TDDD38/726G82 - Advanced programming in C++

Templates III

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Dependent Names

Dependent Names

```
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

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

    // or

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

    // or

    // foo is a variable (or a function)
    T::foo;
  }
};
```

Dependent Names

Binding Rules

- Dependent names
- Non-dependent names

Dependent Names

Binding Rules

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

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

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 <int N>
int bar()
{
  return S1<N>::S2<N>::foo();
}
```

Dependent Names

Which way should the compiler interpret this?

```
int foo() { return 1; }


template <int N> struct S1
{
  static int const S2{};
};
```

```
template <int N> struct S1
{
  template <int M> struct S2
  {
    static int foo() { return M; }
  };
};
```

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

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

Dependent Names

Which way should the compiler interpret this?

```
int foo() { return 1; }


template <int N> struct S1
{
  static int const S2{};
};
```

```
template <int N> struct S1
{
  template <int M> struct S2
  {
    static int foo() { return M; }
  };
};
```

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

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

Dependent Names

Which way should the compiler interpret this?

```
int foo() { return 1; }


template <int N> struct S1
{
  static int const S2{};
};
```

```
template <int N> struct S1
{
  template <int M> struct S2
  {
    static int foo() { return M; }
  };
};
```

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

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

Dependent Names

But what about this?

```
template <int N> struct S1
{
  template <int M> struct S2
  {
    static int foo() { return M; }
  };
};

template <> struct S1<1>
{
  static int const S2{};
};

int foo() { return 1; }

template <int N>
int bar()
{
  // only works if N = 1 (the specialization)
  return S1<N>::S2<N>::foo();
}
```

Dependent Names

But what about this?

```
template <int N> struct S1
{
  template <int M> struct S2
  {
    static int foo() { return M; }
  };
};

template <> struct S1<1>
{
  static int const S2{};
};

int foo() { return 1; }

template <int N>
int bar()
{
  // works for the general case but not for N = 1
  return S1<N>::template S2<N>::foo();
}
```

Dependent Names

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

```
template <typename T>
void bar()
{
  typename T::A a;
  T::B;
  T::C();
}
```

LINKÖPING UNIVERSITY

More on Templates

Template parameters

There are three kinds of template parameters:

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

More on Templates

Template template parameters

```
template <

  typename T
>
```

More on Templates

Template template parameters

```
template <
  template <typename>
  typename T
>
```

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

```
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

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

```
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

```
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

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

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

```
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&);
```

```
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

```
template <typename T>
void fun1(T&& t)
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)
{
  fun2(t);
}
```

lvalue

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t)  lvalue
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t)  lvalue
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t) lvalue (?!)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(t);
}
```

```
template <typename T>
void fun2(T&& t) lvalue (?!)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t)  xvalue
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t)  xvalue
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)
{
  fun2(std::move(t));
}
```

lvalue

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t) xvalue (!)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(std::move(t));
}
```

```
template <typename T>
void fun2(T&& t) xvalue (!)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)  lvalue
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  lvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)  lvalue
{
  // value category?
}
```

```
int x { };
fun1(x);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)  xvalue
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)  xvalue
{
  // value category?
}
```

```
fun1(7);
```

More on Templates

Forwarding Reference

```
template <typename T>
void fun1(T&& t)  xvalue
{
  fun2(std::forward<T>(t));
}
```

```
template <typename T>
void fun2(T&& t)  xvalue
{
  // value category?
}
```

Works!

```
fun1(7);
```

More on Templates

Forwarding Reference

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

More on Templates

Forwarding Reference

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

More on Templates

Forwarding Reference

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

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

What will be printed?

```
void fun(int&) { cout << 1; }
void fun(int const&) { cout << 2; }
void fun(int&&) { cout << 3; }

template <typename T>
void fun2(T&& t) { fun(t); }

int main()
{
  int x{};
  int const y{};
  fun2(1);
  fun2(x);
  fun2(y);
}
```

More on Templates

What about now?

```
void fun(int&) { cout << 1; }
void fun(int const&) { cout << 2; }
void fun(int&&) { cout << 3; }

template <typename T>
void fun2(T&& t) { fun(std::forward<T>(t)); }

int main()
{
  int x{};
  int const y{};
  fun2(1);
  fun2(x);
  fun2(y);
}
```

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;
```

LINKÖPING UNIVERSITY

SFINAE

Suppose the following:

```
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();
}
```

```
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

The best acronym

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

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

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), int> = {})
{
  cout << "Negative" << endl;
}

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

SFINAE

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

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

the comma-operator

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

SFINAE

Expression SFINAE

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

SFINAE

Expression SFINAE

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

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;
}
```

SFINAE

What will be printed?

```
template <typename T>
enable_if_t<(sizeof(T) < 4), int> foo(T&&)
{ return 1; }

template <typename T, T = nullptr>
int foo(T&&)
{ return 2; }

template <typename T>
auto foo(T&& t) -> decltype(t.size(), 3)
{ return 3; }

int main()
{
  vector<int> v{};
  short int s{};
  cout << foo(s)
       << foo(nullptr)
       << foo(v);
}
```

www.liu.se

LINKÖPING
UNIVERSITY