TDDD38/726G82:
Adv. Programming in C++
Templates III

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 Dependent Names
2 SFINAE
3 Concepts
4 Summary

1 Dependent Names
2 SFINAE
3 Concepts
4 Summary

Dependent Names

Dependent Names

```
1  struct X
2  {
3    using foo = int;
4  };
5
6  struct Y
7  {
8    static void foo() { }
9  };
10
11 template <typename T>
12 struct Z
13 {
14   void foo()
15   {
16     T::foo; // what does this refer to?
17   }
18 };
```

Dependent Names

Dependent Names

`T::foo` can refer to these things:

- A type
- A function
- A variable

All of which are names that *depend* on T.

Dependent Names

Dependent Names

- The compiler can have a hard time to distinguish between these uses;
- To specify that it is a *type* use the `typename` keyword;
- If it is a member, then use it as normal.

Dependent Names

Dependent Names

```
1  template <typename T>
2  struct Z
3  {
4    void foo()
5    {
6      // foo should be a type
7      typename T::foo x{};
8
9      // or
10
11     // foo is a function (or a variable)
12     T::foo();
13
14     // or
15
16     // foo is a variable (or a function)
17     T::foo;
18   }
19 };
```

Dependent Names

Binding Rules

- Dependent names
- Non-dependent names

Dependent Names

Binding Rules

- Dependent names
  - Is bound at *instantiation*
  - Name lookup occurs when the template argument is known
  - For member functions in class templates, `this` is a dependent name
- Non-dependent names

Dependent Names

Binding Rules

- Dependent names
- Non-dependent names
  - Is bound at *definition*
  - Name lookup occurs as normal
  - **Note:** if the meaning of a non-dependent name has changed between definition and instatiation, the program is ill-formed

Dependent Names

Binding Rules

```
1  struct Type { };
2  template <typename T>
3  void foo()
4  {
5    // dependent name
6    typename T::type x{};
7
8    // non-dependent name
9    Type t{};
10 }
```

Dependent Names

typename

```
1  template <typename T>
2  class Cls
3  {
4    struct Inner
5    {
6      T x;
7      T::value_type val;
8    };
9  public:
10   static Inner create_inner();
11 };
12
13 template <typename T>
14 Cls<T>::Inner Cls<T>::create_inner()
15 {
16   T x{};
17   T::value_type val;
18   return {x, val};
19 }
```

Dependent Names

typename

```
1  template <typename T>
2  class Cls
3  {
4    struct Inner
5    {
6      T x;
7      typename T::value_type val;
8    };
9  public:
10   static Inner create_inner();
11 };
12
13 template <typename T>
14 typename Cls<T>::Inner Cls<T>::create_inner()
15 {
16   T x{};
17   typename T::value_type val{};
18   return {x, val};
19 }
```

Dependent Names

Ambiguity

```
1 template <int N>
2 int bar()
3 {
4   return S1<N>::S2<N>::foo();
5 }
```

Dependent Names

Which way should the compiler interpret this?

```
1  int foo() { return 1; }
2
3
4  template <int N> struct S1
5  {
6    static int const S2{};
7  };
```

```
1  S1<N>::S2<N>::foo()
```

```
1  template <int N> struct S1
2  {
3    template <int M> struct S2
4    {
5      static int foo() { return M; }
6    };
7  };
```

```
1  S1<N>::S2<N>::foo()
```

Dependent Names

Which way should the compiler interpret this?

```
1  int foo() { return 1; }
2
3
4  template <int N> struct S1
5  {
6    static int const S2{};
7  };
```

```
1  template <int N> struct S1
2  {
3    template <int M> struct S2
4    {
5      static int foo() { return M; }
6    };
7  };
```

```
1  (S1<N>::S2)<(N>::foo())
```

```
1  (S1<N>)::(S2<N>)::(foo())
```

Dependent Names

Which way should the compiler interpret this?

```
1  int foo() { return 1; }
2
3
4  template <int N> struct S1
5  {
6    static int const S2{};
7  };
```

```
1  template <int N> struct S1
2  {
3    template <int M> struct S2
4    {
5      static int foo() { return M; }
6    };
7  };
```

```
1  S2 < (N > foo())
```

```
1  S2<N>::foo()
```

Dependent Names

But what about this?

```
1  template <int N> struct S1
2  {
3    template <int M> struct S2
4    {
5      static int foo() { return M; }
6    };
7  };
8
9  template <> struct S1<1>
10 {
11   static int const S2{};
12 };
13
14 int foo() { return 1; }
15
16 template <int N>
17 int bar()
18 {
19   // only works if N = 1 (the specialization)
20   return S1<N>::S2<N>::foo();
21 }
```

Dependent Names

But what about this?

```
1  template <int N> struct S1
2  {
3    template <int M> struct S2
4    {
5      static int foo() { return M; }
6    };
7  };
8
9  template <> struct S1<1>
10 {
11   static int const S2{};
12 };
13
14 int foo() { return 1; }
15
16 template <int N>
17 int bar()
18 {
19   // works for the general case but not for N = 1
20   return S1<N>::template S2<N>::foo();
21 }
```

Dependent Names

Dependent names of templates

- If a name depends on a template (as was the case with S1<N>::S2<N>) the compiler cannot assume that the dependent name is a template
- Therefore the only reasonable interpretation must be that the second < is a comparison
- *unless* we specify it as a template by adding `template` before the dependent name
- This is true for all operators which can access names; `->`, `.` and `::`

Dependent Names

What type of entities must A, B and C be?

```
1  template <typename T>
2  void bar()
3  {
4    typename T::A a;
5    T::B;
6    T::C();
7  }
```

1 Dependent Names
2 SFINAE
3 Concepts
4 Summary

SFINAE

Suppose the following:

```
1  template <typename T, int N>
2  int size(T const (&arr)[N])
3  {
4    return N;
5  }
6
7  template <typename T>
8  typename T::size_type size(T const& t)
9  {
10   return t.size();
11 }
```

```
1  int main()
2  {
3    int arr[3]{1,2,3};
4    std::vector<int> vec{4,5};
5
6    std::cout << size(arr)
7              << std::endl;
8
9    std::cout << size(vec)
10             << std::endl;
11 }
```

SFINAE

How should the compiler handle this case?

When we pass an array of type `int(&)[N]` into `size` the compiler will:

1. examine each `size` candidate to see if they fit
2. notice that both function templates take one argument
3. notice that the second version have return type `typename T::size_type`
4. see that `int(&)[N]` is of non-class type, so it cannot have members
5. conclude that the second version is invalid

But the first version matches, so should the compiler actually report an error regarding the second version?

SFINAE

The best acronym

Substitution **F**ailure **I**s **N**ot **A**n **E**rror

SFINAE

Excuse me, what?

- During instantiation of templates the compiler will substitue the template parameters with an actual type or value
- This substitution can fail for many reasons
- If it does fail, it is not consider an error
- Instead the compiler will move on and try to find another match elsewhere

SFINAE

So it is just a special case? Why should I care?

Somebody realized, in the distant time of the 90's that this can be exploited for some awesome things!

SFINAE

Controlling the substitution failures

```
1  // if parameter is a container
2  template <
3    typename T,
4    typename = typename T::size_type>
5  int size(T const& t)           // #1
6  {
7    return t.size();
8  }
9  // if parameter is an array
10 template <typename T, size_t N>
11 int size(T const (&)[N])       // #2
12 {
13   return N;
14 }
```

```
1  // if parameter is a pointer
2  template <typename T, T = nullptr>
3  int size(T const& t)           // #3
4  {
5    // we don't know how many elements
6    // the pointer is pointing to
7    return -1;
8  }
9
10 // Everything else, a so called sink
11 T size(...)                    // #4
12 {
13   return 1;
14 }
```

SFINAE

Controlling the substitution failures

- There are 4 overloads of `size`, numbered #1 to #4
- #1 will fail for all cases where `T` does not have a type named `size_type` (i.e. non-container types)
- #2 will only match arrays (not due to SFINAE)

SFINAE

Controlling the substitution failures

- Nontype template parameters can be pointers
- so if T is a pointer it can be a template parameter
- In #3 we take a nontype template parameter of type T and have `nullptr` as default-value
- This will fail if T is not a pointer, since `nullptr` can only be assigned to pointers

SFINAE

Controlling the substitution failures

- #4 is a so called *variadic function*
- variadic functions are a relic from C
- has been made obselete by variadic templates
- a variadic function will only be called if there are no other matching functions
- I.e. it has the lowest priority during overload resolution
- Due to this, it is perfect as a sink

SFINAE

Trigger failure with a `bool` condition

```
1  template <bool, typename T = void>
2  struct enable_if
3  {
4  };
5
6  template <typename T>
7  struct enable_if<true, T>
8  {
9    using type = T;
10 };
11
12 template <bool N, typename T = void>
13 using enable_if_t = typename enable_if<N, T>::type;
```

SFINAE

`std::enable_if`

- `std::enable_if` is a class template that takes two parameters: a `bool` condition and an arbitrary data type T.
- If the `bool` condition is `true`, then `std::enable_if` will contain a type alias `type` which is an alias to T.
- Otherwise, it will be an empty class template.

SFINAE

`std::enable_if`

- This means that if we try to access `type` it will only exist if the `bool` condition is `true`.
- So if we use `enable_if` in function template headers, SFINAE will make sure that the function overload only is valid (and therefore available) if the `bool` condition is `true` by simply trying to access `type`.
- This is because if we try to access a type that does not exist, then SFINAE is triggered.

SFINAE

We need to go deeper!

```
1  template <int N>
2  enable_if_t<(N >= 0) && (N % 2 == 0)> check()
3  {
4    cout << "Even!" << endl;
5  }
6
7  template <int N, typename = enable_if_t<(N >= 0) && (N % 2 == 1)>>
8  void check()
9  {
10   cout << "Odd!" << endl;
11 }
12
13 template <int N>
14 void check(enable_if_t<(N < 0), int> = {})
15 {
16   cout << "Negative" << endl;
17 }
18
19 check<0>();
20 check<3>();
21 check<-57>();
```

SFINAE

We need to go deeper!

Notice that we have three disjoint cases:

- `(N >= 0) && (N \% 2 == 0)`
- `(N >= 0) && (N \% 2 == 1)`
- `N < 0`

SFINAE

We need to go deeper!

- SFINAE only applies in the function header
- Each case occurs in a different places of the header
- SFINAE considers:
  - Default values to template parameters
  - The return type
  - Function parameter types
  - Default values to function parameters

SFINAE

We need to go deeper!

- By default, the second template parameter to `std::enable_if` is `void`.
- You cannot have a default value for `void` parameters in a function.
- Because of this, the last case needs to use somehing other than `void`. In this case we use `int`.

SFINAE

`std::enable_if` is essentially a template `if`-statement!

SFINAE

Nice SFINAE with C++11

```cpp
// if t has a member size()
template <typename T>
auto size(T const& t) -> decltype(t.size())
{
  return t.size();
}
// if t is a pointer
template <typename T>
auto size(T const& t) -> decltype(*t, -1)
{
  return -1;
}
```

```cpp
// if T is an array
template <typename T, size_t N>
auto size(T const (&)[N])
{
  return N;
}

// sink
int size(...)
{
  return 1;
}
```

SFINAE

...What?

Let’s take it step by step:

- Trailing return type
- `decltype`
- comma-operator
- Expression SFINAE

SFINAE

Trailing return type

```
1 auto foo(int x) -> int
2 {
3   return x;
4 }
```

```
1 int foo(int x)
2 {
3   return x;
4 }
```

SFINAE

decltype

```
1 int             i{0};   // int
2 decltype(i+1) j{i+1}; // int
```

SFINAE

`decltype`

- `decltype` is a specifier that collapses to a type
- `decltype(...)` will deduce the type of the supplied expression
- The expression inside `decltype` will never be evaluated, nor compiled. The compiler just checks what type the expression is.

SFINAE

the comma-operator

```
1 char sign{(1, 1.0, 'a')};
2 bool flag{(cout << 1, true)};
```

SFINAE

the comma-operator

- C++ has an operator called the *comma-operator*
- It takes a comma-separated list of expressions
- evaluates all of the expressions
- and return the final one
- So the *type* of the expression is the type of the last one
- ... never use it for evil (nor when you are lazy)!

SFINAE

Expression SFINAE

```
1  // only match types which can be
2  // added with 1 (and default initialized)
3  template <typename T>
4  decltype(T{}+1) inc(T const& t)
5  {
6    return t+1;
7  }
```

SFINAE

Expression SFINAE

- if a template declaration uses `decltype`
- every expression in the `decltype` declaration will trigger a substitution failue if they are invalid
- this is called *expression SFINAE*
- Notice that the return type will be whatever type `T{}+1` is.

SFINAE

Expression SFINAE

```
1  // only match types which can be incremented
2  template <typename T>
3  auto inc(T& t) -> decltype(++t)
4  {
5    return ++t;
6  }
```

SFINAE

Expression SFINAE and trailing return type

- If we add the `decltype` inside a trailing return type instead then we have access to the function parameters.
- This means we don't have to construct a new object of type `T`, instead we can use `t` directly.
- This is better, because now we don't have to assume that `T` can be default initialized.

SFINAE

Putting it all together!

```cpp
// if t has a member size()
template <typename T>
auto size(T const& t) -> decltype(t.size())
{
  return t.size();
}
// if t is a pointer
template <typename T>
auto size(T const& t) -> decltype(*t, -1)
{
  return -1;
}
```

```cpp
// if T is an array
template <typename T, size_t N>
auto size(T const (&)[N])
{
  return N;
}

// sink
int size(...)
{
  return 1;
}
```

SFINAE

Putting it all together!

Overload #1:

- Notice that in the `decltype` we call `.size()`.
- If `t.size()` is invalid (for example if T doesn't have a member function called `size`) then this leads to a substitution failure.
- This means this function is only callable if T has a member called `size`.
- **Note:** `t.size()` is actually never *called*, it's just examined by the compiler.

SFINAE

Putting it all together!

Overload #2:

- Notice that our return type is `decltype(*t, -1)`
- Inside the `decltype` we use the comma-operator for two separate expressions: `*t` and `-1`. If any of these expressions are invalid the substitution fails.
- `*t` is only valid for pointer (or pointer-like) types. So if `T` isn't a pointer this will fail.
- The return type is given by the last expression (`-1`).

SFINAE

What will be printed?

```
1  template <typename T>
2  enable_if_t<(sizeof(T) < 4), int> foo(T const&)
3  { return 1; }
4
5  template <typename T, T = nullptr>
6  int foo(T const&)
7  { return 2; }
8
9  template <typename T>
10 auto foo(T const& t) -> decltype(t.size(), 3)
11 { return 3; }
12
13 int main()
14 {
15   vector<int> v{};
16   short int s{};
17   cout << foo(s)
18        << foo(nullptr)
19        << foo(v);
20 }
```

1 Dependent Names
2 SFINAE
3 Concepts
4 Summary

Concepts

Example

```
1 template <typename T>
2 auto remainder(T a, T b)
3 {
4   return a % b;
5 }
```

Concepts

Example

- `remainder` only works for *integral types* (`int`, `bool`, `char`, `long` etc.).
- Floating point numbers (`float`, `double` and `long double`) doesn't support `operator`%.
- However, the concept of remainders is still applicable to floating point numbers.
- In <`cmath`> there is the function `std::fmod` that calculates the remainder for floating point numbers.

Concepts

Example

```
1  template <typename T>
2  auto remainder(T a, T b)
3  {
4    return a % b;
5  }
6
7  template <typename T>
8  auto remainder(T a, T b)
9  {
10   return std::fmod(a, b);
11 }
```

Concepts

Example

```
example.cc:11:6: error: redefinition of
`template<class T> auto remainder(T, T)`
   11 | auto remainder(T a, T b)
      |      ^~~~~~~~~
example.cc:5:6: note: `template<class T> auto remainder(T, T)`
previously declared here
    5 | auto remainder(T a, T b)
      |      ^~~~~~~~~
```

Concepts

Example

- This will unfortunately not work...
- We have two equally valid overloads of `remainder`
- Any call to `remainder` will therefore be ambigous

Concepts

Example

```
1  template <typename T>
2  auto remainder(T a, T b)
3      -> std::enable_if_t<std::is_integral_v<T>,
4                          decltype(a % b)>
5  {
6    return a % b;
7  }
8
9  template <typename T>
10 auto remainder(T a, T b)
11     -> std::enable_if_t<std::is_floating_point_v<T>,
12                         decltype(std::fmod(a, b))>
13 {
14   return std::fmod(a, b);
15 }
```

Concepts

Example

- This works, but is very terse.
- Since we are forced to use trailing return types for our SFINAE usage, we can't use `auto` as return type anymore.
- So we have to duplicate our return statement in a `decltype` statement if we want the compiler to deduce the right return type.
- If T is something other than an integral or floating point type, the errors are horribly unhelpful.

Concepts

C++20

Simplified with C++20

Concepts

Enter `requires` clauses!

```
1  // requires after template header
2  template <typename T> requires std::is_integral_v<T>
3  auto remainder(T a, T b)
4  {
5    return a % b;
6  }
7
8  // requires after function header
9  template <typename T>
10 auto remainder(T a, T b) requires std::is_floating_point_v<T>
11 {
12   return std::fmod(a, b);
13 }
```

Concepts

Enter `requires` clauses!

- `requires` is a new keyword that specifies requirements on function templates.
- It can be placed immediately after the template header,
- Or immediately after the function header.
- Requirements are expressed as `bool` expressions,
- which means that we can use && and || to chain multiple requirements together.

Concepts

Enter `requires` clauses!

```
1  // requires after template header
2  template <typename T>
3    requires std::is_integral_v<T>
4  auto remainder(T a, T b)
5  {
6    return a % b;
7  }
8
9  // requires after function header
10 template <typename T>
11 auto remainder(T a, T b)
12   requires std::is_floating_point_v<T> || std::is_convertible_v<T, double>
13 {
14   return std::fmod(a, b);
15 }
```

Concepts

Enter `requires` clauses!

- Now we are checking that T is either an integral/floating point type,
- **or** convertible to `int` and `double` respectively.
- This works since a `requires` clause just expects a `bool` expression.

Concepts

Even simpler with Concepts

```
 1 template <typename T>
 2 auto remainder(T a, T b)
 3   requires std::integral<T>
 4 {
 5   return a % b;
 6 }
 7
 8 template <typename T>
 9 auto remainder(T a, T b)
10   requires std::floating_point<T>
11 {
12   return std::fmod(a, b);
13 }
```

Concepts

Even simpler with Concepts

- Concepts are a collection of requirements bundled together into one entity, called a *concept*.
- Concepts are used to induce requirements on template parameters. They also act as `bool` conditions.
- The standard library provides several pre-defined concepts in `<concepts>`,
- For example: `std::integral` and `std::floating_point`.
- But you can also make your own.

Concepts

A simpler way to use concepts

```
1  template <std::integral T>
2  auto remainder(T a, T b)
3  {
4    return a % b;
5  }
6
7  template <std::floating_point T>
8  auto remainder(T a, T b)
9  {
10   return std::fmod(a, b);
11 }
```

Concepts

A simpler way to to use concepts

- We can replace `typename` in template parameter declarations with a specific concept instead.
- This means that whatever that template parameter is instantiated as, it must fulfill the requirements specified by the concept.

Concepts

Implementation of `std::integral` and `std::floating_point`

```
1  namespace std
2  {
3    template <typename T>
4    concept integral = std::is_integral_v<T>;
5
6    template <typename T>
7    concept floating_point = std::is_floating_point_v<T>;
8  }
```

Concepts

Creating own concepts

- You create your own concept with the `concept` keyword.
- Each concept must take template parameters which are then constrained.
- After the concept name there must be a = that are then followed by the constraints, which are all `bool` expressions.

Concepts

Creating our own concepts

```
1  template <typename T>
2  concept like_int = std::integral<T> ||
3                     std::is_convertible_v<T, int>;
4
5  template <like_int T>
6  concept remainder(T a, T b)
7  {
8    return a % b;
9  }
```

Concepts

Creating our own concepts

- We can join multiple constraints together with && and `||`, which allows for more complicated concepts.
- We should also note that concepts acts as `bool` conditions, so we can also use those to define other, more constrained concepts.
- In this case we are saying that `T` must either fulfill `std::integral`, or it must be convertible to `int` in order to fulfill the `like_int` concept.
- Let's look at another example:

Concepts

Another example

```
1  template <has_at T>
2  auto const& get(T const& t, int i)
3  {
4    return t.at(i);
5  }
6
7  template <typename T> requires (!has_at<T>)
8  auto const& get(T const& t, int i)
9  {
10   return t[i];
11 }
```

Concepts

Another example

- In this example we want to make a `get` function that takes an container structure and an index and returns the element at the corresponding index in the container.
- If the container has an `at` function, we want to use that, otherwise we want to use `operator[]`. Note that the `[]` case only occurs if we *don't* have an `at` function due to the requirement (`!has_at<T>`).
- How do we implement the `has_at` concept?

Concepts

Implementing `has_at`

```
1 template <typename T>
2 concept has_at = requires(T t, int i)
3 {
4   t.at(i);
5 };
```

Concepts

Implementing `has_at`

- A concept can also use an *requires expresson*,
- which is a way for us to specify requirements on the interface of the passed in types.
- This is done by checking that certain expressions are valid for the type.
- These expressions are never actually evaluated, they are just examined by the compiler to check if they are valid for the specified type.

Concepts

Implementing `has_at`

- In order to make such checks we need access to objects that are involved in the expression we are checking.
- Because of this, a requires expression can take parameters. We can add how many parameters we want, and they can be whatever type we want.
- These parameters are never actually created and we never pass anything to them. They are just there so we have access to symbolic objects that are used in the validation of the expressions.

Concepts

Implementing `has_at`

- This can be seen in the `has_at` concept where we take a T object called `t` and an `int` parameter called `i`. These parameters are never created, and we don't have to care about them when using our concept.
- These are then used in the body of the requires expression to check if we can call `t.at(i)`.
- This simply means that the compiler checks: does T have a member function which can be called on an instance of T with an `int` parameter?

Concepts

Requires expression in the wild

```
1  template <typename T>
2  void print(std::ostream& os, T const& data)
3    requires std::is_class_v<T> &&
4             requires { os << data; }
5  {
6    os << data;
7    if (requires { T{data}; })
8    {
9      os << " (copyable)";
10   }
11 }
```

Concepts

Requires expression in the wild

- A requires expression is just a `bool` expression, so we can use them outside the context of concepts.
- Parameters are optional for requires expressions.
- Here we are using a requires expression to constrain the template parameter T. We are first checking if T is a class, and then we make sure that we can print it with `operator`<<.
- Then we are using it in a normal if-statement to check if T has a copy constructor.

Concepts

Concepts + classes = ♥

```
1  template <typename From, typename To>
2  concept convertible_to = std::is_convertible_v<From, To>;
3
4  template <convertible_to<int> T>
5    requires std::copy_constructible<T>
6  class Cls
7  {
8  };
```

Concepts

Concepts + classes = ♥

- Of course, template parameters for class templates can also be constrained with the help of concepts.
- Notice here that our concept takes two parameters, `From` and `To`.
- When we constrain a template parameter with a concept the first one will always implicitly be the template parameter itself.
- All other template parameters to the concept must however be set.
- So `From` is the same as `T` while `To` is set to `int`.

1 Dependent Names
2 SFINAE
3 Concepts
4 Summary

Summary

- SFINAE: C++98 (so works everywhere)

Summary

- SFINAE: C++98 (so works everywhere)
- Concepts: C++20 (easier, but requires modern compiler)

Summary

- SFINAE: C++98 (so works everywhere)
- Concepts: C++20 (easier, but requires modern compiler)
- Both techniques are used in real code

Summary

- SFINAE: C++98 (so works everywhere)
- Concepts: C++20 (easier, but requires modern compiler)
- Both techniques are used in real code
- Concepts can *usually* replace SFINAE

Summary

- SFINAE: C++98 (so works everywhere)
- Concepts: C++20 (easier, but requires modern compiler)
- Both techniques are used in real code
- Concepts can *usually* replace SFINAE
- Main focus of course is SFINAE (but concepts are allowed)

Summary

- SFINAE: C++98 (so works everywhere)
- Concepts: C++20 (easier, but requires modern compiler)
- Both techniques are used in real code
- Concepts can *usually* replace SFINAE
- Main focus of course is SFINAE (but concepts are allowed)
- Do exercises using both SFINAE and concepts and compare

www.liu.se

LINKÖPING UNIVERSITY