constexpr: Introduction

By Scott Schurr for Ripple Labs at CppCon September 2015

constexpr: Introduction

By Scott Schurr for Ripple Labs at CppCon September 2015

Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- 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
- 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

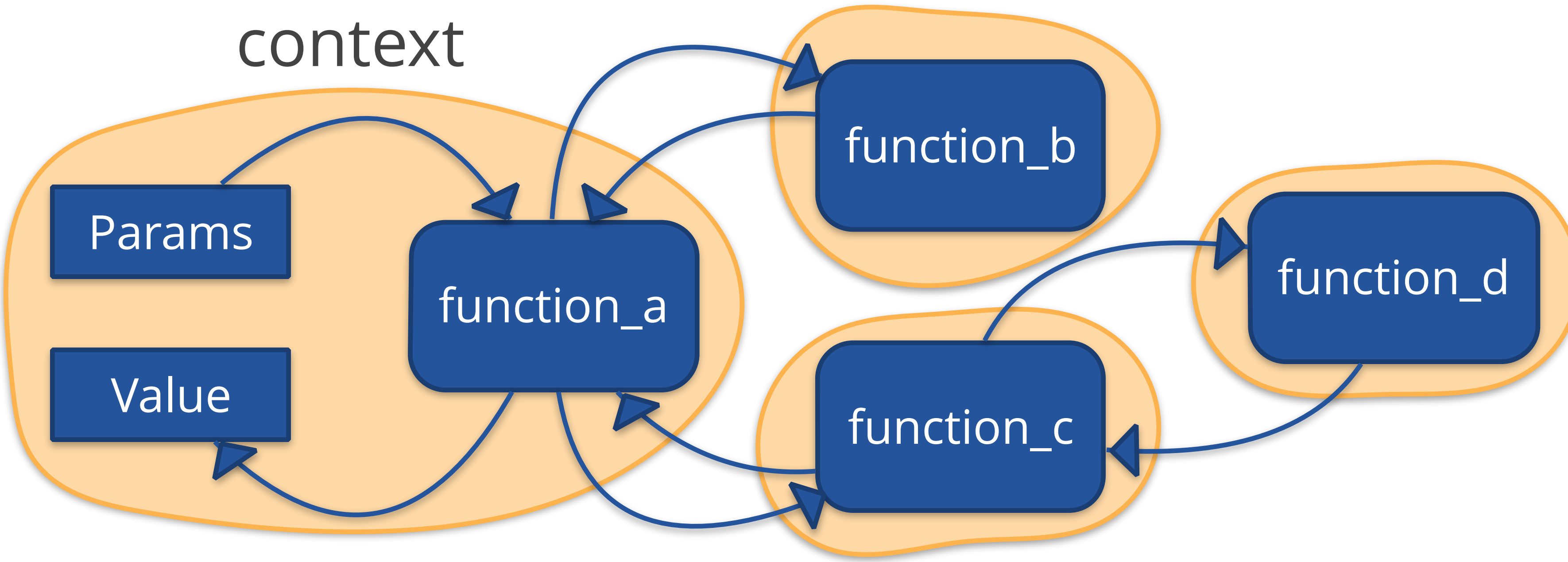
```
int main()
{
    static_assert (               // Compute at compile time
        constexpr_pow_int_cpp11(2.0,  0) ==  1.0,     "Yipe!");
    static_assert (
        constexpr_pow_int_cpp11(2.0,  5) == 32.0,     "Yipe!");
    static_assert (
        constexpr_pow_int_cpp11(2.0, -5) == 0.03125, "Yipe!");

    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
- 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
- type_id 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

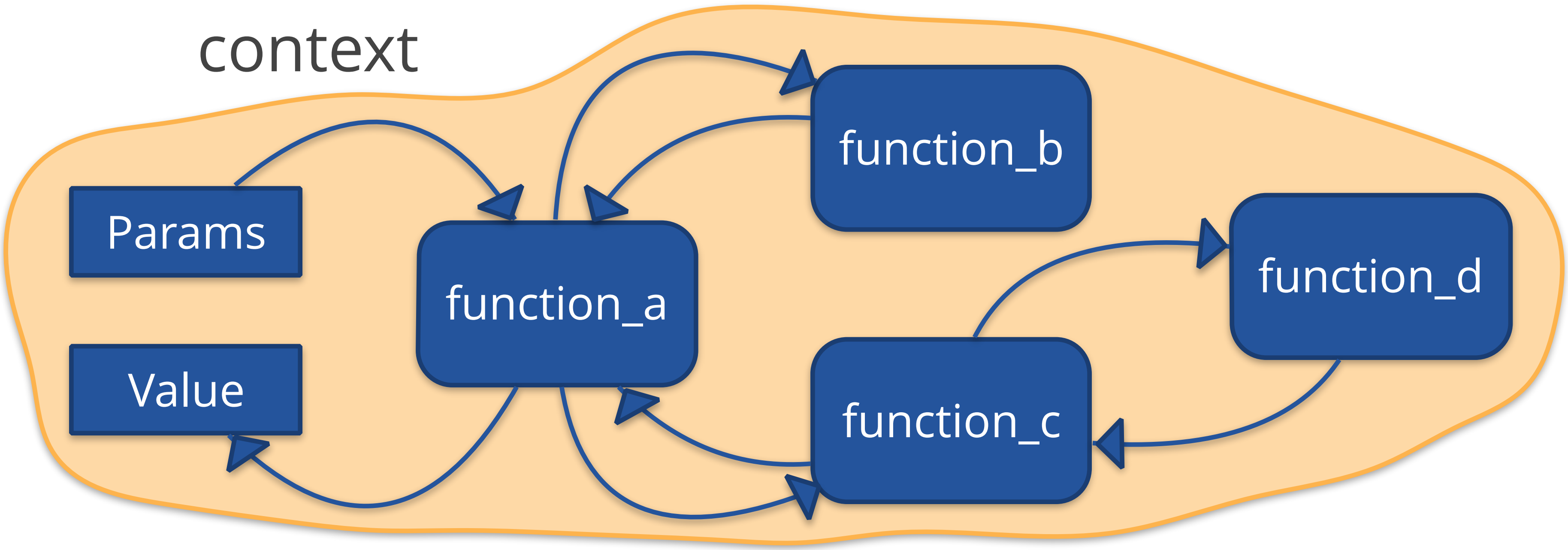
```
int main()
{
    static_assert (              // Compute at compile time
        constexpr_pow_int_cpp14(2.0,  0) ==  1.0,     "Yipe!");
    static_assert (
        constexpr_pow_int_cpp14(2.0,  5) == 32.0,     "Yipe!");
    static_assert (
        constexpr_pow_int_cpp14(2.0, -5) == 0.03125, "Yipe!");

    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

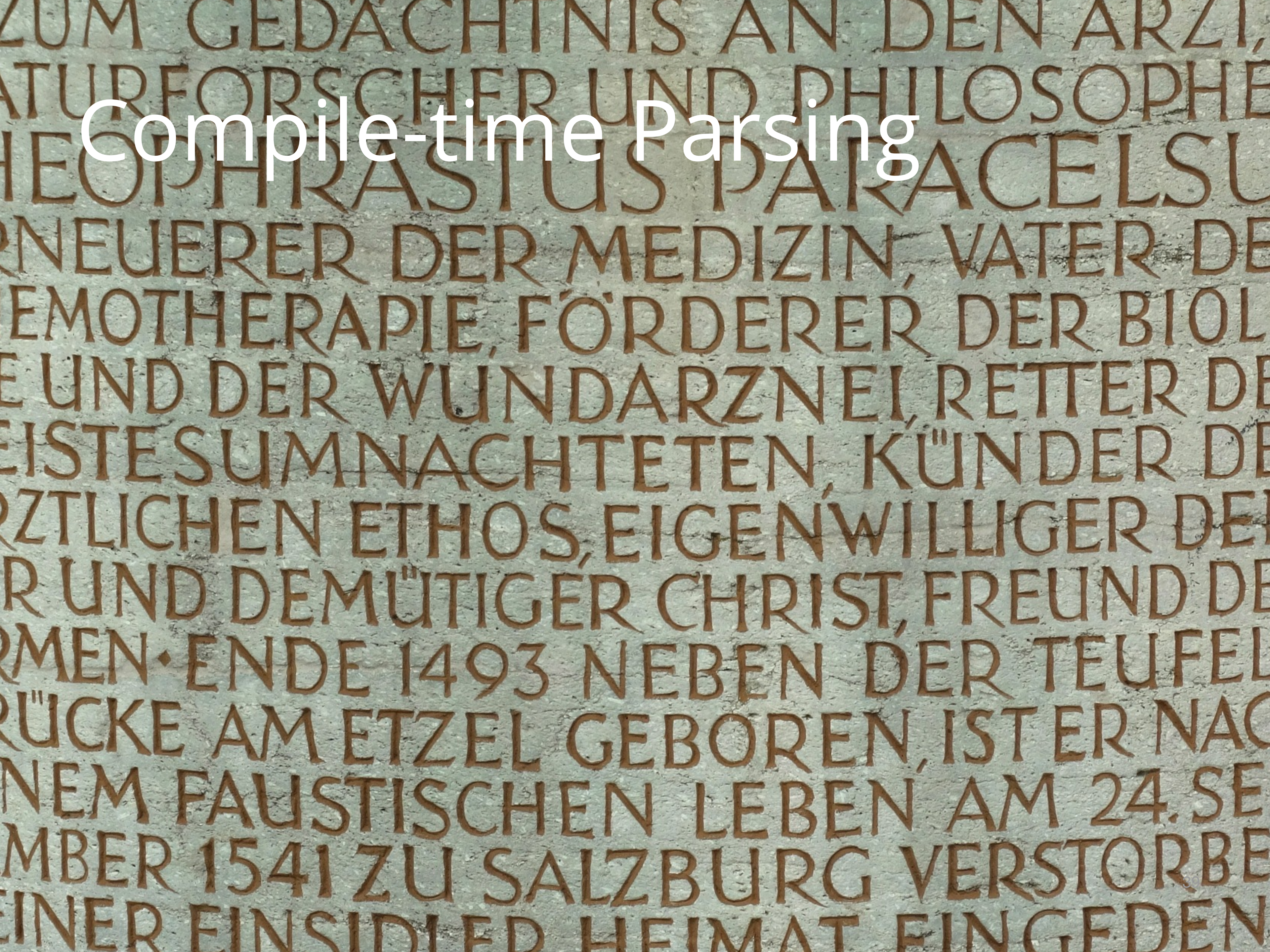


Think like a C++ programmer without new

Topics

- constexpr beginning
- constexpr in C++11
- constexpr in C++14
- Compile-time parsing
- Summary

Compile-time Parsing



C++11 binary literal

- No native binary literal in C++11
- Can we make one?

constexpr11_bin

```
template <typename T = std::uint32_t>
constexpr T constexpr11_bin(
    const char* t,
    std::size_t b = 0,       // bit count
    T x = 0)                 // accumulator
{
    return
        *t == '\0' ? x :     // end recursion
        b >= std::numeric_limits<T>::digits ?
            throw std::overflow_error("Too many bits!") :
        *t == ',' ? constexpr11_bin<T>(t+1, b,    x) :
        *t == '0' ? constexpr11_bin<T>(t+1, b+1, (x*2)+0) :
        *t == '1' ? constexpr11_bin<T>(t+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;
}
```

constexpr_bin Weakness?

- If evaluated at runtime, subject to buffer overrun attacks
- A way to fix that?

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_; }
};
```

Adapted from http://en.cppreference.com/w/cpp/language/constexpr class conststr

Applying constexpr_txt

```
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");
}
```

More constexpr Questions

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

constexpr Limits

Minimum recommended implementation quantities [Annex B]:

- Recursive constexpr function invocations: 512
- Full-expressions evaluated within a core constant expression: 1,048,576

Actual limits are up to your compiler(s)

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
- Summary

Summary

Summary

New keyword: constexpr

- Introduced with C++11
- Improved with C++14

constexpr…

- Values are computed at compile time
- Code allowed to run at compile time
- Uses include parsing
- Lots of other uses. See the Applications talk

Accidental use of constexpr code at runtime can be bad

- Consider unresolved extern hack

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

- All errors belong to Scott Schurr

Questions?

Thanks for attending