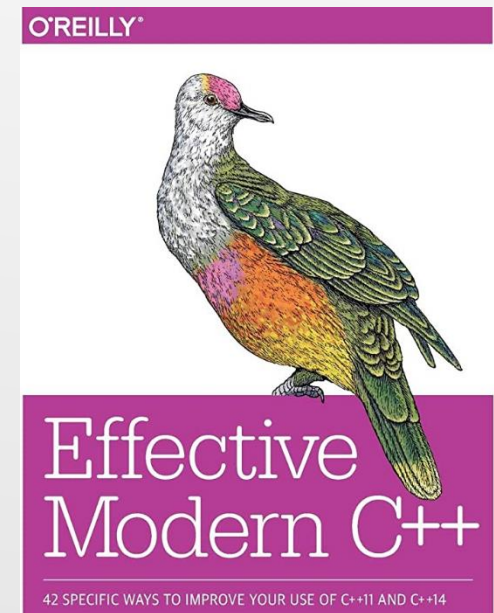


ITEM 14:

Declare Functions **noexcept** If They Won't Emit Exceptions.

Scott Meyers - Effective Modern C++

Presented By
@7eRoM



- ▶ In C++11, unconditional **noexcept** is for functions that guarantee they won't emit exceptions.
- ▶ Permits compilers to generate better object code.
- ▶ Consider a function *f* that promises callers they'll never receive an exception:

```
int f(int x) throw();    // no exceptions from f: C++98 style  
int f(int x) noexcept;  // no exceptions from f: C++11 style
```

- ▶ If, at runtime, an exception leaves *f*, *f*'s exception specification is violated.
- ▶ The **stack unwinding** is a process where the function call stack entries are removed at runtime. To remove stack elements, we can use exceptions. If an exception is thrown from the inner function, then all of the entries of the stack is removed, and return to the main invoker function.



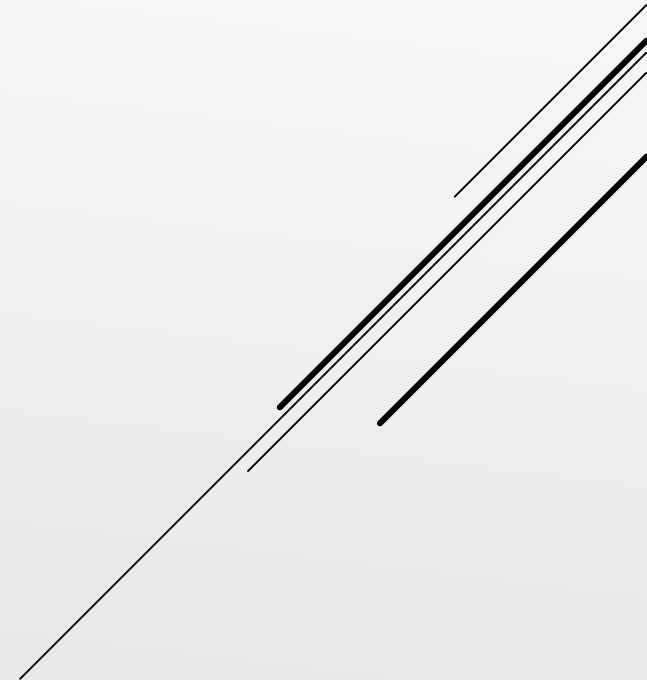
```

1  #include <iostream>
2  using namespace std;
3
4  void function1() throw (int) {
5      // This function throws exception
6      cout << "\n Entering into function 1";
7      throw 100;
8      cout << "\n Exiting function 1";
9  }
10
11 void function2() throw (int) {
12     // This function calls function 1
13     cout << "\n Entering into function 2";
14     function1();
15     cout << "\n Exiting function 2";
16 }
17
18 void function3() {
19     // Function to call function2, and handle exception thrown by function1
20     cout << "\n Entering function 3 ";
21     try {
22         function2(); //try to execute function 2
23     }
24     catch (int i) {
25         cout << "\n Caught Exception: " << i;
26     }
27     cout << "\n Exiting function 3";
28 }
29
30 int main() {
31     function3();
32     return 0;
33 }

```

Entering function 3
 Entering into function 2
 Entering into function 1
 Caught Exception: 100
 Exiting function 3

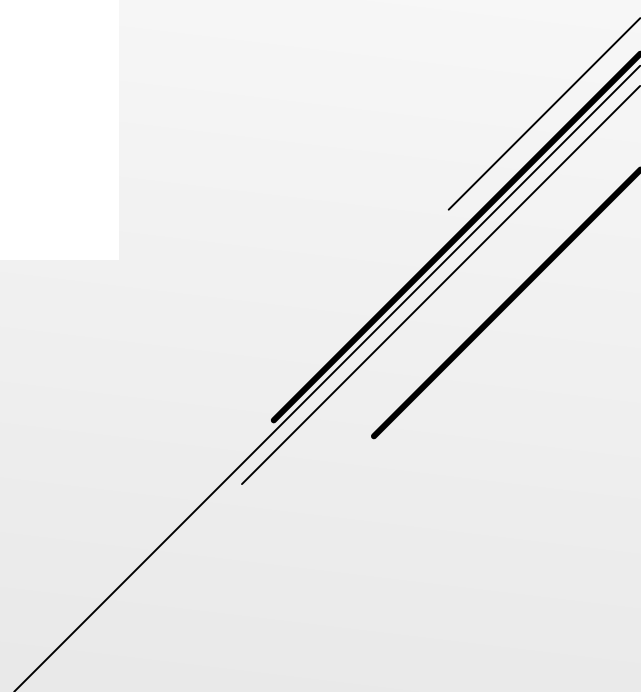
- ▶ **With the C++98 exception specification:**
 - ▶ The call stack is unwound to f's caller, and
 - ▶ After some actions not relevant here, program execution is terminated.
- ▶ **With the C++11 exception specification:**
 - ▶ The stack is only possibly unwound
 - ▶ Program execution is terminated.
- ▶ The difference between unwinding the call stack and possibly unwinding it has a surprisingly large impact on code generation. In a **noexcept** function,
 - ▶ optimizers need not keep the runtime stack in an unwindable state if an exception would propagate out of the function,
 - ▶ optimizers need not ensure that objects in a **noexcept** function are destroyed in the inverse order of construction should an exception leave the function.



RetType function(params) noexcept; // most optimizable

RetType function(params) throw(); // less optimizable

RetType function(params); // less optimizable

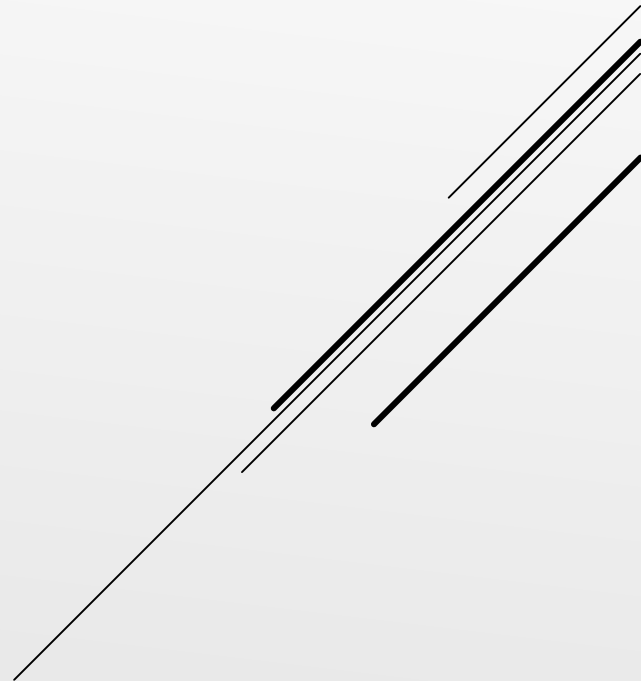


- ▶ **No-throw guarantee (failure transparency):** Operations are guaranteed to succeed and satisfy all requirements even in exceptional situations. If an exception occurs, it will be handled internally and not observed by clients.
- ▶ **Strong exception safety (commit or rollback semantics):** Operations can fail, but failed operations are guaranteed to have no side effects, leaving the original values intact.
- ▶ **Basic exception safety:** Partial execution of failed operations can result in side effects, but all invariants are preserved. Any stored data will contain valid values which may differ from the original values. Resource leaks (including memory leaks) are commonly ruled out by an invariant stating that all resources are accounted for and managed.
- ▶ **No exception safety:** No guarantees are made.

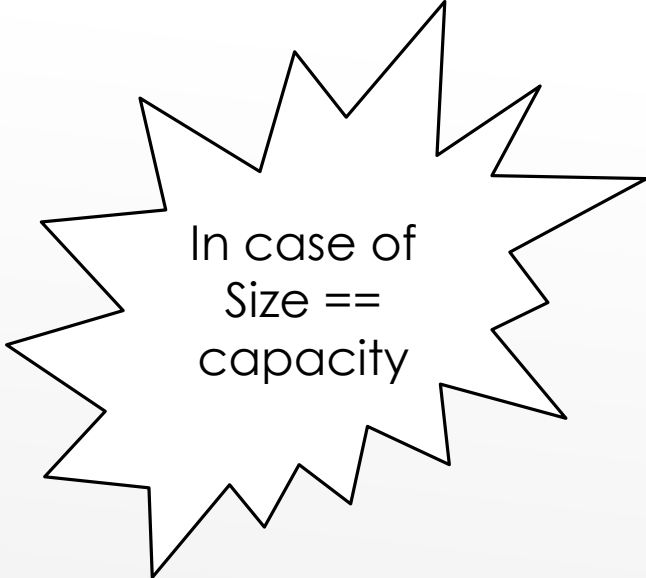
Classifications of Exception Safety

A series of three parallel diagonal lines extending from the bottom left towards the top right, located in the bottom right corner of the slide.

- ▶ Consider a `std::vector`, when an item `x` is added to a vector `v` if the existing capacity isn't sufficient.
- ▶ **No-throw guarantee.** Implemented by ensuring that memory allocation never fails, or by defining the `insert` function's behavior on allocation failure (for example, by having the function return a boolean result indicating whether the insertion took place).
- ▶ **Strong exception safety.** Implemented by doing any necessary allocation first, and then swapping buffers if no errors are encountered (the copy-and-swap idiom). In this case, either the insertion of `x` into `v` succeeds, or `v` remains unchanged despite the allocation failure.
- ▶ **Basic exception safety.** Implemented by ensuring that the count field is guaranteed to reflect the final size of `v`. For example, if an error is encountered, the `insert` function might completely deallocate `v` and reset its count field to zero. On failure, no resources are leaked, but `v`'s old value is not preserved.
- ▶ **No exception safety.** An insertion failure might lead to corrupted content in `v`, an incorrect value in the count field, or a resource leak.



```
std::vector<Widget> vw;  
  
...  
  
Widget w;  
  
... // work with w  
  
vw.push_back(w); // add w to vw
```



In case of
Size ==
capacity

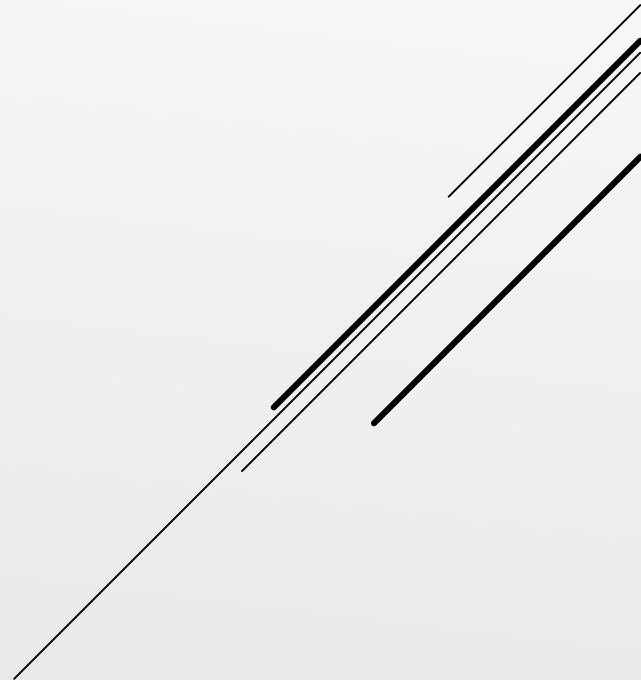
- ▶ In C++98, the transfer was accomplished by copying each element from the old memory to the new memory, then destroying the objects in the old memory. (strong exception safety guarantee)
- ▶ In C++11,
 - ▶ a natural optimization would be to replace the copying of elements with moves.
 - ▶ because the behavior of legacy code could depend on `push_back`'s strong exception safety guarantee. Therefore, C++11 implementations can't silently replace copy operations inside `push_back` with moves unless it's known that the move operations won't emit exceptions.
 - ▶ “move if you can, but copy if you must” strategy

- ▶ How can a function know if a move operation won't produce an exception?
- ▶ If the operation is declared **noexcept**
- ▶ These functions are conditionally **noexcept**

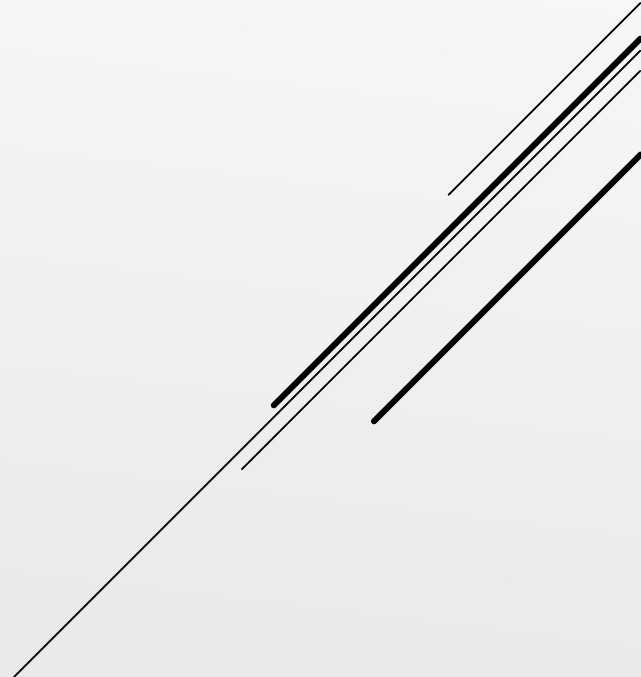
```
template <class T, size_t N>
void swap(T (&a)[N],                      // see
          T (&b)[N]) noexcept(noexcept(swap(*a, *b))); // below

template <class T1, class T2>
struct pair {
    ...
    void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                                noexcept(swap(second, p.second)));
    ...
};
```

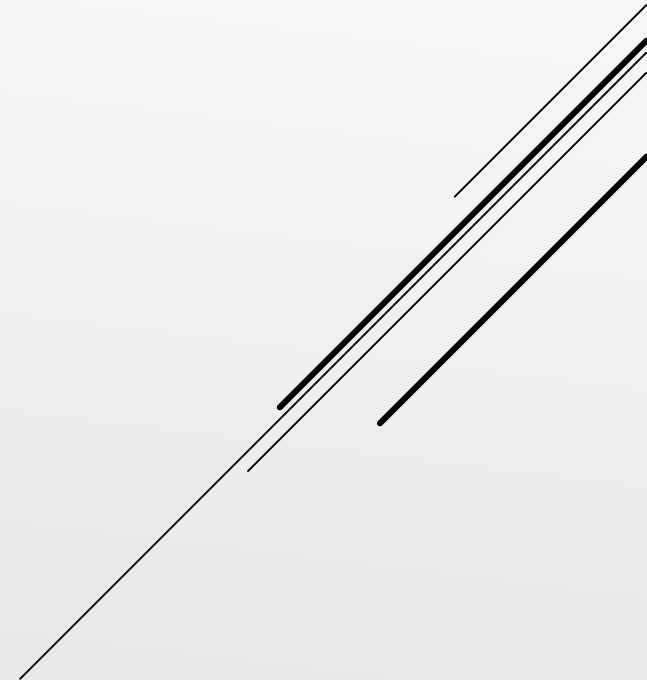
- ▶ Most functions are **exception-neutral**.
 - ▶ throw no exceptions themselves, but functions they call might emit one.
 - ▶ allows the emitted exception to pass through on its way to a handler further up the call chain
 - ▶ are never **noexcept**, because they may emit such “just passing through” exceptions
- ▶ When you can honestly say that a function should never emit exceptions, you should definitely declare it **noexcept**
- ▶ But twisting a function’s implementation to permit a **noexcept** declaration is bad practice.
 - ▶ Complicate code at callee (e.g., catching all exceptions and replacing them with status codes or special return values)
 - ▶ Complicate code at caller (e.g. checking for status codes or special return values)
 - ▶ The runtime cost of those complications (e.g., extra branches, larger functions that put more pressure on instruction caches)



- ▶ Memory deallocation functions (i.e., `operator delete` and `operator delete[]`) and Destructors:
 - ▶ in C++98, it was considered bad style to permit to emit exceptions
 - ▶ in C++11, this style rule has been all but upgraded to a language rule
 - ▶ no need to declare them `noexcept`
 - ▶ the only time a destructor is not implicitly `noexcept` is when a data member of the class (including inherited members and those contained inside other data members) is of a type that expressly states that its destructor may emit exceptions (e.g., declares it “`noexcept(false)`”)
 - ▶ such destructors are uncommon and there are none in the Standard Library



- ▶ **Wide Contract** functions have well-defined behavior for all possible inputs. In other words, A function with a wide contract has no preconditions. Such a function may be called regardless of the state of the program, and it imposes no constraints on the arguments that callers pass it. Functions with wide contracts never exhibit undefined behavior.
- ▶ **Narrow Contracts** mean that the functions can only be called when certain preconditions are met. For such functions, if a precondition is violated, results are undefined.
- ▶ Some examples:
 - ▶ `std::vector<int>`'s `.size()` member function has a wide contract because it can be called on any instance of a vector
 - ▶ The `operator[](size_t index)` (as in `int x = v[10]`) has a narrow contract, because it can only be called with a parameter that is less than `.size()`, otherwise it is undefined.
 - ▶ The `.at(size_t i)` member function (as in `int y = v.at(10)`) however has a wide contract, because it is specified to throw an exception when the index is out of range.



► Wide Contracts:

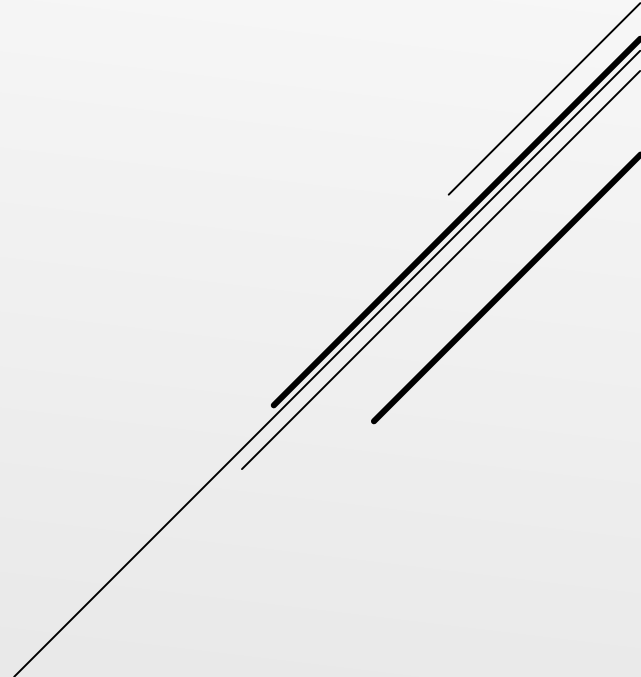
- If you're writing a function with a wide contract and you know it won't emit exceptions, declare it `noexcept`

► Narrow Contracts:

- suppose you're writing a function `f` taking a `std::string` parameter, and suppose `f`'s natural implementation never yields an exception. That suggests that `f` should be declared `noexcept`.
- suppose that `f` has a precondition: the length of its `std::string` parameter doesn't exceed 32 characters. declaring `f noexcept` seems appropriate ☺

```
void f(const std::string& s) noexcept;    // precondition:  
                                         // s.length() <= 32
```

- **Note:** `f` is under no obligation (but could be useful e.g., during system testing) to check this precondition, because functions may assume that their preconditions are satisfied. (Callers are responsible for ensuring that such assumptions are valid.)



- Compilers typically offer no help in identifying inconsistencies between function implementations and their exception specifications.

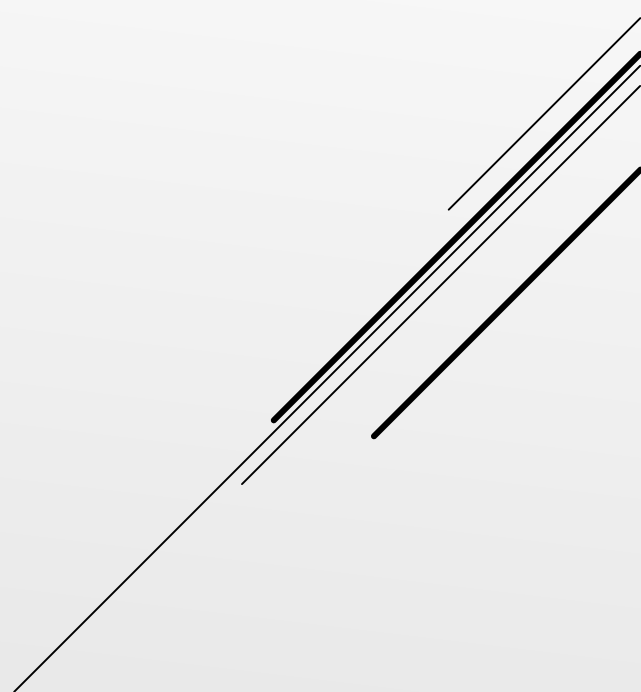
```
void setup();           // functions defined elsewhere
void cleanup();

void doWork() noexcept
{
    setup();            // set up work to be done

    ...                // do the actual work

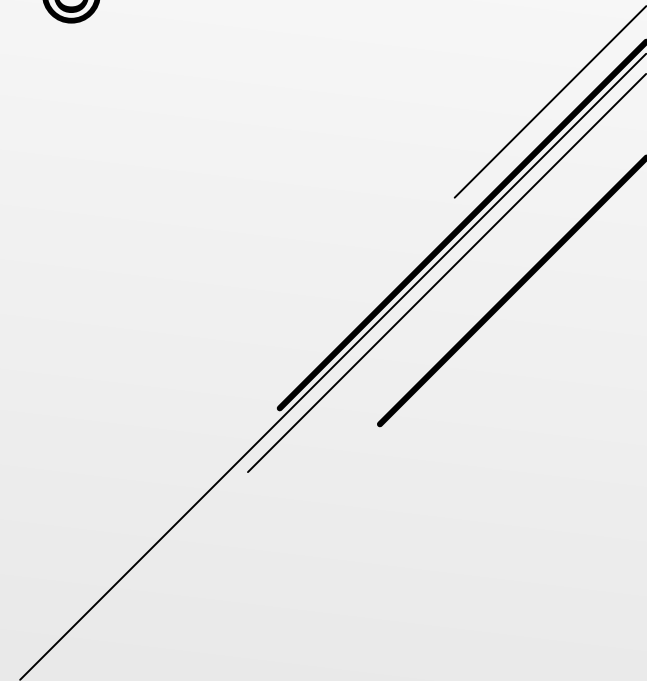
    cleanup();          // perform cleanup actions
}
```

- It could be that `setup` and `cleanup` document that they never emit exceptions, even though they're not declared that way.
- There could be good reasons for their non-`noexcept` declarations. For example, they might be part of a library written in C Or they could be part of a C++98 library that decided not to use C++98 exception specifications and hasn't yet been revised for C++11.
- Because there are legitimate reasons for `noexcept` functions to rely on code lacking the `noexcept` guarantee, C++ permits such code, and compilers generally don't issue warnings about it.





This item was quite challenging!



Ref.

- ▶ <https://www.tutorialspoint.com/stack-unwinding-in-cplusplus>
- ▶ https://en.wikipedia.org/wiki/Exception_safety
- ▶ <https://stackoverflow.com/questions/51292673/what-is-in-simple-understanding-narrow-contract-and-wide-contract-in-terms-of>

