constexpr for compile-time computation in modern C++

Banin, M.; Kochanova, K.; Skrebkov, A.; Vodopyanov, D.

January 18, 2018

Constexpr

Motivation

- Template metaprogramming is hard to write and slow
- Macros and other code generation methods are their own can of worms
- Embedded developers want to do as much as possible during compilation
- Can express data that has regular structure, like S-boxes in AES cypher
- Diminish need to be careful with multi-thread intialization
- Feeling relieved that due to not having to learn template metaprogramming

Kinds of constexpr

- constexpr values:
  - Definition of an object
  - Declaration of a static data member of literal type
- constexpr computations:
  - Functions
  - Constructors

constexpr vairables

On their own, are like static consts

```
constexpr double approx_pi = 3.141;

constexpr char reply[] = "Yep.";
static_assert(name[3] == '.', "That's unexpected.");
```

- constexpr auto works
  - just auto won't pick constexprness from initialization

constexpr computation

- Free functions
- Member functions
- Constructors
- Allowed code:
  - Constrained in C++11, e.g. single statement, can't retrun void, etc.
  - Relaxed somewhat in C++14
  - share code with run-time version of the function
    - allows for unit-testing and debugging
    - can be called in static asserts
  - Amount of execution is at least 1M full expression evalutations, way more than templates
  - Definitely no dynamic memory, e.g. std::vector

constexpr std::array generation

C++11 would require re-implementing std::array with constexpr in right places, use C++14.
Actually, C++14 doesn't seem to have constexpred the copy and constructor for it either, so C++17.

```
constexpr size_t SZ = 10;
constexpr std::array<int, SZ> fibs(){
  ::std::array<int, SZ> ret = {};
  unsigned int a = 1, b = 1;
  for(size_t i = 0; i < SZ; ++i){
    ret[i] = a;
    unsigned int c = a + b; a = b; b = c;
  }
  return ret;
}

constexpr ::std::array<int,SZ> arr(fibs());
...
```

more constexpr code

```
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;
}
```

constexpr in an object declaration

- Member functions and variables have the same requirements as usual functions and variables. But there is a note:
  - constexpr member function implicitly obtains a const qualifier (C++11)
- Constexpr constructor have the same requirements as member functions except the one about return and the additional one.
  - Every base class and every non-static member must be initialized, either in the constructor's initialization list or by a member brace-or-equal initializer.

Is constexpr function const?

- C++11: The two declarations declare two function overloads: a const and a non-const one.
  - Output: -1 1
- C++14: The two declarations will define the same non-const member function with two different return types: this will cause a compilation error.
- Suggestion: Add const explicitly to be compatible

```
#include <iostream>

struct Number {

  int i;

  constexpr const int& get() /*const*/ { return i; }

  int& get() { return --i; }

};

int main() {

  Number n1{0}; std::cout << n1.get() << “ “;

  const Number n2{0}; std::cout << n2.get() << std::endl;

  return 0;

}
```

constexpr in an object declaration. C++11

- Member functions and variables have the same requirements as for usual functions and variable. But there is a note:
  - constexpr member function implicitly obtains a const qualifier
- Constexpr constructor must satisfy the following requirements:
  - Each of its parameters must be LiteralType
  - Constructor must not have a function-try-block
  - Every base class and every non-static member must be initialized
  - Constructor must contain only:
    - Null-statements
    - Static_assert declarations
    - Using declarations and directives

C++17 brings us

- constexpr if
- constexpr lambdas

Constexpr lambda

```
template <typename I>
constexpr auto adder(I i) {
  //use a lambda in constexpr context
  return [i](auto j){ return i + j; };
}

//constexpr closure object
constexpr auto add5 = adder(5);

template <unsigned N>
class X{};

int foo() {
  //use in a constant expression
  X<add5(22)> x27;

}
```

if constexpr

```
template<class L, class R>
auto fold(L l, R r) {
  using lTag = typename L::tag;
  using rTag = typename R::tag;
  if constexpr (is_base_of<rTag, BarTag>::value) {
    if constexpr (is_same<lTag, FooTag>::value) {
      return foldFB(l, r);
    } else {
      return foldBB(l, r);
    }
  } else {
    return foldFF();
  }
}
```

References

- N3690
- Scott Shurr's excellent talks at CppCon 2015
- `https://stackoverflow.com/questions/14116003/difference-between-constexpr-and-const`
- `http://cpptruths.blogspot.com/2011/07/want-speed-use-constexpr-meta.html`