Andrei Alexandrescu and David B. Held

Exception Safety Analysis

Programming is understanding, and exceptions are no exception

This installment of Generic<Programming>, coauthored with Dave Held, takes a short break from the "Smart Pointers Reloaded" miniseries to discuss how to analyze the exception safety of a function. Although my book *Modern C++ Design* [1] pioneered policy-based smart pointers, it glossed over the details of figuring out behavior in the presence of exceptions thrown by various policy implementations. To, ahem, its author's pride, two years after publication, the only known bug of `SmartPtr` (now under the undercover name `smart_ptr`) was that it incorrectly managed resources during construction in the presence of exceptions. Dave took care of fixing this bug, with much help (in the form of ideas, suggestions, criticisms, and code) from the Boost community.

After having a little fun with macros, we discuss how to analyze a function for exception safety. We'll resume the miniseries in the future, when we'll apply the exception-safety analysis to `smart_ptr`.

Mailcontainer

Rob Mann sent in a nice idea that applies to the assertions framework described in Generic<Programming> [2,3]. The idea is more general though, so it might be of interest to many people who enjoy defining new and improved macros.

Macros and C++ have always been in a codependent relationship—C++ did a lot to rid itself of macros, yet it still depends on them in a few key ways. Rob's macro trick, when generalized, lets you define a macro that injects a variable in a yet-to-be-entered scope. For example:

Andrei is a graduate student in Computer Science at the University of Washington and author of *Modern C++ Design*. **David** is a consultant specializing in custom software development. They can be contacted at andrei@metalanguage.com and dheld@codelogicconsulting.com, respectively.

```
INJECT_VARIABLE(string, x)
{
   ... use string variable x here ...
}   // x's scope ends here
// if there's an outer x defined, its
// scope resumes here
```

This trick is quite powerful because it lets macros define variables without the fear of name clashes. My article on assertions [2] uses a complicated scheme that generates variable names by (ab)using `__LINE__`, with the caveat that you can't use the macro twice on the same line—an unusual requirement even for seasoned programmers.

The workings of `INJECT_VARIABLE` are simple, and many people have "discovered" it before. There are two to inject a variable—you can abuse the `if` statement, or you can abuse the `for` statement.

The first way of injecting a variable via a macro (see [4]) is to use the relatively recent ability to define a variable in an `if` test expression:

```
if (int x = Fun())
{
  ... x is defined here ...
}
else
{
  ... x is defined here (and amounts to
      0 upon entering) ...
}
// if there's an outer x defined, its
// scope resumes here
```

The `x` variable exists through the True and False branches of the `if` statement. There are two drawbacks to this trick:

- You can't define static variables.
- `x` must be testable and, as such, it must define an operator `bool()` or something similar. If you do define for your type `MyType` an operator `bool()` that always returns False, then you can inject a variable of your type with a macro such as:

```
#define INJECT_VARIABLE(name) \
   if (MyType name) ; else
```

Or, you simply define your macro to start a `for` statement that defines the needed variable. For example:

```
#define INJECT_VARIABLE(type, name) \
   for (type name; ; )
```

Of course, if you use `INJECT_VARIABLE` in this form, you'll forever execute the statement that follows it. To make sure you execute the statement only once, you'd need to wrap the `for` loop in an `if` statement, which defines a variable to help with execution control:

```
#define INJECT_VARIABLE(type, name) \
   if (bool obfuscatedName = false) ; else \
      for (type name; !obfuscatedName;
           obfuscatedName = true)
```

Everything works fine now, with two requirements on client code. One is that clients don't try to use the name `obfuscatedName` themselves. (As its name implies, you can obfuscate that name in any way you wish, perhaps by using the `__LINE__`-based trick in [2]). The other issue is that, by design, the `break` and `continue` statements inside an `INJECT_VARIABLE` block terminate the block's execution, which might be confusing to the macro's users. On the efficiency front, fortunately, your compiler can most likely optimize-away the two `obfuscatedName` tests.

An interesting application of `INJECT_VARIABLE` is to emulate Java's `synchronized` keyword, which prescribes serial execution for a piece of code or a function. We assume we have a class `Mutex` that implements, well, a mutex, and a class `Lock` that automates the locking/unlocking of a `Mutex` in a "resource acquisition is initialization" manner. Then, you can define the following macro:

```
#define SYNCHRONIZED \
   INJECT_VARIABLE(static Mutex,
                   obfuscatedMutexName) \
   INJECT_VARIABLE(Lock, obfuscatedLockName
                   (obfuscatedMutexName))
      Now you can write:
void Foo()
{
   ... reentrant code ...
   SYNCHRONIZED
   {
      ... nonreentrant code ...
   }
   ... reentrant code ...
}
```

and be sure that the nonreentrant portion of the code is always executed in only one thread at a time. The job of initializing the static `Mutex` in a thread-safe manner is left as an exercise to the reader. (Yes, it is doable; and no, it isn't trivial.)

Java also provides a one-argument version of `synchronized`, which synchronizes operation on a certain object. You can emulate that Java construct by defining `SYNCHRONIZED_OBJ`:

```
#define SYNCHRONIZED_OBJ(obj) \
   INJECT_VARIABLE(Lock, obfuscatedLockName((obj).GetMutex()))
```

and by using it as follows:

```
class Widget
{
   Mutex myMutex_;
public:
   Mutex& GetMutex() { return myMutex; }
   ...
};
void Foo()
{
   Widget widget;
   ... reentrant code ...
   SYNCHRONIZED_OBJ(widget)
   {
      ... widget-synchronized code ...
   }
   ... reentrant code ...
}
```

Pretty neat. In a more involved implementation, you might want to protect `Mutex`'s functions so that only `Lock`s can access your `Mutex`.

Finally, Java's `synchronized` keyword is applicable to entire methods, such as:

```
// Java code follows
class Javanilla
{
   public void DoSomething() synchronized
   {
      ... this is a nice thread-safe function ...
   }
   ...
}
```

You might think that emulating this usage of the `synchronized` keyword isn't possible in C++. You might even think—and rightly so—that you don't even need that keyword and that you can satisfy all of your scoped locking needs with the `SYNCHRONIZED` and `SYNCHRONIZED_OBJ` macros. However, as the Greeks might say, the cup of disgrace has no bottom; people without honor can drink from it without ever emptying it. You did not hear from me about the following macro:

```
#define SYNCHRONIZED_METHOD \
    try { throw Lock(this->getMutex()); } catch (Lock&)
```

Now, if you used it like this (but, of course, you'd never use it):

```
class Widget
{
public:
    void DoSomething() SYNCHRONIZED_METHOD
    {
        ... this is a thread-safe function ...
    }
}
```

The `SYNCHRONIZED_METHOD` macro causes the function to look like (reformatted):

```
        void DoSomething()
        try
        {
            throw Lock(this->getMutex());
        }
        catch (Lock&)
        {
            ... this is a nice thread-safe function ...
        }
```

This form is a valid "function-try-block." It is legal to have a `try` statement as the body of the function, without having to include it inside brackets. This concession is unique to the `try` statement—there's no "function-for-block" or anything similar. That's why the only way to inject code into the body of a function from the outside is to abuse the `try` statement as shown.

The long and short of `SYNCRHONIZED_METHOD` is, don't do it. It's here as a curiosity. The good thing to do is to simply synchronize inside the function by using `SYNCHRONIZED_OBJ(*this)`.

Purity and Exception Safety

Whenever you write a function, you need to have an understanding of its behavior on exceptional paths and make a statement about that function's behavior in the presence of exceptions [5]. That summary is relevant to your function's callers. To briefly recap the canonical exception-safety guarantees, from weakest to strongest [8]:

- The basic guarantee: the invariants of the function's arguments are preserved and no resources are leaked.
- The strong guarantee: the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
- The nothrow guarantee: the operation will not throw an exception.

You should, of course, strive for the strongest guarantee that's possible and affordable. There could also be "the wacky guarantee," which means that the system is in an undefined state in which you have no idea what might happen next, so you better call `abort` ASAP. In spite of the popularity of the wacky guarantee, we don't discuss it in depth here.

We felt in need of another characterization of functions: A function is pure when it doesn't engender any side effects. Pure functions (not to be confused with pure virtual functions) are well-known concepts; in functional languages such as Haskell or ML, all functions are pure, leading to a stateless model of computation.

Put another way, a pure function does not depend on program state and does not change the program state in any way. A simple example is `int operator+(int, int)`. Two `int`s go in, one `int` comes out—and nothing had better change!

Now consider a function `Fun` that simply calls two other functions in a row:

```
void Fun()
{
   Gun();
   Hun();
}
```

We set out to analyze the exception-safety level (basic, strong, or nothrow) of `Fun`, assuming we already know the exception-safety level of `Gun` and `Hun`.

If both `Gun` and `Hun` are nothrow, `Fun` is nothrow and we're home free. If `Gun` is strong and `Hun` is nothrow, then `Fun` is strong. This is

because we have an operation with transactional behavior followed by an operation that never fails. So if Gun fails, then Fun fails in a "nothing happened" manner; and if Gun doesn't fail, Hun never fails and thus successfully concludes Fun's execution. This is strong behavior.

Now, what if both Gun and Hun are strong? Well, if Gun succeeds and Hun throws, then we have a problem. What if Gun already modified the program state? That would be a state of partial success for Fun, and so Fun's exception-safety level decays to…

To what? Basic or wacky? It could be either, really. If Gun and Hun alter subobjects of a larger object, the intermediate state between Gun's success and Hun's failure might be undefined as far as the larger object's invariant is concerned.

However—and here's where purity comes into play—if Gun is strong and pure, then Fun stays strong. This is because if Hun fails, there's no change in the program state whatsoever, so Fun behaves in a strong manner just as Hun does.

It's clear that analyzing the exception safety of a function involves the purity of the functions that that function calls. But beware; purity is not absolute (just like in real life, Machiavelli would hasten to add). Even if impure by construction, Gun can be pure as far as Fun's analysis is concerned, as long as it only modifies local automatic variables of Fun. During execution, Fun can freely modify, directly or via other function calls, its own automatic variables, without putting its purity in jeopardy. Here's an example:

```
int Sum(int i1, int i2) // silly, but pure
{
    std::vector<char> v; // automatic variable
    v.push_back(i1);  // impure function call
    v.push_back(i2);  // impure function call
    return std::sum(v.begin(), v.end());
}
```

Note that said function-local data must be automatic—that is, not preserve state between calls. If you make v in the aforementioned example static, Sum's return value depends on previous calls to itself, which violates the notion of purity.

The important conclusion to drive home from this discussion is that, to make a sequence of function calls strong, you need to structure it as a pure subsequence (possibly altering local automatic data), followed by no more than one strong call, followed by a nothrow subsequence. This is exactly what the "assignment-through-swap idiom" [6] does (actually, Dave proved that all strong functions have this form, but the proof does not fit in the margin of this text).

Gaining Strength

Back to Fun, Gun, and Hun. In certain situations, structuring code in pure + nothrow subsequences is difficult due to the way Gun and Hun are implemented. For example, Gun might always manipulate global state, and you cannot redirect it to write to Fun's local automatic data.

In that case, the try/catch statement can be used to restore strength:

```
void Fun()
{
    Gun();
    try
    {
        Hun();
    }
    catch (...)
    {
        ... undo Gun's effects ...
        throw;
    }
}
```

If there's no way of undoing Gun, you cannot make Fun strong no matter how hard you try. If Gun throws, Fun's purity cannot be restored, and as such, Fun doesn't have transactional behavior anymore. Moreover, the undoing code must be nothrow.

This idiom is a workable alternative to sequencing calls, but you should use it only when you cannot sequence your code as prescribed in the previous section. It has three drawbacks:

- It is more verbose and does not scale well. You need to add a lot of code per function call in the sequence.
- It is harder to check, either by hand or automatically. This is because it's usually not easy to make sure that the undo operations actually undo whatever Gun did.
- The generated code might be less efficient with many of today's compilers (such as Microsoft Visual C++) than code that has no try/catch in sight.

For a solution that does not have the first and third drawbacks, you may want to refer to the already well-established ScopeGuard [7].

Restoring Invariants

Consider this simple function:

```
void ReNew(char*& buffer, size_t s)
{
```

```
        delete [] buffer;
        buffer = new char[s];
    }
```

Besides resizing `buffer`, we promise the caller that `buffer` will always return valid or `null`. If you applied what you know so far, you'd conclude that `ReNew` offers the basic guarantee. It calls an impure nothrow function, then a strong function.

However, this is not the case. If `new` fails by throwing an exception, then `ReNew` breaks its promise and leaves `buffer` with a singular value. So in reality, you're dealing with the wacky guarantee.

Thus, when invariants are temporarily broken, you should always use nothrow calls to restore them. Here are two possible rewrites of `ReNew`:

```
// basic guarantee
void ReNew(char*& buffer, size_t s)
{
    delete[] buffer;
    buffer = 0;
    buffer = new char[s];
}
// strong guarantee: uses an automatic local variable
void ReNew(char*& buffer, size_t s)
{
    char* temp = new char[s];
    delete[] buffer;
    buffer = temp;
}
```

Both versions perform a nothrow operation (in the first case, `buffer = 0`; in the second case, `buffer = temp`) that restores our invariant after the `delete[]` operation.

Sheesh! Writing exception-safe code is hard (but so is error handling in general). You can learn a lot just by staring at two function calls. Now, we'll apply our newly found insights to devise a general algorithm that analyzes the exception safety of any function.

Exception-safety Analysis

We want to create an algorithm that, given a function `Fun`, returns basic, strong, or nothrow. This is as hairy as you'd expect—hey, with only two functions, there was already plenty to think about! We need to apologize in advance for the dryness and laboriousness of this section, but we do believe that such an algorithm is useful. We mainly intend it for "hand-execution," meaning that it is an aid to a human reader to figure out the exception safety of a function. A more rigorously defined version of this algorithm could be devised for automated execution.

This algorithm operates step-by-step on each operation inside `Fun` (in the order that a compiler would evaluate them), incrementally computing its exception-safety level. This means that we need to handle every statement, including expressions, `if`s, `for`s, `switch`es, `try`s, and so on.

The algorithm needs to keep some state during execution:

- `safety`. The current safety level. We initialize that to nothrow, because a function that hasn't done anything can't throw, either.
- `purity`. The current purity level. That is initially "pure," because in any democratic analysis, a function is considered pure unless proven to do something impure.
- `exception_set`. The set of exceptions that `Fun` might throw (we'll see in a moment why that's needed). We initialize that to the empty set.

- `caught_set`. The set of expressions that `Fun` can catch. This set will grow/shrink as `try` statements are entered/exited. Initially `caught_set` is empty.

The "worst state" would be defined as impure and basic. As soon as we reach this state, the analysis can stop. The "bliss state" is the initial state, pure and nothrow.

We also define a `meet` operation, which is the union of two tuples containing `safety`, `purity`, `exception_set`, `caught_set`. A meet is performed whenever you merge two branches of code, such as the True and False branches of an `if` statement. The `meet` operation of two such tuples consists of: 1. taking the worst purity and safety of the two tuples; 2. computing the union of the two `exception_sets`; and 3. computing the intersection of the two `caught_sets`.

Let's proceed with the algorithm, written in pseudoC++:

- If the next statement is an `if`, then recursively apply the algorithm to analyze the True/False branches. Then merge the results on the two branches as we just explained.
- If the next statement is a `switch`, then perform analysis on all branches of the `switch`, then merge all results.
- If the next statement is a loop (`while` or `for`), apply the algorithm iteratively to the body of the loop until there is no change in the three state variables. Usually, this fixed point is reached in one to two iterations. Any break continues the loop analysis to where that statement passes control.
- If the next statement is a call to a function `Gun`, then: 1. You need to see if `caught_set` catches all of `Gun`'s exceptions—if that's the case, you can consider `Gun` a nothrow function followed by a series of conditional jumps to the catch handlers; 2. if `purity == pure` and `safety == nothrow` and `Gun()` is the last expression in `Fun`, then assign `purity = Purity(Gun)` and `safety = Safety(Gun)`, then return; 3. else, if `Purity(Gun) == pure` and `Safety(Gun) == nothrow`, then continue; 4. else, if `Purity(Gun) == impure` and `Safety(Gun) == nothrow`, then assign `purity = impure`; 5. else, if `Safety(Gun) == strong`, then:
  - If `purity == pure` then assign `safety = min(strong, safety)` and `purity = Purity(Gun)`.
  - Else, assign `safety = basic`: 1. `else, if Safety(Gun) == basic`, then assign `purity = Purity(Gun)` and `safety = basic`; 2. in any case, if the current `caught_set` catches any of `Gun`'s exceptions, you need to meet the result obtained above with the result obtained on all possible `catch` blocks inside `Fun` that `Gun`'s call might transfer control to.
- If the next statement is a throw, then: 1. if `caught_set` catches it, then continue analysis with the appropriate catch handler; 2. else, if `purity == pure`, assign `safety = strong`; otherwise, assign `safety = basic`. Also, add whatever exception is thrown to `exception_set`.
- If the next statement is a `try/catch` block, then: 1. add all catch handlers to `caught_set`; 2. analyze the `try` block—the catch blocks will be automatically analyzed as a consequence of possible exceptions being thrown from inside the block; 3. on top of this analysis, use heuristics to see if the `catch` block(s) restore purity by means of undoing.
- Whatever the statement is, if it writes to variables other than local automatic variables, assign `purity = impure`.
- At the exit of each block, check whether destructors are restoring purity.

There would be more to add if the algorithm were to be rigorous, but for now we wanted to give a good feel for how to perform the analysis by hand. In essence, you analyze each statement of the function, progressively changing the current state of the analysis as you make progress. The algorithm might go down recursively into blocks for compound statements, such as `if`, `while`, or `try/catch`. For simplification purposes, we analyze each function in isolation; you analyze complex expressions by imaginarily calling one function at a time and storing intermediate results in temporary variables.

Now, the astute reader might notice that our algorithm gives no way to stumble onto the wacky guarantee. To do so would require even more state to track invariants and allocated resources in order to determine when the basic guarantee has been lost or regained. Because our space is limited, we're going to leave that exercise for the reader.

At the end of the day, the algorithm returns a useful characterization of a function: whether it is pure, what its exception-safety level is, and what exceptions it might throw.

Conclusion

Writing exception-safe code is hard. Analyzing the exception-safety level of a function is a highly nontrivial and heuristic task. Yet exception safety is a very important summary of a function, and making sure your code has defined behavior in the presence of exceptions is the only way to correctness.

The important things to remember about analyzing exception safety are:

- In a sequence of function calls, you obtain the strong guarantee if you write your code as a sequence of pure calls, followed by no more than one strong call, followed by a sequence of nothrow calls.
- Purity is context dependent; you can transform a strong function into a pure function by having it operate on local automatic data. Note that pure functions are always strong.
- Alternatively, you can use `try/catch` with purity restoration in the `catch` clause. A variant of that uses ScopeGuard [7] for automatically restoring purity in case of an exception.
- When the system has broken invariants, always restore them with nothrow calls.
- To determine a function's exception-safety behavior, analyze it with the algorithm in the previous section, having at hand the summaries of the functions that it invokes.

References

[1] Alexandrescu, Andrei. *Modern C++ Design*, Addison-Wesley, 2001.

[2] Alexandrescu, Andrei. "Assertions." *CUJ Online Experts*, April 2003 (http://www.moderncppdesign.com/publications/cuj-04-2003.html).

[3] Alexandrescu, Andrei and John Torjo. "Enhancing Assertions." *CUJ Online Experts*, August 2003 (http://www.moderncppdesign.com/publications/cuj-08-2003.html).

[4] Niebler, Eric and Anson Tsao. "FOR_EACH and LOCK," *C/C++ Users Journal*, November 2003.

[5] Sutter, Herb. "Exception Safety and Exception Specifications: Are They Worth It?" *Guru of the Week #82* (http://www.gotw.ca/gotw/082.htm).

[6] Sutter, Herb. "Exception-Safe Class Design, Part 1: Copy Assignment." *Guru of the Week #59* (http://www.gotw.ca/gotw/059.htm).

[7] Alexandrescu, Andrei. "Change the Way You Write Exception-Safe Code—Forever." *CUJ Online Experts*, December 2000 (http://www.moderncppdesign.com/publications/cuj-12-2000.html).

[8] Abrahams, David. "Exception-Safety in Generic Components." Boost (http://www.boost.org/more/generic_exception_safety.html). ❑