Exploring **`constexpr`** at Runtime

Doc No: SC22/WG21/N3583
Date: 2013-03-13
Author: Scott Schurr
Reply To: s dot scott dot schurr at gmail dot com

Abstract

The paper explores motivations for either a) constraining selected `constexpr` functions and constructors so they may only be used during translation, or b) overloading on `constexpr` so different implementations could be provided for translation-time vs. run-time execution. Two examples are examined for motivation. Then the paper explores four approaches to providing the constraints.

Contents

1 Introduction

The constexpr feature, added in C++11, is a powerful and easy-to-use tool for compile-time programming. Thanks to constexpr, things that were done with the preprocessor or with templates can now just be written as code and evaluated by the compiler at compile time. Here's an example of code that might have previously been implemented using the preprocessor or at runtime, but now can be handled directly during translation:

```
template <typename T = double>
constexpr T eulers_num()
{ return static_cast<T>(2.71828182845904523536028747l); }

constexpr double e_d = eulers_num();
constexpr float e_f = eulers_num<float>();
constexpr int e_i = eulers_num<int>();

// The following happens at compile time!
static_assert(e_d != e_f, "Precision matters!");
static_assert(e_i < e_f, "Precision really matters!");
```

But constexpr is much more capable. Prior to the introduction of constexpr, compile-time computations required using macros or template metaprogramming tricks. Macros have the disadvantage of not being scoped. Template metaprogramming techniques tend to be opaque at best. By using constexpr, average coders can now write their own compile-time computations. Even better, they can look at someone else's computations and have a chance of understanding them.

But constexpr is even more than that. A constexpr function or constructor can be passed values that are only known at run-time. In this case the compiler generates code so the evaluation of the constexpr function or constructor can happen at runtime. The constexpr feature is a tool for all seasons.

This is all good as far as it goes, but while constexpr behavior is great most of the time, there are some situations where it has disadvantages. This paper looks at two of those situations. The paper also looks at ways a future version of the C++ standard might address these situations.

2 Motivating Examples

It's easier to talk about software when there's a real example to point at. Let's make one.

2.1 A constexpr Function That Computes a BCD Value

Suppose we'd like to build a collection of packed binary-coded-decimal (BCD) values at compile time. BCD packs one decimal digit into each 4-bits of a nibble. BCD encoding strikes one particular balance of storage efficiency against computational ease. BCD encoded values might, for example, be used in an embedded system. The following is a possible implementation. There's nothing unusual about the code if you wish to skip to the next normal text. The code is only included for completeness.

```
namespace constexpr_bcd_detail
{
    using namespace std;
    // Literal char[] class.  Attribution:
    // http://en.cppreference.com/w/cpp/language/constexpr
    class str_const {
    private:
        const char* const p_;
        const size_t sz_;
    public:
        template<size_t N>
        constexpr str_const(const char(&a)[N]) : p_(a), sz_(N-1) {}
        constexpr char operator[](size_t n) {
            return n < sz_ ? p_[n] : throw out_of_range("Bad index");
        }
        constexpr size_t size() { return sz_; }
    };

    // Return maximum number of bits T allows
    template <typename T>
    constexpr size_t max_bits()
    {
        using lim = numeric_limits<T>;
        static_assert(lim::radix == 2, "bcd requires a base 2 type");
        static_assert(lim::is_integer == true,
            "bcd requires an integer type");
        return
            lim::is_signed  == true ? lim::digits + 1 : lim::digits;
    }

    // Put the BCD sign code into the least significant nibble
    template <typename T>
    constexpr T bcd_end(size_t bits, int sign, T x)
    {
        return
            (bits+4) > max_bits<T>() ?
                throw range_error("Too many bits in bcd") :
            (x * 16) + sign;
    }
    // Recursively compute packed BCD
    template <typename T>
    constexpr T bcd_recurse(
        const str_const& bcd, size_t n, size_t bits, int sign, T x)
    {
        return
            (bits+4) > max_bits<T>() ?
                throw range_error("Too many bits in bcd") :
            n == bcd.size() ?  bcd_end<T>(bits, sign, x) :
            bcd[n] == ',' ? bcd_recurse<T>(bcd, n+1, bits, sign, x) :
            bcd[n] == ' ' ? bcd_recurse<T>(bcd, n+1, bits, sign, x) :
            (bcd[n] >= '0') && (bcd[n] <= '9')  ?
                bcd_recurse<T>(bcd,n+1,bits+4,sign,(x*16)+bcd[n]-'0') :
            throw std::domain_error(
                "Only '0'...'9', ',', and ' ' may be in bcd body");
    }

    static const int pos = 0xC;  // BCD code for a '+' sign (Credit)
    static const int neg = 0xD;  // BCD code for a '-' sign (Debit)

    // Skip leading spaces and capture the sign if there is one
    template <typename T>
    constexpr T capture_sign(const str_const& bcd, size_t n)
    {
        return
            n == bcd.size() ? bcd_end<T>(0, pos, 0) :
            bcd[n] == ' ' ? capture_sign<T>(bcd, n+1) :
            bcd[n] == '+' ? bcd_recurse<T>(bcd, n+1, 0, pos, 0) :
            bcd[n] == '-' ? bcd_recurse<T>(bcd, n+1, 0, neg, 0) :
            bcd[n] >= '0' && bcd[n] <= '9' ?
                bcd_recurse<T>(bcd, n, 0, pos, 0) :
            throw domain_error(
                "Only '0'...'9', '+', '-', and ' ' may start bcd");
    }
} // end constexpr_bcd_detail namespace

// Public face of the packed BCD computation
template <typename T = std::uint32_t>
constexpr T constexpr_bcd(constexpr_bcd_detail::str_const bcd)
{
    return constexpr_bcd_detail::capture_sign<T>(bcd, 0);
}
```

So, you may ask what did we get with our two-or-so pages of C++ code. We got a way of describing BCD encoded literals using text. The compiler takes the text and produces an integral type filled in with the appropriate nibbles to represent the specified value. This is not intended as a run-time feature. It is intended as a compile-time evaluation only function that provides error checking and formatting. An example follows.

```
int main()
{
    // Packed BCD conversion at compile time
    constexpr std::uint32_t good1 = constexpr_bcd("+8,765,432");
    static_assert(good1 == 0x8765432C, "Yipes!");

    constexpr std::uint64_t good2 =
        constexpr_bcd<std::uint64_t>("-123 234 345 456 567");
    static_assert(good2 == 0x123234345456567D, "Yipes!");

    // If the target variables were declared constexpr you'd get a
    // compile-time error.  Using gcc 4.7.0, they are run-time errors.
    std::uint32_t bad1 = constexpr_bcd("12,345,678");
    assert(bad1 == 0x12345678C);
    std::uint32_t bad2 = constexpr_bcd("+A");
    assert(bad2 == 0xAC);

    return 0;
}
```

As the preceding example shows, it's not that hard to hand-pack a packed BCD value into an integer if you use hex. Yeah, you have to put the sign at the wrong end. And with a different endian-ness the coding might get more awkward, but what's the big deal?

The point is this. The BCD encoding is constrained to follow a particular format. Using `constexpr_bcd()` provides a convenient way for every packed BCD literal to be validated by the compiler. If a tired programmer tries to use an invalid value in the BCD encoding then the compiler will catch it at compile time. The compiler identifies the problem before the code can possibly leak out into the wild.

But there's a hole in the argument. You'll note that the variables `bad1` and `bad2` are not declared `constexpr`. Since they are not `constexpr`, the compiler has the option of evaluating them at run time, not compile time. What was once a compile-time error can now become a run-time error with a thrown exception. All of the necessary information was available at compile time. But the C++11 standard does not require translation-time evaluation of `constexpr` functions except under very specific circumstances. We'll look more closely at those circumstances later. For the moment it is sufficient to say that the designers of `constexpr_bcd()` did not get all of the compile-time checking that they hoped for. To get the full benefit of compile-time checking the user of `constexpr_bcd()` must diligently follow a specific rule with each use: the variable that receives the `constexpr_bcd()` result must also be declared `constexpr`.

While declaring each variable `constexpr` is not onerous, it is error prone. It's very easy to forget, and it must be applied in every case. If the `constexpr` is accidentally omitted, then the code compiles correctly and the string may be evaluated at runtime. That's usually okay. But if the BCD string contains an error, that error may turn into a run-time exception.

2.2 A `constexpr` Function that Computes a Square Root

While the previous example showed code that was only ever intended to execute at compile time, this next example is different. Here we'll look at code that can compute a square root. Square root computations are really useful at run-time as well as at compile time. Often a processor will provide intrinsics that help speed computation of a square root. However if we want the square root during translation, say to provide initialization for a static double, then we need to do the work ourselves.

The following is not a full-blown square root implementation. It is, however, representative of what an ordinary non-numerics coder might resort to. Once again, there's nothing unusual about the code if you wish to skip to the next normal text. The code is only included for completeness.

```
namespace constexpr_sqrt_detail
{
    class neg_sqrt_exception : public std::bad_exception
    {
        virtual const char* what() const noexcept override
        { return "A negative number has no square root"; }
    };

    class big_sqrt_exception : public std::bad_exception
    {
        virtual const char* what() const noexcept override
        { return "Value too big for constexpr_sqrt"; }
    };

    class modulus_numeric_type : public std::bad_exception
    {
        virtual const char* what() const noexcept override
        { return "Modulus types not allowed by this function."; }
    };

    // "Babalonian method" of successive square root approximation.
    template <typename T>
    constexpr T sqrt_approx(T value, T approx)
    {
        return ((approx + (value / approx)) / 2);
    }

    // The recursing part of the recursive square root calculation.
    template <typename T>
    constexpr T sqrt_recurse(int count, T value, T approx)
    {
        return
            ((count <= 0) ||                           // if limit
             (approx == sqrt_approx(value, approx)) ?  // or exact
            approx :                                   // then done
            sqrt_recurse(count-1, value, sqrt_approx(value, approx)));
    }
    // Before we run the "Babalonian method" it's good to get in the
    // ball park.  The following lookup works for modulus types.
    template <typename T>
    constexpr T sqrt_rough(T approx)
    {
        return
            approx <=                  10LL ?      approx :
            approx <=                 100LL ?          10 :
            approx <=               10000LL ?         100 :
            approx <=             1000000LL ?        1000 :
            approx <=           100000000LL ?       10000 :
            approx <=         10000000000LL ?      100000 :
            approx <=       1000000000000LL ?     1000000 :
            approx <=     100000000000000LL ?    10000000 :
            approx <=   10000000000000000LL ?   100000000 :
            approx <=  1000000000000000000LL ? 1000000000 :
            approx <=  9223372036854775807LL ? 2147483647 :
            throw big_sqrt_exception();
    }

    // Before we run the "Babalonian method" it's good to get in the
    // ball park.  This recursive approach works for non-modulus types.
    template <typename T>
    constexpr T sqrt_rough_recurse(
        int count, T target, T bound = 16, T approx = 4)
    {
        return std::numeric_limits<T>::is_modulo == true ?
            throw modulus_numeric_type() :
            target <= 1 ? target :
            target <= bound ? approx :
            count <= 0 ? throw big_sqrt_exception() :
            sqrt_rough_recurse(count-1,target,1024*bound,32*approx);
    }
} // end constexpr_sqrt_detail namespace

// Public facade for the constexpr_sqrt implementation
template <typename T>
constexpr T constexpr_sqrt(T value)
{
    using namespace constexpr_sqrt_detail;
    return value == 0 ? value :                      // No divide-by-0
        (value < 0) ? throw neg_sqrt_exception() :  // No sqrt of neg
        (std::numeric_limits<T>::is_modulo == true) ?
        sqrt_recurse<T>(100, value, sqrt_rough<T>(value)) :
        sqrt_recurse<T>(100, value, sqrt_rough_recurse<T>(100, value));
}
```

This time out, our two-or-so pages of code give us a way to compute square roots of many kinds of numbers during translation. Integers are well covered and a wide range of floats and doubles are covered as well. Here's an example of the code in use.

```
int main1()
{
    // Example of square root calculated at compile time.
    constexpr double sqrt_2 = constexpr_sqrt(2.0);
    static_assert((sqrt_2 > 1.41421356) && (sqrt_2 < 1.41421357),
        "Bad square root of 2");

    constexpr double sqrt_half = constexpr_sqrt(0.5);
    static_assert((sqrt_half > 0.70710678) && (sqrt_half < 0.70710679),
        "Bad square root of 0.5");

    constexpr unsigned int sqrt_82 = constexpr_sqrt(82U);
    static_assert(sqrt_82 == 9, "Bad integer square root of 82");

    constexpr long long sqrt_big = constexpr_sqrt(0x7FFFFFFFFFFFFFFFLL);
    static_assert(sqrt_big == 759250124, "Bad sqrt_big value");

    constexpr double sqrt_dbl = constexpr_sqrt(1e20);
    static_assert((sqrt_dbl < (1e10 + 0.1)) &&
        (sqrt_dbl > (1e10 - 0.1)), "Bad square root 1e20");
    assert(sqrt_dbl == std::sqrt(1e20));

    constexpr double sqrt_big_dbl = constexpr_sqrt(1e200);
    static_assert((sqrt_big_dbl < (1e100 + 1e84)) &&
        (sqrt_big_dbl > (1e100 - 1e84)), "Bad square root 1e200");
    assert(sqrt_big_dbl == std::sqrt(1e200));

    // The following won't compile because the rough
    // approximation recursion runs out of steam.
//  constexpr double sqrt_too_big = constexpr_sqrt(1.79e308);

    // The following won't compile because of the throw
//  constexpr double sqrt_neg = constexpr_sqrt(-1.0);

    return 0;
}
```

There are those who might question the value of computing a square root during translation. What's the use? The use may be in a chain of other calculations that validate limits during compile time. Or there may be tables of values that can be computed prior to execution where the square root plays a role. Are these huge use cases? No. But they are, nevertheless, valid use cases.

So what's the big deal? We now have the function that we want. We'll use it at compile time. Yes, we have what we want. But like the previous example, this one is also easily prone to accidental misuse. If the variable that captures the result of a `consexpr_sqrt()` call is not declared `constexpr`, then our compile-time only code may execute at runtime. Let's do some timing to see what happens. We'll call our `constexpr_sqrt()` function in a loop. One loop declares the target as constexpr, the other does not. Again there is nothing out of the ordinary in the code. You may wish to skip the code and look at the results.

```
int main()
{
    using clk_t = std::chrono::steady_clock;
    static constexpr double sq {1.1};

    // Time the constexpr computations
    const std::int32_t loop_count = 1000000000;
    volatile double constexpr_sqrt_result = 0;
    {
        const clk_t::time_point ta_0 = clk_t::now();
        for (int i = 0; i < loop_count; ++i)
        {
            constexpr double constexpr_value =        // constexpr here
                constexpr_sqrt(sq);
            constexpr_sqrt_result = constexpr_value;
        }
        const clk_t::time_point ta_1 = clk_t::now();
        const auto deltaA = ta_1 - ta_0;
        std::cout << "True constexpr_sqrt ticks:      "
            << deltaA.count() << std::endl;
    }
    volatile double runtime_sqrt_result = 0;
    {
        const clk_t::time_point tb_0 = clk_t::now();
        for (int i = 0; i < loop_count; ++i)
        {
            runtime_sqrt_result = constexpr_sqrt(sq); // not constexpr
        }
        const clk_t::time_point tb_1 = clk_t::now();
        const auto deltaB = tb_1 - tb_0;
        std::cout << "Runtime constexpr_sqrt ticks: "
            << deltaB.count() << std::endl;
    }
    assert(constexpr_sqrt_result == runtime_sqrt_result);
    return 0;
}
```

This code will show us the penalty extracted for forgetting to declare the target of the `constexpr_sqrt()` calculation as `constexpr`. If we compile and execute the previous code on an old Dell laptop using gcc 4.7.0 with full optimization we get the following results:

```
True constexpr_sqrt ticks:      312500
Runtime constexpr_sqrt ticks: 81609375
```

Of course your results will certainly vary, depending on your compiler and computing device. But, at least in this particular case, there is about a 250:1 time penalty for calling `constexpr_sqrt()` at runtime.

But perhaps that's a reasonable price to pay. Let's adjust our test code so we can compare the time for a call to std::sqrt() to the time taken by our constexpr_sqrt() function. The test code is included here, again for completeness. You may wish to skip the code and jump to the results.

```
int main()
{
    using clk_t = std::chrono::steady_clock;
    static constexpr double c_sq {1.1};
    volatile double v_sq {c_sq};      // force each call of std::sqrt()

    // Time the computations
    const std::int32_t loop_count = 1000000000;
    volatile double sqrt_lib_result = 0;
    {
        const clk_t::time_point ta_0 = clk_t::now();
        for (int i = 0; i < loop_count; ++i)
        {
            sqrt_lib_result = std::sqrt(v_sq);   // runtime library
        }
        const clk_t::time_point ta_1 = clk_t::now();
        const auto deltaA = ta_1 - ta_0;
        std::cout << "std::sqrt ticks:               "
            << deltaA.count() << std::endl;
    }
    volatile double runtime_constexpr_sqrt_result = 0;
    {
        const clk_t::time_point tb_0 = clk_t::now();
        for (int i = 0; i < loop_count; ++i)
        {
            runtime_constexpr_sqrt_result =
                constexpr_sqrt(c_sq);            // not constexpr
        }
        const clk_t::time_point tb_1 = clk_t::now();
        const auto deltaB = tb_1 - tb_0;
        std::cout << "Runtime constexpr sqrt ticks: "
            << deltaB.count() << std::endl;
    }
    assert(sqrt_lib_result == runtime_constexpr_sqrt_result);
    return 0;
}
```

Now we'll compile this test with the same optimization settings and execute on the same old Dell laptop. This will give us a sense for the runtime execution speed difference between `std::sqrt()` and our `constexpr_sqrt()`.

```
std::sqrt ticks:               7015625
Runtime constexpr sqrt ticks: 80234375
```

This time the time difference is not so stark. We see that `std::sqrt()` is a bit more that 10 times faster than our `constexpr_sqrt()` executed at runtime. Not quite as breathtaking as a 250:1 ratio, but still not peanuts.

Why is it that `std::sqrt()` runs so much faster, at least in this particular case? If we dig into the disassembly with this particular compiler on this particular old Dell laptop we see that the compiler has availed itself of (what is effectively) an intrinsic. It's using the X86 `fsqrt` assembly instruction. All our two pages of C++ code are wrapped up in a single assembly instruction. That's tough to beat.

At this point it is hopefully clear why, if we're computing a square root during runtime, we'd rather not use `constexpr_sqrt()`. It's only a matter of speed of execution. Still, the very best choice would be to compute the square root during translation, if that's possible.

A thoughtful person might at this point say, "Let's declare `std::sqrt` as `constexpr` in the standard." We'll let the compiler straighten it out.

That solves an individual problem, but it ignores any number of other intrinsic instructions that might be available to a specific processor. There may be sine, cosine, and tangent intrinsics. There may be an intrinsic that computes $\log_2$ of a value. And different processors will have different capabilities. It would be hard for the standard to predict all of the intrinsics that processor implementations might include. So declaring `std::sqrt` as `constexpr` would only be a patch to one small corner of a larger concern.

2.3 A Note about Recursion

One last point worth noting is that both of the preceding examples use recursion, not iteration, to perform repeated calculations. The rules for constexpr require that such computations be made recursively to accommodate compile-time evaluation. The `constexpr_sqrt()` function explicitly limits its recursion depth to 100, although the FDIS sets a lower limit of 512 recursive `constexpr` function invocations [see FDIS (N3242) **Annex B (informative) Implementation quantities**]. In most situations on modern processors, a recursion depth of 512 is nothing to worry about. But in an embedded environment with limited memory such recursion might exceed the stack limits. In such cases it could be vitally important for `constexpr_sqrt()` to be evaluated during translation.

3 Isn't `constexpr` a Compile-time Thing?

The C++11 standard gives `constexpr` the ability to perform computations during translation. But it only requires those computations to be done at translation under very specific circumstances. Here are some specific circumstances identified in the FDIS (N3242).

Section 3.6.2 Initialization of non-local variables, paragraph 2 says if an object with static or thread local storage duration is initialized with a `constexpr` constructor then the constructor is evaluated during translation:

> *Constant initialization* is performed:
>
> — if an object with static or thread storage duration is initialized by a constructor call, if the constructor is a `constexpr` constructor, if all constructor arguments are constant expressions (including conversions), and if, after function invocation substitution (7.1.5), every constructor call and full-expression in the *mem-initializer*s is a constant expression;

Section 7.1.5 The `constexpr` specifier, paragraph 9 says if an object declaration includes the `constexpr` specifier, that object is evaluated during translation (i.e., is a literal):

> A `constexpr` specifier used in an object declaration declares the object as const. Such an object shall have literal type and shall be initialized. If it is initialized by a constructor call, that call shall be a constant expression (5.19). Otherwise, every full-expression that appears in its initializer shall be a constant expression. Each implicit conversion used in converting the initializer expressions and each constructor call used for the initialization shall be one of those allowed in a constant expression (5.19).

Some people argue that this paragraph leaves room for an implementation to postpone the initialization until runtime unless the effect can be detected during translation due to, say, a `static_assert`. That may not be an accurate view because whether a value is initialized during translation is observable under some circumstances. This view is reinforced by **Section 5.19 Constant expressions** paragraph 4:

> [*Note:* Although in some contexts constant expressions must be evaluated during program translation, others may be evaluated during program execution. Since this International Standard imposes no restrictions on the accuracy of floating-point operations, it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program execution... — *end note*]

Section 9.4.2 Static data members, paragraph 3 says if a const static data member of literal type is initialized by a `constexpr` function or constructor, then that function or constructor must be evaluated during translation:

> If a static data member is of const literal type, its declaration in the class definition can specify a *brace-or-equal-initializer* in which every *initializer-clause* that is an *assignment-expression* is a constant expression. A static data member of literal type can be declared in the class definition with the `constexpr` specifier; if so, its declaration shall specify a *brace-or-equal-initializer* in which every *initializer-clause* that is an *assignment-expression* is a constant expression. [*Note:* In both these cases, the member may appear in constant expressions. —*end note*]

Outside of those circumstances the C++11 standard allows computations in `constexpr` functions and constructors to be performed at compile time. But it does not require it. The computations could occur at runtime. Which computations the compiler performs at compile time are, to a certain extent, a quality of implementation question.

The end result of these observations is that the person writing code has only a few ways to guarantee that a `constexpr` function or constructor is evaluated during translation. The most straight forward way is to declare the target of the constructor or function result as `constexpr`. This specifier is easy to forget. And if the `constexpr` specifier is omitted the code compiles beautifully. The primary observable consequences of the omission will be:

- Whether an error is caught at compile-time or at runtime,
- The size of the resulting code,
- The speed of code execution, and possibly (in small embedded systems)
- A runtime error due to exceeding the stack size.

4 Do These Observations Motivate a Change to the Standard?

First, let's summarize the observations so far.

- **Concern A**: it's easy to accidentally invoke a `constexpr` function or constructor so that its code executes at runtime rather than during translation. This is unimportant for most `constexpr` code, but is particularly relevant for `constexpr` code that is intended to catch errors during translation.

- **Concern B**: a `constexpr` function or constructor, invoked at runtime, may run substantially slower than non-`constexpr` code that accomplishes the same end.

- **Concern C**: a `constexpr` function or constructor, invoked at runtime, may use excessive amounts of stack that would not be used at all by non-recursive code that accomplishes the same end. This excessive stack usage may cause premature termination of a program in an environment with limited memory (as exemplified by some embedded environments).

Clearly the author of this paper feels that a change to the standard is justified, or he would not have droned on for so long. Each standard committee member must make their own call, however. Next we'll consider reasons that it might be good to change the standard.

4.1 Prefer Compile- and Link-Time Errors to Run-Time Errors

The heading for this section is stolen from Item 14 in *C++ Coding Standards* by Sutter and Alexandrescu. Similar sentiments have been attributed to other notables, like Scott Meyers and Bjarne Stroustrup.

The C++11 formulation of `constexpr` gives us almost the perfect tool for opening vast vistas of compile-time error checking. With `constexpr` such tools become easy to write. But such tools, once written, violate Item 18 in *Effective C++ Third Edition* by Scott Meyers:

Item 18: Make interfaces easy to use correctly and hard to use incorrectly.

Any `constexpr` function or constructor that is intended for compile-time error checking is easy to use incorrectly. If the target of a `constexpr` function result is not declared `constexpr`, then the error checking that was intended to be done at compile-time may become a run-time error. It is easy to forget that additional `constexpr` declaration which is required at every invocation site.

With a bit of adjustment, `constexpr` can become a tool that is as reliable at performing compile-time error checking as macros and template metaprogramming are today. And `constexpr` can do so in a way that is readable and maintainable, unlike the other two tools. But until that adjustment is made, `constexpr` cannot be used to reliably perform such error checks during compilation. It is simply too easy for any `constexpr` constructor or function to be accidentally evaluated at run-time, not at compile-time.

4.2 Is Execution Time Observable?

Earlier we noted that our constexpr_sqrt() example ran about 10 times slower than std::sqrt() when they are both invoked at runtime. For those of us who worry about execution speed it would be nice to get a compile-time hint that we may be using the wrong function. But should this kind of concern influence the standard?

According to the FDIS **Section 1.9 Program execution** paragraph 8 the observable behavior of a program is quite limited.

> The least requirements on a conforming implementation are:
>
> — Access to volatile objects are evaluated strictly according to the rules of the abstract machine.
>
> — At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.
>
> — The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.
>
> These collectively are referred to as the *observable behavior* of the program. [*Note:* More stringent correspondences between abstract and actual semantics may be defined by each implementation. *—end note*]

So, no, strictly speaking the standard does not consider speed of code execution to be observable.

But speed of execution is certainly relevant to many users of C++. Users often choose C++ as a language because it runs fast. The `constexpr` feature plays to this strength since, as long as it is evaluated during translation, it requires almost no runtime cycles or space in the code image.

4.3 Can Optimization Paper Over the Issues?

Much of the focus on `constexpr` to date has revolved around two things:

- What can we reasonably expect a C++ compiler to evaluate during translation?
- Given valid `constexpr` code, how can an optimizer make such code run fast?

These are great, and necessary, concerns to address. But they miss two important points:

1. If error checking, rather than speed, is the motivation for writing a `constexpr` constructor or function then it is still easy to misuse the function, even though the function executes quickly at runtime.
2. It seems unlikely that the optimizer folks, as clever as they are, could ever catch up with what can be hand coded invoking hardware accelerators like intrinsics.

The preceding statement is not intended to dismiss efforts in expanding the C++ code that a compiler can evaluate during compilation, or determining how the compiler can make such code execute quickly during runtime if that's necessary. Only certain specific `constexpr` constructors and functions need to be evaluated during translation. And efforts in those areas will reduce the number of situations where coders will worry about whether separate compile-time and run-time implementations are needed. But improved runtime optimization can never help in situations where compile-time error checking is desired.

4.4 How Could the Standard Be Changed?

In what ways could the C++ standard be changed to address issues with `constexpr` at runtime? During discussions four different types of changes to the standard have been considered. In no particular order the approaches are:

- **Approach A**: introducing a qualifier or attribute that allows a `constexpr` function or constructor to be marked as only useable during translation, not at runtime.

- **Approach B**: changing the requirements on `constexpr` code so it can no longer use general recursion. It must limit itself to tail recursion. Furthermore, compilers would be required to use the tail recursion optimization when generating runtime code for `constexpr` functions and constructors.

- **Approach C**: introducing a trait or a concept that specialized `constexpr` functions and constructors for either translation-time or runtime use.

- **Approach D**: extending function overloading so both `constexpr` and non-`constexpr` functions could share the same signature. The compiler would choose the correct function based on context.

The remainder of this paper explores these four options for viability. Each section will evaluate its approach for three concerns:

- **Concern A**: does it provide a mechanism to enforce compile-time error checking?
- **Concern B**: does it provide a means for invoking an appropriate function implementation at runtime?
- **Concern C**: does it provide a way to avoid excessive recursion on the stack at runtime?

5 Approach A: a constexpr Qualifier

Suppose we added a qualifier to constexpr that required the compiler to only use that code during translation. Code with such a qualifier would never be considered for runtime execution. The author of a constexpr function or constructor could use such a qualifier if they wanted to guarantee that their code was never used a runtime. This gives the author the ability to enforce that all errors caught by their code would always be caught during translation.

To make the idea more concrete, suppose we introduce a new attribute: [[constant_initialization_only]]. This attribute would only be applicable to constexpr function and constructor declarations and implementations. To keep things simple, let us insist that any declaration using the attribute must match the usage of the attribute by the implementation. The conscientious coder of our previous constexpr_bcd() function example would then add the qualifier to the five functions that make up its implementation. Like this:

```
namespace constexpr_bcd_detail
{
    [[constant_initialization_only]]
    template <typename T>
    constexpr size_t max_bits()
    { ... }

    [[constant_initialization_only]]
    template <typename T>
    constexpr T bcd_end(size_t bits, int sign, T x)
    { ... }

    [[constant_initialization_only]]
    template <typename T>
    constexpr T bcd_recurse(
        const str_const& bcd, size_t n, size_t bits, int sign, T x)
    { ... }

    [[constant_initialization_only]]
    template <typename T>
    constexpr T capture_sign(const str_const& bcd, size_t n)
    { ... }
}

[[constant_initialization_only]]
template <typename T = std::uint32_t>
constexpr T constexpr_bcd(constexpr_bcd_detail::str_const bcd)
{ ... }
```

With the qualifier in place the compiler would fail with a diagnostic if constexpr_bcd() were ever called on to execute at runtime. This would force the user of the function to fix the invocation, probably by adding a constexpr declaration to the result target. Otherwise they may need to decide on some other mechanism that is appropriate for computing packed BCD at runtime, probably using a different function.

How would such a qualifier help with the concerns raised earlier?

- **Concern A**: does it provide a mechanism to enforce compile-time error checking?

  Yes. Such qualified `constexpr` functions and constructors would only be evaluated during translation, so any detected errors would be reported during translation.

- **Concern B**: does it provide a means for invoking an appropriate function implementation at runtime?

  Yeah, sort of. If the code author wants to provide similar functionality at runtime they would have to provide a separate non-`constexpr` function or constructor with a different name or signature. This non-`constexpr` code could be designed to be as efficient as possible at runtime with regard to speed and stack usage.

- **Concern C**: does it provide a way to avoid excessive recursion on the stack at runtime?

  Yes. We explicitly forbid the recursive version of the function from ever executing at runtime.

What would be the downsides to such a qualifier?

- If two implementations of a function are provided, one for use during translation and one for runtime, then those functions must be distinguished by name, or different parameter types, or by placing the functions in different namespaces. In effect, the user of the function needs to take responsibility for invoking the correct version in the correct circumstance. The good news is that if the user of the function makes a mistake, then the compiler will straighten out the situation. The two implementations can never be confused because the situations where they execute (compile-time vs. run-time) have no overlap.

- The previous issue becomes a bit worse for constructors than for functions. Two constructors for the same type cannot have differing names or be in different namespaces. So `constexpr` constructors with the qualifier would need to take different arguments from other non-`constexpr` constructors of the same type. Sometimes the different arguments would have to be unused arguments included only to make the constructors distinct. It is worth noting that the extra argument, while a bit silly to the caller of the function, would have no runtime execution cost.

So this approach to the problem solves all of the previously identified problems. However it introduces the necessity of new names that are visible to both the implementer and the user of the functions of interest.

5.1 How Could Such a Qualifier Be Spelled?

The previous example spells the qualifier as an attribute. Attributes have the advantage of being dedicated to a specific circumstance. But it might be possible to spell the `constexpr` qualifier in other ways.

It's tempting to suggest `explicit constexpr` as a possible spelling. But that won't work. Using `explicit` would cause trouble with `constexpr` constructors and conversion operators.

A possible, albeit quirky, spelling would be `true constexpr`, since `true` is never a return type and is not a function or constructor qualifier.

This topic can be discussed further if using a qualifier is considered a good approach to solving the problem, but members of the committee find using an attribute to be distasteful.

6 Approach B: Mandatory Tail Recursion

Two of the major concerns lofted earlier were the speed of execution and use of stack space at runtime due to recursion. One way to address these problems would be to take the approach used by the Scheme programming language. Scheme requires that all implementations must be properly tail recursive. Here is a quote from the current Scheme programming language specification (found at (http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-8.html#node_sec_5.11).

> Implementations of Scheme must be ***properly tail-recursive***. Procedure calls that occur in certain syntactic contexts called *tail contexts* are *tail calls*. A Scheme implementation is properly tail-recursive if it supports an unbounded number of active tail calls. A call is ***active*** if the called procedure may still return. Note that this includes regular returns as well as returns through continuations captured earlier by `call-with-current-continuation` that are later invoked. In the absence of captured continuations, calls could return at most once and the active calls would be those that had not yet returned. A formal definition of proper tail recursion can be found in Clinger's paper [5]. The rules for identifying tail calls in constructs from the `(rnrs base (6))` library are described in section 11.20

C++ would not need to be as draconian as Scheme to solve the `constexpr` runtime issue. One possibility would be to:

- Limit `constexpr` recursion so only tail recursion was allowed, and
- Require compilers to implement the tail recursion optimization only for `constexpr` functions and constructors.

In terms of the success of the optimization, it appears that g++ 4.7.0 already takes this approach to optimizing certain `constexpr` code when the optimization level is high enough. So g++ demonstrates prior art of this technique with a history of success.

Requiring optimizations would be a bit of a break from tradition for the C++ standard. The standard has usually said very little about requirements on implementations.

How would such a qualifier help with the concerns raised earlier?

- **Concern A**: does it provide a mechanism to enforce compile-time error checking?

  No.

- **Concern B**: does it provide a means for invoking an appropriate function implementation at runtime?

  Not really, but it reduces the number of situations when a constexpr function or constructor implementation is inappropriate for runtime execution.

- **Concern C**: does it provide a way to avoid excessive recursion on the stack at runtime?

  Yes. Every recursive constexpr function must meet the tail recursion requirement, and every compiler must implement the tail recursion optimization. Even though the code looks like it uses the stack heavily, the compiler sidesteps the recursive function calls.

Downsides to requiring tail recursion would include:

- The new rules would make some amount of existing code (`constexpr` functions using recursion that is not tail recursion) illegal. During the transition to the new rule compiler implementations might choose to issue a warning regarding non-tail recursion in `constexpr` code. What is unknown at this time is whether there is an important class of potentially `constexpr` functions that cannot be implemented using only tail recursion.

- Insisting on only tail recursion would require compilers to identify and flag recursion of `constexpr` calls that was not tail recursion. For example, A() calls B() calls A() is recursive, but is not tail recursion. This suggested rule would make such recursion illegal in `constexpr` functions, and the compiler would be required to identify the problem. Compiler writers might find this constraint onerous.

All things considered, limiting `constexpr` functions and constructors to only using tail recursion and requiring compilers to provide the tail recursion optimization would be an elegant and largely transparent approach to solving one out of the three concerns described.

7 Intermission

The previous two approaches to addressing the runtime shortcomings of `constexpr` have been a bit non-traditional from the C++ perspective. Function overloading, particularly overloading on `const`, has a very long tradition with C++ and solves similar kinds of problems.

Traits have a tradition almost as long as overloading on `const`. Traits have been part of the C++ lexicon at least since 1993. Traits allow template implementations to be specialized based on characteristics of template parameter types.

It seems natural that one of these two techniques would be an excellent fit for addressing the concerns raised about `constexpr`. We'll see how that goes when we examine the two remaining approaches.

In anticipation of exploring those two approaches, it is worth talking through how a compiler might decide when to attempt to perform `constexpr` computations during translation, and when it might decide to postpone those computations until runtime. As shown in Section 3 of this paper, the FDIS itself offers very little guidance. Each compiler design group must determine their own approach. Here we'll consider two likely approaches:

- Evaluation based on call site context, and
- Evaluation based on availability of literals.

There are, no doubt, other models that could be taken. Hopefully these two approaches will be sufficient for our upcoming discussions of traits and overloading.

7.1 Evaluation Based on Call Site Context

Section 3 of this paper shows that the standard only mandates translation-time evaluation of `constexpr` code in three circumstances:

1. If a value or object with static or thread local storage duration is initialized with a `constexpr` function or constructor and all arguments are constant expressions.
2. If a value or object declaration includes the `constexpr` specifier.
3. If a const static data member of literal type is initialized by a `constexpr` function or constructor, then that function or constructor must be evaluated during translation.

Additionally, although the FDIS does not explicitly mandate it, there seems to be an expectation that if a `constexpr` computation result provides the extent of an array, that evaluation will occur during translation.

All four of these requirements for translation-time evaluation are based on the characteristics of the call site. Let's make this distinction a bit more concrete with an example from Daniel Krügler.

```
#include <iostream>

// A constexpr function that can produce undesirable behavior
constexpr int inverse(int v) { return 1/v; }

int main()
{
#if 1
    constexpr int c = inverse(0);           // Compile-time error
    static_assert(c != 1, "Unexpected value");
#endif
    const int r = inverse(0);               // Possibly run-time error
    std::cout << r << std::endl;
    return 0;
}
```

If this code is compiled with g++ 4.7.0 the code inside the `#if` causes compilation to fail with a diagnostic. If we disable the failing code (by entering `#if 0`) then compilation succeeds. The failure now happens at runtime.

Note that the two calls to inverse() are identical. In both cases inverse() is called with a literal known at compile time. The change in the failure mode is induced by the demands of the call site. In the case of `c`, the compiler is required to determine a value for `c` during translation (due to the `constexpr` and `static_assert`). In the case of `r`, the compiler chooses to postpone evaluation of inverse() until runtime since the standard allows it to do so. So in the `r` case we won't find out about the divide-by-zero until runtime.

This point is worth belaboring because in C++11 both traits and overloading make most of their decisions based on participants in the call – not due to the target at the call site. So, as long as the rules for runtime evaluation of `constexpr` don't change, we'll need to hold our tongues very carefully if we want to use traits or overloading to address shortcomings in `constexpr`.

7.2 Evaluation Based on Availability of Literals

A compiler doesn't have to support `constexpr` the way g++ 4.7.0 does in order to comply with the standard. Another approach, which would be fully compatible with the C++11 standard, would be to aggressively identify literals during translation. Then whenever all of the arguments to a `constexpr` function or constructor are literals, the compiler would always just go ahead and evaluate them during translation. In the end, all of the situations where the standard requires `constexpr` evaluation will be fully evaluated if at all possible. If `constexpr` evaluation was not possible where it was required then the compiler would recognize that and fail with a diagnostic.

The above description would probably turn out to be harder to implement than it initially looks. At least in part this happens because every time the compiler sees a literal, or something that might be a literal, the compiler must aggressively attempt to convert anything that consumes that literal into more literals. Only after such conversions fail would the compiler resort to generating runtime code. A compiler designed around this approach would probably suffer long compilation times while attempting to generate literals and, more often than not, failing to produce those literals.

Returning to the previous example, this more aggressive approach to `constexpr` evaluation would have a compile time error when evaluating `r` where the usual approach does not.

```
#include <iostream>

// A constexpr function that can produce undesirable behavior
constexpr int inverse(int v) { return 1/v; }

int main()
{
    // Assuming a compiler that aggressively evaluates constexpr
    // during translation...
    const int r = inverse(0);                  // Compile-time error
    std::cout << r << std::endl;
    return 0;
}
```

Since `inverse()` is declared `constexpr` it *can* be evaluated during translation. Since the argument (0) is a literal known during translation the compiler aggressively performs the evaluation during translation. The evaluation results in a divide-by-zero, so the compilation fails.

A quick look around did not reveal any C++11 compilers that take this aggressive approach to `constexpr` evaluation. So there is no evidence of prior art.

If the C++ standard were modified so all compilers were required to implement this more aggressive stance regarding `constexpr`, then we could have very good luck using overloading to address the concerns raised earlier in the paper. We'll see that later in the paper.

Unless there are other significant improvements as a result, the effort of changing the way `constexpr` functions and constructors are evaluated cannot be justified by the comparatively feeble complaints that motivate this paper. The discussion is only included for completeness.

8 Approach C: a Trait

If you'll recall, from Section 5 of this paper, a feasible solution to the `constexpr` issues was to use a qualifier or attribute to tell the compiler that the code could only be considered to be used during translation. But why do that with a qualifier? Why not use a trait?

Recall that, in C++11, `constexpr` functions and constructors are evaluated during translation based on the demands of the call site. The types of the arguments to the `constexpr` code are irrelevant for this determination. So we are wandering away from the usual use of traits. Instead we need the compiler to tell us when it is choosing to evaluate the code during translation or during runtime. The compiler certainly knows this information and it could make it available to the implementer.

To this end, suppose there were a standard library function that evaluated `true` at compile time and `false` at runtime. We could name the function `std::evaluated_during_translation()`. For the moment, let's ignore how such a function would be implemented. For our `constexpr_bcd()` function we could use it like this:

```
template <typename T = std::uint32_t>
constexpr T constexpr_bcd(constexpr_bcd_detail::str_const bcd)
{
    static_assert(std::evaluated_during_translation(),
        "constexpr_bcd may not be used at runtime.  "
        "Please declare the target variable as constexpr.");
    ...
}
```

Elegant, no? It fits in well with existing C++ practice. Non-`constexpr` code would never have need of such a `static_assert` because non-`constexpr` code can only ever be evaluated at runtime. And by using a `static_assert` we allow the code designer to provide guidance to the user regarding the remediation. And we did it all by introducing a single new (magic) library function.

But then some enterprising library writer starts to think about SFINAE. Why should we have two different functions with different names? Thinking about our `consexpr_sqrt()` case, we might consider writing…

```
template <typename T>
constexpr
typename std::enable_if<std::evaluated_during_translation(), T>::type
sqrt(T value)
{ /* constexpr_sqrt implementation */ }

template <typename T>
typename std::enable_if<! std::evaluated_during_translation(),T>::type
sqrt(T value)
{ std::sqrt(value); }
```

What you see here is an attempt to select an implementation for `sqrt()` based on how it is used at the invocation site. From a naïve perspective, it seems as though a compiler could interpret this code in one of two possible ways:

1. If the compiler re-evaluates the template instantiation each time it is invoked, then what we've created is probably a violation of the one definition rule. The types of the template parameters have no bearing on whether `std::evaluated_during_translation()` evaluates to `true` or `false`; the result is controlled exclusively by the call site's requirements. Consequently the code could easily generate both implementations. The implementations would come from two different call sites: one requiring evaluation during translation, and the other requiring runtime evaluation. The only differences in the two signatures would be whether or not the functions are declared `constexpr`.

2. More likely, the compiler only fully evaluates the template instantiation once for each set of template parameters. The compiler probably saves the result of that first evaluation and re-uses it when the same template parameters recur. In that case we've created a pseudo coin toss as to which implementation will be instantiated. If the first use that the compiler sees

evaluates the function during translation, then the `constexpr` version is generated. Otherwise the non-`constexpr` version is generated. And which was seen first would vary from one translation unit to the next. So we again get a violation of the one definition rule, but it is one that the translation tools are not required to diagnose.

Certainly there are other possible outcomes. But if the committee chooses the path of introducing a trait, then some clever person will need to spend time determining the best way of handling attempts at using the trait in an SFINAE context.

How would such a qualifier help with the concerns raised earlier?

- **Concern A**: Does it provide a mechanism to enforce compile-time error checking?

  Yes.

- **Concern B**: Does it provide a means for invoking an appropriate function implementation at runtime?

  Yeah, sort of. The trait provides the same kind of capabilities as the qualifier, but improves on them slightly. By using a well written static assert, the function's author can tell the user which function should be used in place of the one that was called. The function's author must still supply two distinct functions that are distinguished by name or signature.

- **Concern C**: Does it provide a way to avoid excessive recursion on the stack at runtime?

  Yes. We can explicitly forbid the recursive version of the function from ever executing at runtime.

What would be the downsides to such a trait?

- As noted earlier, the trait is quite general and is open to abuse. One obvious abuse case was identified, and there are probably others. Possible abuses should be identified and, if appropriate, remediation put in place if the committee selects a trait as the best approach.

So we see that our `std::evaluated_during_translation()` function provides an elegant solution. But, because the solution is quite general, it leads to other interesting questions. The `constexpr` qualifier, described in Section 5 of this paper, is less elegant but it has a smaller impact since it can only be used in limited contexts.

9 Approach D: Overloading on `constexpr`

A truly elegant solution to the runtime efficiency issues with `constexpr` would be to allow an implementer to provide two implementations: one for use during translation and one for use at runtime. This sounds like a perfect opportunity for overloading.

9.1 How Does constexpr Participate in Overloading in C++11 Today?

Before we consider overloading on constexpr, we should consider the existing rules within C++11.

In C++11 constexpr does play in overloading, but only because constexpr is a superset of const. Overloading on constexpr is only supported for member functions. Since a constexpr declaration is also const, it's as though the constexpr declaration is a const declaration with a special capability – it can be evaluated during translation. So C++11 does not support overloading a given member function on both const and constexpr.

```
// Overload non-static non-const class member on constexpr compiles
struct test_a {
    bool           is_constexpr() { return false; }
    constexpr bool is_constexpr() { return true; }        // Succeeds
};

// Overload member on both const and constexpr fails
struct test_b {
    bool is_constexpr()           { return false; }
    bool is_constexpr() const     { return false; }
    constexpr bool is_constexpr() { return true; }        // Error
}
```

Given this, our examination of overloading on constexpr need only make sure that const overloading still works correctly when we're done.

9.2 Overload Resolution for constexpr

Overload resolution for constexpr is based on three ideas:

1. A constexpr function, method, or constructor is the only way to evaluate a value during translation. So, for situations involving evaluation during translation, a constexpr function, method, or constructor is the only (and hence best) choice.

2. For non-static member functions, the constness of the object determines which overload is called. This precisely models, but extends, overloading non-static member functions on const. It would be allowed to overload a member function on constexpr, const, and not-const. If all three overloads were provided for a given class member signature, the…

    a. constexpr overload would be called for evaluation during translation, the
    b. const overload would be called for runtime evaluation on a const object, and the
    c. non-const overload would be called for runtime evaluation on a non-const object.

3. During runtime, a constexpr function, method, or constructor is likely to be the least efficient choice – otherwise a non-constexpr version would not be provided. Therefore for runtime execution, if a non-constexpr version of an overloaded function, method, or constructor is available, then the non-constexpr version is preferred.

With the addition of these three simple rules, conceptually no other rules need to change. Of course the actual textual changes to the standard would be much more extensive.

You'll notice that `constexpr` overloading ends up being much more wide spread than overloading on `const`. Overloading on `const` can only happen with non-static member functions. Overloading on `constexpr` would need to apply to:

- free functions,
- static member functions, and
- constructors as well as with
- non-static member functions.

9.3 Examples of `constexpr` Overloading

Given these suggested rules, let's examine overload resolution for free functions, static member functions, constructors, and non-static member functions. The examples are short and sweet to show intent. The functions would need to involve significant computations to provide motivation.

9.3.1 Overloading of Free Functions

The following would be legal:

```
// Overload free function on constexpr
bool           is_constexpr() { return false; }
constexpr bool is_constexpr() { return true; }

int main()
{
    // Since test1 is declared constexpr, its value must be evaluated
    // during translation.
    constexpr bool test1 = is_constexpr();
    static_assert(test1 == true, "Yipes!");

    // In C++11 the compiler has discretion for whether to evaluate at
    // runtime or during translation.  The assert may fail depending
    // on the mood of the compiler.
    const bool test2 = is_constexpr();
    assert(test2 == false);                       // May fail (or not)
    return 0;
}
```

And, since overloading on `constexpr` is not required, the following would still be legal:

```
constexpr bool can_constexpr() { return true; }

int main()
{
    constexpr bool test1 = can_constexpr();
    static_assert(test1 == true, "Yipes!");
    bool test2 = can_constexpr();
    assert(test2 == true);
    return 0;
}
```

9.3.2 Overloading of Static Member Functions

The following would be legal:

```
struct test_ba {
    // Overload static member function on constexpr
    static bool is_constexpr() { return false; }
    static constexpr bool is_constexpr() { return true; }

    // Non-overloaded constexpr static member function
    constexpr static bool can_constexpr() { return true; }
};

int main()
{
    // Using the overloaded static member functions...
    static_assert(test_ba::is_constexpr() == true, "Yipes!");

    // In C++11 the compiler has discretion for whether to evaluate at
    // runtime or during translation.  The assert may fail depending
    // on the mood of the compiler.
    const bool tba_1 = test_ba::is_constexpr();
    assert(tba_1 == false);                         // May fail (or not)

    // Using the non-overloaded constexpr static member function...
    static_assert(test_ba::can_constexpr() == true, "Yipes!");
    const bool tba_2 = test_ba::can_constexpr();
    assert(tba_2 == true);
    return 0;
}
```

9.3.3 Overloading of Constructors

The following would be legal:

```
// Overload constructor on constexpr
struct test_aa {
    test_aa() : v_{false} { }
    constexpr test_aa() : v_{true} { }

    constexpr bool is_constexpr() { return v_; }

private:
    bool v_;
};

// Class that only has a constexpr constructor
struct test_ab {
    constexpr test_ab() { }
    constexpr bool can_constexpr() { return true; }
};

int main()
{
    // Overloaded constructor
    constexpr test_aa taa_1;
    static_assert(taa_1.is_constexpr() == true, "Yipes!");

    // In C++11 the compiler has discretion for whether to evaluate at
    // runtime or during translation.  The assert may fail depending
    // on the mood of the compiler.
    const test_aa taa_2;
    assert(taa_2.is_constexpr() == false);         // May fail (or not)

    // Non-overloaded constexpr constructor
    constexpr test_ab tab_1;
    static_assert(tab_1.can_constexpr() == true, "Yipes!");

    test_ab tab_2;
    assert(tab_2.can_constexpr() == true);
    return 0;
}
```

9.3.4 Overloading of Non-Static Member Functions

The following would be legal:

```
enum class my_constness {
    is_constexpr,
    is_const,
    is_not_const
};

// Overload non-static class member on constexpr and const
struct test_da {
    constexpr test_da() { }

    my_constness constness() { return my_constness::is_not_const; }
    my_constness constness() const { return my_constness::is_const; }
    constexpr my_constness constness()
        { return my_constness::is_constexpr; }
};

int main()
{
    // The constness of the object determines the overload used,
    // just like with const overloading...
    constexpr test_da tda_1;
    static_assert(
        tda_1.constness() == my_constness::is_constexpr, "Yipes!");

    // In C++11 the compiler has discretion for whether to evaluate at
    // runtime or during translation.  So tda_2 may have been
    // constructed with either the constexpr or the const constructor
    // depending on the compiler's mood.  The assert may (or may not)
    // fail.
    const test_da tda_2;
    assert(tda_2.constness() == my_constness::is_const);    // May fail

    // Since tda_3 is not const, the compiler must use the non-const
    // constructor.  The following assert is guaranteed to succeed.
    test_da tda_3;
    assert(tda_3.constness() == my_constness::is_not_const);
    return 0;
}
```

9.4 Forbidding Runtime Execution with Overloading

One of the goals for any of the approaches described is to allow a developer to force a function or constructor to either be evaluated during compilation or to fail to compile. Nothing in `constexpr` overloading leads directly to that goal. We must be a little tricky. However, by using templates and SFINAE we can achieve that goal. Here's an example:

```
// Non-constexpr function that static_asserts
template <typename T>
bool during_compilation_with_arg(T arg)
{
    static_assert(
        sizeof (T) == 0,
        "Assign return value to a constexpr.");
    return false;
}

// Constexpr function overload that compiles, but only with bool
template <typename T>
constexpr bool during_compilation_with_arg(T arg)
{
    static_assert(std::is_convertible<T, bool>::value, "Pass a bool");
    return arg;
}

int main()
{
    // The following evaluates during translation
    constexpr bool compiling1 = during_compilation_with_arg(true);
    static_assert(compiling1 == true, "Yipes!");

    // The following fails with a static_assert
    volatile bool runtime_bool = true;
    bool compiling2 = during_compilation_with_arg(runtime_bool);
    assert(compiling2 == true);
}
```

The preceding example shows a possible technique using `constexpr` overloading to require that a function either be evaluated during compilation or fail to compile. The goal has been achieved, but it takes a bit of work to pull off. Here are two points worth noting:

1. Only template functions or constructors can perform this trick. In the example, we wish to enforce the only-during-compilation rule on a plain (non-template) function that accepts a bool. That won't work because SFINAE only applies to templates. To only cause the `static_assert` when the non-`constexpr` version is invoked, the non-`constexpr` version must be a template.

2. The declarations of the failing and succeeding cases must be identical other than the leading `constexpr`. If they are not identical then overload resolution may thwart the attempt. Therefore the successful (`constexpr`) version of the function must also be declared as a template. If it is not also declared as a template then overload resolution will prefer the non-template version which would side-step our attempt at requiring compile-time only evaluation.

The technique gets even trickier if the function or constructor takes no arguments.

So `constexpr` overloading does a great job of effortlessly substituting `constexpr` and non-`constexpr` versions of functions or constructors. But `constexpr` overloading is much harder to apply to the problem of guaranteeing that a function or constructor can only be evaluated during compilation.

9.5 Consequences of Overloading

A number of the preceding examples have indeterminate results. That's unfortunate. It's hard to use a language feature in a program when the result of using the feature is unpredictable. The unpredictability of `constexpr` overloading, which results from current C++11 rules for `constexpr` evaluation, makes it undesirable as a programming tool of any sort.

If we ignore the unpredictability of overloading, we can return to the original set of three concerns about a proposed approach. How would overloading on `constexpr` help with the concerns raised earlier?

- **Concern A**: Does it provide a mechanism to enforce compile-time error checking?

  Yes, but it takes some work. See Section 9.4 of this paper.

- **Concern B**: Does it provide a means for invoking an appropriate function implementation at runtime?

  Yes, beautifully and, for the user, transparently.

- **Concern C**: Does it provide a way to avoid excessive recursion on the stack at runtime?

  Yes. By providing an appropriate non-`constexpr` overload the author can prevent the recursive version of the function or constructor from ever executing at runtime.

What would be the downsides to overloading on `constexpr`?

- The unpredictability regarding which overload the compiler will choose seems like a difficult issue to overcome. As noted in Section 7 of this paper, it might be possible to change the rules for `constexpr` evaluation in such a way that constexpr overloading would become predictable. Such an extensive change is far beyond the scope of this paper and would need strong motivation.
- Using overloading it is possible, but awkward, to require a function or constructor to only be evaluated during translation. Either the `constexpr` qualifier approach or the `std::evaluated_during_runtime` trait would be easier to use in this arena.
- The added compiler complexity of `constexpr` overloading may be difficult for compiler developers to support. Overloading rules are already quite complicated, and this would increase the complexity.

10 Summary

The `constexpr` feature added by C++11 is great. We have, however, looked at two situations where `constexpr` may fall a bit short:

1. It would be useful for there to be a way to guarantee that a certain select `constexpr` functions and constructors are evaluated during translation. This capability would considerably enhance detecting errors during compilation rather than at runtime.

2. There are situations where the techniques required for writing a `constexpr` function or constructor must result in less than the fastest possible runtime code. In these cases it would be useful to provide an exclusively `constexpr` version of the function or constructor and a distinct runtime version of the same.

This paper argues that these considerations are significant enough that it would be appropriate to change the standard in some way to improve the situation.

The trick is determining how the standard should be changed.

Four different approaches to changing the standard were examined:

- **Approach A**: introducing a qualifier or attribute that allows a `constexpr` function or constructor to be marked as only useable during translation, not at runtime.

- **Approach B**: Changing the requirements on `constexpr` code so it can no longer use general recursion. It must limit itself to tail recursion. Furthermore, compilers would be required to use the tail recursion optimization when generating runtime code for `constexpr` functions and constructors.

- **Approach C**: introducing a trait or a concept that could be used to specialize `constexpr` functions and constructors for either translation-time or runtime use.

- **Approach D**: extending function overloading so both `constexpr` and non-`constexpr` functions could share the same signature. The compiler would choose the correct function based on context.

All four of these approaches have their own advantages and drawbacks. The drawbacks range from the introduction of ugly keywords to the introduction of unpredictable behavior.

11 Acknowledgements

Many thanks to the folks who reviewed and contributed to this paper:

- Special thanks to Walter E. Brown. He suggested investigating overloading as a way to address these `constexpr` issues. He also provided other editing and helpful comments on several early drafts.

- Daniel Krügler kindly and thoroughly answered a question regarding `constexpr` on comp.std.c++. The divide-by-zero example in Section 7.1 was directly lifted from his response. See https://groups.google.com/forum/?fromgroups=#!topic/comp.std.c++/QQ34k_1b_Hs.

- Brian Schiller provided many useful comments.

- Marc Glisse provided information about anticipated `consexpr`-izations of `std::array`. His information had significant impact on the motivating example and removed a concern from the paper.

- Jeremiah Willcock correctly pointed out that compile-time error checking is a more significant consideration than speed of code execution.

- Rick Coates contributed useful suggestions for organizing the paper.

- Thanks also to all the reviewers at std-proposals@isocpp.org. Their comments were invaluable in finalizing the paper.

12 References

Information about packed BCD encoding is at http://en.wikipedia.org/wiki/Binary-coded_decimal

The implementation for the `str_const` class, used in the `constexpr_bcd` example, came from http://en.cppreference.com/w/cpp/language/constexpr

Information about the Babylonian Method of finding square roots may be found at http://en.wikipedia.org/wiki/Methods_of_computing_square_roots