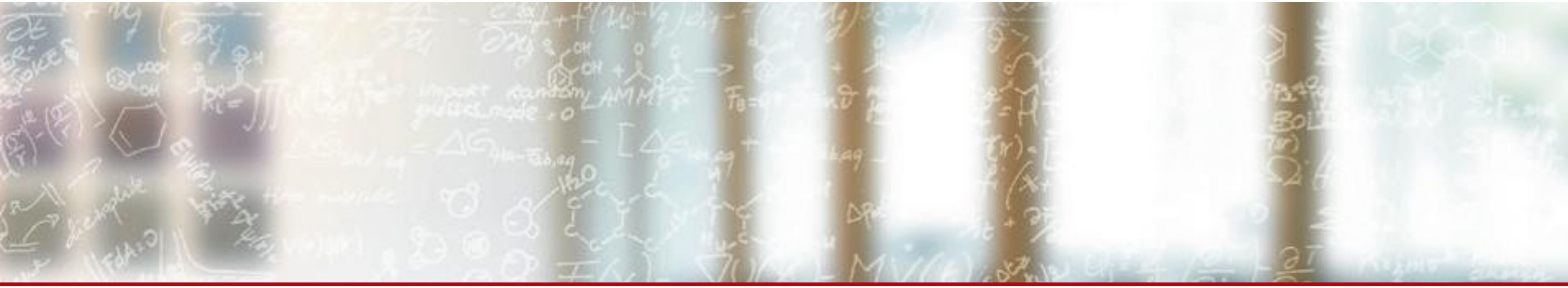

constexpr (and also consteval and constinit)

Anton Afanasyev, CSCS (afanasyev@cscs.ch)
October, 2021

Resources:

Definition

The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time. Such variables and functions can then be used where only compile time constant expressions are allowed (provided that appropriate function arguments are given).

Disclaimer: if constexpr is a different thing!

Definition

The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time. Such variables and functions can then be used where only compile time constant expressions are allowed (provided that appropriate function arguments are given).

Where to use constexpr functions/variables?

- In explicit template instantiations, template specializations and default template parameters.
- In array type declarations.
- In static asserts.

Where to use constexpr functions/variables?

- In explicit template instantiations, template specializations and default template parameters.
- In array type declarations.
- In static asserts.

Are there any other use cases?

Where to use constexpr functions/variables?

- In explicit template instantiations, template specializations and default template parameters.
- In array type declarations.
- In static asserts.

Are there any other use cases? **No**

Dummy examples

```
constexpr int a = 9;
constexpr int foo(int x) { return x * x; }

template <int = a> struct Foo;
template <> struct Foo<foo(3)> {};
Foo<a> x;

int arr[foo(a)];

static_assert(foo(a) == 81);
```

Less dummy example

```cpp
constexpr size_t default_alignment = 128;

constexpr size_t do_expand(size_t n, size_t size, size_t alignment) {
   size_t unit = std::lcm(size, alignment) / size;
   return  (n + unit - 1) / unit * unit;
}

template <class T, size_t Alignment> struct expand { using type = T; };

template <class T, size_t N, size_t Alignment>
struct expand<T[N], Alignment> {
   using type = T[do_expand(N, sizeof(T), Alignment)];
};

template <class T, size_t Alignment = default_alignment>
using expand_t = typename expand<T, Alignment>::type;

template <class T, size_t Outer, size_t Inner, size_t Alignment>
struct expand<T[Outer][Inner], Alignment> {
   using type = expand_t<T[Inner], Alignment>[Outer];
};

static_assert(std::is_same_v<expand_t<float[33][33][33]>, float[33][33][64]>);
static_assert(std::is_same_v<expand_t<double[33][33][33]>, double[33][33][48]>);
```

Invoking constexpr function in runtime

```
auto& trace(std::source_location const &loc = std::source_location::current()) {
   static int count = 0;
   return std::cout
       << "\n#" << count++
       << "\t["<< loc.file_name() << ":" << loc.line() << ":" << loc.column() << "] "
       << "In function \'" << loc.function_name() << "\': ";
}

constexpr int foo(int val) {
   if (!std::is_constant_evaluated())  trace() << "going to return " << val;
   return val;
}

int main() {
   int x[foo(10)];
   static_assert(std::size(x) == 10);
   auto y = foo(34);
   assert(y == 34);
}
```

Output: `#0 [/app/example.cpp:15:46] In function 'constexpr int foo(int)': going to return 34`

consteval

```
auto& trace(std::source_location const &loc = std::source_location::current()) {
   static int count = 0;
   return std::cout
       << "\n#" << count++
       << "\t["<< loc.file_name() << ":" << loc.line() << ":" << loc.column() << "] "
       << "In function \'" << loc.function_name() << "\': ";
}

consteval int foo(int val) {
   if (!std::is_constant_evaluated())  trace() << "going to return " << val;
   return val;
}

int main() {
   int x[foo(10)];
   static_assert(std::size(x) == 10);
   auto y = foo(34);
   assert(y == 34);
}
```

Output:

consteval

```
consteval int foo(int val) {
   if (!std::is_constant_evaluated())  trace() << "going to return " << val;
   return val;
}

int main() {
   int x[foo(10)];
   static_assert(std::size(x) == 10);
   auto y = foo(34);
   assert(y == 34);
   auto tmp = 88;
   auto z = foo(tmp);  // error: the value of 'tmp' is not usable in a constant expression
   assert(z == 88);
}
```

consteval

```
constexpr int foo(int val) {
   if (!std::is_constant_evaluated())  trace() << "going to return " << val;
   return val;
}

consteval auto as_consteval(auto val) { return val; }

int main() {
   int x[foo(10)];
   static_assert(std::size(x) == 10);
   auto y = as_consteval(foo(34));
   assert(y == 34);
   auto tmp = 88;
   auto z = foo(tmp);
   assert(z == 88);
}
```

Output: `#0 [/app/example.cpp:15:46] In function 'constexpr int foo(int)': going to return 88`

constinit

constinit - asserts that a variable has static initialization, i.e. zero initialization and constant initialization, otherwise the program is ill-formed.

The only usage I aware about is to fight against static initialization order fiasco.

puzzle

https://godbolt.org/z/nz86j41Gx

Thank you for your attention.