



constexpr

C++ at Compile Time

By Scott Schurr for Ripple Labs at C++Now May 2015

constexpr

C++ at Compile Time

By Scott Schurr for Ripple Labs - May 2015



Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- Compile-time floating point
- Compile-time containers
- Future and Summary



constexpr beginning

Constant Expression

- Evaluate expressions at compile time
- Like template metaprogramming
 - But uses familiar C++ syntax
 - Therefore easier to maintain
- Only produces constant values
 - constexpr objects can't change at runtime



Why Is constexpr Interesting?

- No runtime cost:
 - No execution time
 - Minimal executable footprint
- Errors found at compile or link time
- No synchronization concerns



constexpr Contexts

- New keyword: constexpr
- Introduced in C++11
- constexpr values:
 - Definition of an object
 - Declaration of a static data member of literal type
- constexpr computations:
 - Functions
 - Constructors



constexpr Values

```
constexpr int const_3 = 3;           // Object definition
constexpr double half = 0.5;        // Object definition
static_assert (half < const_3, "Yipe!");
constexpr char tile_fixative[] = "grout"; // Object definition
static_assert (tile_fixative[5] == '\0', "Yipe!");
```

```
void free_func () {
    constexpr float pi = 3.14159265; // Object definition
    static_assert ((3.1 < pi) && (pi < 3.2), "Yipe!");
}
```

```
struct my_struct {
    // Static data member of literal type
    static constexpr char who[] = "Gabriel Dos Reis";
    static_assert (who[0] == 'G', "Yipe!");
    static constexpr const char* a = &who[1];
    static_assert (*a == 'a', "Yipe!");
};
```



constexpr Value Rules

- May be any literal type including:
 - Floating point types
 - Character literals
 - Pointer literals
 - Literal objects
- Requires no storage declaration
- constexpr parameters not allowed!

```
int bad_func (constexpr int v) { // Error!  
    return v * 5;  
}
```



constexpr Value Usage

- Use anywhere a literal may be used:
 - Non-type template parameters
 - Array dimensions
 - Enum initialization
 - Standard runtime code
- Implicitly const
 - Casting away const is undefined behavior



constexpr Computations

- constexpr declaration allowed on:
 - Free functions
 - Member functions
 - Constructors
- Allowed code:
 - Constrained in C++11
 - Relaxed somewhat in C++14
- constexpr constructor allows user-defined literal types



Compile-time Evaluation Allowed

Remove computations from runtime

Why?

- Reduce runtime execution time
- Reduce total program footprint
- Errors caught at compile or link time

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

Item 14 in *C++ Coding Standards* by Sutter and Alexandrescu



Um, Allowed?

constexpr code can run at both...

- Compile time
- And runtime



constexpr Evaluation Example

```
constexpr double half_of(double x)
{
    return x / 2;
}

void example()
{
    // Evaluate at compile time.
    constexpr double half = half_of (1.0);
    static_assert ((half < 0.51) && (half > 0.49), "Yipe!");

    // Evaluate at runtime.
    char c;
    std::cin >> c;
    const double run = half_of (c);
    assert (run == (c * 0.5));
}
```



constexpr Evaluation

Evaluation may be at runtime

To force evaluation during translation:

- Declare object or value constexpr

```
constexpr int nasty_computation (double v) {  
    ...;  
}
```

```
constexpr int nasty = nasty_computation (1.0);
```

- Use result where a literal is required

```
int nasty_array[nasty_computation(2.0)] {};
```



constexpr is Part of the Definition

The following will not compile...

```
int const_5();    // Forward declaration

// Same declaration: one constexpr, one not.
constexpr int const_5()    // Error!
{
    return 5;
}
```

Different definitions in different translation units violate One Definition Rule. No diagnostic required.



Implicitly Inlined

Definition must be visible in the translation unit

Before the first invocation



constexpr and Floating Point

Compile-time floating point calculations might not have the same results as runtime calculations

Looking inside the implementation of a floating point number (e.g., with `reinterpret_cast`) not allowed



Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- Compile-time floating point
- Compile-time containers
- Future and Summary



constexpr in C++11



C++11 constexpr Function

- Not virtual
- Returns
 - Literal type or
 - Reference to literal type
- Parameters must be
 - Literal types or
 - References to literal types
- Body is one compound statement:
 { **return** *expression* ; }
- Unevaluated subexpressions ignored



Just One Statement???

Yup.

- Compound statement allowed
- Function calls allowed
- Ternary $[A ? B : C]$ operator allowed

We'll use...

Say it...

Recursion



constexpr_pow_int_cpp11

```
namespace constexpr_pow_int_cpp11_detail {  
    // Implementation  
    constexpr double pow_int_cpp11 (double base, int exp) {  
        return (exp == 0 ? 1.0 : // Terminate  
                base * pow_int_cpp11 (base, exp - 1)); // Recursion  
    }  
}  
// User-facing interface  
constexpr double  
constexpr_pow_int_cpp11 (double base, int exp)  
{  
    using namespace constexpr_pow_int_cpp11_detail;  
    return (exp > 100) || (exp < -100) ?  
        throw std::range_error ("abs(exp) exceeds 100") :  
        exp >= 0 ?  
            pow_int_cpp11 (base, exp) :  
            pow_int_cpp11 (1.0 / base, -1 * exp);  
}
```



throw in constexpr?

Throw idiom for constexpr errors

- Compile error if throw is evaluated during compilation
- Legitimate throw if error during runtime



Using constexpr_pow_int_cpp11

```
int main()
{
    static_assert (                // Compute at compile time
        constexpr_pow_int_cpp11(2.0, 0) == 1.0,    "Yipe!");
    static_assert (
        constexpr_pow_int_cpp11(2.0, 5) == 32.0,    "Yipe!");
    static_assert (
        constexpr_pow_int_cpp11(2.0, -5) == 0.03125, "Yipe!");

    std::random_device rd;        // Compute at runtime time
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0, 1);

    const double r = dis (gen);
    assert (constexpr_pow_int_cpp11(r, 2) == r * r);
    return 0;
}
```



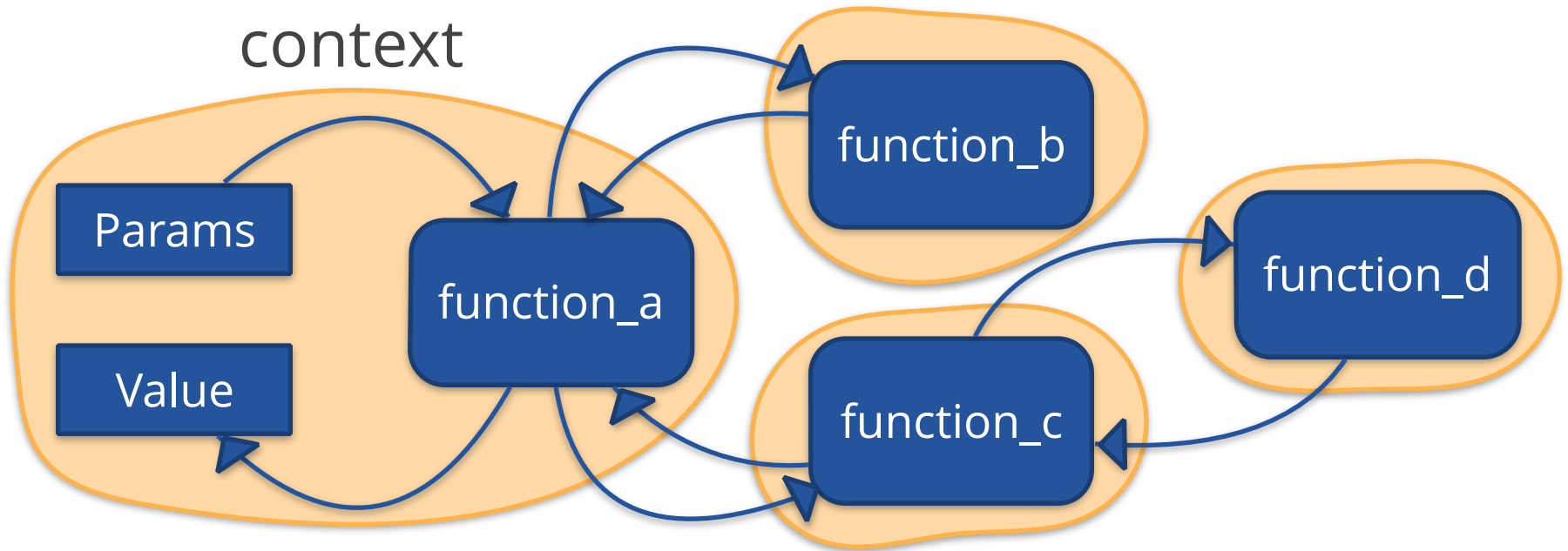
C++11 constexpr Constructor

- Params are literal or ref to literal
- No function-try-block
- Constructor body is empty
- Non-static data members and base-class sub-objects must be init-ed
- All invoked ctors must be constexpr
- Every assignment in initializer list must be a constant expression



C++11 constexpr Model

- C++ interpreter
- Each constexpr function has its own context



Think like a functional programmer

C++11 constexpr Summary

- Highly constrained
- Surprisingly useful with some effort
- C++14 makes it easier



Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- Compile-time floating point
- Compile-time containers
- Future and Summary



constexpr in C++14



C++14 constexpr Functions Can't

C++11 constexpr says what you **can** do

C++14 says what you **can't** do:

- Most examinations of this
- Calling non-constexpr functions
- Operations with undefined behavior
- Lambda expressions
- Most lvalue-to-rvalue conversions
- Referencing uninitialized data
- Conversion from void* to object*
- Modification of non-local objects



C++14 constexpr Can't Continued

- Comparison with unspecified results
- `typeid` of a polymorphic class
- `try` block
- `asm` declaration
- `goto`
- `dynamic_cast`
- `reinterpret_cast`
- `new`
- `delete`
- `throw`



constexpr Functions Can...

- The rules protect the interpreter
- Mostly, it's like regular code
- Much easier than in C++11



constexpr_pow_int_cpp14

```
constexpr double
constexpr_pow_int_cpp14 (double base, int exp)
{
    if ((exp > 100) || (exp < -100)) {
        throw std::range_error ("abs(exp) exceeds 100");
    }

    if (exp < 0) {
        base = 1.0 / base;
        exp = -1 * exp;
    }

    double result = 1.0;
    for (int i = 0; i < exp; ++i) {
        result *= base;
    }
    return result;
}
```



Using constexpr_pow_int_cpp14

```
int main()
{
    static_assert (                // Compute at compile time
        constexpr_pow_int_cpp14(2.0,  0) ==  1.0,    "Yipe!");
    static_assert (
        constexpr_pow_int_cpp14(2.0,  5) == 32.0,    "Yipe!");
    static_assert (
        constexpr_pow_int_cpp14(2.0, -5) == 0.03125, "Yipe!");

    std::random_device rd;          // Compute at runtime time
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0, 1);

    const double r = dis (gen);
    assert (constexpr_pow_int_cpp14(r, 2) == r * r);
    return 0;
}
```



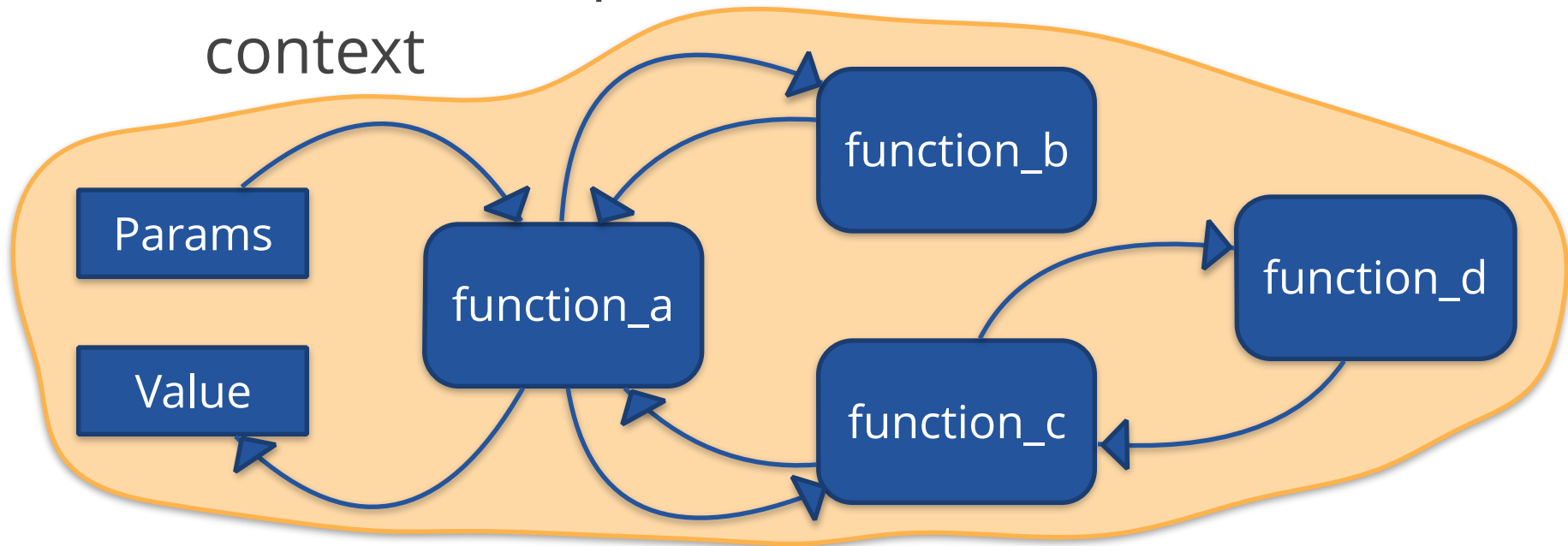
C++14 constexpr Constructor

- Body follows constexpr function rules
- Every constructor for bases and non-static members is constexpr
- Class or struct must initialize:
 - Every base-class sub-object
 - Every non-static data member
- Non-empty union must initialize:
 - Exactly one non-static data member



C++14 constexpr Model

- C++ interpreter
- Each constexpr result has its entire context



Think like a C++ programmer without new

Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- Compile-time floating point
- Compile-time containers
- Future and Summary



Compile-time Parsing

ZUM GEDACHTNIS AN DEN ARZT,
NATURFORSCHER UND PHILOSOPH
THEOPHRASTUS PARACELSUS
NEUERER DER MEDIZIN, VATER DER
CHEMOTHERAPIE, FÖRDERER DER BIOLOGIE
UND DER WUNDARZNEI, RETTER DER
LEISTESUMNACHTETEN, KÜNDER DES
ÄRZTLICHEN ETHOS, EIGENWILLIGER DER
ARMEN. ENDE 1493 NEBEN DER TEUFEL
RÜCKE AM ETZEL GEBOREN, IST ER NACH
SEINEM FAUSTISCHEN LEBEN AM 24. SEPTEMBER
1541 ZU SALZBURG VERSTORBEN
EINER FINSIDIGEN HEIMAT FINGEDEN

C++11 Binary Literal

Two parts:

- Pass text into a constexpr function
- Parse text



constexpr_txt

```
// literal char[] class
class constexpr_txt {
private:
    const char* const p_;
    const std::size_t sz_;
public:
    template<std::size_t N>
    constexpr constexpr_txt(const char(&a)[N]) :
        p_(a), sz_(N-1) {}

    constexpr char operator[](std::size_t n) const {
        return n < sz_ ? p_[n] :
            throw std::out_of_range("");
    }

    constexpr std::size_t size() const { return sz_; }
};
```



constexpr11_bin

```
template <typename T = std::uint32_t>
constexpr T constexpr11_bin(
    constexpr_txt t,
    std::size_t i = 0,           // index
    std::size_t b = 0,           // bit count
    T x = 0)                     // accumulator
{
    return
        i >= t.size() ? x : // end recursion
        b >= std::numeric_limits<T>::digits ?
            throw std::overflow_error("Too many bits!") :
        t[i] == ',' ? constexpr11_bin<T>(t, i+1, b, x) :
        t[i] == '0' ? constexpr11_bin<T>(t, i+1, b+1, (x*2)+0) :
        t[i] == '1' ? constexpr11_bin<T>(t, i+1, b+1, (x*2)+1) :
        throw std::domain_error(
            "Only '0', '1', and ',' may be used");
}
```

Using constexpr11_bin

```
int main()
{
    using u8_t = std::uint8_t;

    constexpr u8_t maskA = constexpr11_bin<u8_t>("1110,0000");
    constexpr u8_t maskB = constexpr11_bin<u8_t>("0001,1000");
    constexpr u8_t maskC = constexpr11_bin<u8_t>("0000,0110");
    constexpr u8_t maskD = constexpr11_bin<u8_t>("0000,0001");

    static_assert(
        maskA + maskB + maskC + maskD == 0xFF, "Yipe!");

    constexpr double d = constexpr11_bin<double>("1000");
    static_assert(d == 8.0, "Yipe!");

    return 0;
}
```



C++14 binary literal

- Much easier to code than in C++11
- Not so useful...
 - C++14 has built-in binary literals
 - `0b1101'0011`



constexpr14_bin

```
template <typename T = std::uint32_t>
constexpr T constexpr14_bin(const char* t)
{
    T x = 0;
    std::size_t b = 0;
    for (std::size_t i = 0; t[i] != '\0'; ++i) {
        if (b >= std::numeric_limits<T>::digits)
            throw std::overflow_error("Too many bits!");
        switch (t[i]) {
            case ',': break;
            case '0': x = (x*2); ++b; break;
            case '1': x = (x*2)+1; ++b; break;
            default: throw std::domain_error(
                "Only '0', '1', and ',' may be used");
        }
    }
    return x;
}
```



Using constexpr14_bin

```
int main()
{
    using u8_t = std::uint8_t;

    constexpr u8_t maskA = constexpr14_bin<u8_t>("1110,0000");
    constexpr u8_t maskB = constexpr14_bin<u8_t>("0001,1000");
    constexpr u8_t maskC = constexpr14_bin<u8_t>("0000,0110");
    constexpr u8_t maskD = constexpr14_bin<u8_t>("0000,0001");

    static_assert(
        maskA + maskB + maskC + maskD == 0xFF, "Yipe!");

    constexpr double d = constexpr14_bin<double>("1000");
    static_assert(d == 8.0, "Yipe!");

    return 0;
}
```



Thoughts On constexpr So Far

- How do developers debug this stuff?
- What do user errors look like?
- Do users want runtime execution?



Debugging: Three Approaches

- Bull through
- Run the code in a debugger
 - Init a non-literal with the function
 - Run that code in a debugger
 - Runtime constexpr is handy!
- Add print statements
 - Not so easy. I/O is never constexpr
 - Remove constexpr qualifier, or
 - Copy the code and rename it



User Errors at Compile Time

```
int main()
{
    constexpr auto mask =
        constexpr11_bin<std::uint8_t>("1110 0000");
    static_assert(mask == 0xE0, "Yipe!");
    return 0;
}
```

error: constexpr variable 'mask' must be initialized by a
constant expression

```
constexpr auto mask =
    ^
```

note: subexpression not valid in a constant expression
throw std::domain_error(
^

note: in call to 'constexpr11_bin({&"1110 0000"[0],
9}, 4, 4, 14)'
t[i] == '0' ? constexpr11_bin<T>(t, i+1, b+1, (x*2)+0) :
^



User Errors At Runtime

```
int main()
{
    auto mask = // <- Not constexpr!
                constexpr11_bin<std::uint8_t>("1110 0000");
    assert(mask == 0xE0);
    return 0;
}
```

libc++abi.dylib: terminating with uncaught exception of
type std::domain_error: Only '0', '1', and ',' may be used
Abort trap: 6

- Runtime exception for forgetting constexpr



Runtime Execution?

- Really handy for debugging
- In this case not so good for users
- Little reason for runtime conversion
- Which one causes a runtime error?

```
constexpr auto maskA =  
    constexpr11_bin<std::uint8_t>("1110 0000");  
  
auto maskB =  
    constexpr11_bin<std::uint8_t>("0001 1111");
```

- Every invocation has error potential



Guidance

Make interfaces easy to use correctly
and hard to use incorrectly.

Item 18 *Effective C++ Third Edition* by Scott Meyers

Prefer compile- and link-time errors to
run-time errors.

Item 14 *C++ Coding Standards* Sutter and Alexandrescu



A Way to Force Compile-Time Only?

- Not within the standard
- But a hack that sometimes works



Unresolved Symbol In Throw

```
extern const char* compile11_bin_invoked_at_runtime;
template <typename T = std::uint32_t>
constexpr T compile11_bin(
    constexpr_txt t,
    std::size_t i = 0,           // index
    std::size_t b = 0,           // bit count
    T x = 0)                     // accumulator
{
    return
        i >= t.size() ? x : // end recursion
        b >= std::numeric_limits<T>::digits ?
            throw std::overflow_error("Too many bits!") :
        t[i] == ',' ? compile11_bin<T>(t, i+1, b, x) :
        t[i] == '0' ? compile11_bin<T>(t, i+1, b+1, (x*2)+0) :
        t[i] == '1' ? compile11_bin<T>(t, i+1, b+1, (x*2)+1) :
        throw std::domain_error( // Only '0', '1', and ','
            compile11_bin_invoked_at_runtime);
}
```



User Errors at Link-Time

```
int main()
{
    auto mask = // <- Not constexpr!
        compile11_bin<std::uint8_t>("1110 0000");
    assert(mask == 0xE0);
    return 0;
}
```

Undefined symbols for architecture x86_64:

 "_compile11_bin_invoked_at_runtime", referenced from:

 unsigned char compile11_bin<unsigned char>(constexpr_txt,
unsigned long, unsigned long, unsigned char) in main11.o

ld: symbol(s) not found for architecture x86_64

clang: error: linker command failed with exit code 1 (use -v to
see invocation)

Why Does It Work?

The throw must not be evaluated at compile time

The throw must be included in a runtime implementation

The runtime implementation cannot link because of the unresolved extern



Is This The Best You Can Do?

- Error is ugly
- Doesn't identify line that caused error
- May not work:

...a function that is called in a potentially-evaluated constant expression is still odr-used, so the compiler is permitted to emit it...

Richard Smith

- Any technique failure is false positive



Technique Summary

- constexpr function must have a throw
- Declare unresolved extern const char*
- Reference unresolved extern in throw



Compile-Time Parsing Summary

- Interesting technique
- Limited applicability
- But still useful
- Consider unresolved extern hack



Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- Compile-time floating point
- Compile-time containers
- Future and Summary





Compile-Time Floating Point

Motivation

- constexpr supports floating point
- Why bother?
 - Errors caught at compile-time
 - Reduce executable footprint
 - Improve runtime execution speed
- Actual use cases?



Use Case: Biquad Coefficients

- Biquad is a common DSP filter
- Two parts:
 1. Coefficients
 - Determine filter response
 - Typically hard to compute
 - Constant for a given response
 2. Recursive computation
 - Provides filter output
 - Typically fast to compute
 - Executes once per sample



Buckle Up!

I wanted a **real** example

We need some infrastructure

Coefficient computing using constexpr



What We're Computing...

```
enum class BiquadType : unsigned char {  
    lowpass = 0,  
    highpass,  
    bandpass,  
    notch,  
    peak,  
    lowshelf,  
    highshelf  
};  
template <typename T>  
struct Coeffs {  
    T a0 = 0;  
    T a1 = 0;  
    T a2 = 0;  
    T b1 = 0;  
    T b2 = 0;  
};
```



BiquadCoeffs

```
template <typename T = double> // T <- storage type
class BiquadCoeffs
{
public:
    template <typename U>
    constexpr BiquadCoeffs(
        BiquadType type, U fc, U q, U peakGainDB)
        : type_(type)
        , coeffs_(computeCoeffs<T>(type, fc, q, peakGainDB))
        { }
private:
    template <typename U> friend class Biquad;
    constexpr T process(T in, T& z1, T& z2) const;
private:
    BiquadType type_;
    Coeffs<T> coeffs_;
};
```



computeCoeffs

```
template <typename T, typename U> // T <- storage type
constexpr Coeffs<T> computeCoeffs (
    BiquadType type, U fc, U q, U peakGainDB)
{
    using W = typename std::common_type<T, U>::type;
    if ((fc < 0) || (fc >= 0.5f )) {
        throw std::domain_error(
            "fc must be less than the Nyquist frequency.");
    }

    Coeffs<T> coeffs;
    W norm = 0;
    const W v = constexpr_pow<W>(
        10, constexpr_abs<W>(peakGainDB) / 20);
    const W k = constexpr_tan<W>(pi_num<W>>() * fc);
    switch (type) {
        ...
    }
}
```



Coefficients by Type

```
case BiquadType::lowpass:  
    norm = 1 / (1 + k / q + k * k);  
    coeffs.a0 = static_cast<T>(k * k * norm);  
    coeffs.a1 = static_cast<T>(2 * coeffs.a0);  
    coeffs.a2 = coeffs.a0;  
    coeffs.b1 = static_cast<T>(2 * (k * k - 1) * norm);  
    coeffs.b2 = static_cast<T>((1 - k / q + k * k) * norm);  
break;
```

```
case BiquadType::highpass:  
    norm = 1 / (1 + k / q + k * k);  
    coeffs.a0 = static_cast<T>(1 * norm);  
    coeffs.a1 = static_cast<T>(-2 * coeffs.a0);  
    coeffs.a2 = coeffs.a0;  
    coeffs.b1 = static_cast<T>(2 * (k * k - 1) * norm);  
    coeffs.b2 = static_cast<T>((1 - k / q + k * k) * norm);  
break;
```

...



Functions Created / Used

- T `pi_num`<T> ()
- T `constexpr_abs`<T> (U x)
- T `constexpr_pow`<T> (U base, V exp)
 - T `eulers_num`<T> ()
 - T `constexpr_log`<T> (U x)
 - T `constexpr_exp`<T> (U x)
 - pair<T> `constexpr_modf`<T> (U x)
- T `constexpr_tan`<T> (U x)
 - T `constexpr_sin`<T> (U x)
 - T `constexpr_cos`<T> (U x)

Using the Coefficients

```
template <typename T>
constexpr T
BiquadCoefFs<T>::process(T in, T& z1, T& z2) const
{
    T out = in * coeffs_.a0 + z1;
    z1 = in * coeffs_.a1 + z2 - coeffs_.b1 * out;
    z2 = in * coeffs_.a2 - coeffs_.b2 * out;
    return out;
}
```

- This is the runtime computation:
 - Adds
 - Subtracts
 - Multiplies



Biquad (not constexpr) Wraps It

```
template <typename T>
class Biquad {
public:
    explicit Biquad(BiquadCoeffs<T> const& coeffs)
        : coeffs_(coeffs)
        , z1_(0)
        , z2_(0) { }

    inline T process(T in) {
        return coeffs_.process(in, z1_, z2_);
    }

private:
    BiquadCoeffs<T> coeffs_;
    T z1_, z2_; // Hold changing recursion values
};
```



Using Biquad

```
void test_biquad()
{
    constexpr BiquadCoeffs<float> coeffs_l
        (BiquadType::lowpass, 0.05, 0.707, 0.0);
    Biquad<float> lowpass (coeffs_l);
    constexpr std::size_t ramp_size = 10;
    constexpr float ramp[ramp_size] {
        0.0f, 0.2f, 0.4f, 0.6f, 0.8f, 1.0f, 1.2f, 1.4f, 1.6f, 1.8f
    };
    constexpr int sample_count = 101;
    float samples[sample_count];
    std::size_t i = 0;
    for (int j = 0; j < sample_count; ++j) {
        if (i >= ramp_size) {
            i = 0;
        }
        samples[j] = lowpass.process(ramp[i++]);
    }
}
```



constexpr BiquadCoeffs Benefits

Declare constexpr BiquadCoeffs

Assembly file 144 lines

Forget constexpr on BiquadCoeffs

Assembly file 2017 lines

Lots of additional machine instructions

Constrained platforms take heed!

Consider the unresolved symbol hack



constexpr Float Timing

constexpr_pow() at compile-time

VS

std::pow()

VS

constexpr_pow() at runtime



Timing Code 1

```
constexpr int repetitions = 10'000'000;

void compile_time()
{
    using namespace std::chrono;
    constexpr double two = 2.0;
    volatile static double sink;
    const auto start_t = high_resolution_clock::now();
    for (int i = 0; i < repetitions; ++i) {
        constexpr double sqrt2 = constexpr_pow (two, 0.5);
        sink = sqrt2;
    }
    const auto end_t = high_resolution_clock::now();
    auto elapsed = end_t - start_t;
    std::cout << "constexpr_pow at compile time: "
        << duration_cast<nanoseconds>(elapsed).count()
        << " ns" << std::endl;
}
```



Timing Code 2

```
constexpr int repetitions = 10'000'000;

void run_time()
{
    using namespace std::chrono;
    volatile double two = 2.0;
    volatile static double sink;
    const auto start_t = high_resolution_clock::now();
    for (int i = 0; i < repetitions; ++i) {
        double sqrt2 = std::pow (two, 0.5);
        sink = sqrt2;
    }
    const auto end_t = high_resolution_clock::now();
    auto elapsed = end_t - start_t;
    std::cout << "std::pow: "
        << duration_cast<nanoseconds>(elapsed).count()
        << " ns" << std::endl;
}
```



Timing Code 3

```
constexpr int repetitions = 10'000'000;

void constexpr_run_time()
{
    using namespace std::chrono;
    volatile double two = 2.0;
    volatile static double sink;
    const auto start_t = high_resolution_clock::now();
    for (int i = 0; i < repetitions; ++i) {
        double sqrt2 = constexpr_pow (two, 0.5);
        sink = sqrt2;
    }
    const auto end_t = high_resolution_clock::now();
    auto elapsed = end_t - start_t;
    std::cout << "constexpr_pow at runtime: "
        << duration_cast<nanoseconds>(elapsed).count()
        << " ns" << std::endl;
}
```



Timing Results: Intel i5 2.4 GHz

clang++ -std=c++1y -O3

```
constexpr_pow at compile time: 3828350 ns
std::pow:                        78555220 ns
constexpr_pow at runtime:      1772557008 ns
```

Normalized...

```
constexpr_pow at compile time:    1
std::pow:                        20
constexpr_pow at runtime:        463
```



Why the Time Differences?

- Compile time has no runtime cost
- `pow()` uses intrinsics on Intel machine
- `constexpr_pow()` can't use intrinsics

Consider the unresolved symbol hack



Numeric Errors at Compile Time

```
void sqrt_neg2_compile_time ()
{
    constexpr double neg_sqrt2 = constexpr_pow (-2.0, 0.5);
    static_assert (neg_sqrt2 != 0, "Yipe!");
}
```

main.cpp:63:22: error: constexpr variable 'neg_sqrt2' must be initialized by a constant expression

```
constexpr double neg_sqrt2 = constexpr_pow (-2.0, 0.5);
                        ^~~~~~
```

./../constexpr_float/constexpr_log.h:21:9: note: subexpression not valid in a constant expression

```
    throw std::domain_error (
    ^
```

./../constexpr_float/constexpr_pow.h:28:23: note: in call to 'constexpr_log(-2.000000e+00)'

```
const W ln_base = constexpr_log<W> (base);
```



Numeric Errors at Runtime?

```
void sqrt_neg2_run_time ()  
{  
    const double neg_sqrt2 = std::pow (-2.0, 0.5);  
    assert (!std::isnan (neg_sqrt2));  
}
```

Assertion failed: (!std::isnan (neg_sqrt2)),
function sqrt_neg2_run_time, file main.cpp, line 73.
Abort trap: 6

std::pow() doesn't throw

- Sets errno — not allowed in constexpr
- Returns NaN



Floating Point Compiler Magic?

Could the compiler provide a `constexpr` value at compile time and an intrinsic at runtime?

Not without changing the interface of:

- functions that set `errno`
- functions that return a value through an output parameter



Math With Non-constexpr Interfaces

| | | |
|-------------------------|-----------------------|--------------------------|
| <code>sin(x)</code> | <code>cosh(x)</code> | <code>floor(x)</code> |
| <code>cos(x)</code> | <code>tanh(x)</code> | <code>fabs(x)</code> |
| <code>tan(x)</code> | <code>exp(x)</code> | <code>ldexp(x,n)</code> |
| <code>asin(x)</code> | <code>log(x)</code> | <code>frexp(x,*e)</code> |
| <code>acos(x)</code> | <code>log10(x)</code> | <code>mod(f,*ip)</code> |
| <code>atan(x)</code> | <code>pow(x,y)</code> | <code>fmod(x,y)</code> |
| <code>atan2(y,x)</code> | <code>sqrt(x)</code> | |
| <code>sinh(x)</code> | <code>ceil(x)</code> | |



No Peeking at Float Internals!

```
void float_bits_ptr ()
{
    static constexpr float f1 {1.0f};
    constexpr const unsigned char* raw =
        reinterpret_cast<const unsigned char*>(&f1);
}
```

error: constexpr variable 'raw' must be initialized by a
constant expression

```
constexpr const unsigned char* raw =
                                ^
```

note: reinterpret_cast is not allowed in a constant expression

```
    reinterpret_cast<const unsigned char*>(&f1);
    ^
```

1 error generated.



No Peeking at Float Internals!

```
void float_bits_union ()
{
    union f_to_i {
        float f32;
        unsigned char raw[sizeof (float)];
        constexpr f_to_i (float f) : f32(f) { }
    };
    constexpr f_to_i u (1.0f);
    static_assert (u.raw[0] != 0, "Yipe!");
}
```

error: static_assert expression is not an integral constant expression

```
static_assert (u.raw[0] != 0, "Yipe!");
               ^~~~~~
```

note: read of member 'raw' of union with active member 'f32' is not allowed in a constant expression

1 error generated.



Compile-Time Floating Point Summary

- constexpr floating point can...
 - Identify errors at compile time
 - Improve runtime speed
 - Reduce code footprint
 - Omit entire libraries from runtime
- C++14 makes it very natural
- Little standard library support so far
- Accidental use at runtime can be bad
 - Consider unresolved extern hack



Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- Compile-time floating point
- Compile-time containers
- Future and Summary



Compile-Time Containers

A black and white photograph of a long, ornate covered walkway, likely a shopping arcade or a historical passageway. The ceiling is a complex, vaulted structure made of glass and steel, with a grid-like pattern of ribs. The walkway is flanked by classical buildings with multiple stories, arched windows, and decorative architectural details. The perspective is looking down the length of the walkway, creating a strong sense of depth. The lighting is bright, with shadows cast on the ground and the buildings.

Motivation

- constexpr supports arrays
- Why bother?
 - Errors caught at compile-time
 - Improved code readability
 - Reduce executable footprint
 - Avoid global init order issues
 - Avoid initialization thread races
- Actual use cases?



First constexpr Array

```
void test_biquad()
{
    constexpr BiquadCoeffs<float> coeffs_l
        (BiquadType::lowpass, 0.05, 0.707, 0.0);
    Biquad<float> lowpass (coeffs_l);
    constexpr std::size_t ramp_size = 10;
    constexpr float ramp[ramp_size] {
        0.0f, 0.2f, 0.4f, 0.6f, 0.8f, 1.0f, 1.2f, 1.4f, 1.6f, 1.8f
    };
    constexpr int sample_count = 101;
    float samples[sample_count];
    std::size_t i = 0;
    for (int j = 0; j < sample_count; ++j) {
        if (i >= ramp_size) {
            i = 0;
        }
        samples[j] = lowpass.process(ramp[i++]);
    }
}
```



More Interesting Array

```
constexpr float pi = pi_num<float>();  
constexpr std::size_t sine_size = 10;  
constexpr float sine[sine_size] {  
    constexpr_sin(0.0f * 2 * pi),  
    constexpr_sin(0.1f * 2 * pi),  
    constexpr_sin(0.2f * 2 * pi),  
    constexpr_sin(0.3f * 2 * pi),  
    constexpr_sin(0.4f * 2 * pi),  
    constexpr_sin(0.5f * 2 * pi),  
    constexpr_sin(0.6f * 2 * pi),  
    constexpr_sin(0.7f * 2 * pi),  
    constexpr_sin(0.8f * 2 * pi),  
    constexpr_sin(0.9f * 2 * pi),  
};
```



Returning a constexpr Array

```
// Can't use C++14 std::array, since element access is not  
// defined as constexpr.
```

```
template <typename T, std::size_t N>  
class array_result {
```

```
private:
```

```
    constexpr static std::size_t size_ = N;
```

```
    T data_[N] {}; // T default constructor essential!
```

```
public:
```

```
    constexpr std::size_t size() const { return N; }
```

```
    constexpr T& operator[](std::size_t n)  
        { return data_[n]; }
```

```
    constexpr const T& operator[](std::size_t n) const  
        { return data_[n]; }
```

```
    using iterator = T*;
```

```
    constexpr iterator begin() { return &data_[0]; }
```

```
    constexpr iterator end()   { return &data_[N]; }
```

```
    // ...
```



Non-const constexpr Methods

```
template <typename T, std::size_t N>
class array_result {
    // ...
    constexpr T& operator[](std::size_t n)
        { return data_[n]; }
    constexpr const T& operator[](std::size_t n) const
        { return data_[n]; }
    using iterator = T*;
    constexpr iterator begin() { return &data_[0]; }
    constexpr iterator end()   { return &data_[N]; }
    // ...
}
```

- Non-const methods to modify array contents
- Use **before** array_result is returned



generate_sine

```
template <typename T, std::size_t N>
constexpr auto generate_sine() -> array_result<T, N>
{
    array_result<T, N> ret; // Init ret

    constexpr T pi = pi_num<T>();
    for (int i = 0; i < N; i += 1) {
        T f = i;
        ret[i] = constexpr_sin(f * 2 * pi / N); // Modify ret
    }
    return ret; // Return ret
}
```



Try generate_sine

```
constexpr float sine_lookup (std::size_t i)
{
    constexpr auto table =
        generate_sine<float, 256>();

    if (i >= table.size()) {
        throw std::out_of_range ("Index too big.");
    }
    return table[i];
}

void try_sine_lookup ()
{
    constexpr float f1 = sine_lookup (100); // Compile time
    const float f2 = sine_lookup (100);     // Run time
    assert (f1 == f2);
}
```



constexpr_to_upper

```
template <std::size_t N>
constexpr auto constexpr_to_upper (const char(&in)[N])
    -> array_result<char, N>
{
    array_result<char, N> out;

    for (std::size_t i = 0; in[i] != 0; ++i)
    {
        char c = in[i];
        out[i] = ((c < 'a') || (c > 'z')) ?
            c : c - ('a' - 'A');
    }
    return out;
}
```



Test constexpr_to_upper

```
void test_constexpr_to_upper()
{
    static constexpr auto a =
        constexpr_to_upper ("Frank Lloyd Wright");
    static_assert (a.size() == 19, "Yike!");
    static_assert (a[a.size() - 2] == 'T', "Yike!");
    std::cout << &a[0] << std::endl;
}
```

```
$ ./test_to_upper
FRANK LLOYD WRIGHT
$
```

```
.section __TEXT,__const
__ZZ23test_constexpr_to_uppervE1a:
.asciz "FRANK LLOYD WRIGHT"
```



So far we have...

- Computed array entries
- Generated an array
- Returned an array based on input

All at compile time

Let's validate that a constexpr array is alphabetical



constexpr_str_less

```
constexpr bool constexpr_str_less (  
    const char* lhs, const char* rhs)  
{  
    std::size_t i = 0;  
    do  
    {  
        if (lhs[i] != rhs[i])  
        {  
            return lhs[i] < rhs[i];  
        }  
    } while (lhs[i++] != '\0');  
    return false;  
}
```



constexpr_is_alphabetical

```
constexpr bool constexpr_is_alphabetical (  
    const char* const* capo, const char* const* fine)  
{  
    for (const char* const* i = capo + 1; i < fine; ++i)  
    {  
        if (constexpr_str_less(*i, *(i - 1)))  
        {  
            return false;  
        }  
    }  
    return true;  
}
```



Test constexpr_is_alphabetical

```
void test_is_alphabetical()
{
    constexpr std::size_t count = 4;
    constexpr const char* architects[count] =
    {
        "Daedalus",
        "Michelangelo",
        "Pei, I. M.",
        "Wright, Frank Lloyd"
    };

    static_assert (
        constexpr_is_alphabetical(
            &architects[0], &architects[count]),
        "architects not alphabetical!");
}
```



Let the Compiler Do the Sort

```
// Can't default cmp to std::less; it's not constexpr.
template <typename Itr, typename Cmp>
constexpr void constexpr_insertion_sort (Itr b, Itr e, Cmp cmp)
{
    static_assert (std::is_same <bool,
        decltype (cmp (*b, *e))>::value, "cmp not valid.");
    using T = typename std::iterator_traits<Itr>::value_type;
    if (b == e) return;
    for (Itr i = b + 1; i != e; ++i) {
        Itr j = i;
        const T temp = i[0];
        for (j = i;
            (j != b) && cmp(temp, j[-1]); --j) {
            j[0] = j[-1];
        }
        j[0] = temp;
    }
}
```

Hold It!

A constexpr function that returns void?

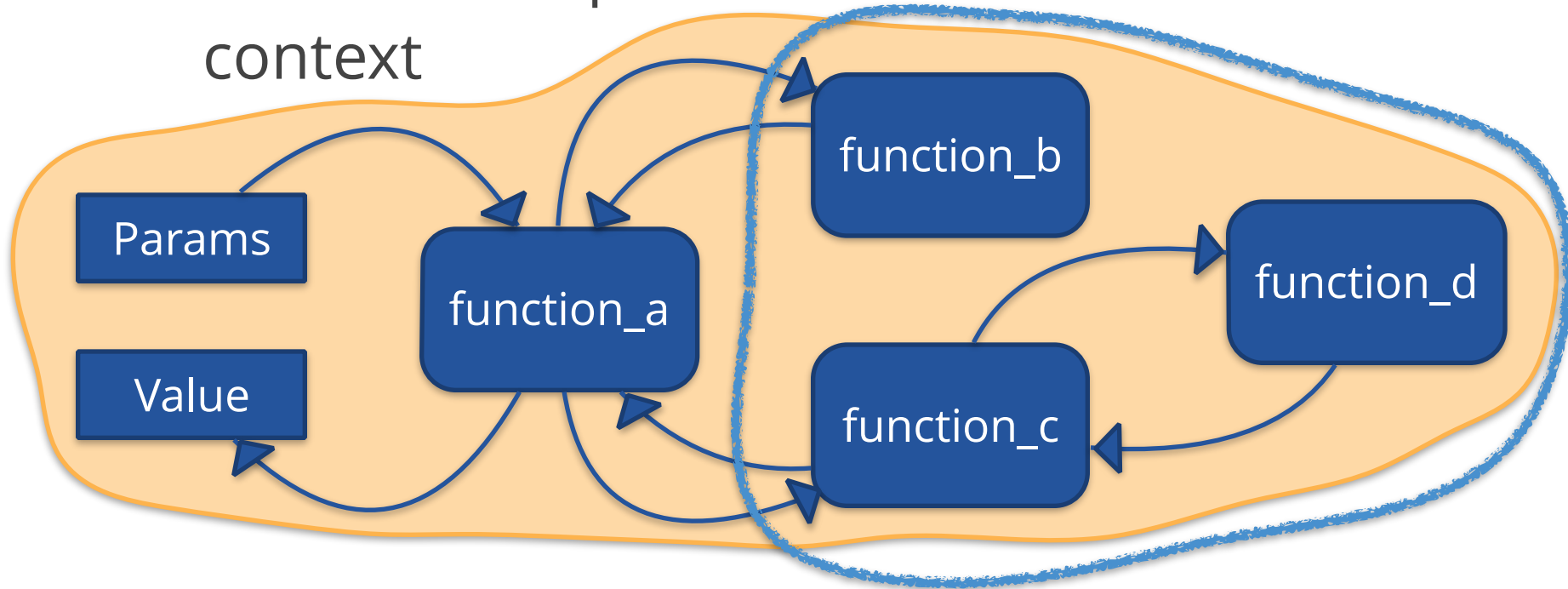
```
template <typename Itr, typename Cmp>  
constexpr void  
constexpr_insertion_sort (Itr b, Itr e, Cmp cmp)  
{  
    // ...
```

- Illegal in C++11
- Useful in C++14 if...
- Not the outermost constexpr function



C++14 constexpr Model

- C++ interpreter
- Each constexpr result has its entire context



- Inner calls can return void

constexpr_str_sort

```
template <std::size_t N>
constexpr auto constexpr_str_sort (
    const char* const(&in)[N])
    -> array_result<const char*, N>
{
    array_result<const char*, N> out;

    // Copy from in to out
    for (std::size_t i = 0; i < N; ++i) {
        out[i] = in[i];
    }

    // Sort
    constexpr_insertion_sort (
        out.data(), out.size(), constexpr_str_less);
    return out;
}
```



Test constexpr_str_sort

```
void test_str_sort()
{
    constexpr const char* architects[] =
        { "Pei", "Daedalus", "Wright", "Michelangelo" };

    constexpr auto a = constexpr_str_sort (architects);
    static_assert (a.size() == 4, "Yike!");

    for (std::size_t i = 0; i < a.size(); ++i) {
        std::cout << i << " : " << a[i] << std::endl;
    }
}
```

```
$ ./test_str_sort
0 : Daedalus
1 : Michelangelo
2 : Pei
3 : Wright
```



Validate or Sort?

Depends on your compile time

- Validation is $O(n)$
- A good sort is $O(n \log n)$

Is the convenience of compile-time sort worth the compile time?



Compile-Time Associative Containers

Consider replacing associative containers with sorted vectors.

Item 23 *Effective STL* by Scott Meyers

Um, or sorted arrays...

Anon



Add More Iterators to array_result<>

```
template <typename T, std::size_t N>
class array_result {

    // ...
    using const_iterator = const T*;
    constexpr const_iterator
    begin() const { return &data_[0]; }

    constexpr const_iterator
    cbegin() const { return &data_[0]; }

    constexpr const_iterator
    end() const { return &data_[N]; }

    constexpr const_iterator
    cend() const { return &data_[N]; }

};
```



std::equal_range() With array_result<>

```
void test_str_sort_find()
{
    constexpr const char* architects[] =
        { "Pei", "Daedalus", "Wright", "Michelangelo" };
    constexpr auto a = constexpr_str_sort (architects);
    static_assert (a.size() == 4, "Yike!");

    auto range = std::equal_range(
        a.begin(), a.end(), "Pei", constexpr_str_less);

    assert ((range.first + 1) == range.second); // Found Pei
    assert (range.first - a.begin() == 2);      // Index 2
    assert (strcmp (*range.first, "Pei") == 0); // It's Pei
}
```



constexpr Unordered Containers?

- A much bigger stretch
 - Potential hash collisions require an unknown number of buckets
- But values known at compile time
- Consider a minimal perfect hash



hash_and_str

```
// Can't use std::pair because the assignment
// operator is not constexpr.
struct hash_and_str
{
    std::uint32_t hash;
    const char* str;

    static constexpr bool
    less (const hash_and_str& lhs, const hash_and_str& rhs) {
        return lhs.hash < rhs.hash;
    }
};
```



Fill In the Hash Container

```
using hash_fn_t = std::uint32_t (*)(const char*);

template <std::size_t N>
constexpr auto constexpr_str_hash (
    const char* const(&in)[N], hash_fn_t hash_fn)
    -> array_result<hash_and_str, N>
{
    array_result<hash_and_str, N> out;

    // Copy from in to out.  Compute hash.
    for (std::size_t i = 0; i < N; ++i) {
        out[i].hash = hash_fn (in[i]);
        out[i].str  = in[i];
    }
    // ...
}
```



Sort and Verify the Hash

```
// ...
// Sort. Can't use a lambda. No lambdas in constexpr.
constexpr_insertion_sort (
    out.begin(), out.end(), hash_and_str::less);

// Verify hash is perfect and minimal.
for (std::size_t i = 0; i < N; ++i) {
    if (out[i].hash != i) {
        throw std::invalid_argument(
            "Hash is not minimal and perfect.");
    }
}
return out;
}
```



Perfect Minimal Hash Function

```
// Minimal perfect hash found manually.  
constexpr std::uint32_t architects_hash (const char* in)  
{  
    std::uint32_t h = 2166136261;  
    for (std::size_t i = 0; in[i] != '\0'; ++i)  
    {  
        h = (h * 16777619) ^ in[i];  
    }  
    h = h & 0x03;  
    return h;  
}
```

Adapted from http://www.etrnallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx



Testing constexpr_str_hash

```
void test_str_hash()
{
    constexpr const char* architects[] =
        { "Pei", "Daedalus", "Wright", "Michelangelo" };

    constexpr auto a =
        constexpr_str_hash (architects, architects_hash);

    for (std::size_t i= 0; i < a.size(); ++i) {
        std::cout << a[i].hash
                    << " : " << a[i].str << std::endl;
    }
}

$ ./test_str_hash
0 : Daedalus
1 : Pei
2 : Wright
3 : Michelangelo
```



constexpr Container Downsides

- Container is immutable
- Call site must know container size
 - Usually from template parameter
- May not contain `std::string`
 - `std::string` requires allocator



constexpr std::string?

- I looked at it
- It could almost, but not quite, work
- If you're curious we should talk



Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- Compile-time floating point
- Compile-time containers
- Future and Summary



Future and Summary



Future

Expect to see lots more constexpr in the standard library

There's work on constexpr lambdas



Summary

Good for:

- Errors caught at compile-time
- Improved code readability
- Reduce executable footprint
- Avoid global init order issues
- Avoid initialization thread races

Accidental use of constexpr code at runtime
can slow execution

- Consider unresolved extern hack

C++ at compile time?

Absolutely!



Thanks and Acknowledgements

- Gabriel Dos Reis: constexpr C++11
- Richard Smith: constexpr C++14
- Scott Meyers: great references
- Howard Hinnant: unresolved external
- Scott Determan: slide review
- cppreference.com: constexpr_txt
<http://en.cppreference.com/w/cpp/language/constexpr>
- Nigel Redmon: Biquad code
<http://www.earlevel.com/main/2012/11/26/biquad-c-source-code/>
- All errors belong to Scott Schurr



Questions?



Thanks for attending