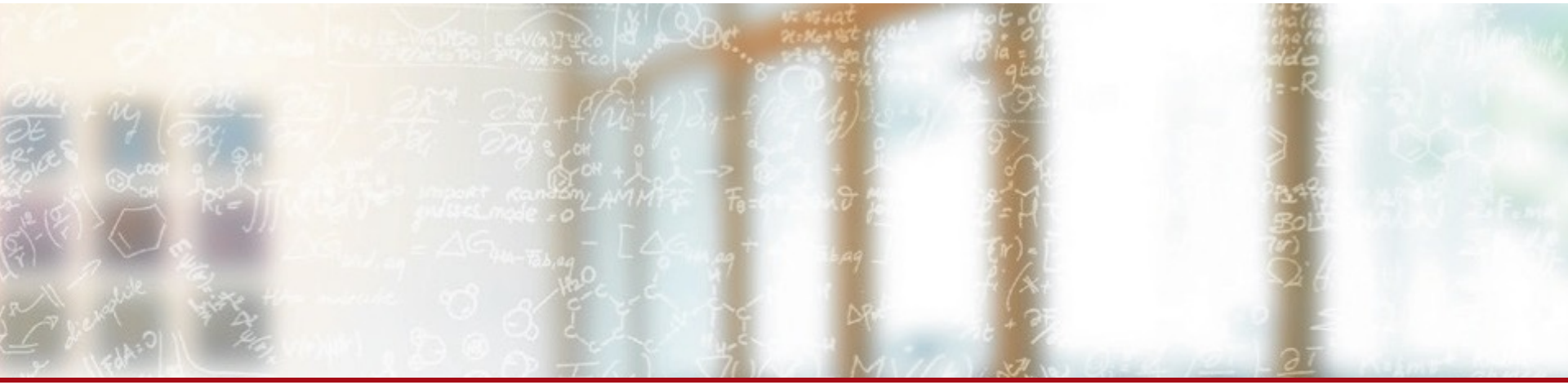
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Name Resolution and Type Deduction

Advanced C++ for HPC

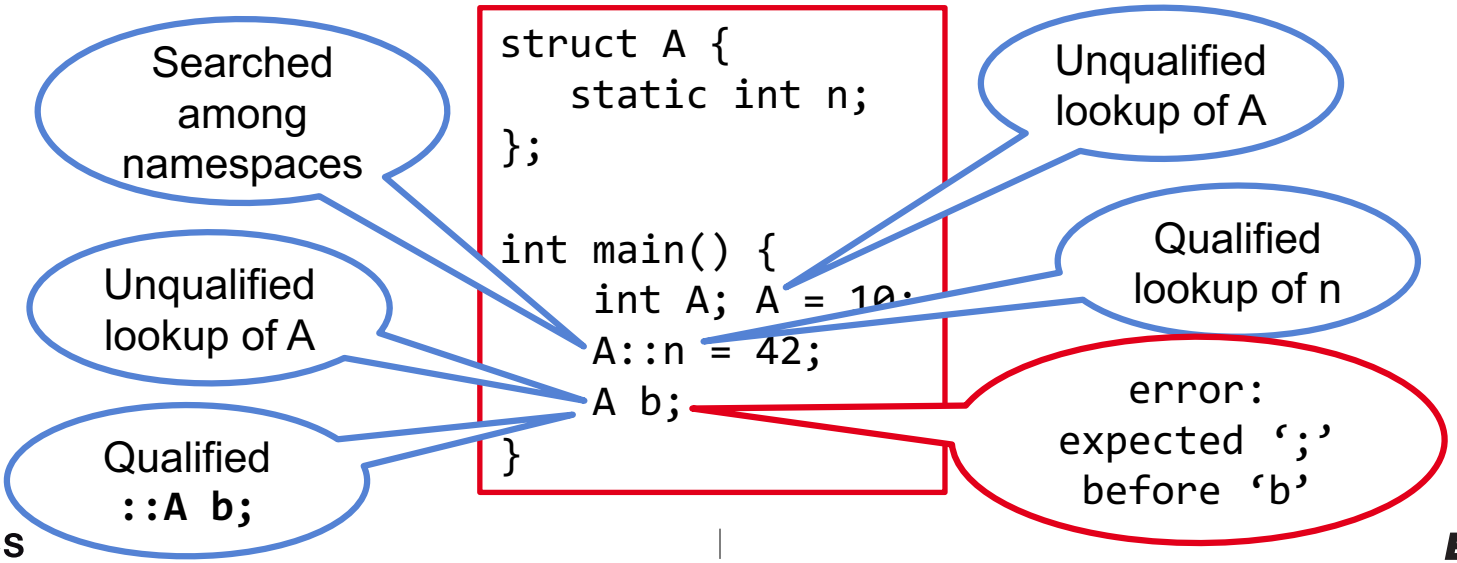
CSCS, Lugano, CH

What is name resolution?

- When a name is found the suitable declaration is looked for
  - Important for function but also variables
- First it looks at the arguments
  - Argument Dependent Lookup (ADL)
- Then employs template argument deduction
- If there is not a single solution: Overload resolution
- If everything fails a compilation error is produced
  - Cannot find a matching candidate
  - Ambiguity among multiple candidates

What is name resolution?

- When a name is found the suitable declaration is looked for
- Unqualified Name Lookup
  - For names not on the right of scope resolution operator `::`
- Qualified Name Lookup
  - For names on the right of scope resolution operator `::`
  - The “thing” on the left is looked for among
    - namespaces, class types, enumerators, templates



ADL

- Why does this work?
- `cout << “Hello\n”;`
- Function equivalent (*infix* notation for shift operator)
- `operator<<(std::cout, “Hello\n”);`
  - Now `operator<<` is looked for in namespace `std`
- Why is this not working? (without using namespace std)
- `std::cout << endl;`
- `operator<<(std::cout, endl)`
- ADL looks for the function names not arguments

ADL

- Used to find candidates for function expressions

```
#include <vector>
#include <algorithm>

template <typename IT, typename F>
void for_each(IT begin, IT end, F f) {
    std::for_each(begin, end, f);
};
int main(){
    std::vector<int> v(10);
    for_each(v.begin(), v.end(),
             [](int &i){ ++i;});
}
```

ADL

- Used to find candidates for function expressions

```
#include <vector>
#include <algorithm>

template <typename IT, typename F>
void for_each(IT begin, IT end, F f) {
    std::for_each(begin, end, f);
};
int main(){
    std::vector<int> v(10);
    ::for_each(v.begin(), v.end(),
               [](int &i){ ++i;});
}
```

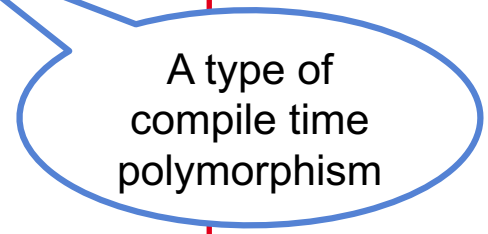
Qualified lookup of for_each

ADL

- Can be used for good

```
namespace X {
    struct A {};
    void foo(A) {
        std::cout << "I'm X\n";
    }
}

namespace Y {
    struct A {};
    void foo(A) {
        std::cout << "I'm Y\n";
    }
}
```

```
template <typename T>
void bar(T const& x) {
    foo(x);
}

int main() {
    X::A xa;
    Y::A ya;

    bar(xa);
    bar(ya);

}
```



Overload Resolution

- To determine which function to call
- Before this can happen:
  - Name look up to find all possible candidates for a call
- Select the best match
  - The most specialized

```
/* 1 */ int    foo(int, int) {return 0;}

/* 2 */ double foo(float, int) {return 0.;}

template <typename T>
/* 3 */ char foo(T, int) {return '\n';}
```

```
/* A */ foo(3,4);       /* 1 */

/* B */ foo(3,4.3);     /* 1 */

/* C */ foo(3.4,4);     /* 3 */

/* D */ foo(3.4f,4);    /* 2 */

/* E */ foo(3.4,4.5);   /* 3 */

/* F */ foo(3.4f,4.5f); /* 2 */
```

Overload Resolution

- To determine which function to call
- Before this can happen:
  - Name look up to find all possible candidates for a call
- Select the best match
  - The most specialized

```
/* 1 */ int    foo(int, int) {return 0;}

/* 2 */ double foo(float, int) {return 0.;}
```

There are not preferred conversions

```
/* A */ foo(3,4);       /* 1 */

/* B */ foo(3,4.3);     /* 1 */

/* C */ foo(3.4,4);     /* ERROR */

/* D */ foo(3.4f,4);    /* 2 */

/* E */ foo(3.4,4.5);   /* ERROR! */

/* F */ foo(3.4f,4.5f); /* 2 */
```

Overload Resolution

- To determine which function to call
- Before this can happen:
  - Name look up to find all possible candidates for a call
- Select the best match
  - The most specialized

```
/* 1 */ int    foo(int, int) {return 0;}

/* 2 */ double foo(int, float) {return 0.;}
```

```
/* A */ foo(3,4);       /* 1 */

/* B */ foo(3,4.3);     /* ERROR! */

/* C */ foo(3.4,4);     /* 1 */

/* D */ foo(3.4f,4);    /* 1 */

/* E */ foo(3.4,4.5);   /* ERROR! */

/* F */ foo(3.4f,4.5f); /* 2 */
```

Overload Resolution

- To determine which function to call
- Before this can happen:
  - Name look up to find all possible candidates for a call
- Select the best match
  - The most specialized

```
/* 1 */ int    foo(int, int) {return 0;}

/* 2 */ double foo(int, float) {return 0.;}

template <typename T>
/* 3 */ char foo(T, int) {return '\n';}
```

```
/* A */ foo(3,4);       /* 1 */

/* B */ foo(3,4.3);     /* ERROR! */

/* C */ foo(3.4,4);     /* 3 */

/* D */ foo(3.4f,4);    /* 3 */

/* E */ foo(3.4,4.5);   /* 3 */

/* F */ foo(3.4f,4.5f); /* ERROR! */
```

auto & decltype

- Uses template deduction mechanism
- Why auto?
  - Return type of makers

```
tuple<int, float, char, std::string> t1{42, 3.14f, 'z', string{"hello"}};
auto t2 = make_tuple(42, 3.14f, 'z', string{"hello"});
```

  - Enabling simple compile time polymorphism w/o traits

```
template <typename T>
auto foo(T const& f) {
    auto x = f.value();
    return x;
}
```

vs.

```
// Requires T::value_type to exist
template <typename T>
typename T::value_type foo(T const& f) {
    typename T::value_type x = f.value();
    return x;
}
```

decltype

- Retrieve the type of an expression

```
decltype(10)
int
```

```
int x;
decltype(x)
int
```

```
decltype(x+10)
int
```

```
decltype((x))
int&
```

```
decltype((x+10))
int
```

```
int* foo(float, double) {return nullptr;}
decltype(foo)
int*(float, double)
```

```
int* (*f)(float, double) = nullptr;
decltype(f)
int*(*)(float, double)
```

```
int* foo(float, double) {return nullptr;}
decltype(foo(float{}, double{}));
int*
```

```
int* (*f)(float, double) = nullptr;
decltype(f(float{}, double{}));
int*
```

```
int* foo(float&, double&) {return nullptr;}
decltype(foo(float{}, double{}));
COMPILATION ERROR
```

```
int* foo(int, int&) {return nullptr;}
decltype(foo(declval<int>(), declval<int&>()));
int*
```

std::declval implementation

- Declaration

```
template<class T>
typename std::add_rvalue_reference<T>::type declval();
```

- Definition not provided!

decltype(auto)

- Syntax
  - **decltype(auto)** */initializer|function/*
- Mechanism
  - Applies auto to the initializer|return expression
  - Then, applies decltype

```
int x = 10;
int& z = x;
decltype(auto) y = x;

decltype(auto) w = z;
```

decltype(y) is int

decltype(w) is int&

- Why decltype(auto)? Reusing templates w/o traits

```
template <typename Array, typename Pred>
void mark_if(Array & a, unsigned index, Pred const& pred) {
    typename Array::value_type& ref = a[index];
    if (pred(ref.value())) {
        ref.mark() = true;
    }
}
```

What if operator[] returns a proxy, or if Array does not have ::value_type?

decltype(auto)

- Syntax
  - **decltype(auto)** */initializer|function/*
- Mechanism
  - Applies auto to the initializer|return expression
  - Then, applies decltype

```
int x = 10;
int& z = x;
decltype(auto) y = x;

decltype(auto) w = z;
```

- Why decltype(auto)? Reusing templates w/o traits

```
template <typename Array, typename Pred>
void mark_if(Array & a, unsigned index, Pred const& pred) {
    decltype(auto) ref = a[index];
    if (pred(ref.value())) {
        ref.mark() = true;
    }
}
```