Templates & Concepts I; constexpr; C++17/20 Library Additions

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

autumn 2020

Templates

Motivation

Templates

Motivation

- generic programming; polymorphism
- metaprogramming; compile-time programming

Templates

Motivation

- generic programming; polymorphism
- metaprogramming; compile-time programming

Syntax

```
template <typename T, typename U = bool, size_t N = 10>
template <std::integral T, std::copyable U>
```

- `template < parameters >`
- parameters can be:
  - types: `typename T` or `class T`
  - concepts (constrained types): `std::integral T`, `std::input_iterator T` – C20
  - values: integers, enums, (pointers, references): `int a`, `auto v`
  - templates: more about this later
- parameters can have default values

Concepts – Motivation

- templates alone do not allow to specify what the type must provide

Concepts – Motivation

- templates alone do not allow to specify what the type must provide
  - movable `T` for `std::vector<T>`
  - comparable `K` for `std::map<K, V>`
  - iterator and predicate arguments for `std::copy_if`

Concepts – Motivation

- templates alone do not allow to specify what the type must provide
  - movable `T` for `std::vector<T>`
  - comparable `K` for `std::map<K, V>`
  - iterator and predicate arguments for `std::copy_if`
- requirements checked implicitly on instantiation

```
In file included from .../bits/stl_tree.h:65,
                 from .../map:60,
                 from map.cpp:1:
.../bits/stl_function.h: In instantiation of // ... 4 lines
map.cpp:7:23:   required from here
.../stl_function.h:386:20:
    error: no match for ‘operator<’
    (operand types are ‘const A’ and ‘const A’)
  386 |       { return __x < __y; }
      |                ~~~~^~~~~
```

Concepts I C++20

C++20 extends templates with *concepts* that allow constraining types

```
#include <concepts>

template<std::totally_ordered T>
const T &min(const T &a, const T &b) {
    return (a < b) ? a : b;
}
```

- `min` cannot be instantiated with non-comparable types
  - error refers to the use and interface, not implementation of `min`

Concepts II C++20

- make requirements about operations supported by the type explicit

```
template< typename T, std::predicate<T> Fun >
bool any(std::vector<T>, Fun f);
```

Concepts II C++20

- make requirements about operations supported by the type explicit

```
template< typename T, std::predicate<T> Fun >
bool any(std::vector<T>, Fun f);
```

- cleaner error messages that do not reveal implementation details

Concepts II C++20

- make requirements about operations supported by the type explicit

```
template< typename T, std::predicate<T> Fun >
bool any(std::vector<T>, Fun f);
```

- cleaner error messages that do not reveal implementation details
- simplify overloading

```
template< std::signed_integral T >
T my_abs(T x) { return x < 0 ? -x : x; }

template< std::unsigned_integral T >
T my_abs(T x) { return x; }
```

Concepts Syntax C++20

there are several ways to use concepts in a definition

- constrained template parameters:

```
template< std::integral T > void foo(T x);
```

Concepts Syntax C++20

there are several ways to use concepts in a definition

- constrained template parameters:

```
template< std::integral T > void foo(T x);
```

- `requires` clause after template parameter list:

```
template< typename T > requires std::integral<T>
void foo(T x);
```

Concepts Syntax C++20

there are several ways to use concepts in a definition

- constrained template parameters:

```
template< std::integral T > void foo(T x);
```

- `requires` clause after template parameter list:

```
template< typename T > requires std::integral<T>
void foo(T x);
```

- trailing `requires`:

```
template< typename T >
void foo(T x) requires std::integral<T>;
```

Concepts Syntax C++20

there are several ways to use concepts in a definition

- constrained template parameters:
  ```
  template< std::integral T > void foo(T x);
  ```
- `requires` clause after template parameter list:
  ```
  template< typename T > requires std::integral<T>
  void foo(T x);
  ```
- trailing `requires`:
  ```
  template< typename T >
  void foo(T x) requires std::integral<T>;
  ```
- abbreviated function template declaration:
  ```
  void foo(std::integral auto x);
  ```
  - `auto` used to make it possible to distinguish templates syntactically

Requires Concept Syntax C++20

- general, most verbose, can use multiple constraints for the same type

```
template< typename T >
requires std::integral<T> && (sizeof(T) > 4)
void foo(T x);
```

Requires Concept Syntax C++20

- general, most verbose, can use multiple constraints for the same type

```
template< typename T >
requires std::integral<T> && (sizeof(T) > 4)
void foo(T x);
```

- can use **requires**-expression (details later)
  - should be used to define named concepts, not recommended for direct use in functions/classes

```
template< typename T1, typename T2 >
requires requires(T1 a, T2 b) { a + b; }
auto add(T1 a, T2 b);
```

(yes, there is **requires** twice)

Constrained Template Parameters Syntax C++20

- instead of `typename`/`class` uses concept name without the first template parameter
  - the name of the type parameter is then used as the first argument for the constraint
  - `template< std::copyable T >` ≡
    `template< typename T > requires std::copyable<T>`
  - `template< std::derived_from<A> T >` ≡
    `template< typename T > requires std::derived_from<T, A>`

Abbreviated Template Syntax & Constrained auto C++20

- define templates without the `template` keyword
  - `void foo(auto x)`, `void bar(std::predicate<int> auto pr)`
  - like `auto`-lambdas
- define constrained `auto` variables
  - like with `auto` – type is derived, but must conform to the concept
  - `std::predicate<int> auto pr = [&](int x) { /*...*/ };`
- also for automatically-derived return types

Defining Concepts I C20

```
#include <concepts>

template< typename T >
concept large_int = std::integral<T> && sizeof(T) >= 4;
```

- concept definition is a variable-like template (with keyword `concept` in place of type)
- compile-time-evaluated boolean expressions, their conjunction or disjunction
- template cannot be constrained

Defining Concepts II C20

```
#include <concepts>

template< typename T >
concept addable = requires(const T &cref, T &ref) {
    { cref + cref } -> std::convertible_to< T >;
    { ref += cref } -> std::convertible_to< T & >;
};

static_assert( addable< int > );
static_assert( addable< float > );
```

Defining Concepts II C++20

```
#include <concepts>

template< typename T >
concept addable = requires(const T &cref, T &ref) {
    { cref + cref } -> std::convertible_to< T >;
    { ref += cref } -> std::convertible_to< T & >;
};

static_assert( addable< int > );
static_assert( addable< float > );
```

- `requires` clause introduces variable names: "let cref be a constant reference to T..."
- the concept of the expression's result is optional

A Note on Iterators and Concepts C++20

- C++ 20 introduces new way of working with iterators – ranges
  - to be presented later
  - some functionality (especially algorithms) is now duplicated
- concepts for iterators reflect the ranges

```
template< std::forward_iterator It,
          std::sentinel_for<It> End >
auto from_iterators( It from, End to )
```

  - sentinel is a possibly distinct type that represents end of the sequence
  - ranges-style iterator need not be comparable to itself, only to its sentinel
  - for conventional ($<$ C++20) iterators, `It` $\equiv$ `End`

More Templates

What can be templated?

More Templates

What can be templated?

- functions
- classes
- type aliases (`using`) – C++11
- variables – C++14

```
template<typename T>
const T pi = T(3.1415926535897932385);

int main() {
    std::cout.precision(17);
    std::cout << pi<double> << '\n';
    std::cout << pi<float> << '\n';
    std::cout << pi<int> << '\n';
}
```

More Templates

Template Instantiation

More Templates

Template Instantiation

- templates instantiated at compile time
  - templated functions/classes declared in header files should be also defined there
- methods of templated classes only instantiated if actually used (why?)
- the previous point is not completely true; why?

```
template <typename T> struct X {
    T t;
    void f() { t.f(); }
    void g() { t.g(); }
};
struct A { void f() {} };
int main() {
    X<int> xi;
    X<A>   xa;
    xa.f();
}
```

More Templates

Template Instantiation

- templates instantiated at compile time
  - templated functions/classes declared in header files should be also defined there
- methods of templated classes only instantiated if actually used (why?)
- the previous point is not completely true; why? *virtual methods*

```
template <typename T> struct X {
    T t;
    void f() { t.f(); }
    void g() { t.g(); }
};
struct A { void f() {} };
int main() {
    X<int> xi;
    X<A>   xa;
    xa.f();
}
```

Template Deduction

- until C++14 only for templated functions
- in C++17 also for class initialization
- template arguments may be deduced from the usage

```
template <typename To, typename From>
To lexical_cast(const From& val) {
    std::stringstream str;
    str << val;
    To result;
    str >> result;
    return result;
}
int main() {
    double d = lexical_cast<double, int>(17);
    int x = lexical_cast<int>("17");
    std::string s = lexical_cast(10); // ERROR: why?
}
```

Template Deduction

```
template <typename T> void foo1(T t);  // value parameter
template <typename T> void foo2(T& t); // reference parameter
```

Value parameters

- arrays and functions decay into pointers

```
int a[7];
foo1(a); // T is deduced to be int*
```

Reference parameters

- no decay

```
int a[7];
foo2(a); // T is deduced to be int[7]
// works as if foo2 was declared as
// void foo(int (&t) [7]); // reference to array
```

Template Deduction

```
template <typename T> void foo1(T t);
template <typename T> void foo2(T& t);
template <typename T> void foo3(const T& t);
```

- what is T deduced as?

```
int x;
const int answer = 42;
foo1(x);
foo1(answer);
foo2(x);
foo2(answer);
foo3(x);
foo3(answer);
```

Template Deduction

```
template <typename T> void foo1(T t);
template <typename T> void foo2(T& t);
template <typename T> void foo3(const T& t);
```

- what is T deduced as?

```
int x;
const int answer = 42;
foo1(x);
foo1(answer);
foo2(x);
foo2(answer);
foo3(x);
foo3(answer);
```

- `int`, `int`, `int`, `const int`, `int`, `int`

Class Template Deduction C++17

- in C++17, we can finally write things like these:

```
std::pair my_pair{ 1, "hello" };
std::vector some_ints{ 1, 2, 7, 42, 17, -1, 0 };
```

  - we do not need `std::make_pair` etc. anymore
- this also works with user-defined classes

```
template <typename T>
class Wrapper {
    T value;
public:
    Wrapper(T val) : value(std::move(val)) {}
};

Wrapper x = 3; // x is Wrapper<int>
```

- *note:* does not work with `unique_ptr`

Class Template Deduction: Deduction Guides C++17

- sometimes the compiler might be unable to deduce what we want
- sometimes we may want a different deduction behaviour
- we can change the deduction behaviour using *deduction guides*

```
template<typename Iterator>
MyContainer(Iterator, Iterator)
    -> MyContainer<typename Iterator::value_type>;

// also non-templated ones
Wrapper(const char*) -> Wrapper<std::string>;

Wrapper w{ "hello" }; // w is now Wrapper<std::string>
```

- see http://en.cppreference.com/w/cpp/language/class_template_argument_deduction

Compile-Time Computations

- templates let the compiler create a function/class/variable for given type at compile time
- sometimes, it is useful to do more computation at compile time
  - templates are actually Turing complete, but not very practical for computation

Compile-Time Computations

- templates let the compiler create a function/class/variable for given type at compile time
- sometimes, it is useful to do more computation at compile time
  - templates are actually Turing complete, but not very practical for computation
- `constexpr` is used to define functions and variables that can be used at compile-time and for compile-time `if`

```
template< typename T >
void construct_range( T *from, T *to ) {
  if constexpr ( !std::is_trivially_constructible_v< T > )
    for ( ; from != to; ++from )
      new ( from ) T();
}
```

constexpr

- functions that can be evaluated at compile-time
  - single-statement functions in C++11
  - more general in C++14 (control flow, no allocation)
  - even more general `constexpr` functions in C++20 (allocations, constructors, destructors)
- `constexpr if` in C++17

```
template< typename T >
constexpr T fill( T x ) {
    T out = x;
    for ( ; x > 0; x >> 1 )
        out |= x;
    return out;
}

Foo< fill(42) > x; // instantiates Foo< 63 >
```

consteval C++20

- `constexpr` functions can be evaluated both at compile-time and at run-time
  - there is no guarantee they will be evaluated at compile-time if they are in the context that does not require it (e.g., as template argument)
- we might want a function that *has to* be evaluated ad compile-time
- this can be done using `consteval` C++20

```
consteval int square(int n) { return n * n; }
constexpr int x = square(10);
consteval int foo(int n) { return square(n) * 4; } // OK

// ERROR - n not constant:
int bar(int n) { return square(n) * 4; }
```

constinit C++20

- specifies that *a variable* needs to be statically initialized
  - that is, its value needs to be initialized at compile-time
- can be used to ensure we don't have problems with dynamic initialization in the cases where it can be avoided

C++17 Library Additions

`std::optional`

- motivation:
  - return either a value or an information that there is none
  - no good default value for a type (or too expensive)
  - other languages: `Maybe` (Haskell), nullable types (C#), ...
- old solutions:

C++17 Library Additions

`std::optional`

- motivation:
  - return either a value or an information that there is none
  - no good default value for a type (or too expensive)
  - other languages: `Maybe` (Haskell), nullable types (C#), ...
- old solutions:
  - `std::pair<T, bool>`
  - `std::unique_ptr<T>` (needs heap allocation)
- `std::optional<T>` allocates space for a value of type `T` locally (on stack if local variable)
- pointer-like access (operators `*`, `->`); `std::nullopt`

C++17 Library Additions

std::variant

- motivation:
  - type-safe **union**
  - other languages: `Either` (Haskell), Rust enums, ...

```
std::variant<int, std::string> v = 42; // v contains int
std::get<int>(v);              // returns 42
std::get<0>(v);                // also works
std::get<std::string>(v); // throws
v.index();                     // returns 0
std::get_if<int>(&v);          // returns pointer to int
std::get_if<std::string>(&v); // returns nullptr
```

C++17 Library Additions

std::variant – visitor access

```
struct Visitor {
    void operator()(int val) const {
        std::cout << "got int: " << val << '\n';
    }
    void operator()(std::string val) const {
        std::cout << "got string: \"" << val << "\"\n";
    }
};

int main() {
    std::variant<int, std::string> v{ "hello" };
    std::visit(Visitor, v);
}
```

C++17 Library Additions

std::string_view

- motivation:
  - lightweight (non-owning) string wrapper with reference semantics
  - eliminate temporary `std::string` (needs allocation)
  - take substrings in a cheap way (without allocation)

```
void do_something(const std::string& str) {
    // do something read-only with str
}

do_something("what is this?");

void do_something_with_subst(const std::string& str) {
    auto substr = str.substr(1, 7);
    // this always creates a copy
    // even if I only want to read the substring
}
```

C++17 Library Additions

`std::string_view`

- older solutions:

C++17 Library Additions

`std::string_view`

- older solutions:
  - `std::pair<const char*, size_t>`
- `std::string_view` does exactly that!
  - with a nicer interface
- can be created from `std::string`, `const char*`
  - libraries can add their own conversions (`QString`)
- how to use:
  - always pass by value
  - make sure that the original string is still alive (think of `std::string_view` as a *reference*)
  - do read-only string operations
  - take substrings using `substr`

C++20 Library Additions

- not everything is implemented, we will now only present features available in GCC 10 or clang 10
- some parts will be presented later (e.g., ranges on the next lecture)

Partial Application of Functions – `std::bind_front`

```
int add(int a, int b) { return a + b };
auto add5 = std::bind_front(&add, 5);
```

- partially applies first arguments of given callable
- creates a callable object that holds arguments of `bind_front` *by value*

C++20 Library Additions

std::span

- a slice of an contiguous sequence (array, vector)
- reference-like object – does not own the elements
- static or dynamic size (`std::span<int>`, `std::span<int, 16>`)
- indexing, slicing, conversion to bytes

```
std::vector<int> vec = {0, 1, 2, 3, 4, 5};
std::span<int> sp(vec);
auto sub1 = sp.subspan<2, 2>();
auto sub2 = sp.subspan(2, 2);
```

C++20 Library Additions

Miscellaneous

- `starts_with`, `ends_with` for `std::string` and `std::string_view`
- various mathematical constant in `<numbers>`
- bit operations `<bit>` (bit counting, rotation, `bit_cast`)
- `std::shift_left` and `std::shift_right` algorithms
- `contains` member function of associative containers (`std::set`, `std::map`, ...)
- `std::erase` overloads for `vector`
- lot more `constexpr` (algorithm, utility, string, ...)