CONSTEXPR SPECIFIER

constexpr Specifier

by Prasanna Ghali

Plan for Today

- Understanding `constexpr`

Why `constexpr`?

- Confusing new word since C++11
- When applied to objects, `constexpr` is beefed up version of `const`
- Different meaning when applied to functions
- Important to know because:
  - Computations can be enabled during compilation
  - Conditional compilation of code is simplified

From const to constexpr (1/3)

- Prior to C++11, const machinery was restricted to two things:
  - Qualifying a type as const, and thus any instance of that type is immutable
  - Qualifying a nonstatic member function so that *this is const in its body

```
class int_wrapper {
public:
  explicit int_wrapper(int);
  void mutate(int);
  int inspect() const;
private:
  int mi;
};
```

```
const int_wrapper iwc{7};
iwc.mutate(5);            // error
int i = iwc.inspect(); // ok
```

From `const` to `constexpr` (2/3)

- Values known during compilation are privileged especially *integral constant expressions*
  - Array sizes, integral template arguments, lengths of `std::array` objects, enumerator values, alignment specifiers, …
  - Mathematical constants …
- `constexpr` object is like `const` object that has values known at compile time

From const to constexpr (3/3)


```
int sz{}; // non-constexpr variable

// error: sz's value not known at compilation
constexpr auto arr_sz1 = sz; // ok???

// error: same problem
std::array<int, sz> a1; // ok???

// fine, 10 is a compile-time constant
constexpr auto arr_sz2 = 10; // ok???

// fine, arr_sz2 is constexpr
std::array<int, arr_sz2> a2; // ok???
```

Difference Between const and constexpr (1/2)

- const doesn't offer same guarantee as constexpr
  - const objects need not be initialized with values known during compilation

```
int sz; // non-constexpr variable

// fine: arr_sz is const copy of sz
const auto arr_sz = sz; // ok???

// error: arr_sz's value not known at compilation
std::array<int, arr_sz> data; // ok???
```

Difference Between `const` and `constexpr` (2/2)

- Simply put, all `constexpr` objects are `const`, but not all `const` objects are `constexpr`
- If you want compilers to guarantee that a variable has a value that can be used in contexts requiring compile-time constants, use `constexpr`, not `const`!!!

Header-Only Libraries: Inlined Variables (1/4)

- C++17 introduced `inline` variables to allow for header-only libraries with variable definitions in header file
  - ODR not invoked when header file is included by many source files
  - Instead, all source files including that header file will have same address for `inline` variable

```
// possibly defined in multiple header files
inline long double pi{3.141'592'653'589'793'238'462'643'383'279L};

// in source file that includes a header file shown above
long double circ_area(long double const& r) {
  return pi*r*r;
}
```

Header-Only Libraries: Inlined Variables (2/4)

- constexpr [and const] objects can be defined in header files
  - By default, such objects have static or internal linkage

```
// possibly defined in multiple header files
constexpr
long double pi{3.141'592'653'589'793'238'462'643'383'279L};

// in source file that includes a header file shown above
long double circ_area(long double r) {
  return pi*r*r;
}
```

Header-Only Libraries: Inlined Variables (3/4)

- If you require address of constant to be same everywhere, you mark it as `inline`

```
// possibly defined in multiple header files
inline constexpr
long double pi{3.141'592'653'589'793'238'462'643'383'279L};

// in source file that includes a header file shown above
long double circ_area(long double const& r) {
  return pi*r*r;
}
```

Header-Only Libraries: Inlined Variables (4/4)

- C++17 allows static data members to be defined and initialized in class

```
struct Counter {
  // static data member is now defined and initialized
  // in-class without the need to provide definition in
  // source file
  static inline int counter = 0;

  Counter()  { ++counter; }
  ~Counter() { --counter; }
};
```

Variable Templates (1/5)

- Since C++14, variables can be parameterized by specific type

```
// can be defined in a header file
template <typename T>
constexpr T pi{3.141'592'653'589'793'238'462'643'383'279L};

template <class T>
T circ_area(T const& r) {
  return pi<T>*r*r;
}

std::cout << pi<long double> << '\n';
std::cout << pi<double>      << '\n';
std::cout << pi<float>       << '\n';
```

Variable Templates (2/5)

- Variables templates can also have default template arguments

```
template <typename T = long double>
constexpr T pi = T{3.141'592'653'589'793’238L};

std::cout << pi<> << '\n'; // outputs a long double
std::cout << pi<float>  << '\n'; // outputs a float
```

Variable Templates (3/5)

- Variables templates can also be parameterized by nontype parameters

```cpp
// array with N elements, zero-initialized
template <int N> std::array<int,N> arr{};

// nontype parameter is used to parameterize initializer
template <auto N> constexpr decltype(N) dval{N};

std::cout << dval<'c'> << '\n'; // N has value 'c'
std::cout << dval<42>  << '\n'; // N has value 42

arr<5>[0] = 42;  // set first element of global object arr
arr<51>[0] = 42; // will this compile?
```

Variable Templates (4/5)

- Useful application of variable templates is to define variables that represent members of class templates

```
// given definition of class C
template <typename T>
class C {
public:
  static constexpr int max{100};
};

// you can define variable template c_max:
template <typename T> int c_max = C<T>::max;

// so that you can define different values for
// different specialization of C<>:
auto sc = c_max<std::string>; // instead of C<string>::max
```

Variable Templates (5/5)


```
// better example ...
// for standard class such as
namespace std {
  template <typename T> class numeric_limits {
  public:
    ...
    static constexpr bool is_signed = false;
    ...
  };
}

// you can define variable template
template <typename T>
constexpr bool is_signed = std::numeric_limits<T>::is_signed;

// to be able to write expression
is_signed<char>
// rather than lengthier expression
std::numeric_limits<char>::is_signed
```

`constexpr` Functions

- Functions that produce compile-time constants *when they're called with compile-time constants*
  - `constexpr` functions can be used in contexts that demand compile-time constants
  - Acts like normal function computing its result at runtime when called with one or more values that are not known during compilation

`constexpr` Functions: Example (1/2)

`constexpr` in front of `fibonacci` doesn't say that `fibonacci` returns a `const` value, it says that if `n` is compile-time constant, `fibonacci`'s result may be used as compile-time constant.
If `n` is not compile-time constant, `fibonacci`'s result will be computed at runtime.

```
constexpr long fibonacci(long n) noexcept {
  return n <= 2 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}
```

`constexpr` Functions: Example (2/2)


```
template <typename T>
constexpr T square(T x) noexcept {
  return x*x;
}
```

```
constexpr int pow(int base, int exp) noexcept {
  int result{1};
  for (int i{}; i < exp; ++i) result *= base;
  return result;
}
```

constexpr Functions: Compile-Time Contexts

- constexpr functions can be used in places with compile-time contexts

```
constexpr
int pow(int base, int exp) noexcept {
  int result{1};
  for (int i{}; i < exp; ++i) result *= base;
  return result;
}

// 5 conditions each with 3 possible states
constexpr int conds {5}, states{3};
std::array<int, pow(states, conds)> results;
```

`constexpr` Functions: Examples (1/3)

- Another example of `constexpr` functions used in places with compile-time contexts

```
template <typename T1, typename T2>
constexpr
auto Max(T1 a, T2 b) -> decltype(b<a?a:b) {
  return b < a ? a : b;
}

int ai[Max(sizeof(int), 10L)] {1,2,3};

std::array<std::string, Max(sizeof(int), 8L)>
                                        as{"a","b","c"};
```

`constexpr` Functions: Examples (2/3)

- Another example …

```
constexpr bool is_prime(uint64_t p) {
  for (uint64_t d{2}; d <= p/2; ++d) {
    // found divisor without remainder
    if (p % d == 0) return false;
  }
  // no divisor without remainder found
  return p > 1;
}

bool found =
is_prime(std::numeric_limits<uint64_t>::max());
```

`constexpr` Functions: Examples (3/3)

Compiles with g++ but not with clang++!!!

```
constexpr long floor_sqrt(long n) {
  return floor(sqrt(n));
}
```

`constexpr` Functions for User-Defined Types (1/7)

- `constexpr` functions are limited to taking and returning literal types [types that can have values determined during compilation]
  - All built-in types except `void` qualify
- User-defined types can be literal too …

constexpr Functions for User-Defined Types (2/7)


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}

  // other stuff ...
};
```

`constexpr` Functions for User-Defined Types (3/7)


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}

  // other stuff ...
};

// compiler will run constexpr ctor
constexpr Point p1{9.4, 27.7};
constexpr Point p2{28.8, 5.3};
```

constexpr Functions for User-Defined Types (4/7)


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}
  constexpr double X() const noexcept { return x; }
  constexpr double Y() const noexcept { return y; }

  // other stuff ...
};
```

`constexpr` Functions for User-Defined Types (5/7)


```
constexpr Point p1{9.4, 27.7};
constexpr Point p2{28.8, 5.3};
constexpr Point mid = midpt(p1, p2);
```

```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}
  constexpr double X() const noexcept { return x; }
  constexpr double Y() const noexcept { return y; }

  // other stuff ...
};

constexpr
Point midpt(Point const& p1, Point const& p2) noexcept {
  return { (p1.X()+p2.X())/2.0, (p1.Y()+p2.Y())/2.0 };
}
```

constexpr Functions for User-Defined Types (6/7)


```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}
  constexpr double X() const noexcept { return x; }
  constexpr double Y() const noexcept { return y; }

  constexpr void X(double dx) noexcept { x = dx; }
  constexpr void Y(double dy) noexcept { y = dy; }
};
```

constexpr Functions for User-Defined Types (7/7)


```
constexpr Point p1{9.4, 27.7};
constexpr Point p2{28.8, 5.3};
constexpr Point mid = midpt(p1, p2);
constexpr Point rmid = reflection(mid);
```

```
class Point {
  double x, y;
public:
  constexpr
  Point(double dx=0.0, double dy=0.0) noexcept
  : x{dx}, y{dy} {}
  constexpr double X() const noexcept { return x; }
  constexpr double Y() const noexcept { return y; }

  constexpr void X(double dx) noexcept { x = dx; }
  constexpr void Y(double dy) noexcept { x = dy; }
};
```

```
constexpr Point reflection(Point const& p) noexcept {
  Point result;
  result.X(-p.X());
  result.Y(-p.Y());
  return result;
}
```

constexpr if Statement

- *constexpr if* statement allows template functions to evaluate different scopes in same function at compile time

constexpr if Statement: Motivation

- We'd like compile-time polymorphism:

```
struct Dog {
  auto woof() const { std::cout << "Woof!!!\n"; }
};

struct Cat {
  auto meow() const { std::cout << "Meow!!!\n"; }
};

template <typename Pet>
auto noise(Pet const& pet) {
  if (std::is_same<Pet, Dog>::value) {
    pet.woof();
  } else if (std::is_same<Pet, Cat>::value) {
    pet.meow();
  }
}
```

```
int main() {
  Dog d;
  Cat c;
  // error ...
  noise(d);
  noise(c);
}
```

constexpr if Statement: Motivation


```cpp
struct Dog {
  auto woof() const { std::cout << "Woof!!!\n"; }
};
struct Cat {
  auto meow() const { std::cout << "Meow!!!\n"; }
};

// solution using SFINAE ...
template <typename Pet>
std::enable_if_t<std::is_same_v<Pet, Dog>>
noise(Pet const& pet) {
  pet.woof();
}

template <typename Pet>
std::enable_if_t<std::is_same_v<Pet, Cat>>
noise(Pet const& pet) {
  pet.meow();
}
```

```cpp
int main() {
  Dog d;
  Cat c;

  noise(d); // ok
  noise(c); // ok
}
```

What is SFINAE?


```
template <typename T, size_t N>
std::size_t len(T(&)[N]) { return N; }

template <typename T>
typename T::size_type len(T const& t) { return t.size(); }

std::size_t len(...) { return 0; }

int ai[10];
std::cout << len(ai) << '\n';
std::cout << len("hello") << '\n';
std::vector<int> vi(5);
std::cout << len(vi) << '\n';
double *pd;
std::cout << len(pd) << '\n';
std::allocator<int> mi;
std::cout << len(mi) << '\n';
```

→ noname elipses — accept all.
ensure no error
- just to be check

Cloning std::enable_if_t<>


```
template <bool B, typename T=void>
struct my_enable_if {};

template <typename T>
struct my_enable_if<true, T> { using type = T; };

template <bool B, typename T=void>
using my_enable_if_t = typename my_enable_if<B,T>::type;

template <typename Pet>
my_enable_if_t<std::is_same_v<Pet, Dog>> noise(Pet const& pet) {
  pet.woof();
}

template <typename Pet>
my_enable_if_t<std::is_same_v<Pet, Cat>> noise(Pet const& pet) {
  pet.meow();
}
```

```
int main() {
  Dog d;
  Cat c;
  noise(d);
  noise(c);
}
```

constexpr if Statement: Motivation

- We'd like compile-time polymorphism:

```
struct Dog {
  auto woof() const { std::cout << "Woof!!!\n"; }
};

struct Cat {
  auto meow() const { std::cout << "Meow!!!\n"; }
};

template <typename Pet>
auto noise(Pet const& pet) {
  if constexpr(std::is_same<Pet,Dog>::value) {
    pet.woof();
  } else if (std::is_same<Pet,Cat>::value) {
    pet.meow();
  }
}
```

```
int main() {
  Dog d;
  Cat c;

  noise(d); // ok
  noise(c); // ok
}
```

constexpr if Statement: Example


```
template <typename T>
T sum(T const& t) { return t; }

template <typename T, typename ...Types>
T sum(T const& t, Types const& ...params) {
  return t + sum(params...);
}
```

```
template <typename T, typename ...Types>
T sum(T const& t, Types const& ...params) {
  if constexpr (sizeof...(params) == 0)
    return t;
  else
    return t + sum(params...);
}
```

constexpr if Statement: Example


```
template <typename T>
void print(T const& t) {
  std::cout << t << '\n';
}

template <typename T, typename ...Types>
void print(T const& t, Types const& ...params) {
  print(t);
  print(params...);
}
```

```
template <typename T, typename ...Types>
void print(T const& t, Types const& ...params) {
  std::cout << t << '\n';
  if constexpr(sizeof...(params) > 0) {
    // code only available if sizeof...(args)>0
    print(params...);
  }
}
```

constexpr if Statement: Motivation

- Implementing a polymorphic adder

Variadic Class Template: Example

- Same approach as with variadic function templates
- Use recursion pattern with class template specializations