

Section 2.1 C++11

noexcept Operator

```
int f(int) noexcept; // Function f is noexcept(true).
int g(int);         // Function g is noexcept(false).

static_assert( noexcept( f(17) ), ""); // OK, f is noexcept(true).
static_assert(!noexcept( g(17) ), ""); // OK, 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) ), ""); // OK, g is noexcept(false).
static_assert(!noexcept( g(1) + f(2) ), ""); // OK, " " " "
static_assert(!noexcept( f(1) + g(2) ), ""); // OK, " " " "
```

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

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) ), ""); // OK, g is noexcept(false).
static_assert(!noexcept( g(1) || f(2) ), ""); // OK, " " " "
static_assert(!noexcept( f(1) || g(2) ), ""); // OK, " " " "
static_assert(!noexcept( true || g(2) ), ""); // OK, 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 654.

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 to propagate the **value** of one object to another; see Section 2.1. “*Rvalue References*” on page 712. 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: