

Department I - C Plus Plus

Modern and Lucid C++ Advanced
for Professional Programmers

Week 6 – Advanced Templates

Thomas Corbat / Felix Morgner
Rapperswil, 30.03.2020
FS2020






- **Week 5 Recap**
- **Static vs. Dynamic Polymorphism**
- **Templates Recap**
- **Deduction Guides**
- **Substitution Failure Is Not An Error (SFINAE)**

- **Participants should...**


- ... get a deeper understanding of why template code (static polymorphism) is faster than virtually dispatched code (dynamic polymorphism)
- ... have refreshed their general template knowledge
- ... can implement deduction guides for class template argument deduction
- ... are able to eliminate function template overloads by applying SFINAE



Recap Week 5




 



Static vs. Dynamic Polymorphism




 



Templates Recap




 



Deduction Guides




 



Substitution Failure Is Not An Error (SFINAE)




 

Summary





Pack Expansion (Recap)



Varadic Template Instances Unfolded (Recap)

Recap Week 5

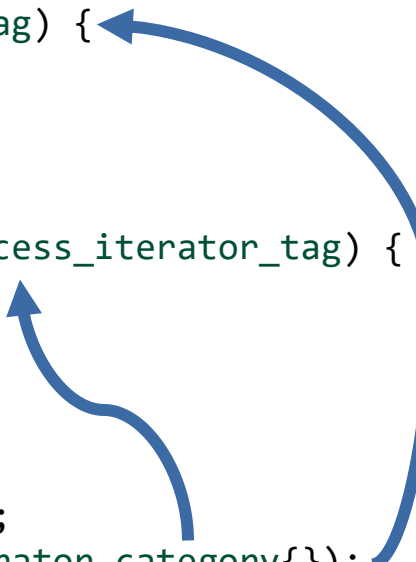


- Simple function overload resolution determines which implementation to use

```
template<typename InputIter, typename Distance>
void advanceImpl(InputIter & i, Distance d, std::input_iterator_tag) {
    while (d--) { i++; }
}

template<typename RandomAccessIter, typename Distance>
void advanceImpl(RandomAccessIter & i, Distance d, std::random_access_iterator_tag) {
    i += d;
}

template<typename InputIter, typename Distance>
void advance(InputIter & i, Distance n) {
    typename std::iterator_traits<InputIter>::difference_type d = n;
    advanceImpl(i, d, typename std::iterator_traits<InputIter>::iterator_category{});
}
```



```
::advance(iter, 15);
```

- Using `boost/operators.hpp` shortens definition

Pass own type
CRTP = Curiously Recurring Template Parameter

Explicit
Constructor

Reuse
predefined
type

```
struct IntIteratorBoost
: boost::input_iterator_helper<IntIteratorBoost, int> {

    explicit IntIteratorBoost(int start = 0)
    : value { start } {}

    bool operator==(IntIteratorBoost const & r) const {
        return value == r.value;
    }

    value_type operator*() const { return value; }

    IntIteratorBoost & operator ++() {
        ++value;
        return *this;
    }

private:
    value_type value;
};
```

Inherit to obtain types and operations
(through CRTP)

`operator==`
required

```
struct IntInputter {  
    using iterator_category = std::input_iterator_tag;  
    using value_type = int;  
    /* Other Member Types Omitted */  
  
    IntInputter();  
    explicit IntInputter(std::istream & in)  
        : input { in } {}  
    value_type operator*();  
    IntInputter & operator++() {  
        return *this;  
    }  
    IntInputter operator++(int) {  
        IntInputter old{*this};  
        ++(*this);  
        return old;  
    }  
    bool operator==(IntInputter const & other) const;  
    bool operator!=(IntInputter const & other) const {  
        return !(*this == other);  
    }  
private:  
    std::istream & input;  
};
```

Default Constructor
for EOF

++ does nothing

Equal only if both
EOF

Caller must guarantee survival
of object, otherwise "dangling"
reference!

Static vs. Dynamic Polymorphism



Pros of static polymorphism

- Happens at compile-time
- Faster execution time
 - No dynamic dispatch required
 - Easier to optimize (inline)
- Type checks at compile-time

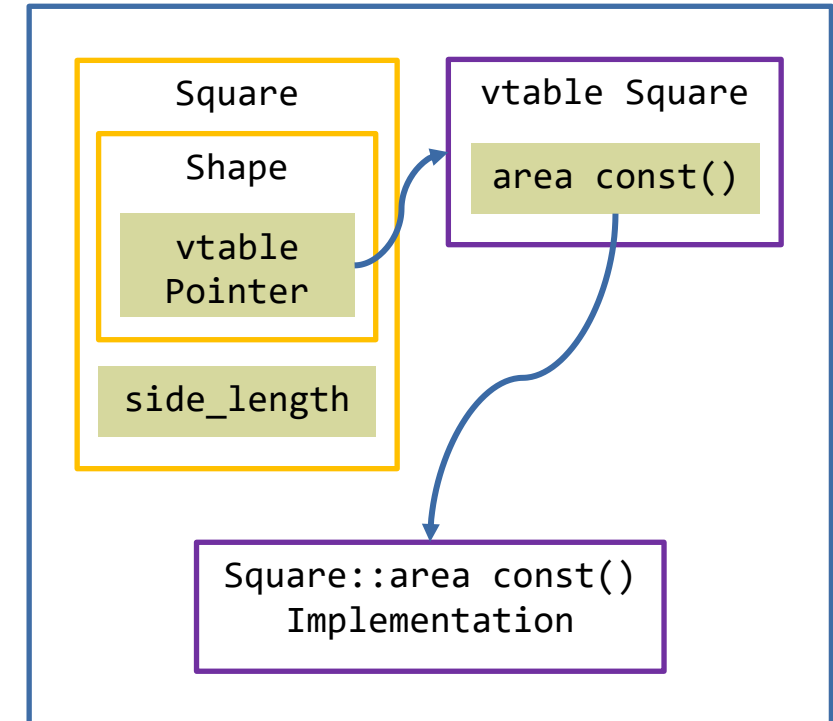
Cons of static polymorphism

- Longer compile-times
- Template code has to be known when used
- Larger binary size
 - Copy of the used parts for each (template) instance

- A polymorphic call of a virtual function requires lookup of the target function

```
struct Shape {  
    virtual unsigned area() const = 0;  
    virtual ~Shape();  
};  
  
struct Square : Shape {  
    Square(unsigned side_length)  
        : side_length{side_length} {}  
    unsigned area() const {  
        return side_length * side_length;  
    }  
    unsigned side_length;  
};
```

```
decltype(auto) amountOfSeeds(Shape const & shape) {  
    auto area = shape.area();  
    return area * seedsPerSquareMeter;  
};
```



- Article on this topic: <http://eli.thegreenplace.net/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c>

- Non-virtual calls directly call the target function

Argument Type

```
struct Square {  
    Square(unsigned side_length)  
        : side_length{side_length} {}  
    unsigned area() const {  
        return side_length * side_length;  
    }  
    unsigned side_length;  
};
```

Template

```
template<typename ShapeType>  
decltype(auto) amountOfSeeds(ShapeType const & shape) {  
    auto area = shape.area();  
    return area * seedsPerSquareMeter;  
};
```

Instance

```
decltype(auto) amountOfSeeds(Square const & shape) {  
    auto area = shape.area();  
    return area * seedsPerSquareMeter;  
};
```

Square

side_length

Square::area const()
Implementation

- **Object is smaller**

- No vtable

- **Compiler flag for (cl.exe)**

- /d1reportSingleClassLayout<ClassName>
- /d1reportAllClassLayout

```
class Shape          size(1):
    +---
    +---

class Square         size(4):
    +---
    0    | +--- (base class Shape)
          | +---
    0    | side_length
          +---
```

```
class Shape          size(4):
    +---
    0    | {vfptr}
          +---

Shape::$vtable@:
          | &Shape_meta
          | 0
    0    | &Shape::area
    1    | &Shape::{dtor}

class Square         size(8):
    +---
    0    | +--- (base class Shape)
    0    | | {vfptr}
          | +---
    4    | side_length
          +---

Square::$vtable@:
          | &Square_meta
          | 0
    0    | &Square::area
    1    | &Square::{dtor}
```

- **Copy-pasting at compile-time**

- Instances for Square, Circle and Triangle
- The optimizer might get rid of them

```
template<typename ShapeType>
decltype(auto) amountOfSeeds(ShapeType const & shape) {
    auto area = shape.area();
    return area * seedsPerSquareMeter;
};
```

```
unsigned amountOfSeeds(Square const & shape) {
    auto area = shape.area();
    return area * seedsPerSquareMeter;
};
```

```
},
```

```
Circle const & shape) {
    a();
    rSquareMeter;
```

```
},
```

```
Triangle const & shape) {
    a();
    rSquareMeter;
```

Templates Recap



- **Template declaration**

- template Keyword
- Template Parameters

template
Keyword

Template
Parameters

```
template<typename ShapeType>  
decltype(auto) amountOfSeeds(ShapeType const & shape) {  
    auto area = shape.area();  
    return area * seedsPerSquareMeter;  
};
```

- **Function is implicitly inline**

- **Template arguments might be deduced from the function call arguments**

ShapeType is deduced
to be Rectangle

```
Rectangle r{5, 8};  
auto seeds = amountOfSeeds(r);
```

Explicitly specified
to be Rectangle

```
Rectangle r{5, 8};  
auto seeds = amountOfSeeds<Rectangle>(r);
```


Template Template
Parameter

Template Type
Parameter

Template Non-Type
Parameter

```
template<template<typename, unsigned> typename Container, typename Target, std::size_t N>
Target extractMiddleElement(Container<Target, N> & container) {
    using std::swap;
    Target middleElement{};
    swap(container.at(N / 2), middleElement);
    return middleElement;
}
```

```
std::array<int, 3> values{1, 2, 3};
extractMiddleElement(values);
```

```
Container => std::array
Target    => int
N         => 3
```

```
BoundedBuffer<int, 3> values{1, 2, 3};
extractMiddleElement(values);
```

```
Container => BoundedBuffer
Target    => int
N         => 3
```

- Before C++17 template template parameter were declared with «template <...> class»

- **We could also implement `extractMiddleElement` differently**

- Instead of `N` we might use `size()`
- Instead of `Target` we have the member type `value_type` of `Container`

- **That changes the Concept of the parameter. How?**

Container must have:

- member type `value_type`
- `size()` member function

It does not need to be a template with type and unsigned parameter anymore.

- **The member type `value_type` is a dependent type**

- The compiler does not know whether `Container::value_type` is a member type, function or variable
- To tell the compiler that it is a type the `typename` keyword is required (there is also a `template` keyword for cases where the member is a template)

```
template<typename Container>
auto extractMiddleElement(Container & container) {
    typename Container::value_type middleElement{};
    std::swap(container.at(container.size() / 2), middleElement);
    return middleElement;
}
```

- In specific cases the number of template parameters might not be fix/known upfront
- Thus the template shall take an arbitrary number of parameters

- **Example:**

```
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
    std::cout << first;
    if (sizeof...(Types)) {
        std::cout << ", ";
    }
    printAll(rest...);
}
```

- **Syntax (ellipses everywhere): ...**

- ... in template parameter list for an arbitrary number of template parameters (Template Parameter Pack)
- ... in function parameter list for an arbitrary number of function arguments (Function Parameter Pack)
- ... after sizeof to access the number of elements in template parameter pack
- ... in the variadic template implementation after a pattern (Pack Expansion)

- Templates allow generic programming in C++
- A template is instantiated for a specific set of template arguments

Type
Parameter

Non-Type
Parameter

```
template<typename Freight, unsigned Space>
struct Carriage {
    std::array<Freight, Space> cargo{};
};

decltype(auto) createSmallTankWagon() {
    return Carriage<Oil, 1>{};
}
```

Creates
Template Instance

```
struct Carriage {
    std::array<Oil, 1> cargo{};
};
```



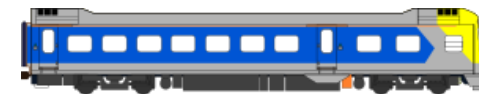
- Templates can be (partially) specialized
- Liskov's Substitution Principle does not apply for specializations, i.e. a specialized template does not need to satisfy the interface of the base template!

```
template<typename Freight, unsigned Space>
struct Carriage;

template<unsigned Space>
struct Carriage<Passenger, Space> {
    unsigned const doors{4};
}

decltype(auto) createPassengerWagon() {
    return Carriage<Passenger, 100>{};
}
```

```
struct Carriage {
    unsigned const doors{4};
}
```



- When a template is instantiated the compiler has to decide whether to use the base template or one of its specializations

● Syntax

```
template<[Parameters]>  
Type name [= initialization];
```

- Can be specialized
- Usually constexpr

● Purpose

- Compile-time predicates and properties of types
- Usually applied in template meta programming
- Before C++14 it was necessary to create a class template with a static member variable
 - Now less code is required for the same effect

```
template<typename T>  
constexpr T pi = T(3.1415926535897932385);  
  
template<typename T>  
constexpr bool is_integer = false;  
  
template<>  
constexpr bool is_integer<int> = true;
```

Deduction Guides



- Class template arguments can usually be determined by the compiler

```
template <typename T>
struct Box {
    Box(T content)
        : content{content}{}
    T content;
};

int main() {
    Box<int> b0{0}; //Before C++17
    Box      b1{1}; //Since C++17
}
```

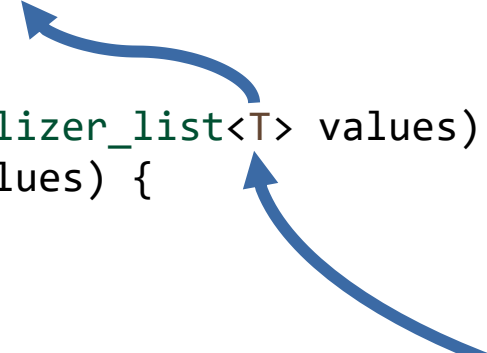
- The behavior is similar to pretending as if there was a factory function for each constructor

```
template <typename T>
Box<T> make_box(T content) {
    return Box<T>{content};
}
```

```
auto gift = make_box(teddy);
```


- In the following example the only template parameter is T, which can be deduced from `std::initializer_list<T>`

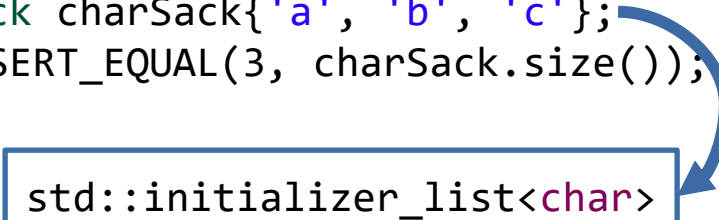
```
template <typename T>
class Sack {
    //...
    Sack(std::initializer_list<T> values)
        : theSack(values) {
    }
    //...
};
```



A blue arrow originates from the `std::initializer_list<T>` parameter in the `Sack` constructor and points to the `T` in the `template <typename T>` declaration, illustrating how the compiler deduces the template parameter from the argument type.

```
void testImplicitDeductionGuide () {
    Sack charSack{'a', 'b', 'c'};
    ASSERT_EQUAL(3, charSack.size());
}
```


`std::initializer_list<char>`



A blue arrow originates from the `{'a', 'b', 'c'}` argument in the `charSack` initialization and points to the `std::initializer_list<char>` box, which then points to the `char` in the `Sack` constructor parameter, showing the deduction of the template parameter `T` as `char`.

- There is no direct relation from `Iter` to `T` (constructor parameters to template parameter)

```
template <typename T>
struct BoundedBuffer {
    //...
    template <typename Iter>
    BoundedBuffer(Iter begin, Iter end);
    //...
};
```



```
void testDeductionFromIterators() {
    std::vector values{3, 1, 4, 1, 5, 9, 2, 6};
    BoundedBuffer buffer{begin(values), end(values)};
    ASSERT_EQUAL(values.size(), buffer.size());
}
```

```
error: class template argument deduction failed:
      Sack aSack(begin(values), end(values));
```

- **User-defined deduction guides can be specified in the same scope as the template**
 - Usually, after the template definition itself

```
TemplateName(ConstructorParameters) -> TemplateID;
```

- **Might be necessary for complex cases, e.g. template constructors if the constructor template parameters do not map directly to the class template parameters**
- **The deduction guide can be (and usually is) a template itself**
- **It looks similar to a free-standing constructor**
- **Unfortunately, C_{develop} does not recognize the deduction guides yet**

Template declaration for
Iter

```
template <typename Iter>  
BoundedBuffer(Iter begin, Iter end) -> BoundedBuffer<typename std::iterator_traits<Iter>::value_type>;
```

Constructor signature

Deduced template instance

- **Test for deducing template argument from iterator works**

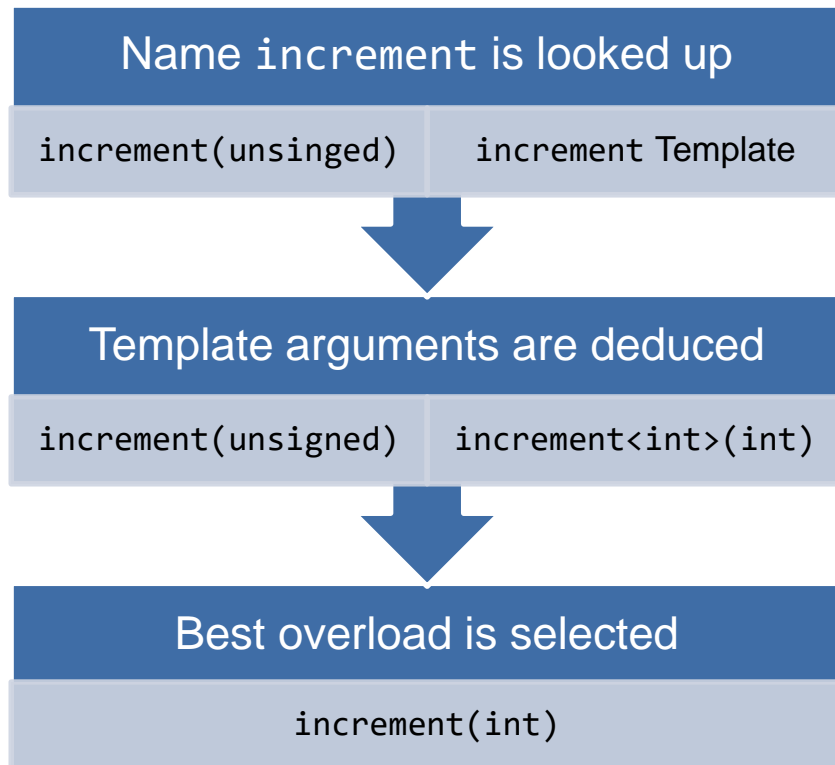
```
void testDeductionFromIterators() {  
    std::vector values{3, 1, 4, 1, 5, 9, 2, 6};  
    BoundedBuffer buffer{begin(values), end(values)};  
    ASSERT_EQUAL(values.size(), buffer.size());  
}
```

Substitution Failure Is Not An Error (SFINAE)



- What do you expect is the return value of the program on the right?

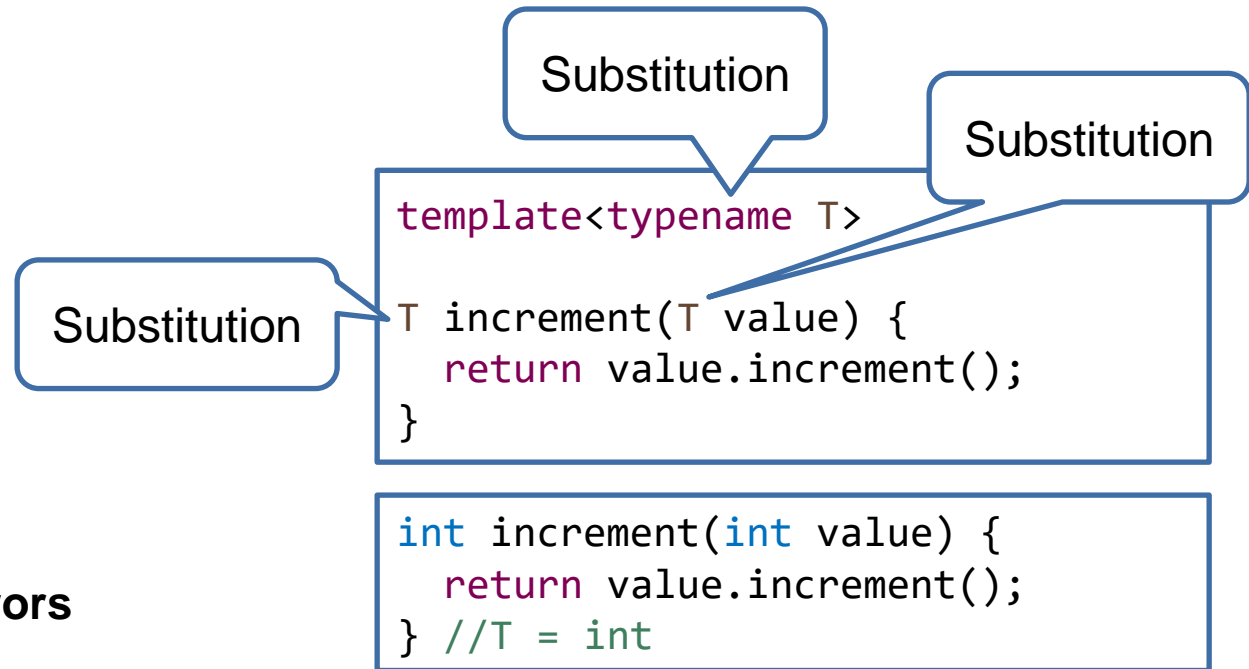
- None, the program does not compile!



```
unsigned increment(unsigned i) {  
    return i++;  
}  
  
template<typename T>  
T increment(T value) {  
    return value.increment();  
}  
  
int main() {  
    return increment(42);  
}
```

```
error: request for member 'increment' in 'value',  
which is of non-class type 'int'
```

- During overload resolution the template parameters in a template declaration are substituted with the deduced types
 - This may result in template instances that cannot be compiled
 - Or otherwise suboptimal selection
- If the substitution of template parameter fails that overload candidate is discarded
- Substitution failure might happen in
 - Function return type
 - Function parameter
 - Template parameter declaration
 - And expressions in the above
- Errors in the instance body are still errors



```
template<typename T>
auto increment(T value) -> decltype(value.increment()) {
    return value.increment();
}
```

- We can break the return type
- If we tell the compiler to use the type of `value.increment()` as return type for `increment<int>`
 - That type cannot be determined during substitution

```
error: no matching function for call to 'increment(int)'
    increment(42);
               ^
```

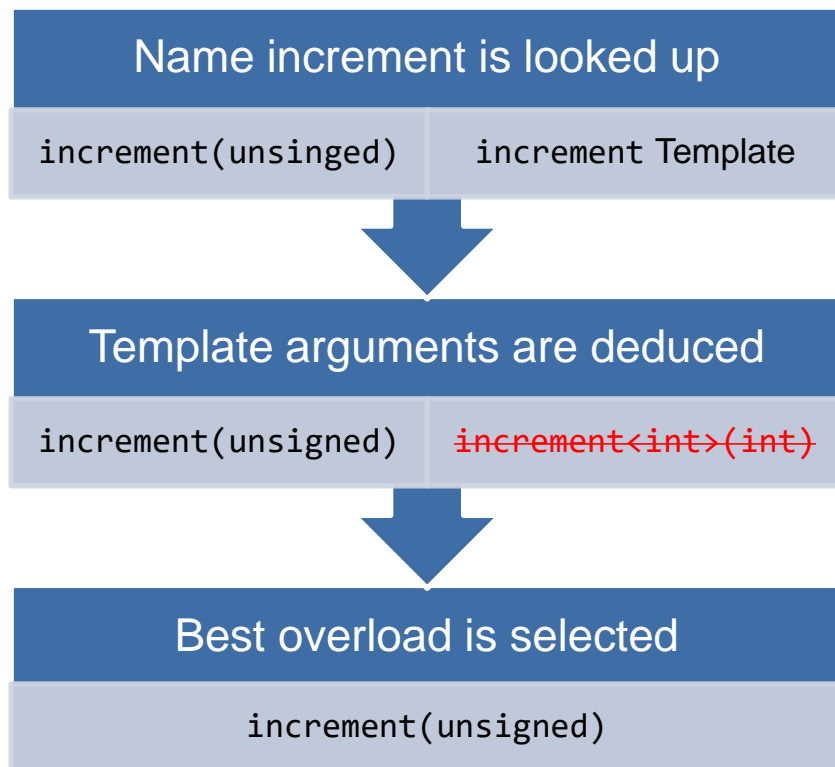
```
note: candidate: template<class T> decltype (value.increment()) increment(T)
    auto increment(T value) -> decltype(value.increment()) {
        ^~~~~~
```

note: template argument deduction/substitution failed:

In substitution of 'template<class T> decltype (value.increment()) increment(T) [with T = int]':

```
error: request for member 'increment' in 'value', which is of non-class type 'int'
    auto increment(T value) -> decltype(value.increment()) {
```


- Since there is a problem during substitution that overload is discarded



- Now the result is 42


```
unsigned increment(unsigned i) {  
    return i++;  
}  
  
template<typename T>  
auto increment(T value) ->  
    decltype(value.increment()) {  
    return value.increment();  
}  
  
int main() {  
    return increment(42);  
}
```

- This approach, using `decltype(...)` as trailing return type, is infeasible in general
 - Function might have return type void
 - It is not elegant for complex bodies

- **Let's examine our example again**
 - We want the increment template to be selected only for class type arguments
- **There exists a template `std::is_class<T>`**
 - contains static constexpr bool value;
 - value is true if T is a class, false otherwise
- **Variable template: `std::is_class_v<T>`**
 - Type bool (direct access of `::value`)
- **Can we apply this to the increment template directly?**
 - No, either value (true or false) is still valid
 - We need something to create an error in the type of the function

```
template<typename T>
T increment(T value) {
    return value.increment();
}

int main() {
    return increment(42);
}
```



```
#include <type_traits>

struct S {};

int main() {
    std::is_class<S>::value; //true
    std::is_class<int>::value; //false
}
```

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

- The `std::enable_if_t` template takes an expression and a type
 - If the expression evaluates to true `std::enable_if_t` represents the given type
 - Otherwise it does NOT represent a type

```
int main() {  
    std::enable_if_t<true, int> i;          //int  
    std::enable_if_t<false, int> error;    //no type  
}
```



- Inside `std::enable_if`


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

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

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

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

```
 T increment( T value) {  
    return value.increment();  
}
```

 Spots to apply enable_if (SFINAE)

● Possibilities

```
template<typename T>  
std::enable_if_t<std::is_class<T>::value, T> increment(T value) {  
    return value.increment();  
}
```

```
template<typename T>  
T increment(std::enable_if_t<std::is_class<T>::value, T> value) {  
    return value.increment();  
}
```

enable_if as parameter
impairs type deduction


```
template<typename T, typename = std::enable_if_t<std::is_class<T>::value, void>>  
T increment(T value) {  
    return value.increment();  
}
```

would be void per default

- **Example: Box-Container with**

- Default constructor
- Copy constructor
- Move constructor
- Size constructor

```
Box<MemoryOperationCounter> b{1};
```



```
template <typename T>
struct Box {
    Box() = default;
    Box(Box const & box)
        : items{box.items}{}
    Box(Box && box)
        : items{std::move(box.items)} {}
    explicit Box(size_t size)
        : items(size) {}
    //...
private:
    std::vector<T> items{};
};
```

- What if we replace the copy/move constructors with a forwarding constructor?

```
Box<MemoryOperationCounter> b{1};
```

```
template <typename T>
struct Box {
    Box() = default;
    template <typename BoxType>
    explicit Box(BoxType && other)
        : items(std::forward<BoxType>(other).items) {}
    explicit Box(size_t size)
        : items(size) {}
    //...
private:
    std::vector<T> items{};
};
```

```
Test.cpp:14:41: error: request for member 'items' in 'std::forward<int>((* & other))',
               which is of non-class type 'int'
    : items(std::forward<BoxType>(other).items) {}
      ~~~~~^~~~~
```

- We don't want the forwarding constructor to match anything else than Boxes

- Type traits can be used to narrow down the valid calls.

```
template <typename T>
struct Box {
    Box() = default;
    template <typename BoxType, typename = std::enable_if_t<std::is_same_v<Box, BoxType>>>
    explicit Box(BoxType && other)
        : items(std::forward<BoxType>(other).items) {}
    explicit Box(size_t size)
        : items(size) {}
    //...
private:
    std::vector<T> items{};
};
```

- The standard library provides many predefined checks for type traits¹

- A trait contains a boolean value

- Usually they are available in two versions

- Example:

- std::is_same<T, U>
- std::is_same_v<T, U>

```
template <typename T, typename U>
struct is_same : false_type {
    // inherits
    // static constexpr bool value = false;
};

template <typename T>
struct is_same<T, T> : true_type {
    // inherits
    // static constexpr bool value = true;
};

template <typename T, typename U>
constexpr bool is_same_v = is_same<T, U>::value;
```

¹ https://en.cppreference.com/w/cpp/header/type_traits


```
template <typename T>
std::enable_if_t<
    std::negation_v<
        std::is_reference<T>
    >
>
consume(T && value) { /*...*/ }

//This function can only be called with rvalues
```

```
template <typename T>
int convert(T value) {
    using namespace std;
    enable_if_t<
        is_constructible_v<int, T>, int
    > converted{value};
    return converted;
}

//This function can be eliminated by SFINAE
```

```
template <typename T>
std::enable_if_t<
    std::negation_v<
        std::is_reference<T>
    >
>
consume(T && value) { /*...*/ }

//This function can only be called with rvalues
```

Correct

`std::is_reference` is `std::true_type` if `T` is a reference otherwise `std::false_type`. If the forwarding reference is initialized with an lvalue `T` is a reference. `std::negation_v` negates the true/false_type. The overload is only enabled for rvalues.

```
template <typename T>
int convert(T value) {
    using namespace std;
    enable_if_t<
        is_constructible_v<int, T>, int
    > converted{value};
    return converted;
}

//This function can be eliminated by SFINAE
```

Incorrect

This is not SFINAE! It will compile if called with something that can be used to construct an int, but the overload will be taken anyway.

Summary



- **Function calls resolved at compile-time can be much faster**
- **Deduction guides allow CTAD for constructors that do not have a direct relation to the template arguments of the class template**
- **SFINAE is used to eliminate overload candidates**

Pack Expansion (Recap)



Variadic Template Instances Unfolded (Recap)

- **Template declaration:**

```
template<typename First, typename...Types>  
void printAll(First const & first, Types const &...rest);
```

- **Implicit instantiation:**

```
int i{42}; double d{1.25}; std::string book{"Lucid C++"};  
printAll(i, f, book);
```

- **Template instance:**

```
void printAll(int const & first, double const & __rest0,  
              std::string const & __rest1) {  
    std::cout << first;  
    if (2) { //sizeof...(Types) - Number of arguments in the pack  
        std::cout << ", ";  
    }  
    printAll(__rest0, __rest1); //rest... expansion  
}
```

- **sizeof...(<PACK>) will be replaced by the number of arguments in the pack parameter**

- 0, 1, 2, ...

```
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
    //...
    printAll(rest...);
}
```

- **Pattern: rest**
- **The pattern must contain at least one pack parameter**
- **An expansion is a coma-separated list of instances of the pattern**
- **For each argument in that pack an instance of the pattern is created**
- **In an instance of the pattern the parameter pack name is replaced by an argument of the pack**

```
void printAll(int const & first, double const & __rest0, std::string const & __rest1) {
    //...
    printAll(__rest0, __rest1); //rest...
}
```

- For the call `printAll(__rest0, __rest1): printAll<double, std::string>`

```
void printAll(double const & first, std::string const & __rest0) {  
    std::cout << first;  
    if (1) { //sizeof...(Types) - Number of arguments in the pack  
        std::cout << ", ";  
    }  
    printAll(__rest0); //rest... expansion  
}
```

- For the call `printAll(<rest0>): printAll<std::string>`

```
void printAll(std::string const & first) {  
    std::cout << first;  
    if (0) { //sizeof...(Types) - Number of arguments in the pack  
        std::cout << ", ";  
    }  
    printAll(); //rest... expansion  
}
```

- What about `printAll()`?

- What about printAll()?
 - The variadic template printAll is not viable, as it requires at least one parameter
- We need a base case for the recursion

```
void printAll() {  
}
```

- Wouldn't it be feasible to just rearrange the code in the variadic template?

```
template<typename First, typename...Types>  
void printAll(First const & first, Types const &...rest) {  
    std::cout << first;  
    if (sizeof...(Types)) {  
        std::cout << ", ";  
        printAll(rest...);  
    }  
}
```

