# Disambiguation the Black Technology

Zhihao Yuan <zhihao.yuan@rackspace.com>

# Why this talk

Bjarne: when you say SFINAE I get an icy feeling down my spine.

# Why this talk

“I rarely have need of anything but the normal function binding. ”

# Why this talk

A “real world” example:

```
void WriteAll(Stream& source, bool encrypted); ❶
void WriteAll(Stream& source, std::string const& md5); ❷

fs.WriteAll(s, "c6b16a0a6582e869d59ea65c79b9a221");
```

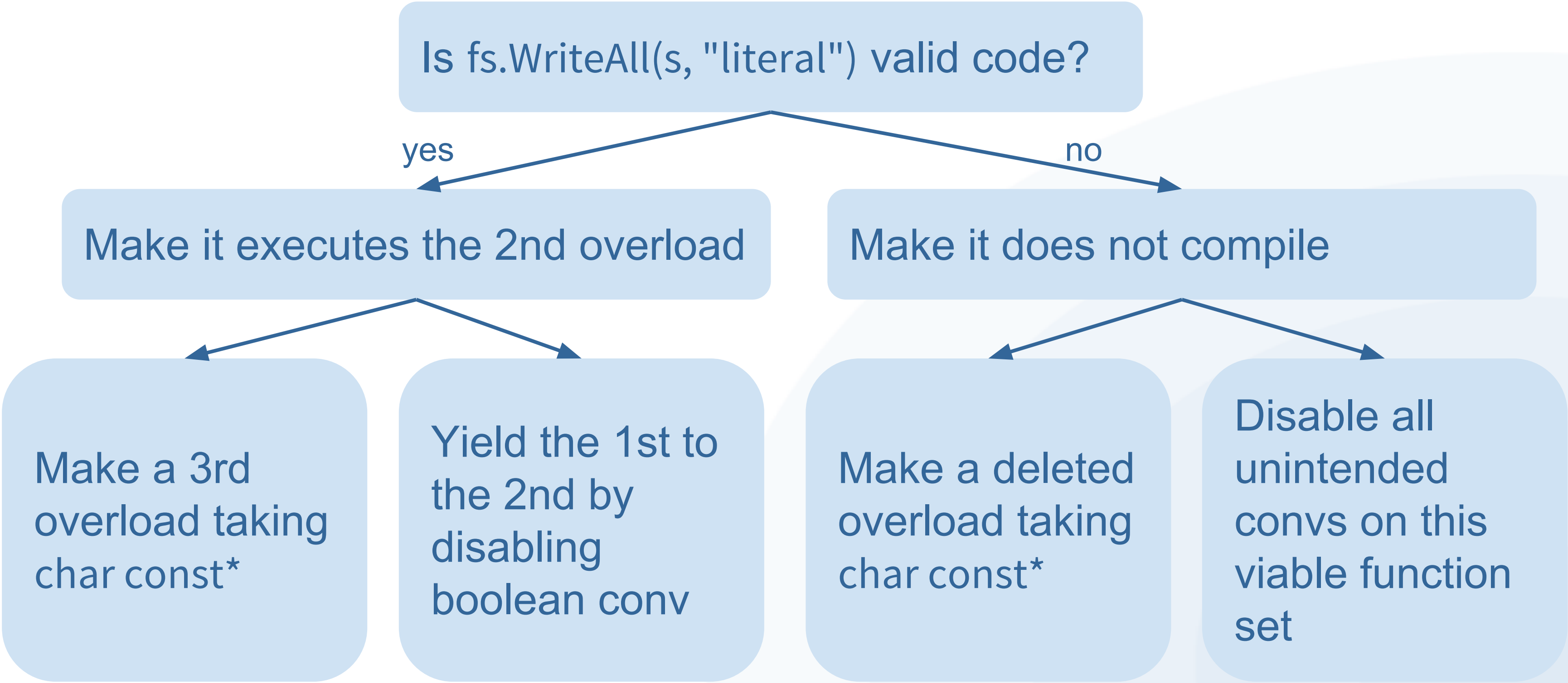
# A basic disambiguation procedure

```
void WriteAll(Stream& source, bool encrypted);
void WriteAll(Stream& source, std::string const& md5);

"c6b16a0a6582e869d59ea65c79b9a221"
char const[33]
    ⇒ char const*        // array-to-pointer, “exact match”
        ⇒ bool           // boolean conversion
        ⇒ std::string    // user-defined conversion
```

# A basic disambiguation procedure

```
void WriteAll(Stream& source, bool encrypted);
void WriteAll(Stream& source, std::string const& md5);

// Quick fix:
fs.WriteAll(s,
   std::string("c6b16a0a6582e869d59ea65c79b9a221"));
```

# A basic disambiguation procedure



# Reflection

… Why the two functions are overloaded?

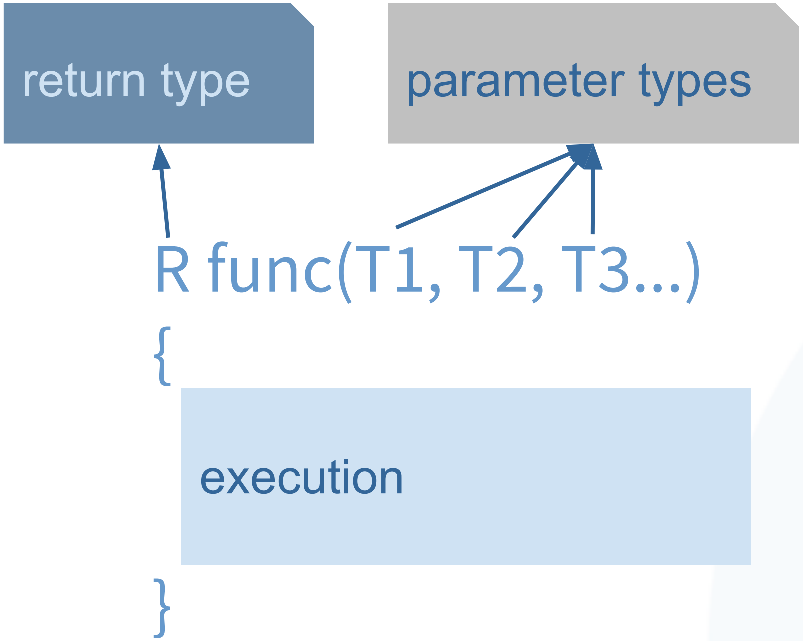
# What this talk does

- Share my thoughts on when to overload
- Introduce disambiguation techniques working for one or two types
- Introduce techniques to control overloading multiple types
- Introduce techniques to control overloading types with various relationship

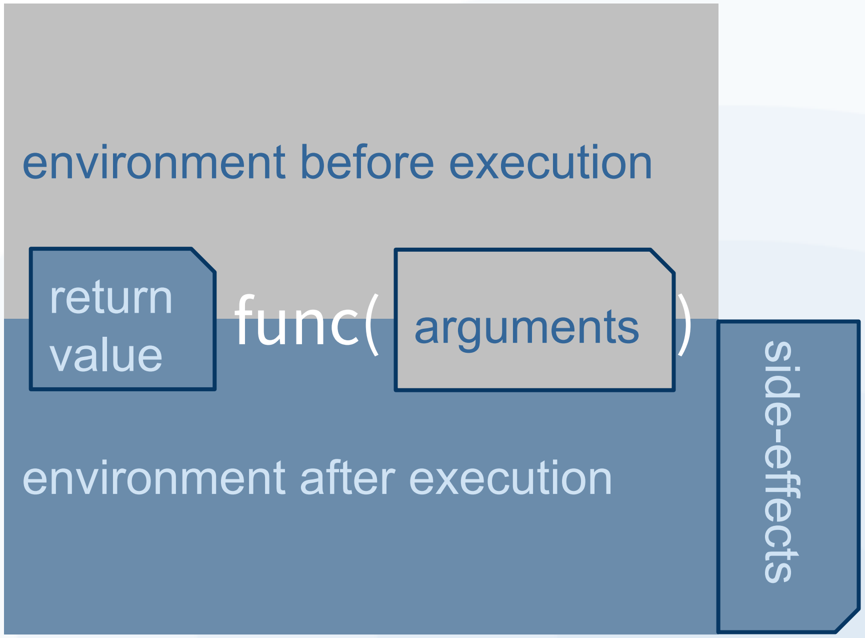
# When to overload

# The way this talk understand functions

function

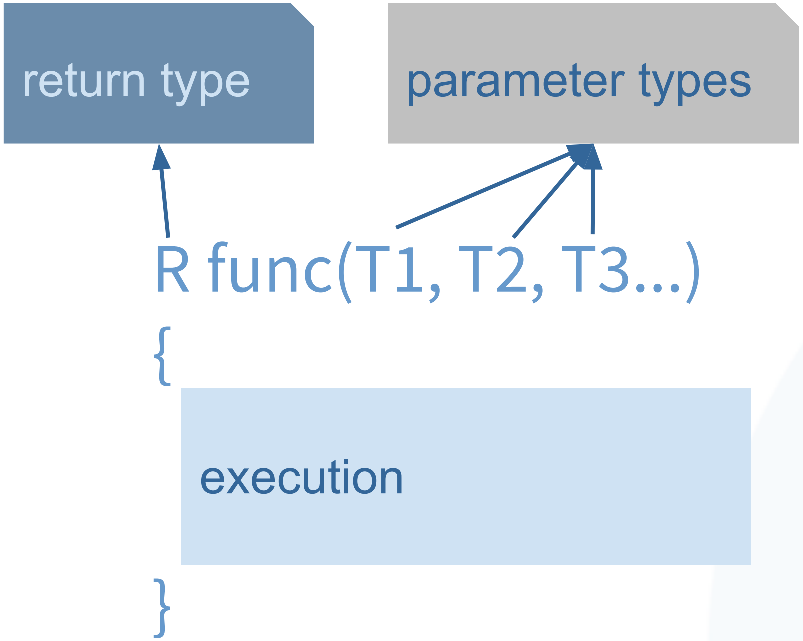



function call



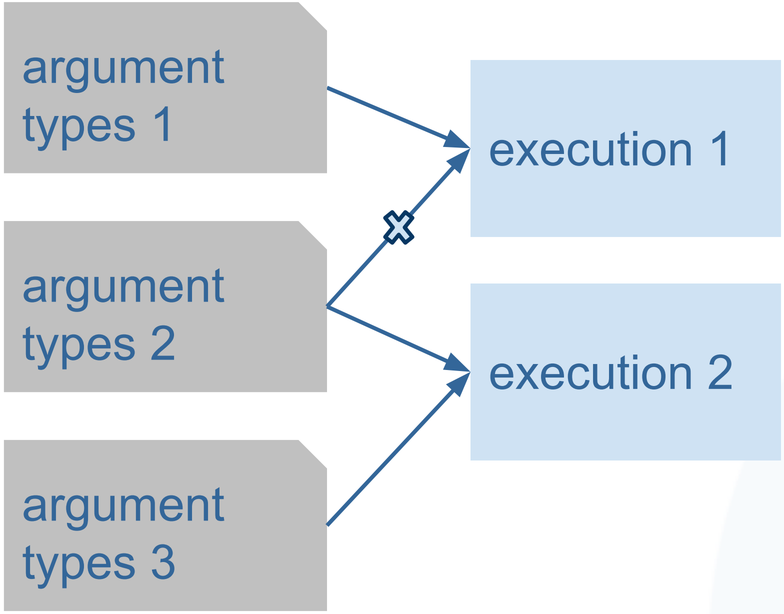

* search C++ Std for “*INVOKE*”

# Overload resolution




Overload resolution selects execution based on the argument types.

# Disambiguation

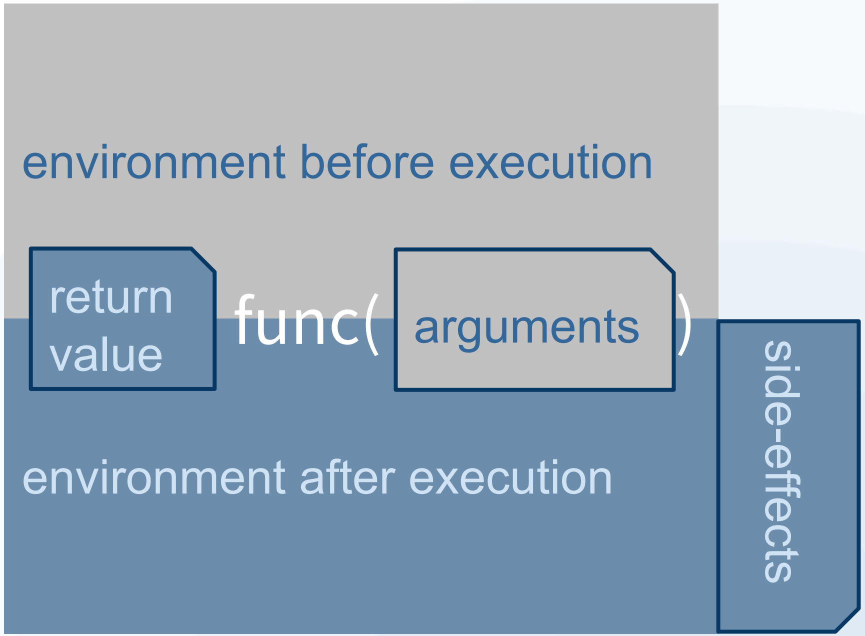



Overload resolution selects execution based on the argument types.

Disambiguation avoids unintended execution(s) **without** changing the argument types.

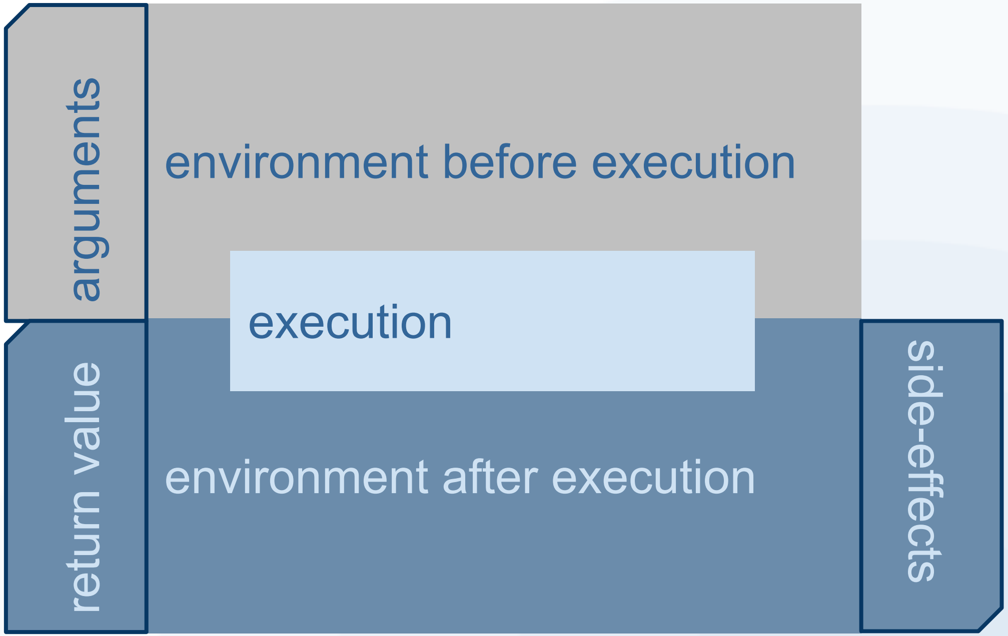
# Input and Output

The environment before execution and the states of the arguments are collectively called $Input$; the environment after execution, the states of the return value, and side-effects, are collectively called $Output$.



# Execution

An execution gives a certain output for a certain input. *



# My thoughts on overloading

Functions in the same overload set should give semantically the same output.

# My thoughts on overloading, translated

If two functions are overloaded, they should provide the same functionality.

# My thoughts on overloading, relaxed

Functions in the same overload set should give semantically the same output.

Minimal requirement: “Functions that may exist in the same viable function set …”
≠ that may accept the same number of arguments

# So…

1. Design your overload set as one function
2. You are not able to disambiguate every case
3. Design your viable function set as one function if 1) is not a choice

# Basic techniques

# Remove a subset from an overload set

$\textit{some function(signature)}$ = delete;

- Work with SFINAE *
- Do not use static_assert for this purpose

# Capture one type

```
void write(std::string const&);
void write(std::experimental::string_view);
void write(char const *);          // add an overload
```

- Does not break ABI
- Combine with = delete to shut a use off
- Downside: scalability

# Capture the other types

```
void write(std::string const&);
template <typename T>
void write(T const&);          // unspecialized template
```

- Does not break ABI
- Side-effect: implicit conversions* are turned off on this viable function set
- Make use of that side-effect: limiting the viable function set to accept only specified types

# … if you don't have an object of that type

```
void write() { write_impl(identity<Type>()); }
void write_impl(identity<char>);
```

- When an object of identity<T> participates in overload resolution instead of T, **all** implicit conversions to T do not apply, including lvalue-to-rvalue conversions.
- Use it in implementation, and be aware of cv-qualification.

# Type function identity

```
template <typename T>
struct identity
{
    typedef T type;
};
```

A proposal to standardize this:

http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3766.html

# Ambiguated template argument deduction

```
template <typename T>
void push_back(std::vector<T>& v, T t)
{
    v.push_back(std::move(t));
}

std::vector<long> v;
push_back(v, 3);          // LOL
```

# Restrict the source of deduction

```
template <typename T>
void push_back(std::vector<T>& v,
    typename identity<T>::type t)        // non-deduced ctx
{
    v.push_back(std::move(t));
}
```

- Can also be used to shut off deduction completely and enforcing passing the template argument (std::forward)

# Limit the genericity of a template

a.k.a "constrained template"

```
template <typename T>
void write(T const&);          // unconstrained

template <typename T>
void write(T const&,
  std::enable_if_t<std::is_pod<T>{}>* = 0);        // c++14
```

# Speak English!

```
template <typename T>
void write(T const&,
   std::enable_if_t<std::is_pod<T>{}>* = 0);
```

Read this in the “standard” way:

This function shall not participate in overload resolution unless T is not a POD type.

# Let’s start from scratch

# Common styles

```
template <typename T>
std::enable_if_t< … > func(T);          // return void

template <typename T>
std::enable_if_t< …, R> func(T);        // return R

template <typename T>
MyClass(T, std::enable_if_t< … >* = 0);
```

# More styles

```
template <typename T,
    std::enable_if_t< …, int> = 0>
R func(T);

template <typename T,
    typename = std::enable_if_t< … >>
R func(T);       // caution: template redefinition
                 // has workaround but, savepoint
```

# … if your function is not a template

Just make it a template:

```
template <typename>
std::enable_if_t< … > func();
```

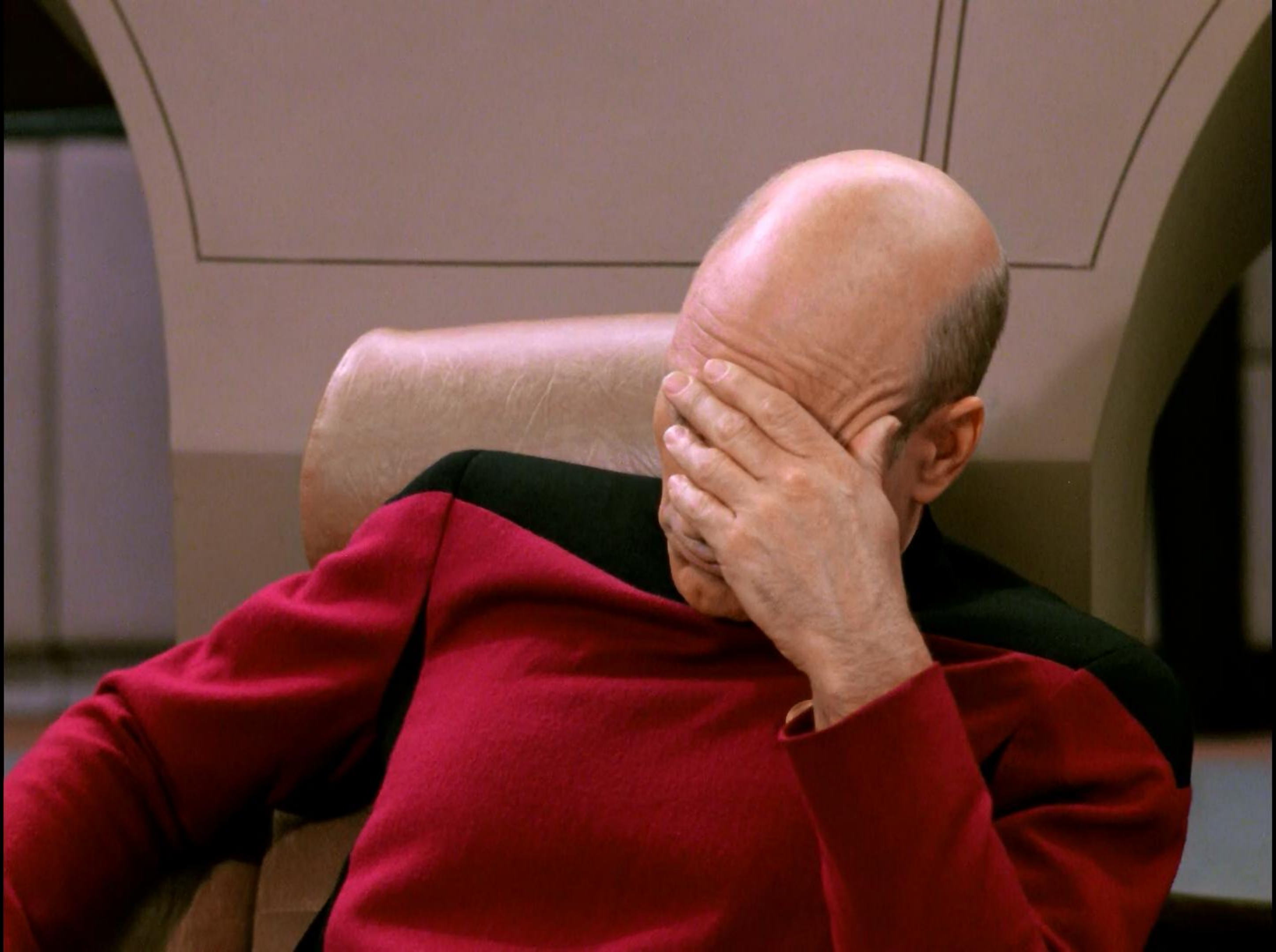
- Caveat: not been recognized as a special member function, see Eric's article: http://ericniebler.com/2013/08/07/universal-references-and-the-copy-constructo/

# SFINAE is a hammar

Gabriel: The wording about “shall not participate in overload resolution” appears to dictate a local modification of language rules […]

# … but don’t use it like a hammer

```
template <typename>
std::enable_if_t<not (std::is_same<C, std::list<V>>{} or
                      std::is_same<C, Vector>{})>
sort() { std::sort(begin(c_), end(c_)); }

template <typename>
std::enable_if_t<std::is_same<C, std::list<V>>{}>
sort() { c_.sort(); }

template <typename>
std::enable_if_t<std::is_same<C, Vector>{}>
sort() { c_.Sort(); }
```



# … just an identity dispatching

```
void sort() { sort_impl(identity<C>()); }

template <typename T>
void sort_impl(identity<T>) { std::sort(begin(c_),end(c_)); }

void sort_impl(identity<std::list<V>>) { c_.sort(); }

void sort_impl(identity<Vector>) { c_.Sort(); }
```

# Patterns seen so far

- Specific-only interface
- General-specific interface
- Constrained general-specific interface
- General-specific implementation

Each function template overload may individually choose its own type of interface.

# Extensible techniques

## Multiple constrained-general interface

# Programming templates

- Type functions
  - Control flows
    - if (std::enable_if)
    - if-else (std::conditional)
  - Type predicates
  - Type modifications and transformations
- Higher order type functions (not in std)
- Data structures (not in std)

# Contract of a type function

Does this work?

```
template <typename T>
std::make_unsigned_t<T>
to_unsigned(T);

seminumeric::bits
to_unsigned(seminumeric::integer);
```

# … picking up the topic mentioned before

1. Type function may have precondition
2. static_assert denotes a compile-time precondition violation, not a constraint
3. A function with a compile-time wide contract should not use static_assert

# Scalability

One type function replaces all overloads, everywhere

```
template <typename T>
std::enable_if_t<std::is_integral<T>{}, T>
abs(T n);

template <typename T>
std::enable_if_t<std::is_integral<T>{}, get *div_t >
div(T n);
```

# Ambiguated overloaded function templates

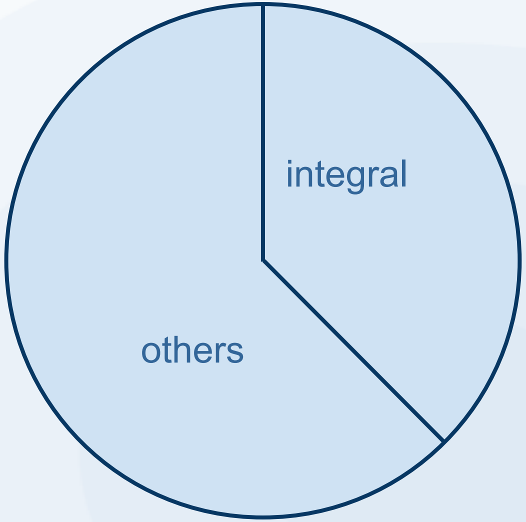
```
template <typename T>
std::enable_if_t<std::is_integral<T>{}>
write(T t);
```

# … until you want to capture the others

```
template <typename T>
std::enable_if_t<std::is_integral<T>{}>
write(T t);

template <typename T>
std::enable_if_t<not std::is_integral<T>{}>
write(T const& t);
```

# … because

1. Two function templates overloads are ambiguous only if neither is more specialized than the other
2. They accept a union set of types
3. To disambiguate them, each overload has to be constrained to accept disjointed subset of types



# Again, turn it into an overloading problem!

```
template <typename T>
void write(T&& t)
{ write_impl(std::forward<T>(t), std::is_integral<T>()); }

template <typename T>
void write_impl(T t, std::true_type);

template <typename T>
void write_impl(T const& t, std::false_type);
```

# Multiple properties

| | is_integal<T>(), | is_signed<T>(), | is_unsigned<T>() |
|---|---|---|---|
| signed integer | true_type, | true_type, | false_type |
| unsigned integer | true_type, | false_type, | true_type |
| floating point | false_type, | true_type, | false_type |
| others | false_type, | … | |

# Really extensible techniques

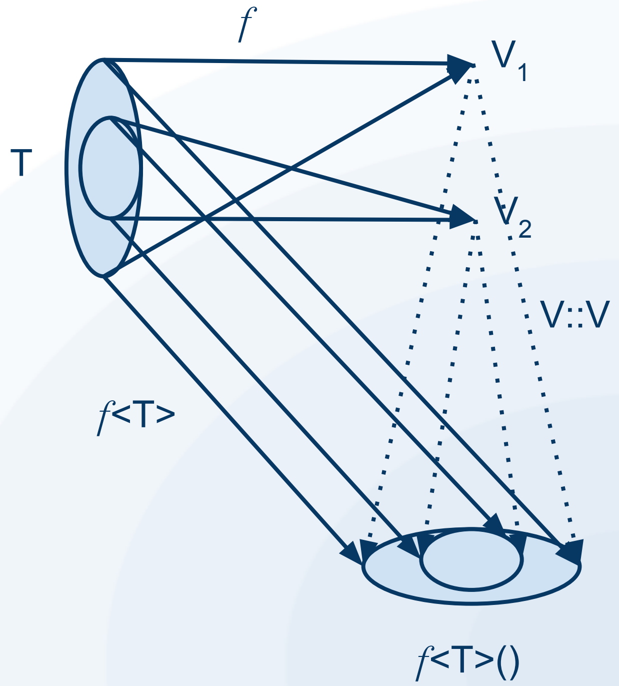
## Hierarchy type relationships

# Ruminations on identity dispatching

1. identity<T>() is an object to represent uniqueness of T
2. is_integer<T>() is an objects of one of two types to represent a boolean property of T
3. What we get if having *f*<T> with a limited number of possible return types?

# Idea behind tag dispatching

1. Let $V = f<T>$
2. Any relationship on V can be observed equivalently on T and $f<T>()$
3. Let $T_a$, $T_b \in T$, $f<T_a>()$ is convertible to $f<T_b>()$, then we consider that $T_a$ **refines** $T_b$ in terms of V.



# Conclusion

- Design overloaded functions as one function
- Design by patterns, disambiguate by patterns
- Prefer using overload resolution to solve overloading problems
- SFINAE as a last resort

# Resources

Stephan T. Lavavej: Core C++, 2 of n (Template Argument Deduction)
http://channel9.msdn.com/Series/C9-Lectures-Stephan-T-Lavavej-Core-C-/Stephan-T-Lavavej-Core-C-2-of-n

Stephan T. Lavavej: Core C++, 3 of n (Overload Resolution)
http://channel9.msdn.com/Series/C9-Lectures-Stephan-T-Lavavej-Core-C-/Stephan-T-Lavavej-Core-Cpp-3-of-n

Function Overloading Based on Arbitrary Properties of Types
http://www.drdobbs.com/function-overloading-based-on-arbitrary/184401659

# Questions