can be reliably assumed that calling `someFunction` on an object of type `T` will never throw anything.

Although syntactically trivial, safe and effective use of the **noexcept** specifier fairly deserves substantial elaboration; see Section 3.1."??" on page ??.

eption-specifications

Compatibility of dynamic and `noexcept` exception specifications

Dynamic exception specifications are required to unwind the program stack — a.k.a. **stack unwinding** — when an unexpected exception is encountered, whereas that behavior is left unspecified for violations of **noexcept** specifications:

```
#include <iostream>  // std::cout

struct S { ~S() { std::cout << "Unwound!" << std::endl; } };

int f() throw()  { S s; throw 0; }  // ~S is invoked.
int g() noexcept { S s; throw 0; }  // ~S may be invoked.
```

As of C++11, the *syntactic* meaning of **throw()** was made identical to **noexcept** and, hence, can coexist on declarations within the same translation unit:

```
int f() throw();    // OK
int f() noexcept;   // OK, redeclaration of same syntactic entity
```

If we now *define* the function, `f`, the nature of the stack unwinding behavior is left unspecified by the Standard.[4]

cial-member-functions

Compiler-generated special member functions

In C++11, exception specification on implicitly declared special member functions is still defined in terms of dynamic exception specification; this purely theoretical distinction is, however, unobservable via the **noexcept** operator. We will therefore ignore this distinction and speak only in terms of the observable binary property: **noexcept(true)** or **noexcept(false)**. For example, consider a class, `A`, containing only fundamental, "built-in" types such as **int**, **double**, and **char***:

```
// original: struct A { int i; double d; char* cp; ~A() } a, a2;  // built-ins only
struct A { int i; double d; char* cp; } a, a2;  // built-ins only

static_assert(noexcept( A()       ), "");  // OK, default constructor
static_assert(noexcept( A(a)      ), "");  // OK, copy constructor
static_assert(noexcept( a = a2    ), "");  // OK, copy assignment
static_assert(noexcept( A(A())    ), "");  // OK, move constructor
static_assert(noexcept( a = A()   ), "");  // OK, move assignment
```

[4] On GCC, for example, the first declaration and, in fact, all pure declarations must be consistent (i.e., either all **noexcept** or all **throw()**), but they make no difference with respect to behavior. It is *only* the declaration on the function *definition* that governs. If the specification is dynamic and `--std=c++17` is not specified, then stack unwinding will occur; otherwise, it will not. On Clang, stack unwinding always occurs. MSVC has never implemented specifying handlers with `std::set_unexpected` and doesn't perform stack unwinding. Intel (EDG) always calls the handler specified by `std::set_unexpected` and unwinds the stack.

```
static_assert(noexcept( a.A::~A() ), "");  // OK, destructor
```

Since this type neither contains nor inherits from other **user-defined types (UDTs)**, the compiler is authorized to treat each of these implicit declarations as if they had been specified **noexcept**.

If a special member function is declared explicitly, then that declaration defines whether the function is to be considered **noexcept** irrespective of its definition (except when using **=default** on first declaration). Consider, for example, an empty class, B, that declares each of the six standard special member functions:

```
struct B  // empty class with all special members declared w/o exception spec.
{
    B();                     // default constructor: noexcept(false)
    B(const B&);             // copy constructor:    noexcept(false)
    B& operator=(const B&);  // copy assignment:     noexcept(false)
    B(B&&);                  // move constructor:    noexcept(false)
    B& operator=(B&&);       // move assignment:     noexcept(false)
    ~B();                    // destructor:          noexcept(true)
};
```

Without regard for their corresponding definitions, the **noexcept** operator will report each of these explicitly declared special member functions as being **noexcept(false)** with the lone exception of the destructor, which defaults to **noexcept(true)**; see *Annoyances — Destructors, but not move constructors, are* **noexcept** *by default* on page 42. More generally, all explicitly declared destructors default to **noexcept(true)** unless they have a base class or member with a **noexcept(false)** destructor. To indicate that the destructor of a class (e.g., BadIdea in the example below) may throw, it must be declared explicitly using the syntax **noexcept(false)** or a nonempty *dynamic* exception specification; see Section 3.1."**??**" on page **??**:

```
struct BadIdea
{
    ~BadIdea() noexcept(false);  // destructor may throw
};
```

C++11 allows the user to declare a special member function and then request the compiler to provide its default implementation using the **=default** syntax; see Section 1.1."**??**" on page **??**. The resulting implementation will be *identical* to what it would have been had the special member function declaration been omitted rather than implicitly suppressed through the declaration of other special member functions:

```
struct C  // empty class declaring all of its special members to be =default
{
    C()                    = default;  // default constructor: noexcept(true)
    C(const C&)            = default;  // copy constructor:    noexcept(true)
    C& operator=(const C&) = default;  // copy assignment:     noexcept(true)
    C(C&&)                 = default;  // move constructor:    noexcept(true)
    C& operator=(C&&)      = default;  // move assignment:     noexcept(true)
    ~C()                   = default;  // destructor:          noexcept(true)
};
```

When a user-defined type (e.g., `D` in the example below) contains or derives from types whose corresponding special member functions are all **`noexcept(true)`**, then so too will any implicitly defined special member functions of that type:

```
struct D : A { A v; } d, d2;  // All special members of A are noexcept(true).

static_assert(noexcept( D()       ), "");  // OK, default constructor
static_assert(noexcept( D(d)      ), "");  // OK, copy constructor
static_assert(noexcept( d = d2    ), "");  // OK, copy assignment
static_assert(noexcept( D(D())    ), "");  // OK, move constructor
static_assert(noexcept( d = D()   ), "");  // OK, move assignment
static_assert(noexcept( d.D::~D() ), "");  // OK, destructor
```

If, however, a special member function in any base or member type of a class (e.g., `E` in the example code below) is **`noexcept(false)`**, then the corresponding special member function of that class will be as well:

```
struct E { B b; } e, e2;  // All special members of B are noexcept(false) apart
                          // from the destructor.

static_assert(noexcept( E()       ), "");  // Error, default constructor
static_assert(noexcept( E(e)      ), "");  // Error, copy constructor
static_assert(noexcept( e = e2    ), "");  // Error, copy assignment
static_assert(noexcept( E(E())    ), "");  // Error, move constructor
static_assert(noexcept( e = E()   ), "");  // Error, move assignment
static_assert(noexcept( e.E::~E() ), "");  // OK, destructor
```

It is permitted for an explicit **`noexcept`** specification to be placed on a special member of a class and then to use the **`=default`** syntax on first declaration to implement it. The explicit **`noexcept`** specification must then match that of the implicitly generated definition, else that special member function will be implicitly deleted:

```
struct F : B  // All special members of B are noexcept(false) apart
              // from the destructor.
{
    F() noexcept(false)    =default;  // default constructor: OK
    F(const F&) noexcept   =default;  // copy constructor:    (deleted)
    F& operator=(const F&) =default;  // copy assignment:     OK
    F(F&&) noexcept(true)  =default;  // move constructor:    (deleted)
    F& operator=(F&&)      =default;  // move assignment:     OK
    ~F() noexcept(true)    =default;  // destructor:          OK
} f, f2;
```

Notice that, in class `F` in the code snippet above, both the *copy* and *move* constructors are mislabeled as being **`noexcept(true)`** when the defaulted declaration would have made them **`noexcept(false)`**. Such inconsistency is not in and of itself an error until an attempt is made to access that function, which is still declared but rendered inaccessible in all contexts; see Section 1.1."**??**" on page **??**:

```
static_assert(!noexcept( F()       ), "");   // OK,    default constructor
static_assert( noexcept( F(f)      ),  "");  // Error, copy constructor
```

```
static_assert(!noexcept( f = f2    ), "");   // OK,    copy assignment
static_assert(!noexcept( F(F())    ), "");   // Error, move constructor
static_assert(!noexcept( f = F()   ), "");   // OK,    move assignment
static_assert( noexcept( f.F::~F() ),  "");  // OK,    destructor
```

Note that the need for an *exact* match between explicitly declared and defaulted **noexcept** specifications is unforgiving in *either* direction. That is, had we, say, attempted to restrict the contract of the class by decorating the destructor of class, `F` in the example above, with **noexcept(false)** when the defaulted implementation would have happened to have been **noexcept(true)**, that destructor would have nonetheless been implicitly deleted, severely crippling use of the class.

Finally, the C++11 specification does not address directly the implicit exception specification for inheriting constructors (see Section 2.1."**??**" on page **??**), yet most popular compilers handle them correctly in that they take into account the exceptions thrown by the inherited constructor and all the member initialization involved in invocation of the inherited constructor.

In C++14, all implicitly declared special member functions, including inheriting constructors, are **noexcept(false)** if any function they invoke directly has an exception specification that allows all exceptions; otherwise, if any of these directly invoked functions has a dynamic exception specification, then the implicit member will have a dynamic exception specification that comprises all of the types that may be thrown by functions it invokes directly. In particular, when a constructor is inherited, its exception specification is nonthrowing if the base class constructor is nonthrowing, and the expressions initializing each of the derived class's additional bases and members are also nonthrowing. Otherwise, an inheriting constructor has a potentially throwing exception specification. Although the original text in the C++11 Standard was worded subtly differently, the wording was repaired via a defect report and incorporated directly into C++14. Note that all known implementations of the feature, even early prototypes, follow these corrected rules of C++14.

As a concrete example, let's suppose that we have a base class, `BB`, that has two *value* constructors, one throwing and the other *non*throwing:

```
struct BB  // base class having two overloaded value constructors
{
    BB(int)  noexcept(false);  //    throwing int  value constructor
    BB(char) noexcept(true);   // nonthrowing char value constructor
};
```

Invoking these value constructors, respectively, on an **int** and a **char** produces the expected results:

```
int i;
char c;

static_assert(!noexcept( BB(i) ), "");  // noexcept(false)
static_assert( noexcept( BB(c) ), "");  // noexcept(true)
    // uses just base constructors' exception specifications
```

Next suppose we derive an empty class, `D1`, from `BB` that inherits `BB`'s base class's constructors:

```
struct D1 : BB  // empty derived class inheriting base class BBs ctors
{
    using BB::BB;  // inherits BB's ctors along with exception specs
};
```

Because the inherited constructors of the derived class are not required to invoke any other constructors, the exception specifications propagate unchanged:

```
static_assert(!noexcept( D1(i) ), "");  // noexcept(false)
static_assert( noexcept( D1(c) ),  "");  // noexcept(true)
    // uses just the inherited constructors' exception specifications
```

Now imagine that we have some legacy class, `SS`, whose default constructor is implemented with a deprecated, dynamic exception specification that explicitly allows it to throw only an `SSException` (assumed defined elsewhere):

```
class SSException { /*...*/ };
struct SS  // old-fashioned type having deprecated, dynamic exception specs.
{
    SS() throw(SSException);  // This default ctor is allowed to throw only an SSException.
};
```

The **`noexcept`** operator, not caring about the flavor of exceptions that are thrown coarsely, reports **`noexcept(false)`**:

```
static_assert(!noexcept( SS() ),  "");  // throw(SSException)
    // uses the dynamic exception specification of the default constructor
```

Now suppose we derive a second type, `D2`, from base class `BB` but this time having, as a data member, `ss`, an object of type `SS` whose default constructor may throw an `SSException`:

```
struct D2 : BB  // nonempty derived class inheriting base class BB's ctors
{
    SS ss;         // data member having default ctor that may throw an SSException
    using BB::BB;  // inherits BB's ctors along with exception specs
};
```

Both inherited constructors are now implicitly obliged to invoke the default constructor of `ss`, which may throw; hence, both inherited constructors are now throwing constructors:

```
static_assert(!noexcept( D2(i) ), "");  // BB(int) is noexcept(false)
static_assert(!noexcept( D2(c) ), "");  // SS() is throw(SSException)
    // Uses both the inherited constructors' exception specifications and
    // the exception specification of the data member's default constructor.
```

In the example above, the implicit exception specification of the `D2(int)` and `D2(char)` constructors are, respectively, **`noexcept(false)`** and **`throw`**`(SSException)`. Implicit dynamic exception specifications, despite being deprecated, are plausible since the invocation of an implicitly declared special member function, say, `D2(char)`, will necessarily invoke all the other special member functions — i.e., `BB(char)` and `SS()` — that contribute to its implicit exception specification. If any of these implicitly invoked functions throws, the exception specification that will be checked first is that of the more restrictive invoked subfunction — i.e., that of `SS()` — and not the potentially more permissive caller.

Widening the exception specification of the implicitly declared special member functions, say, from `throw(SSException)` to `noexcept(false)`, would make no difference. Even if we changed `BB(char)` to `noexcept(false)`, which would give `D2(char)` an implicit exception specification of `noexcept(false)` and allow exceptions of all types to pass, no other exception type thrown by `SS()` would ever propagate to a `D2(char)` constructor's potential exception specification check. Instead, the rogue exception would be stopped by the nonmatching exception specification of the subfunction `SS() throw(SSException)` invoked by the implicitly defined constructor.

o-compound-expressions

Applying the `noexcept` operator to compound expressions

Recall from earlier in this feature section that the **noexcept** operator is applied to an expression, which may itself comprise other subexpressions. For example, consider two functions on integers, `f` and `g`, that are and are not, respectively, **noexcept**:

```
int f(int i) noexcept { return i; }  // Function f is noexcept(true).
int g(int i)          { return i; }  // Function g is noexcept(false).

static_assert(noexcept( f(17) ), "");  // OK,    f is noexcept(true).
static_assert(noexcept( g(17) ), "");  // Error, g is noexcept(false).
```

Now suppose that we have two function calls within a single expression:

```
static_assert(noexcept( f(1) + f(2) ), "");  // OK,    f is noexcept(true).
static_assert(noexcept( g(1) + g(2) ), "");  // Error, g is noexcept(false).
static_assert(noexcept( g(1) + f(2) ), "");  // Error,  "   "      "      "
static_assert(noexcept( f(1) + g(2) ), "");  // Error,  "   "      "      "
```

When we consider composing two functions, the overall expression is **noexcept** if and only if both functions are **noexcept**:

```
static_assert(noexcept( f(f(17)) ), "");  // OK,    f is noexcept(true).
static_assert(noexcept( g(g(17)) ), "");  // Error, g is noexcept(false).
static_assert(noexcept( g(f(17)) ), "");  // Error,  "   "      "      "
static_assert(noexcept( f(g(17)) ), "");  // Error,  "   "      "      "
```

The same applies to other forms of composition; recall from earlier that the specific operators applied in the expression do not matter, only whether any potentially evaluated subexpression might throw:

```
static_assert(noexcept( f(1) || f(2) ), "");  // OK,    f is noexcept(true).
static_assert(noexcept( g(1) || g(2) ), "");  // Error, g is noexcept(false).
static_assert(noexcept( g(1) || f(2) ), "");  // Error,  "   "      "      "
static_assert(noexcept( f(1) || g(2) ), "");  // Error, note g is never called!
```

Importantly, note that the final expression in the example above is *not* **noexcept** even though the only subexpression that might throw is never evaluated. This deliberate language design decision eliminates variations in implementation that would trade off compile-time speed for determining whether the detailed logic of a given expression might throw, but see *Annoyances — Older compilers invade the bodies of* **constexpr** *functions* on page 43.

r-to-move-expressions

Applying the `noexcept` operator to move expressions

Finally we come to the quintessential application of the **noexcept** operator. C++11 introduces the notion of a **move operation** — typically an adjunct to a **copy operation** — as a fundamentally new way in which to propagate the value of one object to another; see Section 2.1."**??**" on page **??**. For objects that have well-defined **copy semantics** (e.g., **value semantics**), a valid copy operation typically satisfies *all* of the contractual requirements of the corresponding move operation, the only difference being that a requested *move* operation doesn't require that the value of the source object be preserved:

`std::move`

```
struct S  // Class S supports both copy and move operations.
{
   // ...
   S();              // default constructor
   S(const S&);      // copy constructor; declared noexcept(false)
   S(S&&) noexcept;  // move constructor; declared noexcept(true)
   // ...
};
```

When a value-preserving copy operation is not specifically needed, requesting that the value of an object be just *moved* could lead to a more runtime-efficient program.[5] For example, a function that takes an argument by *rvalue* reference or by value is capable of exploiting a potentially more efficient move operation if one is available:

```
void f(const S&);  // passing object of type S by const lvalue reference
void f(S&&);       // passing object of type S by rvalue reference
```

There are times when the compiler will automatically prefer a *move* operation over a *copy* — i.e., when it knows that the source object, e.g., a **temporary**, is no longer separately reachable:

```
S h();  // function returning an S by value
S s1;

void test()
{
    f(S());  // The compiler requests a move automatically.
    f(h());  //  "     "         "    "    "         "

    S s2;
    f(s1);   // The compiler will not try to move automatically.
    f(s2);   //  "      "      "   "   "   "    "         "
}
```

For objects whose lifetimes are expiring (e.g., temporary objects), the compiler will automatically attempt to *move* rather than copy the value where applicable.

[5] Importantly, *move* operations can lead to **memory diffusion**, which in turn can severely impact the runtime performance of large, long-running programs; see Meredith, Graham Bleaney, Lakos17 MeetingC++. **[AUs: from VR: LORI NOT SURE WHAT THE LINK SHOULD BE HERE, ASK JOHN]**

When an object such as `s1` or `s2` in the example code above is separately reachable, however, the programmer — knowing the value will no longer be needed — can request of the compiler that the object be moved:

```
void test2()
{
    S s2;
    f(std::move(s1));   // The compiler will now try to move from s1.
    f(std::move(s2));   //  "     "      "    "   "  "    "   "   s2.
}
```

When the programmer uses `std::move`, as shown in the code snippet above, to tell the compiler that the value of an object — such as `s1` or `s2` — is no longer needed, the compiler will invoke the constructor variant taking `S` as an *rvalue* reference if *reachable*.

Not every class necessarily provides a distinct move operation:

```
struct C  // Class C supports copy but not move operations.
{
    C();                    // default constructor
    C(const C&);            // copy constructor
    C& operator=(const C&); // copy assignment
    ~C();                   // destructor
};
```

Merely requesting that an object be moved might have no effect on the generated code — for example, when, as shown in the code snippet above, a proper copy operation is all that is available:

```
void f(C);

void test4()
{
    C c;

    C c1(c);             // invokes C's copy constructor
    C c2(std::move(c));  //    "      "     "       "

    f(c);                // invokes C's copy constructor
    f(std::move(c));     //    "      "     "       "
}
```

Because class `C` in the example above has no distinct `move` operations, normal overload resolution selects `C`'s *copy* constructor as the best match even when a *move* is requested explicitly.

When using **`noexcept`** in conjunction with `std::move`, however, all that matters is whether the *move* operation — whatever it might turn out to be – will be **`noexcept(true)`**; if so, we can perhaps exploit that information to safely employ a more efficient algorithm that requires a nonthrowing move:

```
template <typename T>
void doSomething(T t)
```

```
{
    if (noexcept(T(std::move(t))))
    {
        // may assume no exception will be thrown during a move operation
    }
    else
    {
        // must use an algorithm that can handle a thrown exception
    }
}
```

Alternatively, we can simply *require* that any supplied type have a nonthrowing *move* operation:

```
template <typename T>
void doOrDie(T t)
{
    // may assume no exception will be thrown during a move operation

    static_assert(noexcept(T(std::move(t))), "");  // ill formed otherwise
}
```

Both class `S` and class `C` in the code snippet above have copy constructors that are declared **`noexcept(false)`**; however, class `S` also has a move constructor defined that is **`noexcept(true)`**, whereas `C` declared no `move` constructor at all:

`std::move`

```
S s1;  // declares a noexcept(true) move constructor
static_assert(noexcept(S(std::move(s1))), "");    // OK

C c1;  // declares only a noexcept(false) copy constructor
static_assert(!noexcept(C(std::move(c1))), "");   // OK
```

Although not recommended, we could define a class, `S2`, that is the same as `S` except that the *move* constructor is declared to be **`noexcept(false)`**:

```
S2 s2;  // declares noexcept(false) copy and move constructors
static_assert(!noexcept(S2(std::move(s2))), "");  // OK
```

Similarly, we could imagine a class, `C2`, that is the same as `C` except that the *copy* constructor is declared explicitly with **`throw()`**, making it **`noexcept(true)`**:

```
C2 c2;  // declares only a copy constructor decorated with throw()
static_assert(noexcept(C2(std::move(c2))), "");   // OK
```

There are many ways in which an object might or might not provide a nonthrowing *move* operation. As `C2` in the example above suggests, even a C++03 class that happened to decorate its explicitly declared copy constructor with **`throw()`** would automatically satisfy the requirements of a nonthrowing *move*. A more likely scenario for a class designed prior to C++11 to wind up with a nonthrowing *move* operation is that it followed the **rule of zero**, thereby allowing each of the special member functions to be generated. In this case, all that might be needed to generate a nonthrow *move* constructor is simply to recompile

it under C++11! The takeaway here is that, irrespective of how a type is implemented, we can use the **noexcept** operator in combination with `std::move` to reliably determine, at compile time, whether an object of a given type *may* throw when we ask it to move. Note that while it is typical for C++03 code to either have both copy and move constructors nonthrowing, or both potentially throwing, a C++03 template instantiated with a C++11 move-optimized type may still have exception specifications that match this behavior:

`std::string`

```
template <typename T>
struct NamedValue
{
    std::string d_name;
    T           d_value;
};
```

While `NamedValue` may be a class template shipping from a C++03-authored library, the copy constructor for this class will clearly be **noexcept(false)**, but the move constructor may be throwing or not based solely on the properties of the template argument, `T`.

-the-c-standard-library

Applying the `noexcept` operator to functions in the C Standard Library

According to the C++03 Standard[6]:

> None of the functions from the Standard C library shall report an error by throwing an exception, unless it calls a program-supplied function that throws an exception.

This paragraph is accompanied by a handy footnote:

> That is, the C library functions all have a **throw()** exception-specification. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

Note that this footnote applies only to functions in the C Standard Library, not to arbitrary functions having **extern "C"** linkage. It is not clear what the normative implications of the footnote might be, as it seems to be a non-normative note clarifying something not obviously implied by the normative text. Given the extra costs associated with C++98 exception specifications, there are no known implementations that took advantage of this freedom.

For C++11, the footnote was revised to refer only to permitting the use of the new **noexcept** exception specification, without further clarification of the normative text. There is, however, also general permission to add a nonthrowing exception specification to any nonvirtual C++ Standard Library function, and it might be inferred that this provision gives implementations freedom to add such specifications to their C library wrappers too. Also note that functions taking callbacks, such as `bsearch` and `qsort`, are still specified to have nonthrowing exception specifications.

[6] **[AUs: please elaborate on this citation. We don't have anything called cpp03.]** cpp03,[lib.res.on.exception.handling], p2, pp. 331–332.

Again, there are no known implementations taking advantage of this freedom to add a nonthrowing exception specification to C library functions, although all of the functions in the `<atomic>` header intended for C interoperability are declared as **noexcept**, exploiting an arguable interpretation of the intent of this footnote.

for-virtual-functions

Constraints on the `noexcept` specification imposed for `virtual` functions

You will no doubt have noticed that all of the **noexcept** functions in the examples above have been non**virtual**.

When using C++03-style dynamic exception specifications, the exception specification of any function override cannot be wider than that of the function being overridden. This is perhaps best illustrated by means of a simple example:

```
struct BB03
{
    void n() throw();
    virtual void f();
    virtual void g1() throw();
    virtual void g2() throw();
    virtual void g3() throw();
    virtual void h() throw(int, double);
};

struct DD03 : public BB03
{
    void n() throw(int);                // OK, not virtual
    void n(const SomeType&) throw(int); // OK, hiding nonvirtual function
    virtual void f();                   // OK, base has no exception spec.
    virtual void g1() throw();          // OK, same exception spec.
    virtual void g2() throw(int);       // Error, wider exception spec. (int)
    virtual void g3();                  // Error, wider exception spec. (all)
    virtual void h() throw(int);        // OK, tighter exception spec.
};
```

Interestingly, the rules relating to **virtual** functions and **noexcept** are still defined by the C++11 and C++14 Standards in terms of dynamic exception specifications, despite the fact that dynamic exception specifications are deprecated. It states that "If a virtual function has an *exception-specification*, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall only allow exceptions that are allowed by the *exception-specification* of the base class virtual function."[7]

From a **noexcept** perspective, this means the rules are very straightforward. If the base function forbids exceptions by specifying **noexcept** or **noexcept(true)**, then the override must forbid exceptions by specifying one of **noexcept**, **noexcept(true)**, or **throw()**. In other words, if a base class virtual functions is **noexcept(true)**, then no derived class override of that function can have a throwing exception specification:

```
struct BB11
{
```

[7] **?**, section 15.4, "Exception specifications," paragraph 5, p. 406

```
    void n() noexcept;
    virtual void f();
    virtual void g1() noexcept(true);
    virtual void g2() noexcept;
    virtual void g3() noexcept(true);
    virtual void g4() noexcept(true);
};

struct DD11 : public BB11
{
    void n();                           // OK, not virtual
    void n(const BB11*);                // OK, hiding nonvirtual function
    void f() override;                  // OK, base has no exception spec.
    void g1() noexcept override;        // OK, override is noexcept
    void g2() throw() override;         // OK, override is noexcept
    void g3() override;                 // Error, override allows exceptions
    void g4() noexcept(false) override; // Error, override allows exceptions
};
```

One final thing to note is that these rules also apply to defaulted virtual functions, most notably destructors:

```
struct BB11
{
    virtual ~BB11() = default;  // noexcept(true)
};

struct DD11 : public BB11
{
    virtual ~DD11() noexcept(false);  // Error, BB11::~BB11() is noexcept(true)
};
```

Essentially, the envelope of what can be thrown from a virtual function in a derived class is constrained to be a subset of what can be thrown in the base class. This constraint can be quite restrictive since, in real-world systems, there are times when a base-class contract can serve a syntactic role and have a semantic approximation that can be relaxed if both the consumer and the supplier are in agreement regarding said relaxation. This is further expounded in *Annoyances — Exception-specification constraints in class hierarchies* on page 44.

-cases-noexceptoperator

ement-to-an-std::vector

Use Cases

Appending an element to an `std::vector`

There are certain cases where it is useful to know whether an expression, specifically one involving *copy* operations, *may* throw so that an optimal algorithmic decision can be made, often involving *move* operations. In fact, the very reason *move* operations were thoughtfully added to C++11 was to support more efficient insertion of **allocating objects** into an `std::vector`; see Inserting an element into an `std::vector` efficiently. But insertion into an arbitrary location within a `std::vector` would *not* have been sufficient justification

for hastily adding the **noexcept** operator just prior to shipping C++11; see *Appendix — Genesis of the* **noexcept** *operator: move operations* on page 47.

First, the original C++ Standard provided the **strong exception-safety guarantee** for any element "inserted" at the *end* of an `std::vector`, whether it be via the `insert` member function or the more popular `push_back`. Second, backward compatibility with C++03 meant that any type having explicitly defined copy operations would *not* be given implicit move operations, throwing or otherwise. Hence, when asked to move, any legacy C++ type would instead fall back on its copy operation, which, when it doesn't throw, satisfies all the requirements of an optimizing move operation.

Next, consider that some legacy code, having previously been promised the **strong guarantee**, fairly depends *at run time* on `std::vector::push_back`'s either succeeding or else throwing with no effect whatsoever on the state of the vector. To illustrate what is meant by the strong exception-safety guarantee, let's suppose we have a class, `S`, that has an explicitly declared copy constructor that *may* throw, thereby precluding an implicitly generated move constructor. For our purely pedagogical example, we will force the copy constructor to throw the third time the program attempts to copy an `S` object in the current process:

```
#include <cassert>  // standard C assert macro
#include <vector>   // std::vector

struct S
{
    static int s_nCopy;  // number of objects that have been copied
    int        d_uid;    // unique identifier for each copied object

    S()         : d_uid(-1) { }     // default constructor
    S(const S&) : d_uid(++s_nCopy)  // copy constructor
    {
        if (s_nCopy > 2) throw s_nCopy;  // throws on third attempt to copy
    }
};

int S::s_nCopy = 0; // initialization of static data member of class S
```

When inserting copies of `S` into an `std::vector<S>`, the strong guarantee ensures that the *entire* state of the vector — i.e., not just its **salient values** — remains unchanged despite the **throw**s occurring within the `push_back` operation. Pointers and references to the existing elements remain valid since the vector has not yet expanded, and the entire internal state of those elements too is unaltered:

```
int main()
{
    const S s;                  // default-constructed object (d_uid: -1)
    std::vector<S> v;           // container to fill

    v.reserve(2);               // Do not reallocate until third push_back!
    assert(v.capacity() == 2);  // assert that capacity isn't rounded higher
```

```
    v.push_back(s);
    v.push_back(s);

    // before:
    assert(v[0].d_uid == 1);
    assert(v[1].d_uid == 2);
    assert(v.capacity() == 2);  // assert that capacity is still the same

    // insert third (throwing) element:
    try
    {
        v.push_back(s);          // expected to throw constructing new element
        assert(!!"Should have thrown an exception");
    }
    catch(int n)
    {
        assert(n == 3);         // verify the expected exception value
    }

    // after:
    assert(v[0].d_uid == 1);
    assert(v[1].d_uid == 2);
    assert(v.capacity() == 2);  // even the vector's capacity is unchanged
}
```

Importantly, when the exception is finally thrown, the entire state of the vector prior to attempting to add the third element remains unchanged, thus delivering on the strong exception-safety guarantee as has been required for `std::vector::push_back` by the C++ Standard since its inception. Had the third `push_back` not thrown, the *resize* would have occurred successfully, and each of the elements would have been *copied* into the newly allocated storage, which satisfies the letter of the original C++03 contract, but see *Annoyances — Change in unspecified behavior when an* `std::vector` *grows* on page 41.

The question now becomes how we should implement `std::vector::push_back` efficiently in modern C++.

Ignoring, for simplicity, C++11 memory allocators, recall that a standard vector maintains (1) the `data()` address of its dynamically allocated element storage, (2) the maximum `capacity()` of elements it can hold before having to resize, and (3) the `size()`, i.e., number of elements it currently holds. When the `size()` is either 0 or less than the current `capacity()`, there is no issue: An attempt is made to append the element after first allocating dynamic storage if the `capacity()` too was 0, and, if an exception is thrown either during memory allocation or directly by the element's constructor, there is no effect on the state of the vector object. So far, so good.

Let's now consider what happens when the `size()` is not 0 and there is no more `capacity()` left, that is `size() > 0 && size() == capacity()`. The first step, as ever, is to allocate a larger block of dynamic memory. If that allocation throws, there's nothing to do and the strong guarantee is automatically satisfied. But what happens if that allocation succeeds? If we try to move an existing element to the newly allocated slab of memory and the move operation throws, we have no guarantee that the state of the element is unchanged

in its original location. If the first move succeeds, then, when we go to move a second element, we're past the point of no return: if the second move operation fails by throwing an exception, we have no way to go forward and similarly no guaranteed way to revert since attempting to move the first one back might throw as well. If this were to happen, then the strong exception-safety guarantee would necessarily be violated and in the worst possible way: not at compile time, link time, or start up and not just under a heavy load, but nondeterministically at run time when the vector needs to grow and moving an existing element throws during that operation.

Alternatively, we could take the same conservative approach as in C++03 before move operations were standardized. That is, instead of even trying to efficiently move existing elements from the old dynamically allocated block to the new one, each element would instead be *copied*, e.g., using the **copy/swap idiom**, but now we are making a full copy of every existing element of the vector every time its capacity grows. The complexity of the operation could technically still be considered **amortized constant time**; depending on the complexity of the elements (e.g., whether or not they might allocate), however, the latency cost of a single `push_back` at a memory boundary could still be prohibitive. Unwilling to give up the strong guarantee or optimal performance in the increasingly typical case (support for nonthrowing move operations), the Standards committee chose a third alternative: the **`noexcept`** operator.

Let's now consider how we might use the **`noexcept`** operator to implement an `std::vector`-like `push_back` member function that safely exploits the new *move* operations on potential elements, again, for simplicity of exposition, ignoring C++11 memory allocators. Let's start by assuming a heavily elided definition of our class, `vector`:

```
#include <cstddef>  // std::size_t

template <typename T>
class vector
{
    T*          d_array_p;   // dynamic memory for elements of type T
    std::size_t d_capacity;  // maximum number of elements before resize
    std::size_t d_size;      // current number of elements in this array
public:
    vector() : d_array_p(0), d_capacity(0), d_size(0) { }  // created empty
    // ...
    void push_back(const T& value);  // safe, efficient implementation
    // ...
    void reserve(std::size_t capacity);  // make more space (might throw)
    void swap(vector& other) throw();    // swap state with other vector<T>
};                         // ^^^^^^^        (Today, we would use noexcept.)
```

Assuming the existence of just the member functions in the example code above, let's look at implementing an efficient, exception-safe `push_back` method that preserves the strong exception-safety guarantee even for a type that, when asked just to `move`, *may* nonetheless throw:

```
#include <new>      // placement new
#include <utility>  // std::move
```

```
template <typename T>
void vector<T>::push_back(const T& value)  // safe, efficient implementation
{
    if (d_size < d_capacity)  // sufficient capacity in allocated memory
    {
        void* address = d_array_p + d_size;  // implicit conversion to void*
        ::new(address) T(value);             // may throw on copy
        ++d_size;                            // no throw
        return;                              // early return
    }

    // If we know that attempting to move an object may not throw, we can
    // improve performance compared to relying on a classically throwing copy.

    const std::size_t nextCapacity = d_capacity ? d_capacity * 2
                                                : 1;              // no throw

    if (noexcept(::new((void*)0) T(std::move(*d_array_p))))  // no throw (move)
    {
        vector<T> tmp;                               // may throw
        tmp.reserve(nextCapacity);                   // may throw

        void* address = tmp.d_array_p + d_size;      // no throw
        ::new(address) T(value);                     // may throw (last)

        for (std::size_t i = 0; i != d_size; ++i)  // for each existing element
        {
            void* addr = tmp.d_array_p + i;          // no throw
            ::new(addr) T(std::move(d_array_p[i]));  // no throw (move)
        }

        tmp.d_size = d_size + 1;  // no throw
        tmp.swap(*this);          // no throw, committed
    }
    else                              // otherwise employ the copy/swap idiom
    {
        vector<T> copy;               // may throw
        copy.reserve(nextCapacity);   // may throw
        copy = *this;                 // may throw
        copy.push_back(value);        // may throw on copy; capacity's good
        copy.swap(*this);             // no throw, committed
    }
}
```

As the code snippet above illustrates, as long as there is sufficient capacity in the current block of dynamic memory, there is no need for the **noexcept** operator. Only when it becomes necessary to reallocate to a larger capacity does the need for the **noexcept** operator arise. At that point, we will need to know if, when we ask the object of type `T` to move-construct, it *may* throw, but see *Potential Pitfalls — Using the* **noexcept** *operator directly* on page 37:

```
if (noexcept(::new((void*)0) T(std::move(*d_array_p))))  // no throw (move)
//           ^^^^^^^^^^^^^^^^ avoids noexcept operator's considering T's dtor
```

If copying *may* throw, we must revert to the C++03 algorithm in the **else** clause, which exploits the familiar copy/swap idiom; notice that we call `reserve` to pre-allocate memory in a `copy`, rather than the current vector, to avoid problems with **aliasing** and redundant allocations.

But if we can know that moving (or copying) an object *may not* throw, then we might be able to avoid having to *copy* all the objects over to the newly allocated memory. First, we create a temporary vector and then reserve whatever the next larger capacity is intended to be. If either of these operations were to throw, then there would be no state change in the original vector, and, thus, the strong exception-safety guarantee would be preserved. Then we call **placement new** to construct the new element at the intended address. Note the specific syntax we use when calling placement new: Employing `::` ensures that overloads of **new** in only the global namespace are considered, thus avoiding **ADL**'s finding any ill-advised, class-specific overloads. Moreover, we deliberately allow the address pointer to **decay**, preferring that to an explicit cast, to a plain **void** * to match exactly the specific, Standard-mandated overload of placement new we desire, thus preventing our accidentally calling any other, more specific overload in the global namespace. If, when invoking the constructor for the new element, an exception is thrown, no new element will be created, the partially constructed temporary object will be destroyed as the exception leaves block scope, the temporary vector destructor will reclaim allocated memory, and again the strong guarantee is preserved. Note that the order of operations allows this member function to work properly even if the argument to `value` turns out to be a reference into this vector (a.k.a. aliasing).

We are now past the point at which it is possible for an exception to be thrown, so we proceed accordingly to move constructing each of the original elements over to the new slab of memory with impunity. Once we're done moving all the elements over, we manually set the `d_size` data member of the temporary `vector`, `tmp`, and efficiently (no-throw) `swap` all of its members with those of the current vector. When `tmp` goes out of scope, it first destroys the moved-from carcass of the original elements before deleting the old, dynamically allocated block.

Use of the **noexcept** operator above, although correct, is subtle and not particularly easy to maintain. Almost all uses outside of **conditional noexcept specifications** (see Section 3.1."**??**" on page **??**) involve move operations, the most common of which require even more arcane, metaprogramming shenanigans to make work properly; see *Implementing* `std::move_if_noexcept` on page 28 and *Implementing* `std::vector::push_back(T&&)` on page 31.

t-using-static_assert

Enforcing a `noexcept` contract using `static_assert`

When writing a function template, the possibility of throwing an exception might be determined entirely by operations dependent on the template parameters. In such cases, we may want to have a contract that never throws an exception, so we use the **noexcept** specification. However, this would lead to runtime enforcement of the contract, calling `std::terminate` if an exception is thrown from any of those operations involving the tem-

plate parameters. If we want to enforce the contract at compile time, we would additionally use a **static_assert** testing the relevant expressions with the **noexcept** operator:

```
#include <cmath>  // std::sqrt
template <typename T>
T sine(T const& a, T const& b) noexcept
{
   static_assert(noexcept( T(a / std::sqrt(a * a + b * b)) ), "throwing expr");
   return a / std::sqrt(a * a + b * b);
}
```

Note that this will reject otherwise valid code that does not throw exceptions but is not yet marked up with an exception specification.

Consider the earlier example of preserving the strong exception-safety guarantee in `std::vector` if the move constructor could throw. An alternative design, rejected for concerns of breaking existing code, would be to require all elements inserted into a vector have a nonthrowing move constructor. This restriction could be similarly enforced with a **static_assert** and might be a reasonable design choice for a custom container:

```
#include <new>      // placement new
#include <utility>  // std::move

template <typename T>
void vector<T>::push_back(const T& value)  // efficient implementation
{
    static_assert(noexcept(::new((void*)0) T(std::move(*d_array_p))),
                  "The element type must have a noexcept move constructor");

    if (d_size < d_capacity)  // sufficient capacity in allocated memory
    {
        void* address = d_array_p + d_size;  // implicit conversion to void*
        ::new(address) T(value);             // may throw on copy
        ++d_size;                            // no throw
        return;                              // early return
    }

    // We know that attempting to move an object may not throw, so we can
    // safely move rather than copy elements into the new storage.

    const std::size_t nextCapacity = d_capacity ? d_capacity * 2
                                                : 1;              // no throw

    vector<T> tmp;                                  // may throw
    tmp.reserve(nextCapacity);                     // may throw

    void* address = tmp.d_array_p + d_size;        // no throw
    ::new(address) T(value);                       // may throw (last)

    for (std::size_t i = 0; i != d_size; ++i)      // for each existing element
    {
```

```
        void* addr = tmp.d_array_p + i;          // no throw
        ::new(addr) T(std::move(d_array_p[i]));  // no throw (move)
    }

    tmp.d_size = d_size + 1;  // no throw
    tmp.swap(*this);          // no throw, committed
}
```

std::move_if_noexcept

Implementing `std::move_if_noexcept`

The need for the **noexcept** operator is all but exclusively tied to *move* operations and yet — even for that purpose — direct use of this operator is almost always problematic. C++11 provides an extensive library of type traits that can precisely determine important, relevant properties of a type, such as whether it has an accessible constructor; see *Potential Pitfalls — Using the* **noexcept** *operator directly* on page 37.

Consider the broken implementation of a function intended to construct a new object at a given address, where the *move* constructor will be called if it can be determined that exceptions *may not* be thrown; otherwise, if the object type has an accessible copy constructor, call that, or, as a last resort, call the *move* constructor regardless since the type is simply not copyable (a.k.a. **move-only** type):

newstd::move

```
template <typename T>
void construct(void* address, T& object)
     // object passed by modifiable lvalue to enable both copy and move
{
    if (noexcept(::new(address) T(std::move(object))))  // noexcept move
    {
        ::new(address) T(std::move(object));   // OK, no-throw movable
    }
    else if (std::is_copy_constructible<T>::value)
    {
        ::new(address) T(object);  // Oops, compile-time error if not copyable
    }
    else // T is not declared copyable, is movable, and may throw.
    {
        ::new(address) T(std::move(object));  // move move-only type anyway
    }
}
```

The naive, straightforward implementation above fails to compile for move-only types because the copy branch of the **if** statement needs to compile even when the **else** branch is taken, as *all* branches are compiled for each template instantiation.[8] We need to force

[8] In C++17, the **if constexpr** language feature is a direct solution to such problems:

```
template <typename T>
void construct(void* address, T& object)
{
    if constexpr (noexcept(::new(address) T(std::move(object))))
```

only one of each of the potential branches to be evaluated at compile time, which requires selecting the chosen function through partial template specialization, rather than runtime branching in the function body itself. If it were allowed, we would opt to "partially specialize" the two-parameter `moveParameter` and `copyParameter` functions with respect to the value of its second, boolean argument. As partial template specialization of functions is *not* permitted, we must *hoist* (represent) the parameter to the would-be partially specialized function in a class template sporting the **`static`** template function to be called:

```
template <bool SHOULD_MOVE>
struct ImplementMoveOrCopy  // general declaration/definition of class template
{
    template <typename T>
    static void construct(void* address, T& object)
    {
        ::new(address) T(object); // copy
    }
};

template <>
struct ImplementMoveOrCopy<true>  // explicit specialization of class template
{
    template <typename T>
    static void construct(void* address, T& object)
    {
        ::new(address) T(std::move(object));   // move
    }
};

template <typename T>
void construct(void* address, T& object)
     // object passed by modifiable lvalue to enable both copy and move
{
    ImplementMoveOrCopy<std::is_nothrow_move_constructible<T>::value ||
        !std::is_copy_constructible<T>::value>::construct(address, object);
}
```

The example code above splits the once cohesive `construct` function template into three

---

```
    {
        ::new(address) T(std::move(object));
    }
    else if constexpr (std::is_copy_constructible<T>::value)
    {
        ::new(address) T(object);  // discarded if this branch is not taken
    }
    else // T is not declared copyable, is movable, and may throw.
    {
        ::new(address) T(std::move(object));
    }
}
```

parts, two of which are intended as only private implementation details and hosted in a class template.

The C++ Standard Library provides the `std::move_if_noexcept` function to encapsulate such baroque metaprogramming code, which turns out to be surprisingly useful; see *Implementing* `std::vector::push_back(T&&)` on page 31 and *Implementing* `std::vector::emplace_back(Args&&... args)` on page 34:

`std::movestd::conditionalstd::is_nothrow_move_constructiblestd::is_copy_constructible`

```
template <typename T>
constexpr
typename std::conditional<!std::is_nothrow_move_constructible<T>::value
                          && std::is_copy_constructible<T>::value,
                          const T&,
                          T&&>::type
move_if_noexcept(T& x) noexcept
{
    return std::move(x);
}
```

There's a lot to unpack from the definition in the code snippet above, but the principle is simple: all of the "clever" work is done computing the return type of the function. The `std::conditional` return type is a standard library **metafunction** that evaluates a predicate and produces the second template parameter for its result type if the predicate is **`true`** and the final template parameter for its result type if the predicate is **`false`**. In this case, the predicate is using type traits to examine properties of the deduced type `T` to determine which result should be preferred. If a type is not copy constructible, indicated by the `std::is_copy_constructible` type trait, the `std::conditional` function should always return an *rvalue* reference. If a type is copy constructible, it should not return an *rvalue* reference unless the move constructor is **`noexcept`**. We wisely use the type trait `is_nothrow_move_constructible` to determine whether a type has a **`noexcept`** *move* constructor, rather than attempting ourselves to implement some clever metaprogramming trickery inside a **`noexcept`** operator to avoid consideration of ancillary subexpressions.

A naive attempt to implement the `std::is_nothrow_move_constructible` trait using the **`noexcept`** operator might be:

```
#include <utility>  // std::declval
template <typename T>
struct is_nothrow_move_constructible
    : std::integral_constant<bool,
                             noexcept(::new((void*)0) T(std::declval<T>()))>
{
};
```

Here, `std::declval` is a declared but not defined function that always returns an *rvalue* reference to the type of the template argument, avoiding the need to know of a valid constructor for an arbitrary type. However, this is still an approximation as we must use more obscure template metaprogramming tricks to return **`false`** when the move constructor is not public or is deleted.

Once we have the `move_if_noexcept` function, the `construct` example can be written

even more simply and, this time, will compile without error:

```
template <typename T>
void construct(void* address, T& object)
     // object passed by modifiable lvalue to enable both copy and move
{
    ::new(address) T(std::move_if_noexcept(object));  // factored implementation
}
```

Note that a production form would also use a conditional **noexcept** specification to indicate no exceptions when `T` is a move-only type; see Section 3.1."??" on page ??.

:vector::push_back(t&&)

Implementing `std::vector::push_back(T&&)`

In *Appending an element to an* `std::vector` on page 21, we discussed the primary motivation for having a **noexcept** operator, and in *Implementing* `std::move_if_noexcept` on page 28, we demonstrated how we could implement fully factored functionality that is eminently useful in a variety of *move*-related operations.

In addition to inserting a *copy* of a value into a vector, C++11 adds an overload to permit *moving* an element when inserting into the back of a vector, using the overload **void** `vector<T>::push_back(T&& value)` (see Section 2.1."??" on page ??). This function offers the same strong exception-safety guarantee as the "push-a-copy" overload but has to consider the additional case of a **move-only type** — i.e., a type having a **public** *move* constructor but no **accessible *copy* constructor** — where the *move* constructor *may* throw. In such cases, `push_back` offers only the **basic exception-safety guarantee**; that is, no resources will be leaked and no invariants will be broken, but the state of the vector after the exception is thrown is otherwise unknown.

In addition to weakening the strong exception-safety guarantee in specific circumstances, the `push_back` of an *xvalue* — made possible by an *rvalue* reference (see Appendix ?? on page ?? **[AUs: Is Appendix I: Genesis of a New Value Category in Modern C++: *xvalues* the book appendix? The only info I have is that the title is Value Categories.]**) — adds the complication of implementing a function template that in some circumstances wants to make a copy but in attempting to do so in other circumstances will fail to compile. This conundrum is precisely the problem the `std::move_if_noexcept` library function was designed to address.

Before digging into the implementation of the *rvalue*-reference overload of a vector-like container's `push_back`, we need to introduce the notion of being **exception agnostic**. C++ code is **exception safe** if it provides the **basic guarantee** that, if an exception is thrown out of a function, no resources are leaked and no invariants are broken; it is exception agnostic if it is considered exception safe without having to resort to the use of exception-specific constructs, such as **try**, **catch**, or **throw**, that, in an exception-disabled build, might fail to compile. Striving for exception agnosticism in a library means relying on **RAII** as the means of avoiding resource leaks when an exception is injected into the code via, for example, a user-supplied callback function, a virtual function in a user-derived class, or an object of user-defined type supplied as a template parameter.

The term **scoped guard** is widely recognized as a category of object whose only purpose is to manage the lifetime of some other object, typically supplied at construction. When the

guard object is destroyed, typically by dint of being an automatic variable leaving lexical scope, it destroys the object in its charge:

```
template <typename T>
struct ScopedGuard
{
   T* d_obj_p;

   ScopedGuard(T* obj) : d_obj_p(obj) { }
   ~ScopedGuard() { delete d_obj_p; }
};
```

Even this overly simplified `ScopedGuard` can be used beneficially to make sure that an object allocated using global **new** early in a function invocation will always be cleaned up, even when control flow leaves the function through an exception or an early return:

```
#include <vector>  // std::vector

void test()
{
    ScopedGuard<std::vector<int>> sg(new std::vector<int>);  // Guarded object.
    sg.d_obj_p->push_back(123);
    // ...
    // ...              (Something might throw.)

}  // guarded object will be released automatically as guard leaves scope
```

A special-case use for a scoped guard is one where it acts as an insurance policy up until some commit point, after which the guard is disabled. Consider a function, `evilFactory`, that dynamically allocates an `std::vector<int>`, populates it, and eventually returns a raw pointer to it, which is not recommended, unless an exception is thrown:

```
std::vector<int>* evilFactory()  // return raw address of dynamic memory (BAD)
{
    ScopedGuard<std::vector<int>> sg(new std::vector<int>);  // Guarded object.

    // ...              (something might throw)

    std::vector<int>* tmp = sg.d_obj_p;  // Extract address of managed object.
    sg.d_obj_p = 0;                      // Release ownership to client.
    return tmp;                          // Return ownership of allocated object.
}
```

In this second example, the client makes use of the guard while the object is being configured. During that period, if an exception is thrown, the guard will automatically destroy and deallocate the dynamically allocated object entrusted to it as the exception exits scope. If, as would be the typical case, no exception is thrown, the object's address is extracted from the guard, the guard's pointer is zeroed out (releasing it from its guard responsibilities), and the raw address of the fully configured dynamic object is returned. (Note this is a pedagogical example and is not recommended). We refer to a scoped guard that provides a

way to release ownership of the managed object, typically via a `release` member function, as a **proctor**.

Let's now return to the principle task of implementing an *rvalue*-reference overload of a `push_back` member function for an `std::vector`-like container but, again for simplicity, ignoring memory allocators. First, we will need a simple **proctor class** that can own a dynamic object and ensure it is destroyed at the end of scope — such as when an exception is thrown — unless ownership is adopted by another object:

```
template <typename T>
class DestructorProctor  // generic "scoped guard" class with release method
{
    T* d_obj_p;  // address of object whose destructor might need to be called

public:
    explicit DestructorProctor(T* p) : d_obj_p(p) { }        // initialize
    ~DestructorProctor() { if (d_obj_p) { d_obj_p->~T(); } } // clean up
    void release() { d_obj_p = 0; }                          // disengage
};
```

With this `DestructorProctor` in hand, we can proceed to implement our `push_back` designed specifically for temporary values. Note that there is no longer a branch on the **`noexcept`** operator as all the necessary logic is handled by `move_if_noexcept` returning the right kind of reference:

```
    std::size_tnewstd::moveTvector<T>noexcept

template <typename T>
void vector<T>::push_back(T&& value)  // safe, efficient implementation
{
    if (d_size < d_capacity)  // if sufficient capacity in allocated memory...
    {
        void* address = d_array_p + d_size;  // implicit conversion to void*
        ::new(address) T(std::move(value));  // may throw on construction
        ++d_size;                            // no throw
        return;                              // early return
    }                                        // else...

    vector<T> tmp;                                  // may throw
    tmp.reserve(d_capacity ? d_capacity * 2 : 1);  // may throw

    void* address = tmp.d_array_p + d_size;              // no throw
    T* newElement = ::new(address) T(std::move(value));  // ctor may throw
    DestructorProctor<T> guard(newElement);        // defend against exception

    for ( ; tmp.d_size != d_size; ++tmp.d_size)    // for each current element
    {
        void* addr = tmp.d_array_p + tmp.d_size;   // no throw
        ::new(addr) T(std::move_if_noexcept(d_array_p[tmp.d_size]));
            // may throw only if move is not noexcept
            // Move if either move is noexcept or T is not copyable;
            // otherwise, copy to preserve strong exception-safety guarantee.
```

```
    }

    guard.release();  // no throw
    ++tmp.d_size;     // no throw
    swap(tmp);        // no throw, committed
}
```

Note that if the above function copies, rather than moves, `T` and an exception is thrown, `tmp` will go out of scope, and the vector's destructor will take care of destroying all of the already-copied objects and deallocating the memory at `tmp.d_array_p`. As a result, the `DestructorProctor` is required to call only the destructor for the in-place-constructed object.

This same implementation strategy exploiting `move_if_noexcept`, used here for `std::vector::push_back(T&&)`, could have also been used for the *lvalue*-reference version (see *Appending an element to an* `std::vector` on page 21), the only differences being that we would construct the new element without calling `std::move` — as `::new(address) T(value);` — in two places. The use of `std::move_if_noexcept` would not be changed, as its purpose is to move construct the existing elements rather than necessarily copy construct the newly inserted element. Note that the loop variable to track the elements that are moved or copied is the `d_size` member of the `tmp` vector to ensure that any *copied* or *moved* objects are destroyed by the destructor of `tmp` if a subsequent move or copy operation throws. While the set of operations looks very different because the two branches of the `if (noexcept(...))` statement in the original formulation of `push_back` appear incompatible, in fact the sequence of operations in this new version is almost identical since we have effectively inlined the `reserve` and copy-assignment operations of the original. The only difference is that, in the case of the nonthrowing move, the loop count is incrementing a member variable of the temporary vector rather than a local variable, and this difference should not be observable in practice.

This generally useful pattern can also be pressed into service to implement the `emplace` member functions; see the next section, *Implementing* `std::vector::emplace_back(Args&&... args)` on page 34.

Implementing `std::vector::emplace_back(Args&&... args)`

_back(args&&...-args)

Another useful way to append an element to a vector is the `emplace_back` function that creates the new element directly in the vector's storage using **perfect forwarding** of *rvalue*-reference parameters, avoiding the need for an additional temporary object that would be needed as the argument for a `push_back` function.

For example, consider appending to a vector of strings:

```
  std::vectorstd::string

void testEmplace()
{
    std::vector<std::string> vs;
    vs.push_back("A long string to defeat a small string optimization");
    vs.emplace_back("A long string to defeat a small string optimization");
}
```

In the case of calling `push_back`, first a temporary `std::string` object is constructed and then passed by-value as an argument for the `push_back` function. In the case of `emplace_back`, a **`const char*`** pointer to the string literal is passing into the `emplace_back` function, which will construct the new `std::string` object directly in place, without need for the intervening temporary.

The general implementation approach requires surprisingly little change from the one proposed for `std::vector::push_back` on an *rvalue* reference; see *Implementing* `std::vector::emplace_back(Args&&... args)` on page 34.

We have provided here, for reference, a complete implementation of the `emplace_back` method that comprises several other modern features of C++; see Section 2.1."**??**" on page **??** and Section 2.1."**??**" on page **??**:

```
template <typename T>
template <typename... Args>
void vector<T>::emplace_back(Args&&... args)  // safe, efficient implementation
{
    if (d_size < d_capacity)  // if sufficient capacity in allocated memory...
    {
        void* address = d_array_p + d_size;  // implicit conversion to void*
        ::new(address) T(std::forward<Args>(args)...);  // may throw on ctor
        ++d_size;                            // no throw
        return;                              // early return
    }

    vector<T> tmp;                                  // may throw
    tmp.reserve(d_capacity ? d_capacity * 2 : 1);  // may throw

    void* address = tmp.d_array_p + d_size;        // no throw
    T* newElement = ::new(address) T(std::forward<Args>(args)...); // may throw
    DestructorProctor<T> guard(newElement);        // defend against exception

    for ( ; tmp.d_size != d_size; ++tmp.d_size)    // for each current element
    {
        void* addr = tmp.d_array_p + tmp.d_size;   // no throw
        ::new(addr) T(std::move_if_noexcept(d_array_p[tmp.d_size]));
            // may throw only if move is not noexcept
            // move if either move is noexcept, or T is not copyable,
            // otherwise copy to preserve strong exception-safety guarantee
    }

    guard.release();  // no throw
    ++tmp.d_size;     // no throw
    swap(tmp);        // no throw, committed
}
```

As you can see, the only two lines that change are those for constructing the new element in place (marked with `[Note]` in the code snippet above), in both the sufficient-capacity branch and the exceeding-capacity branch:

```
::new(address) T(std::forward<Args>(args)...);
```

The constructor called here is the perfect forwarding constructor that can consume an arbitrary number of arguments using a **function parameter pack**, where each different set of types of arguments will produce a different instantiation of the template. The `emplace_back` function is another example of such a perfect forwarding function, called a **variadic function template**. The supplied function parameter pack is denoted by use of an ellipsis, which is again used when passing that pack onto the next function in our example above. The `std::forward` function template must always be called by explicitly specifying the template argument. It is intended for use with **forwarding references** that are a `T&&` template parameter, where `T` will be implicitly deduced from the caller. Following the rules for **reference collapsing**, `T` will either deduce to be an *lvalue* reference or an *xvalue*, so the call to `std::forward<T>` will preserve the category of the supplied arguments when passing through this function call to the constructor for the new element.

g-std::vector::insert

Implementing `std::vector::insert`

To complete the study of vector insert operations, let us consider the case of inserting at an arbitrary position of a vector. If the insertion position happens to be the end of the vector, then the strong exception-safety guarantee holds. If we insert into the middle, then the strong guarantee holds unless a copy or move constructor throws, but, as we shall see, a safe implementation will avoid unnecessary copies and so provide undocumented the strong guarantee if the *move* constructor does not throw. This implementation detail falls out of the common implementation technique used to avoid *aliasing* problems when the element to be inserted is a copy of an element, passed by reference, that is already in the vector. The easiest way to solve this problem is to simply insert all new elements at the back of the vector and then call `std::rotate` to move elements into the correct positions. The `rotate` algorithm will `swap` elements to establish the correct order without allocating more storage. The default implementation of `std::swap` is now implemented as three move operations, so if the move constructor and move-assignment operator do not throw, then insertion at an arbitrary position of a vector will also provide the strong guarantee:

```
#include <algorithm>  // std::rotate
template <typename T>
typename vector<T>::iterator vector<T>::insert(const_iterator position, const T& value)
{
    std::size_t offset = position - begin();
        // iterator may be invalidated below

    push_back(value);
    vector<T>::iterator result = begin() + offset;

    std::rotate(result, end() - 1, end());
        // rotate last element to insert position

    return result;
}
```

Notice that the call to `std::rotate` is a null operation when inserting at the end, providing the necessary strong exception-safety guarantee, even if `swap` for the element type throws.

tfalls-noexceptoperator

xcept-operator-directly

Potential Pitfalls

Using the `noexcept` operator directly

One of the early discoveries in specifying the Standard Library was that, as the **`noexcept`** operator considers the whole expression, the result can be **`false`** due to side effects in subexpressions that were not intended as a direct part of the query. Consider the following constructor declaration:

```
template <typename T>
struct MyType
{
    // ...
    MyType(MyType&& rhs) noexcept(noexcept(T(T())));  // T is type of a member
};
```

This seems like a reasonable way to write an exception specification that is conditional on whether a member of type `T` has a nonthrowing move constructor. The expression under test creates a temporary object of type `T` from another default-constructed, temporary object of type `T`. How could this not return the expected result?

First, this code assumes that `T` has an accessible default constructor. This is a best-effort attempt to create an *rvalue* of type `T` without knowing anything about it and specifically not knowing the syntax for an accessible constructor to initialize a temporary object. The Standard solves this problem by introducing the `declval` function into the `<utility>` header:

```
template <typename T>
typename std::add_rvalue_reference<T>::type std::declval() noexcept;
```

While this function is declared by the Standard Library, using it in a context where it might be evaluated — i.e., outside a **`decltype`**, **`noexcept`**, **`sizeof`**, or similar — is an error. Note that the function is declared unconditionally **`noexcept`** to support its intended use in the **`noexcept`** operator without impacting the final result. This function generally returns an *rvalue* reference, but the use of the `add_rvalue_reference` type trait handles several corner cases. If called with an *lvalue* reference, the reference-collapsing rules will be applied, and the result will be an *lvalue* reference. If called with a type that does not support references, such as **`void`**, the type trait will simply return that same type. Note that as the function signature simply returns a reference but the function itself never defined, the question of how to create the returned *xvalue* at run time is avoided.

With the `declval` function in hand, we can rewrite our exception specification:

```
template <typename T>
struct MyType
{
    // ...
    MyType(MyType&&) noexcept(noexcept(T(std::declval<T>())));
};
```

The use of `std::declval` solved the problem of type `T` not being default constructible, but this approach still has a subtle hidden problem. In addition to testing whether the move constructor for the temporary object of type `T` will not throw, we are also testing

the destructor of that temporary, since, by their nature, temporaries are destroyed at the end of the expression that creates them. This insight led to destructors having special rules, different from every other function, when declared without an exception specification (see *Annoyances — Destructors, but not move constructors, are* **noexcept** *by default* on page 42 below), which was important when recompiling code originally developed, tested, and validated against C++03. However, changing the rules for exception specifications on destructors still does not solve the problem when the destructor is explicitly declared as potentially throwing in new code. The workaround for this is to defer destruction of the temporary by use of the **new** operator. As discussed in *Use Cases — Appending an element to an* `std::vector` on page 21, this can be resolved by careful use of the placement **new** operator:

```
template <typename T>
struct MyType
{
    // ...
    MyType(MyType&&) noexcept(noexcept(::new((T*)0) T(std::declval<T>())));
};
```

The use of the null pointer as the target address does not have undefined behavior here, as the expression is an unevaluated operand passed as an argument to the **noexcept** operator, so only the types involved in the expression, not the values, matter.

The simple attempt to write the strongly motivating use case of a nonthrowing move constructor has turned into a complicated experts-only metaprogram. This is generally the pattern observed by direct use of the **noexcept** operator, and typically such metaprograms are packaged up as clearly named **type traits** where they can be developed, tested, and deployed just once.

The `noexcept` operator doesn't consider function bodies

sider-function-bodies

One source of confusion when first learning about the **noexcept** specification is assuming that the specification is determined by the expressions in the function body. This misconception is enhanced by the rules for implicitly declared, or **defaulted**, functions (see Section 1.1."**??**" on page **??**) producing an exception specification based upon the corresponding exception specification of the bases and members of the class.

Similarly, some people are concerned that the compiler does not enforce **noexcept** by parsing the function definition and rejecting expressions that may throw. This was a deliberate language choice (see *Appendix — Genesis of the* **noexcept** *operator: move operations* on page 47) that throwing out of a **noexcept** function be a runtime failure, not a compile-time error, given the history of C++ and other languages attempting to statically enforce an exception markup. Exception specifications for function templates would have become particularly difficult for library authors, and it would be challenging to adopt any use of **noexcept** for projects compiling against C code or with legacy C++03 code. However, some compilers will issue warnings in cases where an exception is known to be thrown through a **noexcept** specification, i.e., in the case where all code paths lead to an exception and there is no regular return path:

```
void does_not_throw() noexcept
```

```
{
    throw "Oops!";  // OK, calls std::terminate, but a good compiler will warn
}

void should_not_throw(bool lie) noexcept
{
    if (lie)
    {
        throw "Fooled you!";  // OK, but conditional so compilers will not warn
    }
}
```

Conflating `noexcept` with the no-fail guarantee

n-the-no-fail-guarantee

A common source of misunderstanding is conflating the *no-throw* guarantee with the *no-fail* guarantee. A **noexcept** specification does not guarantee that the decorated function cannot fail, but rather that if it does fail — that is, if it cannot satisfy its documented postconditions — then it will terminate the program and not return control flow to the caller. In either case, the caller does not have to worry about failure emitting an exception, but if we do not provide additional documentation that the function provides the *no-fail* guarantee, then the user should be aware that calling the function may terminate the program, and the larger system that the program is part of should be prepared to handle that and restart the program if necessary. An example where this has been used to create robust software is in vehicle-control systems that may run three or more processes as a "voting system" to determine the best course of action or to decide whether to trust some surprising sensor data. The software is designed to fail any program hard when it becomes unreliable and to restart quickly and smoothly, as the redundant processes can continue voting and maintain control of the vehicle systems.

When writing our functions that are decorated with **noexcept**, it is important to document whether they offer the no-fail guarantee as well. When reading and maintaining code that calls a **noexcept** function, we must understand whether **noexcept** also conveys a no-fail guarantee, which should be documented by the library. Note that conflating a no-fail guarantee and **noexcept** occurs frequently enough that a convention is arising that **noexcept** might imply a no-fail guarantee unless the potential for failure is also documented, i.e., documenting by omission. It is generally important to understand the implicit defaults of the documentation of any library being used and to aspire to be explicit about default assumptions in our own documentation.

Annoyances

vances-noexceptoperator

The `noexcept` operator is too sensitive for direct use

ensitive-for-direct-use

As the **noexcept** operator takes account of whole expressions, it can be surprisingly difficult to test just the operations about which the code is concerned. The C++ Standard Library provides a number of type traits that effectively package up the metafunctions necessary to determine such results in reusable components:

```
#include <type_traits>  // all of the std::* traits below
```

```
struct S { };  // trivial object, all implicit operations are noexcept

static_assert(std::is_nothrow_assignable<S,S>::value, "");           // OK
static_assert(std::is_nothrow_constructible<S>::value, "");          // OK
static_assert(std::is_nothrow_copy_assignable<S>::value, "");        // OK
static_assert(std::is_nothrow_copy_constructible<S>::value, "");     // OK
static_assert(std::is_nothrow_default_constructible<S>::value, "");  // OK
static_assert(std::is_nothrow_destructible<S>::value, "");           // OK
static_assert(std::is_nothrow_move_assignable<S>::value, "");        // OK
static_assert(std::is_nothrow_move_constructible<S>::value, "");     // OK
```

A sample implementation might look something like:

```
template <typename T, bool = std::is_move_constructible<T>::value>
struct is_nothrow_move_constructible__impl
    : std::integral_constant<bool, false> { };

template <typename T>
struct is_nothrow_move_constructible__impl<T,true>
    : std::integral_constant<bool,
          noexcept(::new((void*)0) T(std::declval<T>()))> { };

template <typename T>
struct is_nothrow_move_constructible
    : is_nothrow_move_constructible__impl<T> { };
```

See *Potential Pitfalls — Using the* **noexcept** *operator directly* on page 37. Notice that even this simple trait requires a level of indirection through a support class to avoid evaluating a **noexcept** expression that would not compile for a type that is not move constructible at all.

In addition to the traits supplied with C++11, a particular concern arose determining whether a `swap` operation has a nonthrowing exception specification. The popular copy/swap idiom relies on a nonthrowing `swap` operation that is called relying on argument dependent lookup (ADL) with the primary `std::swap` template also found by ordinary name lookup. This is generally achieved with a **using** declaration within the scope of the code calling `swap`, but it is not possible to inject **using** `std::swap` within the single expression being tested by use of a **noexcept** operator. This issue was missed when specifying the Standard Library as the library components themselves are all in namespace `std`, so they find the `std::swap` overload without requiring an additional **using**.[9]

However, unlike the C++11 traits, it is not possible for users to provide the equivalent functionality themselves without invading namespace `std`, something explicitly prohibited by the C++ Standard. These traits can be implemented *only* by the Standard Library itself.

[9]This oversight was addressed in C++17 with the addition of two more type traits:

```
static_assert(std::is_nothrow_swappable<S>::value);           // OK, in C++17
static_assert(std::is_nothrow_swappable_with<S&, S&>::value); // OK, in C++17
```

In practice, use of the **noexcept** operator is often delegated to type traits, provided by either the Standard Library or user code, that can implement, test, debug, and package up the precise metaprogram clearly named for its intended usage.

safety-guarantee-itself

The strong exception-safety guarantee itself

A lot of effort has gone into maintaining support for the strong exception-safety guarantee, and whether this guarantee buys significant benefit over the basic exception-safety guarantee to justify the cost is unclear. If an operation has failed such than an exception is thrown, the error recovery path typically catches at a granularity that will reattempt the whole transaction, including re-creating the objects that were providing the strong exception-safety guarantee for their use.

The intended benefit of the strong exception-safety guarantee is to apply transactional reasoning about code, where an operation must succeed, or leave the system in a good state. The strong guarantee requires that the "good state" be the state prior to starting the transaction, which raises the question of what is likely to change such that re-attempting the transaction would succeed on the second attempt. Generally, the fine-grained transactional guarantees give way to abandoning the whole transaction and starting over, in which case the state preserved by the strong guarantee is also lost. Note that fine-grained transactional reasoning does turn out to be important to atomic operations on concurrent code, but that is entirely distinct from the exception safety guarantees.

In practice, the strong guarantee can prove useful when trying to diagnose problems in software, as it helps the debugging process to know that we can rely on inspecting an object to return the same state as before an operation was attempted. Of course, that relies on the assumption that the bug being diagnosed is not also impacting on the strong exception-safety guarantee of the constituent code.

en-an-std::vector-grows

Change in unspecified behavior when an `std::vector` grows

One of the lesser known corner cases of C++03 is that when a vector grows, as it copies all of its elements into a new, larger array, state that is not copied by default is lost. For example, the capacity of a vector is generally not preserved when a vector is copied, even if the user has explicitly called `reserve` to ensure that the original vector object can grow without reallocating. This becomes an issue when trying to prepopulate a vector so that subsequent use in the program should not force a reallocation:

```
#include <cassert>  // standard C assert macro
#include <vector>   // std::vector

void safe_append(std::vector<std::vector<int>>* target)
{
    assert(target);
    target->push_back(std::vector<int>());
    target->back().reserve(100);
}
```

This function ensures that every new `vector<int>` inserted into the `target` vector has a capacity of at least 100. However, if `target` itself is forced to grow, then all of the existing

vectors in `target` will be copied and have a new capacity computed to reasonably hold the elements they already hold, which could well be a lower capacity, forcing new allocations to happen on later use, which this code was explicitly attempting to avoid.

This is resolved by C++11 for the common case that the move constructor for the element of a vector is **noexcept**, as the vector will *move* rather than *copy* each element into the new array, and move operations frequently preserve more information about the state of the original object, including, in our example, the capacity of a vector.

Note that some programmers might think `std::vector<std::string>` would be a more common example to encounter this problem, but a confluence of fortunate design choices mean that this does not happen in practice. The design of `std::string` for C++03 enabled the copy-on-write optimization, which we believe all Standard Libraries implemented. This design means that when we copy a string, we share a reference-counted implementation and only make a real copy when either string calls a modifying operation (such as appending more characters) when the string being modified finally implements the deferred copy. However, manipulating a hidden shared state turns out to be a real performance problem for concurrent code, which was a major design goal for C++11, so the design of `std::string` changed to support the "short string" optimization instead. A vector of this redesigned string would indeed display the original issue, if we had not simultaneously fixed `std::vector` to sidestep the problem using moves.

Destructors, but not move constructors, are `noexcept` by default

e-noexcept-by-default

Once implementations of the **noexcept** operator were available with early compilers, it quickly became apparent that there were issues with common expressions involving temporary objects. These issues occurred because the **noexcept** operator included the whole lifetime of temporary objects, expressions involving temporaries would call their destructors, and the vast majority of existing C++03 code was written without exception specifications on their destructors.

In fact, the C++03 language did implicitly provide an exception specification for implicitly declared destructors, based upon the exception specification of their bases and members. Clearly the language was already trying to help us out, but as soon as the user wrote their own destructor, which is a common thing to do, that user would have to explicitly mark up that destructor as **noexcept**, or many potential uses of the **noexcept** operator would become irrelevant as it would always return **false**. The solution for this problem was to make destructors — *only* destructors — special, so that if there is no explicitly declared exception specification on a user-supplied destructor, it will be given the same exception specification as though it were implicitly declared. Note that many programmers misstate this rule as "destructors are implicitly **noexcept**". While it is the overwhelmingly common case that destructors are implicitly **noexcept**, the rule allows for a class to explicitly mark its destructor as potentially throwing:

```
struct Bad
{
   ~Bad() noexcept(false) { }
};
```

Any class that had a `Bad` base or member would also have an implicitly throwing destructor.

However, unless someone explicitly writes a class like `Bad` in the first place, the misstated rule holds that destructors generally have nonthrowing exception specifications.

The next question to arise is why move constructors do not get the same treatment as destructors and take the exception specification on an implicitly declared move constructor if there is no explicitly supplied specification. The simplest answer may be that nobody suggested it at the time.

`noexcept` arrived extremely late in the C++11 standardization process, and the need to address destructors was discovered only with eager implementations of the new language feature interacting with the vast majority of existing code. Destructors are extremely common, and a part of almost every object's lifetime. Move constructors were a new feature that required new code to be written so there was no large legacy of incompatible code to address. Move constructors are but one constructor of many, whereas a class could have only one destructor, so the rule that "destructors are special" was relatively easy to learn, whereas a new rule that "move constructors are an extra special constructor in the grammar" was less obvious. Also, there were more common cases where an implicit exception specification based upon the exception specification of the bases and members would give the wrong result for user-defined types. For example, the move constructor is responsible for restoring the invariants of any object whose state it consumes, and if one invariant is that an empty object always has an allocated object (e.g., a sentinel node in an implementation of `std::list`), then the user would be responsible for explicitly marking up that move constructor as **`noexcept(false)`**, or else the program would have a hidden termination condition, which did not seem as useful a default as one that merely misses a library optimization opportunity. Finally, without the large body of pre-existing code, it is not clear that even destructors would have been deemed special enough to have a different default to every other function in the language.

-of-constexpr-functions

Older compilers invade the bodies of `constexpr` functions

It has always been clear that the **`noexcept`** operator was not permitted to infer from the body of an ordinary function, even an empty one, whether it would throw. However, for C++11 and C++14, the behavior for a function declared **`constexpr`** (see Section 2.1."**??**" on page **??**) was underspecified, which meant that some compilers, such as GCC prior to version 9.1, would inspect the body of a **`constexpr`** function when evaluating **`noexcept`**.

A clarification was made in C++17, and the specification is retroactive to C++11 and C++14, so we would expect all of the assertions in the following example to succeed on conforming compilers. Such is the case for all versions of Clang and for GCC version 9.1 and later, but, for GCC versions before 9.1 and MSVC up to 2019, some of these cases have nonconforming results:

```
#include <stdexcept>  // std::runtime_error
          int f0()          { return 0; }
constexpr int f1()          { return 0; }
constexpr int f2(bool e)    { if (e) throw std::runtime_error(""); return 0; }
          int f3() noexcept { return 0; }
constexpr int f4() noexcept { return 0; }

static_assert(!noexcept(f0()),      "");  // OK
```

```
static_assert(!noexcept(f1()),      "");  // OK, but fails on old GCC and MSVC
static_assert(!noexcept(f2(false)), "");  // OK, but fails on old GCC and MSVC
static_assert(!noexcept(f2(true)),  "");  // OK
static_assert( noexcept(f3()),      "");  // OK
static_assert( noexcept(f4()),      "");  // OK
```

Exception-specification constraints in class hierarchies

-in-class-hierarchies

There are occasions when we might want to legitimately override a function that never throws with a function that *can* throw. This is, perhaps, best illustrated by means of an example.

As background, a number of mathematical models using correlation matrices require that those matrices are **positive semidefinite**, a description of which is outside the scope of this book but can be found in most texts on statistics and matrix algebra.[10] Testing whether a matrix meets this requirement is a computationally expensive process for matrices of rank greater than 2, and this property automatically holds true for all rank 1, i.e., trivial, and rank 2 correlation matrices.

Suppose, as a hypothetical example, we have some form of mathematical model that depends on a matrix of correlation values requiring that that correlation matrix is "positive semidefinite." For 2x2 matrices, the calculations are straightforward, and there is no need to check the validity of the input matrix.

When a more enhanced model is created that can handle arbitrarily large matrices, we have to check those matrices for validity:

assert

```
class MySimpleCalculator
{

    // a mathematical model that can handle the simple case of only 1 or 2
    // assets

    virtual void setCorrelations(const SquareMatrix& correlations) noexcept
        // This function takes a correlation matrix that must be a valid 1x1
        // or 2x2 matrix.  If the matrix is larger than 2x2, then the behavior
        // is undefined.
    {
        // This simple calculator can handle only 2x2 correlations.
        assert(correlations.rank() <= 2);
        d_correlations = correlations;
    }
};

class MyEnhancedCalculator : public MySimpleCalculator
{
    // an enhancement that can handle arbitrarily large numbers of assets

    void setCorrelations(const SquareMatrix& correlations) override
```

[10] See, for example, **?**.

```
        // This function takes an arbitrarily large correlation matrix,
        // satisfying a positive semidefiniteness constraint.  If the matrix
        // is not positive semidefinite, then an exception will be thrown.
    {
        // Check positive semidefiniteness only if rank > 2 because it is an expensive
        // calculation and is, by definition, true for correlations when rank<=2.
        if (correlations.rank() > 2 && !correlations.isPositiveSemiDefinite())
        {
            throw MyCorrelationExceptionClass();
        }

        d_correlations = correlations;
    }
};
```

We might think it would be a good idea to put **noexcept** on the base class function, knowing that it can never throw. However, those plans would be foiled by the rules, which would force us to remove exception throwing from any overrides of this function.

So, given the above scenario, what are our options? There are, unfortunately, only three, none of which are ideal.

1. Remove the **noexcept** specifier from the base class. This is the easiest option, but we may sacrifice some compiler optimizations as a result of doing so or surprise clients who have come to depend on that function being **noexcept** in some way. Unfortunately, for base classes contained in third-party libraries, this may not be viable.

2. Make the override function **noexcept** and change the **throw** into an `assert`. This would be effectively kicking the problem of data validation upstream of the function call, saying to every caller, "If you don't want the program to die, don't pass bad data." It also means that unit testing could be done only by using "death tests," which has performance implications for the compilation/testing cycle.

3. Make the override function **noexcept** and simply fail to produce a useful result when given an invalid matrix, possibly with some other derived-class specific state to indicate being in an invalid state. This is, likewise, effectively kicking the problem upstream through very indirect error reporting and handling and has added complications when it comes to unit testing.

Generally, if we are writing a class and there is any possibility that we or a client might consider inheriting from that class now or in the future and we are considering making any of the **virtual** member functions **noexcept**, we must consider carefully whether the benefits of **noexcept** outweigh the disadvantages.

Note that when we choose to remove a **noexcept** specifier from a base-class function, there may be cascading consequences:

```
class DataTable
{
    // This class has some data and appropriate virtual accessors.
```

```
    virtual Data getValue(/*...*/) const noexcept;
        // Return some of the held data based on arguments passed in.
};

class Calculator
{
    virtual double getUsefulStatistic(const DataTable& dt) noexcept;
        // given a DataTable dt, makes one or more calls to dt.getValue and
        // performs some calculation to generate a result
};
```

Suppose we want to do the same calculation on data in a database:

```
class LazyLoadingDataTable : public DataTable
{
    // holds a database connection and a mutable data cache

    Data getValue(/*...*/) const override;
        // queries the database for values not in cache and throws
        // DatabaseException in the event of any issues
};
```

Given the rules around **noexcept**, this will give us a compilation error. The compilation error can be resolved by removing the **noexcept** specifier on the `DataTable::getValue` function. This will now compile, so presumably all is well.

Let's examine what could happen if we have, for example, network congestion.

1. The program constructs a `LazyLoadingDataTable` with appropriate database connection information.
2. The program passes that into `Calculator::getUsefulStatistic`.
3. `Calculator::getUsefulStatistic` calls into `LazyLoadingDataTable::getValue`.
4. There is a database timeout, and an exception of type `DatabaseException` is thrown.
5. Because `getUsefulStatistic` is **noexcept**, the exception cannot propagate.
6. The program terminates.

This can be an issue not just in production systems, but also in unit testing. For example, suppose we want to test the above class with a mock database connection, which throws an exception; we would have to again resort to death testing.

Functions declared `noexcept` must be bulletproof

t-must-be-bulletproof

If we mark any function **noexcept**, then we risk terminating the program unexpectedly. For that reason, we need to not only avoid throwing exceptions ourselves, but also be absolutely certain that anything we call and anything we depend on is guaranteed to not throw any

exceptions. When designing mission critical production systems, it is important that any failure case will result in "graceful degradation" rather than a cataclysmic outage.[11]

Suppose, hypothetically, an application writer links in a poorly tested, third-party library that sends status information to a central dashboard display. If, for whatever reason, that library fails to open a socket to the dashboard server, then an exception is thrown. If that were to happen, then the core functionality of the application may be able to continue without the dashboard reporting features, the exception having been caught and logged.

If, on the other hand, that exception were to hit a **noexcept** barrier somewhere within its call stack, then the application would terminate. A worst-case scenario is that multiple applications reporting their status into the same dashboard would all require a fast, risky deployment of new versions with the third-party library dependency completely excised to reinstate the production systems.

See Also

see-also

TODO

Further Reading

further-reading

- ?

Appendix

pendix-noexceptoperator

erator:-move-operations

Genesis of the noexcept operator: move operations

Late in the C++11 standardization process, a problem was discovered with the library optimizations that motivated the addition of *move* operations into C++. The problem was deemed serious enough to adopt a proposal for a new language feature at the Pittsburgh meeting, March 2010, over a year after the cut-off for new features for C++11. [**AUs: from VR: LORI, PLEASE ASK JOHN FOR THE CITATION HERE, I COULDN'T FIND IT.**] That problem was breaking the strong exception-safety guarantee when inserting an element into a vector; see *Use Cases — Appending an element to an* `std::vector` on page 21. Much code has been written and tested expecting this guarantee to hold.

A simple fix would be to require that types provide a nonthrowing *move* constructor to establish the strong exception-safety guarantee. It is relatively simple to provide this guarantee when implementing a *move* constructor for our type, but that involves writing new code. What happens for code from the C++03 era when we recompile under the new rules? This code does not have any *move* constructors defined, as the C++03 language did not support them. Instead, when the vector tries to move each element, overload resolution will find the *copy* constructor and make copies. While we can expect a specially written *move* constructor to provide a no-throw guarantee, we cannot expect the same of *copy* constructors, which often need to allocate memory, such as when a type has a `std::vector` or `std::string` data member.

There are two parts to solving this problem for C++11, and both were adopted at the March 2010 meeting. The first part is an attempt to implicitly upgrade the existing C++03

[11] [**AUs: Please provide this source**] See O'Reilly, Site Reliability Engineering by various authors at Google.

code. Paper N3053[12] provides implicitly declared *move* constructors and *move*-assignment operators for classes that follow the rule of zero, i.e., for classes that rely on the implicit declaration of the *copy* constructor, *copy*-assignment operator, and destructor. Implicit declaration was *not* desired in cases where users have themselves defined any of these functions though, as that suggests that there is some internal state management that must be performed, such as releasing owned resources. The implicit definitions for the move operations are simple member-wise *move*-construct or *move*-assign operations, just as the implicitly defined copy operations call copy constructors/assignment operators. Note that C++03 code for the implicitly defined move operations will generally just *copy*, as the pre-existing C++03 code could not itself provide *move* overloads. However, C++03 class templates instantiated with C++11 move-optimized types will *move* correctly for their implicitly declared *move* operations. The important point is that move-optimized types rarely throw on move construction or move assignment, providing the guarantee that `std::vector` requires but for only that subset of user-supplied class types with optimized move.

The second part was the introduction of the **`noexcept`** operator in paper N3050.[13] This operator acts upon the types returned by expressions, much like the **`sizeof`** operator, to query whether *any* of the subexpressions that comprise the full expression *are allowed to* emit an exception rather than return. If it is known that calling a *move* constructor for the template parameter type, `T`, cannot throw, then there is no need to maintain a duplicate copy of each element when populating the new array after a vector grows to satisfy a new capacity. This guarantee allows the vector to attempt all the potentially throwing operations first (allocating the new array and constructing any new elements) and only then safely moving all the existing elements, preserving the strong exception-safety guarantee, even for old code recompiled with the new library optimization. Otherwise, as we know that a *move* *can* throw, the library falls back on the old nonoptimized behavior that makes a copy of each element into the array before updating the internal pointers to take ownership. Note that paper N3053 even guarantees that library optimization is available for some subset of C++03 types, but most such types will need updating to support nonthrowing *moves* to gain the benefit.

The final part of the puzzle is the **`noexcept`** exception specification that fuels the **`noexcept`** operator, added by the same paper N3050[14]; see Section 3.1."**??**" on page **??**.

[12] **?**
[13] **?**
[14] **?**

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe
sec-unsafe-cpp11 Intro text should be here.

Chapter 3 Unsafe Features

sec-unsafe-cpp14