N3202=10-0192
11/07/2010
Bjarne Stroustrup
Email: bs@cs.tamu.edu

To which extent can noexcept be deduced?

Abstract

If we fail to deem a function **noexcept** even though it doesn't throw the worst that can happen is that sub-optimal, but still correct, code will be executed. Thus, we should craft the simplest rules that render the highest number of functions that does not throw **noexcept**, while never implicitly make throwing code **noexcept**. Given the rules suggested here, a **noexcept** function can only throw if a programmer specifically marked a throwing function **noexcept**.

Introduction

As an introduction, let me reproduce a (slightly modified) version of my initial comment to Thorsten's paper:

http://wiki.dinkumware.com/twiki/pub/Wg21batavia/EvolutionWorkingGroup/reconsider_noexcept.html

My suggested rules below are a refinement of the sentiment in the posting.

On 11/2/2010 9:18 AM, Thorsten Ottosen wrote:

> We may summarize the problems with the current noexcept approach as follows:
>
> - users will be annoyed that the compiler cannot deduce obvious cases, e.g. =default constructors
> - simple interfaces are cluttered with non-essential information, e.g. constexpr functions
> - almost every statement in function templates leak into the noexcept declaration
> - a user-maintained noexcept increases the likelihood that the specification is not correct. In turn, this implies (a) an increased chance that client code terminates unexpectedly, or (b) that optimization opportunities are lost. (Note that providing correct warnings is also undecidable.)
> - client code can still change (fail to compile, different runtime behavior) if noexcept is added or removed from a library.
>
> This is - I think - true and worrying. His examples are very persuasive. His suggestion, which I can be boiled down to:

> "an inline function is (implicitly) noexcept unless it contains a throw or a call of a non-noexcept function."

This is simple

- it requires no flow analysis - if did I would strongly object.
- it does not require recompilation of every user if a potentially throwing operation is added to or removed from a function - if it did I would strongly object

Note that a change to the implementation of a function still "bubbles up" to its "interface" - it just does so implicitly and mostly invisibly to the user. Code that selects on **noexcept** may change meaning and code that does not may throw.

The proposal is focused on inline functions. I think that's a mistake (an unnecessary complication). The ability to deduce **noexcept** does not depend on inlining. The two issues are orthogonal. Independently of "inline", we can

- deduce **noexcept** from the function definition
- have a declaration separated from the function definition (implying a need to reconcile them)

First: in general we cannot know if a function throws - that would require (perfect) flow analysis (and in the worst case solving the halting problem). Second: we cannot build systems where an apparently small change in a function forces recompilations of all users. For example:

```
void f(int i)
{
   // int a[sz];              // before
   vector<int> a(sz);    // after
   a[i] = 7;
   // ...
}
```

Before, **f()** may have been **noexcept**; afterwards, it is not: **vector** may throw.

I know that some wants static checking of **noexcept**, but I don't and I consider static checking *not an option* for C++0x. I don't even want that discussion now.

So, what does **noexcept** mean? It states what a caller may assume of a called function and if the assumption is wrong (and the called function unexpectedly throws) the program terminates. If that's all (and I think it should be), there is no reason to disallow mismatches between a declaration and a definition: An implementation may warn if it finds "mismatches", but unless an

implementation can actually prove that a **noexcept** will get violated for every execution there is no compile or link time error.

Seen as this **noexcept** (deduced or explicit) can be used to speed up execution and to ensure quick termination if a **noexcept** assumption is false. It is not a mechanism for "more robust" or "more reliable" code.

If this is to go anywhere, we *must* keep this about as simple as my simplification of Thorsten's idea/suggestion.

An example

Just to remind us what is the problem, here is the pair from the FCD and Thorsten's paper:

```
template <class T1, class T2>
struct pair {
 typedef T1 first_type;
 typedef T2 second_type;

 T1 first;
 T2 second;
 constexpr pair() noexcept( is_nothrow_constructible<T1>::value &&
                            is_nothrow_constructible<T2>::value );
 pair(const pair&) = default;
pair(const T1& x, const T2& y) noexcept( is_nothrow_constructible<T1, const T1&>::value &&
                                         is_nothrow_constructible<T2, const T2&>::value );
 ...
 void swap(pair& p) noexcept( noexcept(swap(first, p.first)) &&
                              noexcept(swap(second, p.second)));
};
```

A comment

Ville commented:

> For the boilerplate in **noexcept**, I expect every user who has to write that garbage to curse us to everlasting damnation. It's HORRIBLE. I can barely understand if a heroic library writer can manage to cope with it, but I can't imagine any normal user, expert or not, to have patience to write such abominations. I would highly welcome any deduction facilities, if at all attainable.

Let's keep that in mind.

Consistency

Neither Thorsten's proposal nor my (more radical/consistent variant) suggest changes to the meaning of explicitly stated **noexcept** specifications.

The discussion of Thorsten's paper mostly focused on the concern that deduction would imply ODR violations. Thorsten observes (about my suggestion): "If I understand him correctly, then it would be ok to let **noexcept(foo())** return different values depending on context." That seems to be true, and seems to me to be fine. The idea of **noexcept** is to allow code to be written to take advantage of knowing that code will not throw. The key observation is that if we fail to deem a function **noexcept** even though it doesn't throw the worst that can happen is that sub-optimal, but still correct, code will be executed. In other words, as long as we don't mistakenly deem a throwing function **noexcept**, not serious harm is done. Furthermore, if people play around with Booleans derived from **noexcept**, they should be happy as long as that rule is followed.

In fact, I do propose a consistency rule below ("Rule 5"), but from an exception safety point of view, I don't think it is necessary.

So, the question to answer is: can we craft simple rules that never deem a throwing function noexcept and are likely to make more functions **noexcept** than we would get by exclusively relying on programmers decorating functions with **noexcept**. I think that the answer is obviously yes.

Sometimes, programmers can be cleverer than the rules I suggest, getting more optimal code. However, I wouldn't bet on that on a large scale. Please note that the serious mistake: deeming a function that actually throws noexcept can only be done by a programmer. If a **noexcept** function does throw, the program is terminated, which to my mind is far better than giving a wrong result.

Before going into details, let me outline potential rules for consistency of deduced and explicit **noexcept** specifications, so that we have something more concrete to discuss. I would be happier with any rules that obey the fundamental rule and are simpler than my suggestion:

1. If a function declaration has an explicit **noexcept** specification that specification is used. That is, we trust the programmer.
2. If a function declaration that is not a definition has no **noexcept** specification it is considered **noexcept(false)**. That is, if we don't know anything about a function we must assume that it can throw.
3. If a function definition has no **noexcept** specification it is considered **noexcept** provided it contains no operation that could throw (e.g., a **throw**, a **dynamic_cast** to a reference type, or a function that is not **noexcept**) otherwise it is **noexcept(false)**. That is, if we know that a function cannot throw (unless a programmer has wrongly and explicitly deemed a called function **noexcept**) we deem it **noexcept**.

4. An assignment or initialization of a pointer to function with **noexcept** with an explicit **noexcept(P)** function is an error. That is, the constraints on a pointer to function must be at least as strict as those for an assigned function.

I believe rules 1, 2, 3, and 4 to be necessary. Compared to the FCD rules, I believe that these rules leave every program that consistently uses explicit **noexcept** declarations unchanged. The difference is that some functions are deduced to be **noexcept**.

[Note added 11/11/10: The CWG discussion raised the question of whether determining whether an operation can throw should involve template instantiation to determine whether an instantiation could throw. My opinion is that an approach that instantiates is not conservative. A conservative deduction algorithm should only look at declarations – not definitions. Whether that restriction (however phrased) makes this deduction approach useless for most template programs was a topic of debate.]

In addition we need to craft rules that take care of inconsistent use of noexcept for declarations of a function within a translation unit and between translation units. Such rules would serve to avoid confusion and apparently inconsistent behavior; they protect against programmer mistakes. Such added rules would not be to protect against misuses of deduction because those are covered by rules 1 to 4. Presumably, the main technical reason to try to enforce consistency would be to avoid problems with template instantiation. Consider

```
int f();                    // we don't know if f() throws, so noexcept(false)
X<f> x1;
int f() { return 0; };      // obviously noexcept
X<f> x2;
int f() noexcept(cond);     // noexcept depends on cond
X<f> x3;
```

The deduced **noexcept** obviously is the correct (optimal) choice, as it will be in all cases except where a programmer knows that someone has lied about **noexcept** in a function called by a function deduced to be **noexcept**. I don't think we should craft our rules to try to protect against "lying" (incl. obscure programmer errors and deliberate decisions to treat a function that very rarely throws as noexcept).

In principle, we could pick any rules we like (all will lead to correct use of **noexcept**), so we can just as well try the simple rules demanding perfect consistency:

5. If two declarations of the same function differ in their **noexcept** specification the program is invalid. If the two declarations are in different translation unit no diagnosis required (i.e. an ODR violation).

I consider a deduced **noexcept** equivalent to a user-specified **noexcept**. However, to have that work in real programs, we need one more rule:

6. **noexcept** is only deduced if the function definition has no user-specified **noexcept** and if the definition is not preceded by a declaration of the same function.

Without that rule, many classical programs would break. For example:

```
int f();                 // f() might throw
int f() { return 0; }    // actually,  f() doesn't throw
```

I believe that these rules are roughly equivalent to what the FCD requires.

I considered "tweaks" to allow the deduced noexcept to be used together with declarations without noexcept specifications. For example:

```
int f();                 // we don't know if f() throws, so noexcept(false)
// no use of f here
int f() { return 0; };   // obviously noexcept

int g() { return 0; };   // obviously noexcept
int g();                 // OK?
```

I don't think this would buy us much and might complicate linkage rules. So I don't propose such as "rule 5a."

A reflector message:

> Den 03-11-2010 23:50, Alberto Ganesh Barbati skrev:
>
> To: C++ core language mailing list Message c++std-core-18002
>
> // First translation unit
> > void f();
> > inline void f() {}
> > constexpr int x = noexcept(f());
>
> // Second translation unit
> > inline void f();
> > constexpr int y = noexcept(f());

I could live with that; it doesn't lead to throwing code being deemed **noexcept**. However, it is prohibited by "rule 5."

> And further:
>
> Suppose you have a library function that use metaprogramming to select different implementations according to the fact that a certain expression is noexcept or not, for example:

```
// g.h
template <int isnoexcept, class T>
void g_impl(T x);

template <class T>
void g(T x)
{
  g_impl<noexcept(f(x))>(x);
}
```

I wonder how much metaprogramming is done with function templates declarations only and how the programmer would define **g_impl()** so as not to get an error.

```
Now consider this modified version of the example:

// First translation unit
#include "g.h"

inline void f(int) {}

void test1()
{
   g(0);
}

// Second translation unit
#include "g.h"

inline void f(int);

void test2()
{
   g(0);
}

inline void f(int) {}

This program is now ill-formed because of 14.6.4.1/7, with no
diagnostic required. This is most surprising for the user because he
may not know that the library is playing noexcept-tricks and the
program (I mean the two TUs, not g.h) looks perfectly well-formed in
C++03.

>
```

This example could be caught by “Rule 5” in the second translation unit, but I don’t actually see any harm in accepting this code.

Dave Abrahams supplied this example:

```
This proposal essentially requires the instantiation of function
bodies during SFINAE checking, doesn't it?

template <class T>
```

```
typename enable_if_c<noexcept( some_inline_function_template<T>()
)>::type*
f(T x)
```

Yes, I think so. It is easy to write a template function that throws depending on its argument type. If you want to SFINAE based on **noexcept**, you'll have to either explicitly specify **noexcept** (and take the chance that you are wrong because of an argument type that throws) or instantiate to let the compiler find out. Seems fair.

Common cases

Consider the usual separate compilation model:

```
// f.h:
int f();

// def_f.cpp:
#include<f.h>
int f() { } // obviously noexcept

// use_f.cpp:
#include<f.h>
int x = f();
```

With simple deduction there are no way we can get **noexcept** in **use_f.cpp**. That doesn't bother me. This example is the motivation for "Rule 6."

Inline and templates defined in headers also work fine:

```
// f.h:
inline int f() { … }
template<class T> T g(T) { /* no throwing operations here */ }

// use_f.cpp:
#include<f.h>
int x = f();
int y = g(7);       // is noexcept
Throwing z = g(Throwing{});      // is noexcept(false)
```

Simple deduction gives us what we want. Classes behave as would be expected from the examples above:

```
// f.h:
struct X {
        int f() { /* no throwing operations here */  };
        int g();
};
```

```
// use_f.cpp:
#include<f.h>
void user(const X& x)
{
        int r1 = x.f();     // noexcept
        int r2 = x.g();     // noexcept(false)
}
```

We get **noexcept** deduction for the inline member but not from the non-inlined one.