Doc No: SC22/WG21/ N1703 04-0143

Project: JTC1.22.32

Date: Thursday, September 09, 2004

Author: Francis Glassborow

email: francis@robinton.demon.co.uk

Function Qualifiers

1 The Problem & Context

Currently there are at least two things that a programmer can assert about a function which can both be checked statically and which can have positive benefits for optimizers.

While it is not generally possible to do static checking of exception specifications (and experience suggests that attempts to make this possible are counter productive) it is possible to provide rules for static checking of the assertion that a function cannot throw anything. Simply, a function cannot throw anything if its definition:

1) does not explicitly contain a throw expression
2) only calls functions that give the guarantee not to throw
3) Any mechanism to convert OS exceptions to C++ exceptions is switched off

1) & 2) are statically checkable and 3) can be made a required compiler action in the context of the definition of a function that is qualified as not throwing. This will leave the issue of such exceptions as `bad_alloc`. See discussion section.

The second case concerns the concept of a pure function; a function that has no side-effects. The knowledge that a function is a pure function allows extensive optimization when multiple processors and array processors are available. Current hardware trends suggest that desk top machines with multiple processors and/or multiple cores on a single CPU are likely to be the norm during the lifetime of the next C++ Standard. It also seems possible that array processors will become increasingly common. For example given suitable hardware the following code can be unrolled and executed on an array processor given that `foo()` is a pure function. That condition is sufficient though not necessary.

```
double foo(int i, int j);
double array[maxrows][maxcols];
for(int j(0); j != maxrows; ++j)
     for(int k(0); k != maxcols; ++k)
          array[j][k] = foo(j, k);
```

2 The Proposal

Provide a mechanism for adding qualifiers to function declarations that are statically checkable and enforceable. In the first instance there should be two such: `static`

`nothrow` and `static pure`. `static` is already a keyword, and one that is naturally appropriate in this context. As we already have past experience with introducing what is effectively a context keyword (`new(nothrow)`), given a suitable syntax `nothrow` and `pure` should not be a serious problem.

I propose that we add to the grammar so that, for example:

```
int foo() static nothrow;
int foo() static nothrow{/* body */}
int bar() static pure;
int bar() static pure{/* body */}
int foobar() static nothrow, pure;
int foobar() static nothrow, pure{/* body */}
```

Require that the compiler check the relevant assertion(s).

Further we will need to specify that such assertions are part of the type of a function pointer (as is already the case with extern linkage specifications) and with implicit conversion rules so that a qualified function pointer implicitly converts to an unqualified one.

3 Discussion

To make a `nothrow` qualifier useful we will need to consider how to tackle exceptions thrown by elements of the Standard Library. In particular we need to consider `bad_alloc` because that can, in theory, be thrown by any library type that uses dynamic allocation. In effect we need containers that provide a `nothrow` guarantee. Alternatively we can consider providing a mechanism whereby a `nothrow` qualified function can call functions that do not make that guarantee. For example:

```
void foo() static nothrow{
     // region where only nothrow functions are called
     try {
          // possibly throw
     }
     // optional explicit catch clauses
     catch(...){ /* handle all other cases, possibly with a
call to abort() */}
     // region where only nothrow functions are called
}
```

In other words, provide an impermeable barrier so that exceptions cannot propagate from a `nothrow` function.

Note that we should consider making all members of an `explicit class` (assuming that proposal goes into the WP) implicitly qualified as `nothrow` unless explicitly stated otherwise by adding a `throw(...)` exception specifier.

The gains from introducing a pure qualifier seem to be potentially substantial. Unfortunately the potential for aliasing makes it hard to apply to class types. In order for the proposal to fulfill its potential we need to refine it so that, for example, the implementation of such value types as complex can be used as parameters to pure functions. I believe that there are ways to handle aliasing issues by requiring checking at the call site.

Pure functions are necessarily re-entrant and thread safe thus providing further gains in some program areas.

For example, we could allow reference to `const` parameters as long as we:

1) prohibit the uses of `const_cast<>` in the body of a pure function
2) prohibit the use of reference to instances of classes with mutable members
3) require that no non-pure function that has write access to the argument runs in parallel to it.

The final requirement is the key to supporting a wider range of pure functions. The above (together with equivalent requirements for pointer parameters) should allow many const member functions to be implemented as pure functions. However I suspect that, in order to support 3) above multi-threaded code will need a mechanism to ensure that no object is accessed in one thread by a non-const member function whilst another thread is using a pure const-member function.

The argument has been made that very few of the current library functions are 'pure'. However that is not surprising because there has been no motive to go the extra mile to design pure functions.

In considering the amount of work entailed we should remember that the next release of the C++ Standard will effectively be C++ from 2010 to 2020. If we do not provide support for the hardware that is likely to be common in that time interval I believe we will, at the least, have to consider a TR on support for parallel processing.

4 Changes to the Working Paper

The purpose of the current paper is to obtain feedback from both potential users and from implementers. If that is sufficiently encouraging we can address how to incorporate the proposals in the WP.