# TDDD38 - Advanced programming in C++

## Templates III


Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

# Dependent Names

## Dependent Names

```cpp
struct X
{
  using foo = int;
};

struct Y
{
  static void foo() { }
};

template <typename T>
struct Z
{
  void foo()
  {
    T::foo; // what does this refer to?
  }
};
```

# 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

```
template <typename T>
struct Z
{
  void foo()
  {
    // foo should be a type
    typename T::foo x{};

    // or

    // foo is a function
    T::foo();

    // or

    // foo is a variable
    T::foo;
  }
};
```

# 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

```
struct Type { };
template <typename T>
void foo()
{
  // dependent name
  typename T::value x{};

  // non-dependent name
  Type t{};
}
```

# Dependent Names

## Ill-formed program

```
struct Type;

template <typename T>
struct Foo
{
  // Type is incomplete during definition
  Type x;
};

// Type is still incomplete during instantiation
Foo<int> foo;

// Doesn't matter that we define Type here
// the instantiation above is ill-formed
struct Type { };
```

# Dependent Names

Ill-formed program

The worst part?

# Dependent Names

Ill-formed program

The compiler isn't required to report this as an error

# Dependent Names

Ill-formed program

Fortunately, most compilers do!

# Dependent Names

Ill-formed program

```
test.cpp:7:8: error: field 'x' has incomplete type 'Type'
   Type x;
        ^
test.cpp:1:8: note: forward declaration of 'struct Type'
 struct Type;
        ^~~~
```

# Dependent Names

## typename

```
template <typename T>
class Cls
{
  struct Inner
  {
    T x;
    T::value_type val;
  };
public:
  static Inner create_inner();
};

template <typename T>
Cls<T>::Inner Cls<T>::create_inner()
{
  T x{};
  T::value_type val;
  return {x, val};
}
```

# Dependent Names

## typename

```
template <typename T>
class Cls
{
  struct Inner
  {
    T x;
    typename T::value_type val;
  };
public:
  static Inner create_inner();
};

template <typename T>
typename Cls<T>::Inner Cls<T>::create_inner()
{
  T x{};
  typename T::value_type val{};
  return {x, val};
}
```

# Dependent Names

Ambiguity

```
template <typename T> struct S1
{
  template <typename U> struct S2
  {
    static void foo() {}
  };
};
template <typename T>
void bar()
{
  S1<T>::S2<T>::foo();
}
```

# Dependent Names

Which way should the compiler interpret this?

```
S1<T>::S2<T>::foo();
```

# Dependent Names

Which way should the compiler interpret this?

```
(S1<T>::S2)<(T>::foo())
```

```
(S1<T>)::(S2<T>)::(foo())
```

```
S2<T> < (T > foo())
```

```
S2<T>::foo()
```

# Dependent Names

Dependent names of templates

```
S1<T>::template S2<T>::foo();
```

# Dependent Names

## Dependent names of templates

- If a name depends on a template (as was the case with `S1<T>::S2<T>`) 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 `::`

LINKÖPING UNIVERSITY

# More on Templates

## Template parameters

There are three kinds of templates parameters:

- type template parameters
- non-type template parameters
- template template parameters

# More on Templates

## Template template parameters

- What if we want to take a *template* as a parameter?
- That is, a class template that has not been instantiated?
- It is not possible with the two types we have seen so far, type or non-type template parameters
- This is where *template template parameters* come in!

# More on Templates

## Template template parameters

```
template<template <typename> typename T>
struct Wrap_Int
{
  T<int> wrapper;
};

template <typename T>
struct X
{
  T data;
};

int main()
{
  Wrap_Int<X> x;
}
```

# More on Templates

## Template template parameters

- Here `Wrap_Int` takes `X` as a template parameter
- `X` is passed to `Wrap_Int` as a class template
- Inside `Wrap_Int`, `X` will then be instantiated with `int`
- You can think of it as a "template parameters that takes the name of another template"

# More on Templates

## Template template parameters

```
template <typename T, typename U>
struct Y { };

int main()
{
  // does not work, Y takes 2 template parameters
  Wrap_Int<Y> y;

  // does not work, int is not a template
  Wrap_Int<int> z;
}
```

# More on Templates

## A more concrete example

```
template <typename T,
          typename U,
          template <typename,typename> typename C>
ostream& operator<<(ostream& os, C<T, U> const& c)
{
  for (auto const& e : c)
  {
    os << e << ' ';
  }
  return os;
}

int main()
{
  vector<int> v{1,2,3,4};
  cout << v << endl;
}
```

# More on Templates

## A more concrete example

- `C` is template that takes two template parameters
- `T` and `U` are two arbitrary types
- The parameter `c` is of type `C<T, U>`
- We instantiate `operator<<` with:
  - `T = int`
  - `U = allocator<int>`
  - `C = vector`
- Thus we get `vector<int, allocator<int>>`

# More on Templates

A problem appears on the horizon!

```
int main()
{
  map<int, int> m{{1,2},{3,4},{5,6}};
  cout << m << endl;
}
```

- This won't work
- `std::map` takes *three* template parameters

# More on Templates

## Variadic Templates to the rescue!

```
template <template <typename, typename...> typename C,
          typename... Ts>
ostream& operator<<(ostream& os, C<Ts...> const& c)
{
  for (auto const& e : c)
  {
    os << e << ' ';
  }
  return os;
}

int main()
{
  vector<int> v{1,2,3,4};
  map<int, int> m{{1,2},{3,4},{5,6}};
  cout << v << endl;

  // we would need a operator<< for pair<T, U> as well
  cout << m << endl;
}
```

# More on Templates

And now for something completely different...

# More on Templates

## Forwarding References

```
void fun1(int&& x);

template <typename T>
void fun2(T&& x);

// ...

int main()
{
  int x{};
  fun1(5); // works

}
```

# More on Templates

## Forwarding References

```
void fun1(int&& x);

template <typename T>
void fun2(T&& x);

// ...

int main()
{
  int x{};
  fun1(5); // works
  fun1(x); // doesn't work
}
```

# More on Templates

## Forwarding References

```
void fun1(int&& x);

template <typename T>
void fun2(T&& x);

// ...

int main()
{
  int x{};
  fun2(5); // works

}
```

# More on Templates

## Forwarding References

```cpp
void fun1(int&& x);

template <typename T>
void fun2(T&& x);

// ...

int main()
{
  int x{};
  fun2(5); // works
  fun2(x); // works?!
}
```

# More on Templates

## Forwarding References

`T&&` denotes:

- a non-`const` rvalue reference to `T`
- except if `T` is a type template parameter
- then it denotes a *forwarding reference*

# More on Templates

## Forwarding References

```
template <typename T>
void foo(T&&);

// generated functions:
```

```
int main()
{
  int x{};
  int const y{};

  foo(5);
  foo(x);
  foo(y);
}
```

# More on Templates

## Forwarding References

```cpp
template <typename T>
void foo(T&&);

// generated functions:
void foo(int&&);
```

```cpp
int main()
{
  int x{};
  int const y{};

  foo(5);
  foo(x);
  foo(y);
}
```

# More on Templates

## Forwarding References

```cpp
template <typename T>
void foo(T&&);

// generated functions:
void foo(int&&);

void foo(int&);
```

```cpp
int main()
{
  int x{};
  int const y{};

  foo(5);
  foo(x);
  foo(y);
}
```

# More on Templates

## Forwarding References

```
template <typename T>
void foo(T&&);

// generated functions:
void foo(int&&);

void foo(int&);

void foo(int const&);
```

```
int main()
{
  int x{};
  int const y{};

  foo(5);
  foo(x);
  foo(y);
}
```

# More on Templates

## Forwarding References

```
template <typename T>
void foo(T&&);

// generated functions:
void foo(int&&);

void foo(int&);

void foo(int const&);
```

```
int main()
{
  int x{};
  int const y{};

  foo(5);
  foo(x);
  foo(y);
}
```

# More on Templates

## Forwarding Reference

- A forwarding reference can collapse into to any type of reference
- Parameters that are forwarding references will bind to the exact reference that is passed
- Each type of reference passed into the forwarding reference will generate a specific overload

# More on Templates

## Forwarding Reference

```
template <typename... Ts>
vector<T> store(Ts... list)
{
  vector<T> vec {list...};
  return vec;
}
```

# More on Templates

## Problems with `store`

- Every parameter in `list` is passed by value
- Then they are copied into `vec`

# More on Templates

## Problems with `store`

- Every parameter in `list` is passed by value
- Then they are copied into `vec`
- This will cause every parameter passed into the function to be copied twice

# More on Templates

## Problems with `store`

- Every parameter in `list` is passed by value
- Then they are copied into `vec`
- This will cause every parameter passed into the function to be copied twice
- But we have move-semantics; we should leverage it whenever possible

# More on Templates

## Forwarding Reference

```
template <typename... Ts>
vector<T> store(Ts&&... list)
{
  vector<T> vec {list...};
  return vec;
}
```

# More on Templates

## Forwarding Reference

- When we write `Ts&&...` we apply a *pattern* on `Ts`
- This pattern will bind every type in `Ts` to a forwarding reference
- Thus generating a function which takes every parameter as they are
- But we are not yet using move semantics, because we will still copy everything into `vec`

# More on Templates

## Forwarding Reference

```
#include <utility> // std::forward
template <typename... Ts>
vector<T> store(Ts&&... list)
{
  vector<T> vec {std::forward<Ts>(list)...};
  return vec;
}
```

# More on Templates

`std::forward`

- `std::forward` is a function template define in `<utility>`
- it takes a parameter of arbitrary type `T` and passes it as the correct type of reference
- this is necessary since binding an rvalue into an rvalue reference will give it a name
- therefore the rvalue itself has become an lvalue
- `std::forward` is a way to "turn it back"

# More on Templates

## Forwarding References & auto

```
int main()
{
  int x{};

  // will become int&&
  auto&& y{5};

  // will become int&
  auto&& z{x};
}
```

# More on Templates

## Alias Template

In C++11 *alias templates* were introduced

```
template <typename T>
using array = std::vector<T>;
```

A template alias refers to a set of template types.

# More on Templates

## Variable Templates

In C++14 *variable templates* were introduced

```
template <int N>
bool positive {N > 0};

cout << positive<3> << endl;
cout << positive<-1> << endl;
```

# More on Templates

## Variable Templates

- A *variable template* defines a set of variables
- Must be free variables or static member variables
- I.e. data members in a class-type cannot be templated.

LINKÖPING
UNIVERSITY

# SFINAE

Suppose the following:

```cpp
template <typename T, int N>
int size(T const (&arr)[N])
{
  return N;
}

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

```cpp
int main()
{
  int arr[3]{1,2,3};
  std::vector<int> vec{4,5};

  std::cout << size(arr)
            << std::endl;

  std::cout << size(vec)
            << std::endl;
}
```

# 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`

# SFINAE

## How should the compiler handle this case?

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

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

**S**ubstitution **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

```
// if parameter is a container
template <
  typename T,
  typename = typename T::size_type>
int size(T const& t)            // #1
{
  return t.size();
}
// if parameter is an array
template <typename T, size_t N>
int size(T const (&)[N])        // #2
{
  return N;
}
```

```
// if parameter is a pointer
template <typename T, T = nullptr>
int size(T const& t)            // #3
{
  // we don't know how many elements
  // the pointer is pointing to
  return -1;
}

// Everything else, a so called sink
T size(...)                     // #4
{
  return 1;
}
```

# 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

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

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

template <bool N, typename T = void>
using enable_if_t = typename enable_if<N, T>::type;
```

# SFINAE

We need to go deeper!

```
template <int N>
enable_if_t<(N >= 0) && (N % 2 == 0)> check()
{
  cout << "Even!" << endl;
}

template <int N, typename = enable_if_t<(N >= 0) && (N % 2 == 1)>>
void check()
{
  cout << "Odd!" << endl;
}

template <int N>
void check(enable_if_t<(N < 0)> = {})
{
  cout << "Negative" << endl;
}

check<0>();
check<3>();
check<-57>();
```

# SFINAE

## Nice SFINAE with C++11

```
// 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;
}
```

```
// 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

```
auto foo(int x) -> int
{
  return x;
}
```

```
int foo(int x)
{
  return x;
}
```

# SFINAE

`decltype`

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

decltype((i)) k{i};   // int&
decltype((5)) l{5};   // int&&
```

# SFINAE

`decltype`

- `decltype` is a specifier that collapses to a type
- `decltype(...)` will deduce the type of the supplied expression
- `decltype((...))` will deduce the type and value category of the supplied expression and return an appropriate reference

# SFINAE

the comma-operator

```
char sign{(1, 1.0, 'a')};
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
- … never use it for evil (nor when you are lazy)!

# SFINAE

## Expression SFINAE

```
// only match types which can be incremented
template <typename T>
decltype(T{}++) inc(T& t)
{
  return t++;
}
```

# 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*

# SFINAE

Putting it all together!

```
// 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;
}
```

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

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

www.liu.se