Toward Improved Optimization Opportunities in C++0X

Document #: WG21/N1664 = J16/04-0104
Date: July 16, 2004
Revises: None
Project: Programming Language C++
Reference: ISO/IEC IS 14882:2003(E)
Reply to: Walter E. Brown<wb@fnal.gov>
Marc F. Paterno<paterno@fnal.gov>
CEPA Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

Contents

The value of a problem is not so much coming up with the answer as in the ideas and attempted ideas it forces on the would-be solver.

— ISRAEL NATHAN HERSTEIN

1 Introduction

The well-known "new dragon book" [ASU86] presents a number of standard code improvement transformations. Generally known as "optimizations," these techniques are widely applicable and are routinely carried out by most modern compilers. Such transformations include:

- common subexpression elimination,
- copy propagation,
- dead-code elimination,

- code motion,
- strength reduction,

and many more, singly and in combination. Other optimizations, such as those exploiting parallel architectures, are equally well-known.

However, it has long been generally considered a much more difficult task to generate efficient machine code for C++ programs than it is to generate high-performance code for programs expressed in such programming languages as Fortran.[1] This paper explores some of the reasons for this comparative difficulty.

Our goal is to propose modest enhancements to the C++ core language that will permit C++ translators to produce code whose run-time performance is far more competitive.

2 Factors inhibiting optimizations

Informally, an optimization is *safe* if and only if the code transformations involved in performing the optimization cause no change to the program's observable behavior. Unfortunately, it can sometimes be effectively impossible for a C++ compiler to determine whether any particular optimization can be safely applied in the context of a particular code fragment. In such cases, the compiler must generally take a conservative approach and forego the optimization in favor of preserving program correctness.

Two factors seem principally responsible for C++ compilers' difficulty in evaluating optimizations' applicability. Both *aliasing* and *side effects* prevent certain kinds of knowledge from crossing a calling interface. While this paper will principally address issues due to side effects, improved C++ code optimization opportunities are ultimately likely to involve both these factors.

2.1 Aliasing

The subject of considerable recent Committee discussion,[2] *aliasing* "is a long-standing problem both in compiler implementation . . . and in programming language theory" [Pie02, p. 170]. For our present purposes, we are concerned with aliasing in the context of compiling a function that takes any pointer or reference parameters:

```
1 //                                Listing 1
2 void  f( int const & a, int & b )  {
3   // ...
4 }
```

Suppose the compiled body of this function `f` could produce a performance benefit by copying the value of `a` into a high-speed register. However, the compiler can't always determine that such a transformation will preserve the original semantics: a call to this function might, after all, take the form `f(x,x)`. In such a case, the function's parameters (`a` and `b`), although distinct in name, both become aliases for a single underlying object, `x`. If so, it would be generally incorrect to rely exclusively on `a`'s value (really `x`'s value) as copied into a register whenever the function also modified `b` (really modifying `x`).

Thus, without seeing the call, it is in the general case impossible to determine solely from a function's declaration and body whether all its arguments are distinct. The C99 `restrict` keyword is intended to help with this aspect of optimization when pointer parameters are involved.

---

[1] "Matching Fortran's speed remains an unsolved problem. . . . For some problem domains, clever use of template expressions can allow C++ libraries to . . . run faster than Fortran. However, for simple array crunching, Fortran is likely to always be king. . . " [Rob96].

[2] See, for example, the extensive thread on the Committee reflector starting with [Sut04].

Broadly speaking, the presence of `restrict` in a function parameter's declaration provides extra information regarding the client's permitted use of the function. Such additional information may then permit a compiler to discount aliasing and thus achieve additional opportunities for improvement in the code generated for the function body.

In sum, aliasing as an inhibiting factor in code optimization arises due to a lack of information regarding the arguments furnished to a function at a point of call. Given additional information (*e.g.*, `restrict`) regarding constraints on the client's interface, the compiler may be able to generate code to improve the function's performance characteristics.

2.2 Side effects

We have shown that aliasing problems inhibit optimization of a function's body (the "callee") because of a lack of information regarding the arguments furnished (by the "caller") at a point of call. In contrast, difficulties due to *side effects* inhibit optimizing the caller, due to a lack of knowledge regarding the callee function's behavior. In particular, it is in general unknown, *at a point of call*, whether the called function will commit any relevant side effects.

Such information is traditionally available only by inspecting the body of the callee. Because function inlining, by definition, makes function bodies visible at the point of call, compilers can make better decisions regarding both local and global code improvement opportunities relative to such a call site; such additional knowledge therefore contributes significantly toward the improved code very often attributed to inlining technology.

Conversely, in the absence of inlining, a called function's body is generally opaque to its callers. Caller code improvement may therefore be inhibited by such lack of knowledge regarding callee behavior, especially with respect to side effects. Consider, for example:

```
1 //                                  Listing 2
2 int  g( int f(int), int x )  {
3   x *= 2;
4   return f(x) * f(x) * f(x);
5 }
```

If the function call `f(x)` is potentially expensive to evaluate, a compiler may choose a performance-improving transformation such that `g` evaluates the call to `f` only once and caches its result for subsequent use, as if the programmer had instead written:

```
1 //                                  Listing 3
2 int  g( int f(int), int x )  {
3   x *= 2;
4   int __temp = f(x);  return __temp * __temp * __temp;
5 }
```

However, the correctness of such a transformation, whether performed by a programmer or by a compiler, depends on certain assumptions regarding the behavior of `g`'s function parameter `f`. Suppose, for example, we passed to `g` the following function:

```
1 //                                  Listing 4
2 int  f1( int )  {
3   static int  x = 1
4   return x *= 2;
5 }
```

Clearly the two versions (caching *vs.* non-caching) of function `g` would produce different results when called via `g(f1,3)`.

In contrast, a function such as:

```
1 //                                  Listing 5
2 int  f2( int x )  {
3   return 16 * x + 8;
4 }
```

and supplied to `g` (*e.g.*, as `g(f2,3)`) would be a perfect candidate to be evaluated once within `g` and its cached value used in place of subsequent calls.

When functions such as `f1` and `f2` are `inline`d, compilers can and routinely do perform the analysis required to determine applicability of this and even more sophisticated performance-enhancing transformations. However, in the absence of such complete information as inlining affords, compilers must be generally conservative in their application of even standard code improvement techniques.

3 Function behaviors

We will use the term *well-behaved* to describe any free function or any member function that:

- communicates with client code solely via the function's argument list and return value, and
- is incapable of side effects.[3]

Among its other characteristics, a well-behaved function exhibits results that are *reproducible*: no matter how often such a function is called, its results will be identical so long as all values obtained via its argument list remain unchanged.

In contrast, a function that is not well-behaved is said to be *ill-behaved*. An ill-behaved function may violate the above strictures by permitting such behaviors as:

- relying on the value of an object outside its argument list,
- modifying the value of an object outside its argument list,
- throwing an exception without catching it,
- modifying and relying on the value of a local `static` variable,
- failing to return[4], or
- calling any ill-behaved function.

The ability to discriminate between well- and ill-behaved functions seems very important for the generation of high-performance code at and near a call site: If it can be determined at a point of call that the callee is well-behaved (and thus that its results are reproducible), additional caller optimizations may be applicable. Selected optimizations may be applicable even if a callee is ill-behaved, but in such a case the safe application of optimizing code transformations generally requires more detailed knowledge regarding callee behavior.

3.1 Free functions and `static` member functions

A well-behaved free function will exhibit the following characteristics and behaviors of interest, as will a well-behaved `static` member function:

1. It takes arguments passed:
   a) by value, or
   b) by *`const` indirection* (*e.g.*, by `const` reference or by pointer-to-`const`).
2. It uses a modifiable lvalue to refer to (any part of) an object only if that object:

[3] "[S]ide effects . . . are changes in the state of the execution environment" [ISO03, clause 1.9 ¶7].

[4] For example, by calling `exit`, `terminate`, or `longjmp`.

   a) is local to the function, and
   b) has `auto`matic lifetime (storage class).
3. It uses either a non-modifiable lvalue or an rvalue to refer to (any part of) an object only if that object is non-`volatile` and:
   a) is an argument of the function, or
   b) is local to the function and has `auto`matic lifetime, or
   c) is local to the function and has `static` lifetime and is declared `const`.
4. It respects all `const` qualifications.
5. It permits no exceptions to escape.
6. It returns control to the point of invocation.
7. It calls other functions (or member functions; see below) only if such functions are likewise well-behaved.

3.2 Non-`static` member functions

A well-behaved non-`static` member function shares the same characteristics and behaviors exhibited by a well-behaved free function. In addition:

8. It is always declared `const` so that its invoking object is always passed by pointer-to-`const` (*i.e.*, `this` will always have a pointer-to-`const` type).
9. It respects all `const` qualifications, even in the presence of a `mutable` declaration.

4 Analysis of ill-behaved functions

We now explore the behaviors of functions that do not qualify as well-behaved. Derived from the preceding chapter, we focus on the following categories of disqualifying behaviors that, singly or in combination, prevent a function from being deemed well-behaved: *updating*, *writing*, *reading*, or *throwing*.

A code-analysis tool (such as a compiler) can often ascertain the presence of these characteristics by inspecting the body of each relevant function. As pointed out earlier, the result of such inspection is potentially very valuable in the context of call site optimization opportunities, as it can permit a compiler to generate better-performing code. However, such inspection need not be conclusive: for example, a function that is ill-behaved only because it calls an ill-behaved function can in general not be correctly classified in isolation from its callee.

4.1 *Updating* behavior

A function that takes arguments other than as permitted for a well-behaved function (*e.g.*, arguments declared `volatile` or passed by non-`const` indirection) is said to engage in or permit *updating* behavior. It is *updating* behavior that subsumes traditional side effects, altering the value of a function argument. Arguments passed by value are, of course, immune to *updating*, as a callee function receives a copy and thus has no access to the actual argument.

Updating behavior has been long enshrined, for example, in both the C and C++ standard libraries. Indeed, functions such as `memcpy`, `scanf`, `qsort`, and the C++ `operator <<` and `operator >>` overloads engage in *updating* behavior as their principal *raisons d'être*.

Modern compilers can determine, by inspecting a non-defining declaration of a callee, whether that callee permits *updating* behavior. This assumes, of course, that the function abides by its declaration and does not employ a `const_cast` or other means to circumvent the declared

constness of any argument. However, the remaining disqualifying behaviors are not similarly discernible at a call site, under even the most favorable assumptions.

4.2 *Writing* behavior

A function that uses a modifiable lvalue in any manner other than as permitted for a well-behaved function is said to engage in or permit *writing* behavior. Such *writing* behavior encompasses, for example, side effects on non-local objects that are not part of the function's argument list. The following code is representative of such behavior:

```
1 //                                  Listing 6
2 extern int  z;
3 // ...
4 int  f( int x )  {
5   ++z;  // writing
6   return 7 * x;
7 }
```

Numerous functions in both the C and C++ standard libraries have *writing* side effects by design; examples include math functions (*e.g.*, `sqrt`) that report errors by modifying `errno`. In addition, there are standard functions (such as `malloc`, `set_terminate`, and `operator new`) that modify system-level and/or extra-linguistic objects whose lifetimes are `static` or longer.

Objects that are targets of a function's *writing* behavior are effectively a part of that function's interface. While they do not appear in the function's declaration, they may be impacted each time the function is called (and may, in turn, influence future behavior; see the description of *reading* behavior, below).

Note that the scope of a target object's name is not relevant for our purposes. Whether local to the callee function or not, a target object is not necessarily visible at a point of call.

4.3 *Reading* behavior

A function that uses either a non-modifiable lvalue or an rvalue in any manner other than as permitted for a well-behaved function is said to engage in *reading* behavior. The following code is representative of this category of behavior:

```
1 //                                  Listing 7
2 extern int  z;
3 // ...
4 int  g( int x )  {
5   return z * x;  // reading
6 }
```

Reading behaviors have long been generally considered a questionable programming practice, (as are *writing* behaviors; see, for example, the classic [WS73]). They are nonetheless permitted by C++ rules, and so compilers must be prepared to cope with them. Unfortunately, these behaviors are difficult or impossible to detect at a point of call, given only a non-defining declaration of the callee function:

```
//                                  Listing 8
extern int  z;
int  g( int );
// ...
int  h( int x )  {
  int  result = 0;
  for( int i = 0; i != 1000; ++i )
    result += ++z * g(x);  // g's result affected by reading z?
  return result;
}
```

Here, the mere possibility of a *reading* behavior in `g` can inhibit certain optimizing transformations in `h`; the (otherwise innocuous) call to `g` can't be safely hoisted out of the loop and its result cached, thus foregoing any performance benefit from using a single call in place of 1000 calls.

4.4 *Throwing* behavior

A function is said to engage in *throwing* behavior if it permits an exception to escape. Such behavior is well-known as problematic in the context of calls to a class's destructor.[5] However, *throwing* behavior is also problematic with respect to optimization opportunities:

```
//                                  Listing 9
int  f( int );
// ...
int  h( int x )  {
  int  i, result = 0;
  while( cin >> i )  {
    try   { result += i * f(x); }
    catch (...)  { cerr << "Skipped␣" << i << '\n'; }
  }
  return result;
}
```

Assuming `f` were known to give reproducible results[6], careful optimization could today produce code as if `h` had been written:

```
//                                  Listing 10
int  f( int );
// ...
int  h( int x )  {
  int  i, result = 0;
  bool __first_time = true;
  int  __cached_value;
  while( cin >> i )  {
    if( __first_time )  {
      try   { __cached_value = f(x); }
      catch (...)  { cerr << "Skipped␣" << i << '\n'; }
      __first_time = false;
    }
    result += i * __cached_value;
  }
  return result;
}
```

[5] See, for example, [Sut00, Item 16].

[6] Reproducibility includes not only the value returned, but also encompasses such side effects as `thrown` exceptions.

However, were `f` further known not to engage in *throwing* behavior, the code generated near the point of call could be further improved. Not only could the call itself be factored (hoisted) completely out of the loop, but the `catch` clause could be recognized as unreachable (*dead*) code and entirely eliminated:

```
1  //                                Listing 11
2  int  f( int );
3  // ...
4  int  h( int x )  {
5    int  i, result = 0;
6    int  __cached_value = f(x);
7    while( cin >> i )
8        result += i * __cached_value;
9    return result;
10 }
```

5 Introduction to proposed solutions

In subsequent sections, we present two alternative proposals for addressing the above-articulated problems that accompany ill-behaved functions. Each proposal allows the programmer, in the context of a function's declaration, the option to specify additional information about that function's behavior. In turn, a compiler can:

1. validate and make use of such additional information while compiling the function's body, and
2. use such additional information at each of that function's points of call to determine applicability of additional optimizing transformations.

In the first proposal, a function declaration may indicate the function's possible behaviors with rather fine granularity. As appropriate to the function, the declaration may indicate, in any combination:

1. that the function engages in `writing` behavior, naming each object in the set of objects to which it may write;
2. that the function engages in `reading` behavior, naming each object in the set of objects from which it may read; and
3. that the function engages in `throwing` behavior, naming the type of each exception it may allow to escape.

In contrast, the second proposal allows much less granularity. A function's declaration may specify, in any combination as appropriate, only:

1. that the function does not allow any exception to escape, and/or
2. that the function does not engage in any reading, writing, or updating behavior.

The principal advantage of the first proposal is that its finer granularity may allow additional opportunities for optimization. Many of the functions in `<cmath>` provide important examples, since many set `errno` to indicate error status. If the declarations of these functions were modified so as to detail their respective behaviors in this regard, compilers would likely find call site optimization opportunities that they can not in general detect under the current standard. The disadvantage is that the lists of objects and types can become very large, possibly so large as to be unwieldy.

The advantage of the second proposal is that the declaration is much simpler and cleaner, yet still seems to obtain most of the gains of the first proposal. However, many of the functions

in the standard library can not be said to be well-behaved, because of such side effects as their writing behavior toward `errno`. Because opportunities for call site optimization will not arise when calling such functions, the second proposal will include additional features to address such legacy circumstances.

6 Proposal 1

6.1 Overview

We propose that function declaration syntax, including member function declaration syntax, be extended with three new qualifiers[7], each optional, so as to permit declarations such as:

```
1 //                                  Listing 12
2 int  f( int )  reading() writing() throwing();
```

These new qualifiers have been selected to correspond to the function behaviors (a) that were previously identified as bearing on call site optimization, yet (b) that are not generally deducible from current function declarations. The qualifiers are thus intended to convey the extent to which a declared function is well- or ill-formed, and so to provide a compiler with better information to judge the efficacy of a call site's potential optimizing code transformations. Since the qualifiers are optional, when they are not used (*e.g.*, in legacy code) there is no information beyond what is available today; thus a compiler can make the same determinations as it would make today, and so call site optimization is no worse than it is today.

The following additional examples will clarify the usage and intent of the proposed function qualifiers. We first show the relationship between client code and library code in the context of a function that exhibits `writing` behavior on a single nonlocal variable, and exhibits neither `reading` nor `throwing` behavior:

```
1  //                                  Listing 13
2  // client code:
3  #include <cerrno>
4  int  f( int x )  reading() writing(errno) throwing();
5
6  // library code:
7  #include <cerrno>
8  int  f( int x )  reading() writing(errno) throwing()  {
9    errno = 0;
10   return 7 * x;
11 }
```

The following example illustrates a function that exhibits only reading behavior:

```
1  //                                  Listing 14
2  // client code:
3  extern int  z;
4  int  g( int x )  reading(z) writing() throwing();
5
6  // library code:
7  extern int  z;
8  int  g( int x )  reading(z) writing() throwing()  {
9    return z * x;
10 }
```

[7] Specific keywords are, of course, open for discussion; suggestions are certainly welcomed.

The next example presents a function that exhibits `reading`, `writing`, and `throwing` behavior:

```
//                                Listing 15
// client code:
extern int  z;
int  f( int x )  reading(z) writing(z) throwing(int);

// library code:
extern int  z;
int  f( int x )  reading(z) writing(z) throwing(int)  {
  if (++z)  throw z;
  return 7 * x;
}
```

A function that exhibits `reading` and/or `writing` behavior on a local `static` object poses an interesting problem, since local names are not in the scope of any caller. As the next example shows, we might reuse the keyword `this` to denote such localized activity:

```
//                                Listing 16
// client code:
int  h( int x )  reading(this) writing(this) throwing();

// library code:
int  h( int x )  reading(this) writing(this) throwing()  {
  static  int  k = 0;
  return ++k * x;
}
```

Our last example in this section demonstrates a function that manipulates a device's control register. To indicate that a function engages in such behavior, we might reuse the keyword `register`:

```
//                                Listing 17
// client code:
void  go()  reading(register) writing(register) throwing();

// library code:
void  go()  reading(register) writing(register) throwing()  {
  typedef  unsigned  status_word;
  * static_cast<status_word *>(0xFFFF7738) |= 1u;  // set "go" bit
}
```

6.2 Impact on the type system

We intend these new qualifiers to be part of the declared function's type:

```
//                                Listing 18
typedef
  int  F( int )  reading() writing() throwing();
// ...
F *  pf = &f;
```

```
//                                  Listing 19
typedef
  double  Integrand( double )  reading() writing() throwing();
// ...
double  integrate( Integrand f, double lbnd, double ubnd, double eps )  {
  // ...  implementation which calls the Integrand function f
}
```

We intend, however, that these new qualifiers not be part of the function's signature; *i.e.*, that this additional information not be used in overload resolution. This is because no comparable information is part of a function call.

To maintain type safety, we must be careful when copying pointers-to-qualified-functions:

```
//                                  Listing 20
typedef
  int  (*F)( int )  reading() writing() throwing();
typedef
  int  (*G)( int )  reading() throwing();
F fp = & ...;
G gp = & ...;
gp = fp;  // ok
fp = gp;  // error
```

6.3 Impact on function compilation

An additional implication of the proposed syntax, if the programmer employs it, is the exposure of a function's complete interface. By this, we mean that a function declaration can make known all the details of its behaviors: all non-local objects read and/or written, all local `static` objects read and/or written, and all types thrown. As pointed out above, this information is of great relevance and utility to a compiler in producing high-performance code at and near a call site. We intend this information to be checked for consistency when compiling the function's definition.

```
//                                  Listing 21
extern int  z;
// ...
int  f( int x )  reading() writing() throwing()  {
  ++z;  // error; inconsistent with f's declaration
  return 7 * x;
}
```

6.4 A `throwing` qualifier is not a `throw`-specification

Two important features of the proposed `throwing` qualifier distinguish it from the current `throw`-specification:

- A `throwing` qualifier, unlike a `throw`-specification, is part of a function's type.
- A `throwing` qualifier, unlike a `throw`-specification, is enforced by the compiler.

```
//                                  Listing 22
int  f( int ) throw()  {
  throw 2;  // compiles; ultimately calls unexpected()
}

int  g( int ) throwing()  {
  throw 2;  // fails to compile:  inconsistent with g's declaration
}
```

These differences in semantics alleviate the brittleness inherent in today's `throw`-specifications. In particular, unlike the current language, changing a called function so that it `throw`s a new type of object will never lead to a run-time error. Rather, such a change will produce a compile-time error when the caller is next recompiled, or a link-time error (induced by type mismatch) if the caller has not been recompiled.

Further, because the `throwing` qualifier is part of a function's type, current C++ rules will permit overloading when such a type appears in the context of a parameter. In the following example, `cube()` is overloaded based on the type of its first argument (and not based on its own `throwing` qualifier):

```
//                                  Listing 23
typedef
  int  might_throw   ( int );
typedef
  int  does_not_throw( int ) throwing();

int  cube( might_throw f, int x )  {
  return  f(x) * f(x) * f(x);
}

int  cube( does_not_throw f, int x ) throwing()  {
  return  f(x) * f(x) * f(x);
}
```

6.5 Qualified blocks

As part of this Proposal 1, we also recommend the introduction of a new C++ core language construct, tentatively known as a *qualified-block*. This recommendation is made largely to facilitate coping with legacy code.

Today's C++ has a mechanism, the `try`-block, that allows a function to trap the *throwing* behavior of functions that it calls. However, we foresee the need for a comparable mechanism to allow a function to adapt to ill-behavior on the part of other functions that it calls.

For example, many of the `<cmath>` functions, under certain conditions, carry out *writing* behavior by setting `errno`. However, a client function may know (perhaps because it carries out its own error-checking, or perhaps because of its underlying logic) that such *writing* behavior will not occur for one or more particular calls. The otherwise well-behaved client function would nonetheless be considered ill-formed if it called such a `<cmath>` function:

```
//                                  Listing 24
float  hypotenuse( float s1, float s2 ) writing()  {
  return sqrt( s1*s1 + s2*s2 );  // error; sqrt writes to errno
}
```

We therefore recommend introducing a new C++ construct, the *qualified-block*. Used at the programmer's discretion, a qualified-block asserts that the indicated behavior is local to the block it qualifies:

```
//                                  Listing 25
float  hypotenuse( float s1, float s2 ) reading() writing()  {
  writing(errno) {
    return sqrt( s1*s1 + s2*s2 );
  }
}
```

This same function could even use a qualified-block in combination with a `try`-block to ensure that it can be declared as well-behaved:

```
//                                  Listing 26
float  hypotenuse( float s1, float s2 ) reading() writing() throwing()  {
  try {
    writing(errno) {
      return sqrt( s1*s1 + s2*s2 );
    }
  }
  catch(...) { }
}
```

We note that injudicious use of a qualified-block can lead to undefined behavior. For example, via a qualified-block, a function that has side effects could be made to masquerade as well-behaved and thus well-formed. A compiler, seeing only the function's declaration, would likely optimize the function's call sites; any resulting behavior on the part of such an optimized caller is therefore unpredictable.

7 Proposal 2

Unlike the fine granularity of the previous proposal, this second proposal considers only two degrees of ill-behavior:

1. *throwing* behavior, and
2. (as a group) *reading*, *writing*, or *updating* behavior.

We will denote a function as *nothrow* if and only if it engages in no *throwing* behavior. Similarly, we will denote a function as *pure* if and only if it engages in no *reading*, no *writing*, and no *updating* behavior.

7.1 Overview

We propose that function declaration syntax, including member function declaration syntax, be extended with two new qualifiers, each optional, so as to permit declarations such as:

```
//                                  Listing 27
int  f( int ) pure nothrow;
```

These qualifiers are intended to convey, in broad terms, the degree to which a declared function is known to be well-formed.

Compilers today must typically assume, pessimistically, that called functions are ill-behaved. The presence of either or both of these proposed qualifiers would thus permit future compilers to

perform additional optimizing code transformations at call sites. In the qualifiers' absence (*e.g.*, in legacy code), call site optimizations will be limited to precisely those currently available.

7.2 Impact on the type system

As in Proposal 1, we intend these new qualifiers to be part of the declared function's type:

```
//                                Listing 28
typedef
  int  F( int ) pure nothrow;
F *  pf = &f;
```

```
//                                Listing 29
typedef
  double  Integrand( double ) pure nothrow;
// ...
double  integrate( Integrand f, double lbnd, double ubnd, double eps )  {
  // ...  implementation which calls the Integrand function f
}
```

Also as in Proposal 1, we intend that these new qualifiers not be part of the function's signature and thus not impact overload resolution of functions to which they are applied.

As before, to maintain type safety, we must be careful when copying pointers-to-qualified-functions. In addition to the requirements that [ISO03] already imposes on function pointer assignments, we would deem any function pointer assignment ill-formed if:

- its target (left-hand) type is pointer-to-`pure`-qualified-function but its source (right-hand) function pointer type is not `pure`-qualified, or
- its target function pointer type is pointer-to-`nothrow`-qualified-function but its source function pointer type is not `nothrow`-qualified.

Expressed another way, a function pointer assignment (or initialization) is well-formed if and only if the underlying function type of the source pointer is at least as qualified as the underlying function type of the target pointer.

Similarly, a `virtual` function in a derived class is well-formed, overriding the corresponding function in its base class, only if the type of the derived class's function is at least as qualified as the type of the corresponding base class's function. This rule follows directly from the preceding rule governing the copying of pointers-to-qualified-functions, and can perhaps be most clearly understood by considering vtable construction, in significant part, as a sequence of pointer-to-function initializations.

7.3 Impact on function compilation

As before, we intend a function's declared qualifiers to be checked for consistency when compiling the function's definition. A function that is declared `nothrow` is ill-formed if it engages in *throwing* behavior. A function that is declared `pure` is ill-formed if it:

- engages in any *reading* behavior, or
- engages in any *writing* behavior, or
- engages in any *updating* behavior, or
- calls any function that is not declared `pure`.

As an optional part of this proposal, the semantics of the `nothrow` qualifier may be extended such that a function declared `nothrow` would be deemed ill-formed if it failed to `return` control to its calling client via normal flow of control.

7.4 Distinguishing `nothrow` from `throw()`

A function declared with an empty `throw`-specification (*i.e.*, `throw()`) shares one important characteristic with a similar function declared with the proposed `nothrow` qualifier: neither function will `throw` any exception that client code could `catch`. Nonetheless, there are significant technical differences between the two, leading to differences in compilation requirements and behavior.

In today's C++, a function declared with an empty `throw`-specification must call `unexpected()` to deal with any exception the function is not itself prepared to `catch`. Such *throwing* behavior is detected and dealt with at runtime, requiring that a compiler generate code accordingly. In contrast, a function declared with the proposed `nothrow` qualifier is ill-formed if it engages in *throwing* behavior, *i.e.*, permits any exception to escape. Since such analysis is carried out at compile time, a compiler need generate no exception-handling code at all for this circumstance.

Finally, a `nothrow` qualifier is part of a function's type. In contrast, a `throw`-specification is not. As pointed out during discussion of our Proposal 1, this affects overloading when a function parameter has function type.

7.5 Traits

As part of this Proposal 2, we recommend the introduction of two new traits templates. Tentatively named `is_pure` and `is_nothrow`, the presence of these traits in the core language will permit metaprogramming technology to discriminates among functions (including member functions) based on the presence or absence of the respective `pure` and `nothrow` qualifiers.

Such discrimination is desirable, since it may be possible to employ improved algorithms (and thus obtain improved runtime performance) when it is known that certain dependent functions have `pure` and/or `nothrow` characteristics. An example is sketched later in this paper.

7.6 `pure`-blocks

As part of this Proposal 2, we also recommend the introduction of a new C++ core language construct, tentatively known as a `pure`-block. This recommendation is made to facilitate coping with legacy code.

Today's C++ has a mechanism, the `try`-block, that allows a function to trap the *throwing* behavior of functions that it calls. However, we foresee the need for a comparable mechanism to allow a function to adapt to impure behavior on the part of other functions that it calls.

For example, many of the `<cmath>` functions could not be declared `pure` because, under certain conditions, they carry out *writing* behavior by setting `errno`. However, a client function may know (perhaps because it carries out its own error-checking, or perhaps because of its underlying logic) that such impure *writing* behavior will not occur for one or more particular calls. The otherwise `pure` client function would nonetheless be considered ill-formed if it called such a `<cmath>` function:

```
1 //                                Listing 30
2 float  hypotenuse( float s1, float s2 ) pure  {
3   return sqrt( s1*s1 + s2*s2 );  // error; sqrt is not pure
4 }
```

We therefore recommend introducing a new C++ construct, the *`pure`-block*. A `pure`-block, used at the programmer's discretion, can permit a `pure`-qualified function to be deemed well-formed even if its body does call a function that is not `pure`:

```
//                                Listing 31
float  hypotenuse( float s1, float s2 ) pure  {
  pure {
    return sqrt( s1*s1 + s2*s2 );
  }
}
```

This same function could even use `pure`-blocks in combination with `try`-blocks to ensure that it can be declared as well-behaved:

```
//                                Listing 32
float  hypotenuse( float s1, float s2 ) pure nothrow  {
  try {
    pure {
      return sqrt( s1*s1 + s2*s2 );
    }
  }
  catch(...) { }
}
```

We note that injudicious use of a `pure`-block can lead to undefined behavior. A function can be made to masquerade as `pure`, for example, even though it has side effects. A compiler, seeing only the function's declaration, would likely optimize the function's call sites; any resulting behavior on the part of such an optimized caller is therefore unpredictable.

8 Impact on the standard library

As an optional but desirable part of either proposal, we recommend that qualifiers (Proposal 1: `reading`, `writing`, `throwing`; Proposal 2: `pure`, `nothrow`) be applied as appropriate to the declarations of all functions in the C++ standard library, including the C legacy functions. This will permit standard compiler technology to optimize these functions' call sites with no additional effort by a programmer.

In some cases, additional design effort may be warranted. Consider, for example, `vector<T,A>` in the context of our Proposal 2: it may be desirable to produce an implementation whose destructor is declared `nothrow` if and only if the destructors of both `T` and `A` were declared `nothrow`.

In today's C++, this could be accomplished via an extra, `bool`, template parameter used as a basis for specialization. The following code outline illustrates one such approach:

```
//                                  Listing 33
template< class T, class A = ...
        , bool = is_nothrow<T::~T>::value
              && is_nothrow<A::~A>::value
        >
class vector  {  // general case:  throwing
  // ...
  ~vector();
  // ...
};

template< class T, class A >
class vector<T,A,true>  {  // specialization:  nothrow
  // ...
  ~vector() nothrow;
  // ...
};
```

This approach seems potentially unwieldy, since it would require a new specialization for each combination of qualifiers. To avoid such combinatorial explosion of specializations, it might instead be possible to specialize only selected member templates instead of specializing the entire class template. However, this might necessitate reconsideration of today's rule that "A destructor shall not be a member template" [ISO03, clause 14.5.2 ¶2] in order to permit such code as:

```
//                                  Listing 34
template< class T, class A >
class vector  {
  // ...
  template< bool = is_nothrow<T::~T>::value
                && is_nothrow<A::~A>::value
          >
  ~vector();

  template<>
  ~vector<true>() nothrow;
  // ...
};
```

It is certainly undesirable to permit any class template to instantiate multiple destructors, likely one important factor underlying today's restriction as cited above. However, the above example code seems not to violate this need, as any instantiation of `vector<T,A>` clearly produces exactly one destructor.

A third alternative would permit direct dependence of one function's qualifiers upon the qualifiers of one or more other functions:

```
//                                  Listing 35
template< class T, class A >
vector<T,A>::iterator
vector<T,A>::erase(iterator)
  nothrow_if(    is_nothrow<T::~T>::value
              && is_nothrow<T::operator== >::value
            )
{
  // ...
}
```

In this example, `nothrow_if(true)` would be a construct that is synonymous with our previously proposed `nothrow`. While this seems to be a step forward in that it permits clear expression of a function's qualifiers, it does not address the more general situation in which one may wish to implement a function via distinct algorithms, depending upon another function's qualifiers.

Code such as shown above demonstrates the utility of the proposed `is_nothrow` trait. However, such effective use is predicated upon the ability to call upon the trait with an unambiguous template argument. This is difficult when the desired argument, a specific function, is a member of an overload set, for the function name alone merely identifies the set, and not any specific function.

However, this situation is not new. For example, "The address of an overloaded function ... can be taken only in a context that uniquely determines which version of the overloaded function is referred to..." [ISO03, clause 5.3.1 ¶5]. In that situation, among others, "The function selected is the one whose type matches the target type required in the context" [ISO03, clause 13.4 ¶1]. Unfortunately, our traits provide insufficient "context" to apply such a solution. A function's complete signature (rather than merely its name) seems needed for disambiguation, yet C++ precedent seems to have steered away from permitting its use in analogous circumstances. Therefore, this issue remains open for now.

9 Prior art

To our knowledge, at least two existing C++ compiler vendors provide implementations that encompass extensions substantively corresponding to our Proposal 2 above. The Metroworks and gcc compilers each support the optional function attributes known as `__attribute__((pure))` and `__attribute__((nothrow))`.

Analysis of their semantics shows that these attributes closely approximate our proposed `pure` and `nothrow` attributes, respectively. As described in [St04, §5.25: "Declaring Attributes of Functions"] with respect to gcc, "The `nothrow` attribute is used to inform the compiler that a function cannot throw an exception." Similarly, "[functions that] have no effects except the return value .... should be declared with the attribute `pure`." It is our understanding that the Metroworks versions share substantively identical intent and semantics.

This prior art, developed independently of this paper's proposals, differs in one important respect from our proposals herein: In the prior art, the extended attributes are considered as advisory "hints" to the compiler, while our proposal would treat them as declarative (prescriptive) requirements to be verified, with violations diagnosed, at compile-time. We believe the latter to be a superior approach, as evidenced by the widespread lack of use of today's `throw`-specification. Nonetheless, it is our understanding that even these advisory qualifiers have successfully led to improvements in generated code when consistently applied to a substantial portion of the standard library.

It can be argued that the proposed attribute declarations are redundant in the sense that whole-program interprocedural analysis can determine these properties. However, we believe that such analysis is often not feasible and sometimes not possible. For example, pre-compiled code, dynamically-linked libraries, and other time-sensitive compilation issues each provide challenges to the methods underlying interprocedural analysis.

We also understand, informally, that there has been some additional experimentation with compiler extensions that are at least somewhat similar to our Proposal 1 above. However, we are unable to offer further comment since the features are undocumented and the experimental results are unpublished to date.

10 Discussion

In several respects, adding our proposed qualifiers to a function's declaration resembles adding a `const` and/or a `volatile` qualifier to a variable's declaration. Upon inspection, our proposed rules for pointer-to-qualified-function assignment, for example, turn out to be a direct analog of current rules for pointer-to-cv-qualified-object assignment.

The historical debate between declarations based on *logical constness* versus those based on *physical constness* has analogs in our proposed contexts, too. For example, in our Proposal 2, one can consider function declarations based on *logical purity* versus those based on *physical purity*.

Consider a function that acquires, uses, and disposes of a resource: The following sketch employs `auto_ptr` to manage a memory resource in this fashion:

```
1 //                                  Listing 36
2 double  f(...)  pure  {
3   auto_ptr<MyType> ap  =  new MyType(...);
4   // ...  use *ap
5 }
```

The question, of course, is whether it is appropriate to declare such a function `pure`.

On the one hand, the function does preserve *logical purity*: after all, the acquired resource is guaranteed proper disposition, and the net result is effectively invisible to client code. On the other hand, the function may fail to preserve *physical purity*: depending on the algorithms underlying `operator new` and `operator delete`, certain data structures may well differ in their states before and after a call to such a function. This issue is important to resolve: both our proposals would require a compiler to verify the qualified function declarations, yet verification based on a logical view could conceivably give results that differ from those produced on the basis of a physical perspective.

Similar considerations apply to resource management in general. However, there may be resources so pervasive as to fall below the horizon. Devices (a floating-point processor, for example) typically have control registers as well as status registers. A user setting a control register is almost certainly an instance of *writing* behavior. But should the device's response (*e.g.*, setting its status register) be considered a side effect that affects a function's declaration?

While we do not, in this paper, propose solutions to such issues, we respectfully recommend that such matters be addressed in the context of whichever proposal is more favorably received.

11 Summary and conclusion

In this paper, we have analyzed the reasons for missed code improvement opportunities in the context of C++ function calls. We have paid particular attention to the present state of affairs, in which code improvements are often foregone because compilers have insufficient information available at call sites to non-inlined functions.

We have also presented two distinct proposals for C++ core language extensions to address the present situation. The proposals are similar in that each would add to a compiler's knowledge of the behavior of a called function, and would hence permit the compiler to apply that extra knowledge for the purpose of improving the generated code.

The proposals differ in their respective levels of information granularity. The first proposal provides highly detailed information that a compiler might exploit, but does so at the expense of an equally detailed declarative syntax and (potentially) considerable effort by programmers to exploit the syntax. The second proposal reflects a lower level of granularity, but still provides

significantly improved optimization opportunities at a considerably lower cost to programmers. In addition, our second proposal encompasses two additional function traits, as well as a new construct, the `pure`-block, for programmer use.

Finally, we have described prior compiler art that substantively incorporates a significant fraction of our second proposal. We also alluded to unpublished work that may provide prior art analogous to our first proposal.

Of our two alternative proposals, we favor the second. We respectfully urge the C++ standards body to consider our recommendations on a time scale consistent with that of the forthcoming C++0X.

12 Acknowledgments

We would like to thank our Fermilab colleagues Mark Fischler and Jim Kowalkowski for the helpful conversations that sparked the writing of this paper and that aided our development of the ideas herein. We gained additional insights via discussions and/or correspondence with Bob Campbell, Lois Goldthwaite, Howard Hinnant, Andreas Hommel, and Fred Peterson, and we sincerely appreciate their significant and knowledgeable input.

We also thank the Fermi National Accelerator Laboratory's Computing Division, sponsor of our participation in the C++ standards effort, for its support.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-10088-6. x + 796 pp. LCCN QA76.76.C65 A371 1986. Known as the "new dragon book"; see also its predecessor [AU77].

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, MA, USA, 1977. ISBN 0-201-00022-9. x + 604 pp. LCCN QA76.6 .A285 1977. Known as the "dragon book"; see also the much expanded [ASU86].

[ISO03] *Programming Languages — C++, International Standard ISO/IEC 14882:2003(E)*. International Organization for Standardization, Geneva, Switzerland, 2003. 757 pp. Known informally as C++03.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1. x + 604 pp.

[Rob96] Arch D. Robison. C++ gets faster for scientific computing. *Computers in Physics*, 10: 458–462, 1996. A slightly updated version, dated March 7 1997, has been circulated online.

[St04] Richard Stallman and the GCC Developer Community. Using the GNU compiler collection (GCC). Online: `http://gcc.gnu.org/onlinedocs/gcc-3.4.1/gcc/`, May 23 2004.

[Sut00] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, Reading, MA, USA, 2000. ISBN 0-201-61562-2. xiii + 208 pp. LCCN QA76.73.C153.S88 1999.

[Sut04] Herb Sutter. restrict in C++. C++ extensions reflector message c++std-ext-6735, February 17 2004.

[WS73] W[illiam] Wulf and Mary Shaw. Global variables considered harmful. *ACM SIGPLAN Notices*, 8(2):28–34, February 1973.