Templates

Templates

- A template is a generic declaration with compile-time formal arguments of these kinds:
  - any type

```
template< typename T>          // also template< class T>
```

    - [C++20] the type may be constrained using *Concepts*
  - a list of any types (in a *variadic template*)

```
template< typename ... TL>
```

  - integral number (the actual argument must be a compile-time constant)

```
template< std::size_t N>
```

  - a list of integral numbers (in a *variadic template*)

```
template< int ... NL>
```

  - another class template (with the given template argument list)

```
template< template< typename, std::size_t> class T>
```

  - a pointer (the actual argument must be an address of a static variable/function)
    - almost never used, functors work better

```
template< const char * p, inf (*fp)(int,int)>
```

- A function/lambda with **auto** arguments is also a template

```
void f(auto x);                 template< typename T> void f(T x);
```

Templates

- Template
  - a generic piece of code
  - parameterized by types, class templates, and integer constants

- Class templates
  - Global classes
  - Classes nested in other classes, including class templates

```
template< typename T, std::size_t N>
class array { /*...*/ };
```

- Function templates
  - Global functions
  - Member functions, including constructors

```
template< typename T>
inline T max( T x, T y) { /*...*/ }
```

- Type templates [C++11]

```
template< typename T>
using array3 = std::array< T, 3>;
```

- Variable templates [C++14]
  - Global variables and static data members
    - **inline** needed when declared in .hpp

```
template< typename T>
inline factory_class<T> factory;
```

  - Mostly used as global “constants” acting as compile-time functions on types

```
template< typename T>
inline constexpr std::size_t my_sizeof = sizeof(T);    // usage: my_sizeof<double>
```

Templates

- Template instantiation
  - Using the template with particular type and constant parameters
  - Class, type, and variable templates: parameters specified explicitly

```
std::array< int, 10> x;
```

    - [C++17] Deduction guides: Class template parameters may be deducted from initialization

```
std::pair p = { "pi", 3.14 };
```

  - Function templates: parameters specified explicitly or implicitly
    - Implicitly - derived by compiler from the types of value arguments

```
int a, b, c;

a = max( b, c);    // calls max< int>
```

    - Explicitly

```
a = max< double>( b, 3.14);
```

    - Mixed: Some (initial) arguments explicitly, the rest implicitly

```
std::array< int, 5> v;

x = std::get< 3>( v);       // calls std::get< 3, std::array< int, 5>>
```

Templates and compilation

- Compilers [may] check template code when defined
  - Without the knowledge of template arguments
  - Syntactic hints from the author may be required
- Compilers generate code only when templates are instantiated
- Different instantiations do not share code
  - It sometimes causes unwanted code size explosion
- There is no run-time support required for templates
  - Code generated for templates is identical to non-templated equivalents
- There is **no performance penalty** for generic code
  - Except that generic programming may encourage programmers to be lazy

Writing templates

- Compiler needs hints from the programmer
  - **Dependent names** have unknown meaning/contents

```
template< typename T> class X
{
```

- type names must be explicitly designated

```
  using U = typename T::B;
  typename U::D p;                          // U is also a dependent name
  using Q = typename Y<T>::C;
  void f() { T::D(); }                      // T::D is not a type
```

- explicit template instantiations must be explicitly designated

```
  bool g() { return 0 < T::template h<int>(); }
  int j() { return p.template k<3>(); }     // type of p is a dependent name
}
```

- members inherited from dependent classes must be explicitly designated

```
template< typename T> class X : public T
{
  const int K = T::B + 1;                   // B is not directly visible although inherited
  void f() { return this->a; }              // a is not directly visible
}
```

Passing template function arguments by value/reference

- How the arguments are passed in a call to a generic function?

```
std::string a1; std::string & a2 = a1; const std::string & a3 = a1;
f(a1); f(a2); f(a3); f(std::move(a1));
```

  - It depends on the declaration of the function
    - Presence of reference type in the declaration of a2/a3 does NOT matter

```
template< typename T> void f( T x);
```

  - In this case, x is always passed by value

  - If you want pass by reference, always use a "forwarding reference"

```
template< typename T> void f( T && x);
```

    - "Reference collapsing rules" ensure adaptation to both lvalues and rvalues

  - Rationale: plain references cannot adapt to some actual arguments:

```
template< typename T> void f( T & x);
```

    - In these cases, the function cannot be called with an rvalue argument:

```
f(a1+".txt"); f(std::move(a1));
std::vector<bool> k(100); f(k[0]);   // k[0] is an object simulating bool&
```

Templates and compilation

- Implicit template instantiation
  - When a class template specialization is referenced in context that requires a complete object type, or…
  - … when a function template specialization is referenced in context that requires a function definition to exist…
  - … the template is **instantiated** (the code for it is actually compiled)
    - unless the template was already explicitly specialized or explicitly instantiated
    - at link time, identical instantiations generated by different translation units are merged
- Instantiation of template member functions
  - Instantiation of a class template doesn't instantiate any of its member functions unless they are also used
- The definition of a template must be visible at the point of implicit instantiation

Templates and compilation

- The definition of a template must be visible at its instantiation
  - Classes and types
    - The visibility of definition is required also in the non-template case
    - Most class and type definitions must reside in header files
  - Function templates
    - (including non-template member functions of class templates etc.)
    - The template visibility rule is equivalent to rules for *inline* functions
      - Non-inline function templates are almost unusable
    - Most **function template definitions must reside in header files** (and be **inline**)
      - If a declaration only is visible, the compiler will not complain - it will generate a call but not the code of the function called – the linker will complain
    - In rare cases, the visibility rule may be silenced by ***explicit instantiation***:
      - Applicable only if all required argument combinations may be enumerated

```
template int max<int>(int, int);      // forces instantiation of max<int>

template double max(double, double); // forces instantiation of max<double>

template class X<int>;      // forces instantiation of all member functions of X<int>
```

Templates

- Multiple templates with the same name
  - Class templates:
    - one "master" template

```
template< typename T> class vector {/*...*/};
```

    - any number of specializations which override the master template
      - partial specialization

```
template< typename T, std::size_t n> class unique_ptr< T[n]> {/*...*/};
```

      - explicit specialization

```
template<> class vector< bool> {/*...*/};
```

  - Function templates:
    - any number of templates with the same name
    - shared with non-templated functions
  - Type and variable templates, concepts
    - only one definition (which may refer to different specializations of a class template)

Requirements and concepts

Validity of templates

- Templates are checked for validity
  - On definition: syntactic correctness, correctness of independent names
    - Not required by the language specification but supported by rules
  - On instantiation: All rules of the language

  - A template does not have to be correct for all combinations of arguments
    - It would be impossible in most cases
    - Compilers check the correctness only for the arguments used in an instantiation
      - Templates are difficult to test
    - Before C++20, there was no mechanism to specify requirements on template arguments
      - Trial-and-error approach (see SFINAE for advanced misuse)
      - Unreadable error messages when a template is incorrectly used
    - C++20 introduces ***requires*** *clauses* and ***concepts*** for constraining template arguments
      - They also assist in template function overload resolution (like SFINAE, unlike static_assert)

  - Instantiation of a class template does not invoke instantiation of all members
    - A valid class template instance may contain invalid member functions
      - Example: copy-constructor of vector<unique_ptr<T>>

[C++20] Requires clauses

- A **requires-clause** acts as a constraint on template parameters
  - Evaluated by the compiler in the moment of template instantiation

```
template< typename IT, typename F>
requires std::is_invocable_v<F, typename std::iterator_traits<IT>::reference>
F for_each( IT a, IT b, F f);
```

    - In this case, the **requires** clause contains a constexpr bool expression
      - [C++17] std::is_invocable_v is a variable template defined as

```
template< typename F, typename ... ArgTypes>
using is_invocable_v = is_invocable< F, ArgTypes...>::value;
```

      - std::is_invocable is a class template defined to look like this:

```
template< typename F, typename ... ArgTypes> class is_invocable
{ static constexpr bool value = /*...*/; };
```

      - the actual implementation uses partial specialization and other advanced tricks

[C++20] Requires clauses

- A **requires-clause** acts as a constraint on template parameters

```
template< typename IT, typename F>
requires std::is_invocable_v<F, typename std::iterator_traits<IT>::reference>
F for_each( IT a, IT b, F f);
```

  - If violated, this function declaration will be ignored during overload resolution
    - Most likely, the result will be "no function declaration matches the call"
    - This indicates that the problem is not inside the implementation of for_each
  - For non-function templates, the violation will directly trigger an error message
  - The requires clause also acts as documentation

    - Note: The implementation of for_each probably contains the expression f(*a)
      - The requires-clause essentially checks whether this expression is correct
      - If not correct, template instantiation would fail even if the requires clause were not present
      - It would fail after overload resolution, not before (as with SFINAE or requires)
      - The error message would point to something inside the implementation

[C++20] Concepts

- A ***concept*** is, logically, a Boolean function whose arguments are types, templates or constants
  - In most cases, there is just one *typename* argument
  - Evaluated by the compiler
  - Note: C++14 already has a construct with the same underlying logic:

```
template< typename T> inline constexpr bool is_reference_v = /*...*/;
```

    - The difference is in some syntactic sugar associated with concepts
    - Concepts may be defined using bool constants but not (easily) the other way round

[C++20] Concepts

- Definition of a concept:
  - A **concept** may be defined using a **requires-expression**

```
template< typename T> concept Dereferencable = requires (T x) { *x; };
```

    - In this case, the **requires-expression** states that the expression *x must be semantically valid for any x of type T

```
template< typename F, typename ... AL> concept Callable
    = requires (F f, AL ... al) { f(al ...); };
```

  - A **concept** may also be defined using other concepts or constant Boolean expressions, including combining by && and || operators

```
template< typename T> concept Reference = std::is_reference_v<T>;

template< typename T> concept ConstReference =
    Reference<T> && std::is_const_v< std::remove_reference_t< T>>;
```

    - In this context, && and || operators are well-defined even for erroneous operands
      - If remove_reference_t is not defined for T, the result is false

[C++20] Concepts

- Concepts used with all arguments explicit
  - In the **requires-clause**

```
template< typename IT, typename F>
requires Iterator<IT> && Callable<F, typename IT::reference>
void for_each( IT a, IT b, F f);
```

  - In the definition of other **concepts**

```
template< typename IT>
concept Iterator = Dereferenceble<IT> && Incrementable<IT>;
```

- Concepts used with the first argument implicitly inferred from the context
  - Instead of **typename** in template parameter declaration
    - The first argument of the concept is the type being declared here

```
template< Iterator IT, Callable<typename IT::reference> F>
void for_each( IT a, IT b, F f);
```

    - Just a syntactic sugar equivalent to a **requires** clause
  - In **auto** declarations

```
Iterator auto it = k.find(x);
```

    - Triggers an error if the return type of find does not satisfy Iterator

```
[](Iterator auto it){ return *it; }
```

    - Produces a **requires** clause in the generated template operator()
  - In type-checking requirements inside a **requires-expression**

```
template< typename IT> concept SubtractableIterator =
requires (IT a, IT b) { {a-b} -> std::convertible_to<std::ptrdiff_t>; }
```

    - Invokes the concept std::convertible_to<decltype(a-b), std::ptrdiff_t>

[C++20] Concepts

- Example

```
template< typename K, typename V> concept StackOf
requires (K k, V v) {
    {k.push(v)} -> std::same_as<void>;
    {k.top()} noexcept -> std::convertible_to< V>;
    {k.pop()} -> std::same_as<void>;
};
template< typename K> concept Stack
requires {
    typename K::value_type;
    requires StackOf<K, K::value_type>;
};
```

[C++20] Concepts

- Advantages of concepts
  - Explicit and systematic statement of requirements
  - Understandable diagnostic messages
  - Requires clause participates in overload resolution (SFINAE no longer required)
    - Unlike a static_assert inside the template
- Usable since 2022 – not yet really used
  - C++20 specification fixed in February 2020
    - The C++20 library defines some (few) concepts but does not use them
  - The following compilers support concepts:
    - g++-10 (May 2020)
    - clang-10 – partially
    - MSVC 19.30 (Visual Studio 2022 v. 17.0)