Some time ago…

Cppcon 2021 The C++ Conference | October 24-29

Herb Sutter

Extending and Simplifying C++: Thoughts on Pattern Matching using `is` and `as`

Video Sponsorship Provided By:

ansatz

		Queries	P2392
safe	static		
	dynamic	is_same_v<X,Y>	X is Y
		is_base_of_v<B,D>	D is B
		dynamic_cast<D*>(pb)	pb is D*
		std::holds_alternative<T>(v)	v is T
		a.type() == typeid(T)	a is T
		o.has_value()	o is T
		f.wait_for(chrono::seconds(0)) == future_status::ready	f is T

Cppcon 2022 The C++ Conference | September 12th-16th

Herb Sutter

Can C++ be 10x simpler & safer … ?

C++ Core Guidelines

as = implemented using as

Pro.safety: Type-safety profile

- Type.1: Avoid casts:
  1. Don't use `reinterpret_cast`; A strict version of Avoid casts and prefer named casts.
  2. Don't use `static_cast` for arithmetic types; A strict version of Avoid casts and prefer named casts.
  3. Don't cast between pointer types where the source type and the target type are the same; A strict version of Avoid casts.
  4. Don't cast between pointer types when the conversion could be implicit; A strict version of Avoid casts.
- Type.2: Don't use `static_cast` to dow… namic_cast instead.
- Type.3: Don't use `const_cast` to cast a… (i.e., at all): Don't cast away const.
- Type.4: Don't use C-style `(T)expressio`… functional `T(expression)` casts: Prefe… or named casts or `T{expression}`.
- Type.5: Don't use a variable before it ha… initialized: always initialize.
- Type.6: Always initialize a member varia… initialize, possibly using default constru… default member initializers.
- Type.7: Avoid naked union: Use `varian`…
- Type.8: Avoid varargs: Don't use `va_arg`…

TODO classes

Video Sponsorship Provided By:

ansatz think-cell

When will you be able to write code like that?

```
fun: (v : _) -> std::string = {
  if v is std::vector  { return "std::vector";  }
  if v is std::array   { return "std::array";   }
  if v is std::variant { return "std::variant"; }
  if v is my_type      { return "my_type";      }
  else                 { return "unknown";      }
}

if i is (less_than(20)) {
  std::cout << "less than 20" << std::endl;
}

if i is (in(10,30)) {
  std::cout << "i is between 10 and 30" << std::endl;
}
```

Or like that?

```
x as Type;
```

1 Available now!

If you have C++20 compiler

(GCC 10+, Clang 12+, MSVC 16.30+)

FILIP SAJDAK

Head of Engineering

I am helping teams build HMI for premium cars

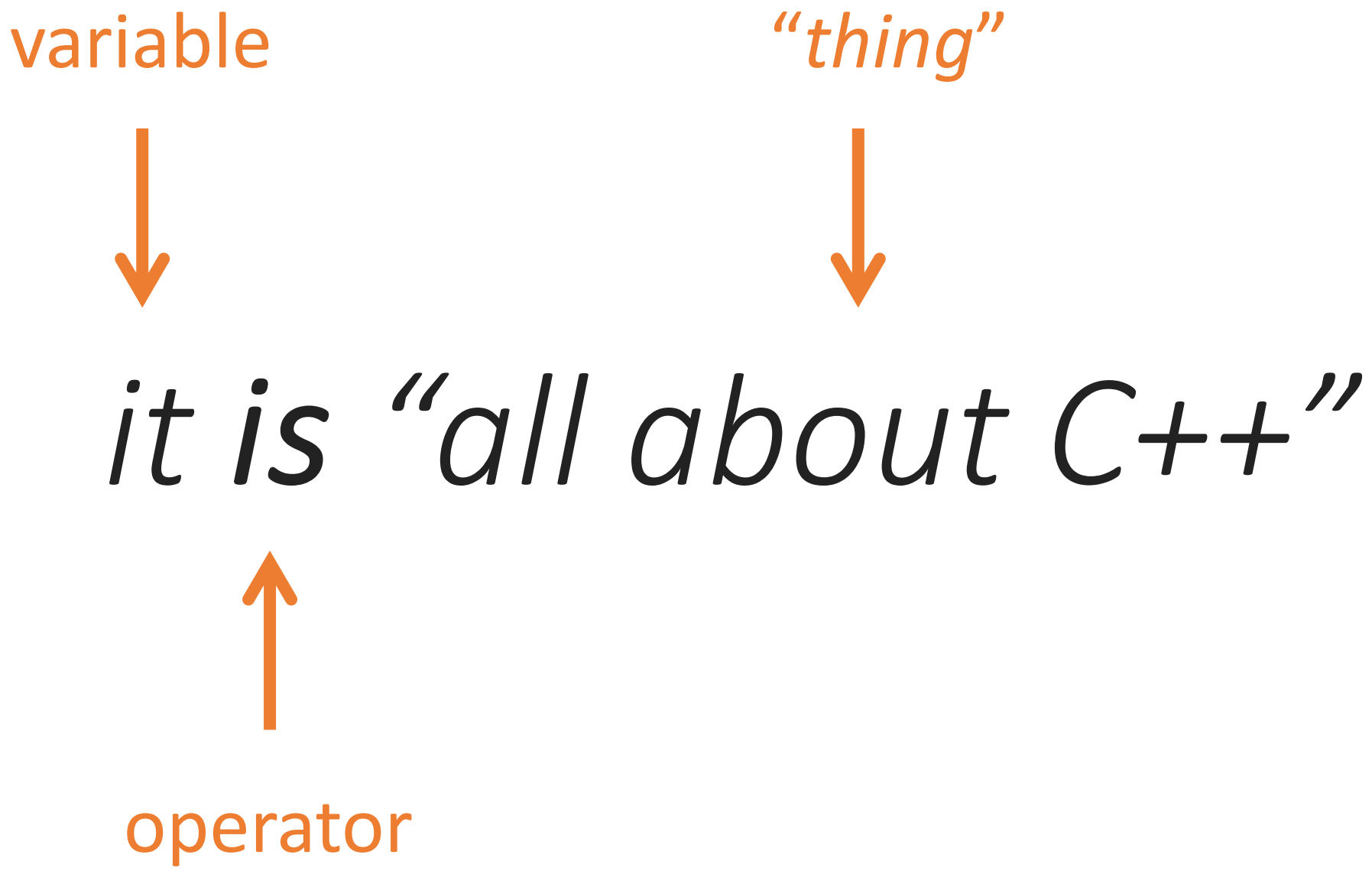
SIILI AUTO

30th | WRO.CPP MEETUP

Advancing cppfront with Modern C++

Refining the Implementation of:

- is,
- as,
- UFCS


variable
"thing"
it is "all about C++"
operator

it ***is*** *value*

it ***is*** *type*

it ***is*** *predicate*

it ***is*** *template*

it ***is*** *concept*

it ***is*** *empty*

2

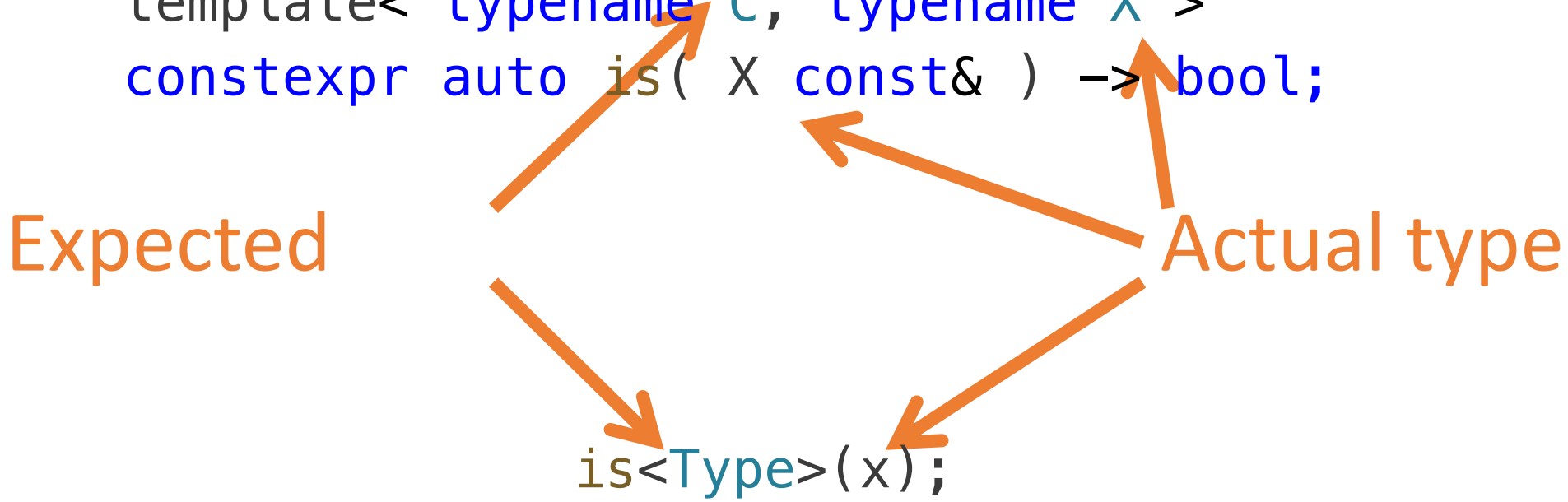
There is always a paper

Usually, a couple of years old.

	P2392	cppfront	C++20
Testing Variable	x is *Type*	x is Type	?
	x is *value*	x is (value)	?
	x is *predicate*	x is (predicate)	?
	x is *empty*	x is empty	?
	x is *Template*	x is Template	?
	x is *Concept*	*x is Concept*	?
Testing Type	X is *Type*	*X is Y*	?
	X is *Concept*	*X is Concept*	?
	X is *Template*	*X is Template*	?

x is *Type*

```
template< typename C, typename X >
constexpr auto is( X const& ) -> bool;
```



x is *Type*

```
template <typename C>
auto is(C const&) -> bool {
    return true;
}

template <typename C, typename X>
constexpr auto is(X const&) -> bool
{
    return false;
}
```

x is *Type*

```cpp
template <typename C>
auto is(C const&) -> bool {
    return true;
}

template <typename C, typename X>
constexpr auto is(X const&) -> bool
{
    return false;
}
```

```
<source>: In function 'int main()':
<source>:118:14: error: call of overloaded 'is<int>(int&)' is ambiguous
  118 |     is<int>(i);
      |              ^
<source>:99:6: note: candidate: 'bool is(const C&) [with C = int]'
   99 | auto is(C const&) -> bool {
      |      ^~
<source>:104:16: note: candidate: 'constexpr bool is(const X&) [with C = int; X = int]'
  104 | constexpr auto is(X const&) -> bool
      |                ^~
Compiler returned: 1
```

x is *Type*

```
template <typename C>
auto is(C const&) -> bool {
    return true;
}

template <typename C, typename X>
    requires (! std::same_as<C, X>)
auto is(X const&) -> bool {
    return false;
}
```

x is *Type*

```
template <typename C>
auto is(C const&) -> bool {
    return true;
}

template <typename C, typename X>
    requires (! std::same_as<C, X>)
auto is(X const&) -> bool {
    return false;
}

template <typename C, typename X>
    requires ( std::is_base_of_v<C, X> )
auto is(X const&) -> bool { return true; }
```

x is *Type*

```cpp
template <typename C>
auto is(C const&) -> bool {
    return true;
}

template <typename C, typename X>
    requires (! std::same_as<C, X>)
auto is(X const&) -> bool {
    return false;
}

template <typename C, typename X>
    requires ( std::is_base_of_v<C, X> )
auto is(X const&) -> bool { return true; }
```

```
<source>: In function 'int main()':
<source>:124:23: error: call of overloaded 'is<Base>(Derived)' is ambiguous
  124 |     is<Base>(Derived{});
      |                       ^
<source>:99:6: note: candidate: 'bool is(const C&) [with C = Base]'
   99 | auto is(C const&) -> bool {
      |      ^~
<source>:105:6: note: candidate: 'bool is(const X&) [with C = Base; X = Derived]'
  105 | auto is(X const&) -> bool {
      |      ^~
<source>:111:6: note: candidate: 'bool is(const X&) [with C = Base; X = Derived]'
  111 | auto is(X const&) -> bool { return true; }
      |      ^~
Compiler returned: 1
```

x is *Type*

```
template <typename C>
auto is(C const&) -> bool {
    return true;
}

template <typename C, typename X>
    requires (! std::same_as<C, X>
              && ! std::is_base_of_v<C, X>
    )
auto is(X const&) -> bool {
    return false;
}

template <typename C, typename X>
    requires ( std::is_base_of_v<C, X> )
auto is(X const&) -> bool { return true; }
```

x is *Type*

```
template <typename C>
auto is(C const&) -> bool { return true; }
```

```
template <typename C, typename X>
    requires (! std::same_as<C, X>
        && ! std::is_base_of_v<C, X>
        && ! ( std::is_polymorphic_v<C> && std::is_polymorphic_v<X> )
    )
auto is(X const&) -> bool { return false; }
```

```
template <typename C, typename X>
    requires (! std::same_as<C, X>)
auto is(X const&) -> bool { return false; }
```

```
template <typename C, typename X>
    requires ( std::is_polymorphic_v<C> && std::is_polymorphic_v<X> )
auto is(X const* x) -> bool {
    return dynamic_cast<const C*>(x) != nullptr;
}
```

3

If you want to learn C++20/23 – contribute!

Find a project that you are passionate about!

Johel Ernesto Guerrero Peña

Thank you!

x is *Type*

Concepts to the rescue!

x is *Type*

Concepts to the rescue!

Subsumption rules!

x is *Type*

Concepts to the rescue!

Subsumption rules!

- Concepts can subsume other concepts,
  - Works only with concepts – no type traits allowed (!)
- Concepts identified by the source location,
- When using negation or parens, you create a new expression
  - this means a new concept (breaks subsumption)

x is *Type*

```
template< typename C, typename X >
constexpr auto is( X const& ) -> bool {
    return false;
}

template< typename C, std::same_as<C> X>
constexpr auto is( X const& ) -> bool {
    return true;
}

template< typename C, inherited_from<C> X >
constexpr auto is( X const& ) -> bool {
    return true;
}

template< polymorphic C, polymorphic X >
    requires std::same_as<C, X>
constexpr auto is( X const& ) -> bool { return true; }

template< polymorphic C, polymorphic X >
constexpr auto is( X const& x ) -> bool {
    return Dynamic_cast<C const*>(&x) != nullptr;
}
```

x is *Type*

```
template <typename C>
auto is(C const&) -> bool { return true; }

template <typename C, typename X>
    requires (! std::same_as<C, X>
        && ! std::is_base_of_v<C, X>
        && ! ( std::is_polymorphic_v<C> && std::is_polymorphic
    )
auto is(X const&) -> bool { return false; }

template <typename C, typename X>
    requires (! std::same_as<C, X>)
auto is(X const&) -> bool { return false; }

template <typename C, typename X>
    requires ( std::is_polymorphic_v<C> && std::is_polymorphic
auto is(X const* x) -> bool {
    return dynamic_cast<const C*>(x) != nullptr;
}
```

```
template< typename C, typename X >
constexpr auto is( X const& ) -> bool {
    return false;
}

template< typename C, std::same_as<C> X>
constexpr auto is( X const& ) -> bool {
    return true;
}

template< typename C, inherited_from<C> X >
constexpr auto is( X const& ) -> bool {
    return true;
}

template< polymorphic C, polymorphic X >
    requires std::same_as<C, X>
constexpr auto is( X const& ) -> bool { return true; }

template< polymorphic C, polymorphic X >
constexpr auto is( X const& x ) -> bool {
    return Dynamic_cast<C const*>(&x) != nullptr;
}
```

C++ source #1

A Save/Load + Add new... Vim CppInsights Quick-bench C++

```
29
30  template< typename C, typename X >
31  constexpr auto is( X const& ) -> std::false_type { return {}; }
32
33  template< typename C, std::same_as<C> X>
34  constexpr auto is( X const& ) -> std::true_type { return {}; }
35
36  template< typename C, inherited_from<C> X >
37  constexpr auto is( X const& ) -> std::true_type { return {}; }
38
39  template< polymorphic C, polymorphic X > requires std::same_as<C, X>
40  constexpr auto is( X const& ) -> std::true_type { return {}; }
41
42  template< polymorphic C, polymorphic X >
43  constexpr auto is( X const& x ) -> bool {
44      return dynamic_cast<C const*>(&x) != nullptr;
45  }
46
47  int main() {
48      Derived d; Base1 const& b1 = d; Base2 const& b2 = d;
49      // TEST((is<int>(42)));
50      // TEST((is<long>(42)));
51      // TEST((is<Derived>(b1)));
52      // TEST((is<Derived>(d)));
53      // TEST((is<Base2>(b1)));
```

(49, 5)

Output of x86-64 gcc 10.5 (Compiler #1)

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
```

4 Compiler Explorer will help you learn concepts

Subsumption rules are not easy to grasp at first – CE gives fast feedbacks

x is *Type*

- Avoid type traits,
- Avoid negations & parens,
- Use Compiler Explorer!

g++-10	Clang-12	MSVC v19.29

```
template< typename C, typename X >
constexpr auto is( X const& ) -> bool {
    return false;
}

template< typename C, std::same_as<C> X>
constexpr auto is( X const& ) -> bool {
    return true;
}

template< typename C, inherited_from<C> X >
constexpr auto is( X const& ) -> bool {
    return true;
}

template< polymorphic C, polymorphic X >
    requires std::same_as<C, X>
constexpr auto is( X const& ) -> bool { return true; }

template< polymorphic C, polymorphic X >
constexpr auto is( X const& x ) -> bool {
    return Dynamic_cast<C const*>(&x) != nullptr;
}
```

x is *Type*

g++-10	Clang-12	MSVC v19.29

- Avoid type traits,
- Avoid negations & parens,
- Use Compiler Explorer!
- Use std::false_type / true_types

```
template< typename C, typename X >
constexpr auto is( X const& ) -> std::false_type {
    return {};
}

template< typename C, std::same_as<C> X>
constexpr auto is( X const& ) -> std::true_type {
    return {};
}

template< typename C, inherited_from<C> X >
constexpr auto is( X const& ) -> std::true_type {
    return {};
}

template< polymorphic C, polymorphic X >
    requires std::same_as<C, X>
constexpr auto is( X const& ) -> std::true_type { return {}; }

template< polymorphic C, polymorphic X >
constexpr auto is( X const& x ) -> bool {
    return Dynamic_cast<C const*>(&x) != nullptr;
}
```

COMPILER EXPLORER Add... More Templates Sponsors intel Solid Sands sonarcloud Share Policies Other

C++ source #1

A Save/Load + Add new... Vim CppInsights Quick-bench C++

```
25  struct Derived : Base1, Base2 {
26      void non_virtual() {std::cout << "Derived" << std::endl;}
27      void const_non_virtual() const {std::cout << "const Derived"
28  };
29
30  template< typename C, typename X >
31  constexpr auto is( X const& ) -> std::false_type { return {}; }
32
33  template< typename C, std::same_as<C> X>
34  constexpr auto is( X const& ) -> std::true_type { return {}; }
35
36  template< typename C, inherited_from<C> X >
37  constexpr auto is( X const& ) -> std::true_type { return {}; }
38
39
40  template< polymorphic C, polymorphic X >
41  constexpr auto is( X const& x ) -> bool {
42      return dynamic_cast<C const*>(&x) != nullptr;
43  }
44
45  int main() {
46      Derived d; Base1 const& b1 = d; Base2 const& b2 = d;
47      // TEST((is<Derived>(d)));
48  }
49
```

(47, 5)

x86-64 gcc 10.5 (Editor #1) Output of x86-64 gcc 10.5 (Compiler #1)

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
```

5

If compiler has too many options – give him another

Subsumption rules selects the most constraint match

	P2392	cppfront	C++20
Testing Variable	x is *Type*	x is Type	is<Type>(x)
	x is *value*	x is (value)	?
	x is *predicate*	x is (predicate)	?
	x is *empty*	x is empty	?
	x is *Template*	x is Template	?
	x is *Concept*	*x is Concept*	?
Testing Type	X is *Type*	*X is Y*	?
	X is *Concept*	*X is Concept*	?
	X is *Template*	*X is Template*	?

x is *value*

```
template <auto value, typename X>
constexpr bool is( auto const& x )

template <typename C, typename X>
constexpr bool is( X const& x, C const& value )

inline constexpr bool is( auto const& x, auto const& value )
```

x is *value*

g++-10	Clang-12	MSVC v19.29

```
inline constexpr bool is( auto const& x, auto const& value ) {
  if constexpr (requires{ bool{x == value}; }) {
    return x == value;
  }
  return false;
}
```

```
auto x = 42;
is(x, 42);   // true
is(x, 0);    // false
```

	P2392	cppfront	C++20
Testing Variable	x is *Type*	x is Type	is<Type>(x)
	x is *value*	x is (value)	is(x, value)
	x is *predicate*	x is (predicate)	?
	x is *empty*	x is empty	?
	x is *Template*	x is Template	?
	x is *Concept*	*x is Concept*	?
Testing Type	X is *Type*	*X is Y*	?
	X is *Concept*	*X is Concept*	?
	X is *Template*	*X is Template*	?

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
inline constexpr bool is( auto const& x, auto const& value ) {
  if constexpr (requires{ bool{value(x)}; }) {
    return value(x);
  } else
  if constexpr (requires{ bool{x == value}; }) {
    return x == value;
  }
  return false;
}

auto i = 42;
is(i, 42); // returns true
is(i, [](int i){ return i <= 42; }); // returns true
```

x is *predicate* (issues)

g++-10	Clang-12	MSVC v19.29

```
requires{ bool{value(x)}; } // stricter than std::predicate

is(42.3, [](int  i){ return i <= 42; }); // returns true
is(42.3, [](auto i){ return i <= 42; }); // returns false

bool free_fun    (int  i) { return i <= 42; }
bool free_gen_fun(auto i) { return i <= 42; }
// ...
is(42, free_fun);     // works
is(42, free_gen_fun); // does not work
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
template< class F, class... Args >
concept predicate =
  std::regular_invocable<F, Args...> &&
  boolean-testable<std::invoke_result_t<F, Args...>>;
```

(since C++20)

```
inline constexpr bool is( auto const& x, auto const& value ) {
  if constexpr (requires{ {value(x)} -> boolean_testable; })
    return value(x);
  } else
  if constexpr (requires{ bool{x == value}; }) {
    return x == value;
  }
  return false;
}
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
inline constexpr bool is( auto const& x, auto const& value ) {
  if constexpr (requires{ {value(x)} -> boolean_testable; })
    if constexpr (
       requires { argument_of<CPP2_TYPEOF(value)>{x}; }
    ) {
      return value(x);
    }
  }
  // ...
  return false;
}

is(42.3, [](int  i){ return i <= 42; }); // returns false

is(42.3, [](auto i){ return i  > 42; }); // returns false
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
inline constexpr bool is( auto const& x, auto const& value ) {
  if constexpr (requires{ {value(x)} -> boolean_testable; })
    if constexpr (requires {
      std::declval<argument_of< CPP2_TYPEOF(value)>>();
    }){
      if constexpr ( requires {
        argument_of< CPP2_TYPEOF(value)>{x};
      }) {
        return value(x);
      }
      return false;
    }
  }
  // ...
}
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
template<typename Ret, typename Arg>
auto argument_of_helper(Ret(*) (Arg)) -> Arg;

template<typename Ret, typename F, typename Arg>
auto argument_of_helper(Ret(F::*) (Arg)) -> Arg;

template<typename Ret, typename F, typename Arg>
auto argument_of_helper(Ret(F::*) (Arg) const) -> Arg;

template <typename F>
auto argument_of_helper(F) ->
      CPP2_TYPEOF(argument_of_helper(&F::operator()));

template <typename T>
using argument_of = CPP2_TYPEOF(argument_of_helper(std::declval<T>()));
```

x is *predicate* (the whole thing)

g++-10	Clang-12	MSVC v19.29

```
inline constexpr bool is( auto const& x, auto const& value ) {
  if constexpr (requires{ {value(x)} -> boolean_testable; })
    if constexpr (requires {
      std::declval<argument_of< CPP2_TYPEOF(value)>>();
    }){
      if constexpr ( requires { argument_of< CPP2_TYPEOF(value)>{x}; }) {
        return value(x);
      }
      return false;
    }
  } else
  if constexpr (requires{ bool{x == value}; }) {
    return x == value;
  }
  return false;
}
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
inline constexpr bool is( auto const& x, auto const& value ) {
  if constexpr ( predicate_like<CPP2_TYPEOF(value), CPP2_TYPEOF(x)> ) {
    if constexpr ( has_defined_argument<CPP2_TYPEOF(value)> ){
      if constexpr ( covertible_to_argument_of<CPP2_TYPEOF(x),
                                               CPP2_TYPEOF(value)>) {
        return value(x);
      }
      return false;
    }
    return value(x);
  } else
  if constexpr (value_comparable<CPP2_TYPEOF(value), CPP2_TYPEOF(x)>) {
    return x == value;
  }
  return false;
}
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
template <typename V, typename X>
constexpr bool is( X const& x, V const& value ) {
  if constexpr ( predicate_like<V, X> ) {
    if constexpr ( has_defined_argument<V> ){
      if constexpr ( covertible_to_argument_of<X, V> ) {
        return value(x);
      }
      return false;
    }
    return value(x);
  } else
  if constexpr (value_comparable<V, X>) {
    return x == value;
  }
  return false;
}
```

Side note: writing concepts

g++-10	Clang-12	MSVC v19.29

```
template <typename X, typename C>
concept inherited_from = std::is_base_of_v<C,X>
                         && !std::same_as<C,X>;


template <typename T, inherited_from<T> F>
void fun(F f) { /* ... */};

template <typename T>
void fun(inherited_from<T> auto f) { /* ... */ };
```

x is *predicate* (we can do better)

g++-10	Clang-12	MSVC v19.29

```
template <typename V, typename X>
constexpr bool is( X const& x, V const& value ) { return false; }

template <typename V, typename X> requires predicate_like<V, X>
constexpr bool is( X const& x, V const& value ) {
  if constexpr ( has_defined_argument<V> ){
    if constexpr ( covertible_to_argument_of<X, V> ) {
      return value(x);
    }
    return false;
  }
  return value(x);
}

template <typename V, typename X> requires value_comparable<V, X>
constexpr bool is( X const& x, V const& value ) { return x == value; }
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
template <typename V, typename X> requires predicate_like<V, X>
constexpr bool is( X const& x, V const& value ) {
  return value(x);
}

template <typename V, typename X>
  requires predicate_like<V, X> && has_defined_argument<V>
constexpr bool is( X const& x, V const& value ) {
  return false;
}

template <typename V, typename X>
  requires predicate_like<V, X> && has_defined_argument<V>
           && covertible_to_argument_of<X, V>
constexpr bool is( X const& x, V const& value ) {
  return value(x);
}
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
template <typename X, typename V>
constexpr bool is( X const& x, V const& value ) { return false; }

template <typename X, predicate_like<X> V>
constexpr bool is( X const& x, V const& value ) { return value(x); }

template <typename X, predicate_like<X> V>
  requires has_defined_argument<V>
constexpr bool is( X const& x, V const& value ) { return false; }

template <typename X, predicate_like<X> V>
requires has_defined_argument<V> && covertible_to_argument_of<X, V>
constexpr bool is( X const& x, V const& value ) { return value(x); }

template <typename X, value_comparable<X> V>
constexpr bool is( X const& x, V const& value ) {
  return x == value;
}
```

6

When writing concept put tested type first

It will allow you to use it instead of typename in the template parameter list and before auto in the argument list

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
template <typename X, typename V>
constexpr bool is( X const& x, V const& value ) { return false; }

template <typename X, std::predicate<X> V>
constexpr bool is( X const& x, V const& value ) { return value(x); }

template <typename X, std::predicate<X> V>
  requires has_defined_argument<V>
constexpr bool is( X const& x, V const& value ) { return false; }

template <typename X, std::predicate<X> V>
requires has_defined_argument<V> && covertible_to_argument_of<X, V>
constexpr bool is( X const& x, V const& value ) { return value(x); }

template <typename X, std::equality_comparable_with<X> V>
constexpr bool is( X const& x, V const& value ) {
  return x == value;
}
```

x is *predicate*

g++-10	Clang-12	MSVC v19.29

```
is(42, free_gen_fun); // does not work
```

```
template <typename X>
constexpr bool is( X const& x, bool (*value)(X const&) ) {
  return value(x);
}
```

```
is(42, free_gen_fun); // works!!!
```

	P2392	cppfront	C++20
Testing Variable	x is *Type*	x is Type	is<Type>(x)
	x is *value*	x is (value)	is(x, value)
	x is *predicate*	x is (predicate)	is(x, predicate)
	x is *empty*	x is empty	?
	x is *Template*	x is Template	?
	x is *Concept*	*x is Concept*	?
Testing Type	X is *Type*	*X is Y*	?
	X is *Concept*	*X is Concept*	?
	X is *Template*	*X is Template*	?

x is *empty*

```
using empty = void;
```

g++-10	Clang-12	MSVC v19.31

```
template< std::same_as<empty> C, pointer_like X >
constexpr auto is( X const& x ) -> bool {
  return x == X();
}

template< std::same_as<empty> C >
constexpr auto is( nullptr_t const& ) -> std::true_type {
  return {};
}

template< std::same_as<empty> C >
constexpr auto is( std::monostate const& ) -> std::true_type {
  return {};
}
```

	P2392	cppfront	C++20
Testing Variable	x is *Type*	x is Type	is<Type>(x)
	x is *value*	x is (value)	is(x, value)
	x is *predicate*	x is (predicate)	is(x, predicate)
	x is *empty*	x is empty	is<empty>(x)
	x is *Template*	x is Template	?
	x is *Concept*	*x is Concept*	?
Testing Type	X is *Type*	*X is Y*	?
	X is *Concept*	*X is Concept*	?
	X is *Template*	*X is Template*	?

x is *Template*

g++-10	Clang-12	MSVC v19.29

```
template <template <typename...> class C, typename... Ts>
constexpr auto is(C< Ts...> const& ) -> std::true_type {
  return {};
}

template <template <typename...> class C, typename T>
constexpr auto is( T const& ) -> std::false_type {
  return {};
}

is<std::vector>(v); // returns true_type
is<std::list  >(v); // returns false_type
```

	P2392	cppfront	C++20
Testing Variable	x is *Type*	x is Type	is<Type>(x)
	x is *value*	x is (value)	is(x, value)
	x is *predicate*	x is (predicate)	is(x, predicate)
	x is *empty*	x is empty	is<empty>(x)
	x is *Template*	x is Template	is<Template>(x)
	x is *Concept*	*x is Concept*	?
Testing Type	X is *Type*	*X is Y*	?
	X is *Concept*	*X is Concept*	?
	X is *Template*	*X is Template*	?

x is *Concept*

g++-10	Clang-12	MSVC v19.31

```
template <typename V, typename X>
concept callable_with_explicit_type = requires (V value)
{ { value.template operator()<X>() }; };

template <typename X, callable_with_explicit_type<X> V>
constexpr auto is( X const& x, V const& value )
  -> std::true_type { return {}; }


is(42.3, []<std::floating_point>(){}); // returns true
is(42.3, []<std::integral>(){});       // returns false
```

7

If you struggle try lambda

Lot of cppfront features are implemented using Lambdas

	P2392	cppfront	C++20
Testing Variable	x is *Type*	x is Type	is<Type>(x)
	x is *value*	x is (value)	is(x, value)
	x is *predicate*	x is (predicate)	is(x, predicate)
	x is *empty*	x is empty	is<empty>(x)
	x is *Template*	x is Template	is<Template>(x)
	x is *Concept*	*x is Concept*	is<[]<Concept>{}>(x)
Testing Type	X is *Type*	*X is Y*	?
	X is *Concept*	*X is Concept*	?
	X is *Template*	*X is Template*	?

	P2392	cppfront	C++20
Testing Variable	x is *Type*	x is Type	is<Type>(x)
	x is *value*	x is (value)	is(x, value)
	x is *predicate*	x is (predicate)	is(x, predicate)
	x is *empty*	x is empty	is<empty>(x)
	x is *Template*	x is Template	is<Template>(x)
	x is *Concept*	*x is Concept*	is<[]<Concept>{}>(x)
Testing Type	X is *Type*	*X is Y*	is<X, Type>() / is<X, Type>
	X is *Concept*	*X is Concept*	is<X, []<Concept>{}>()
	X is *Template*	*X is Template*	is<X, Template>(x)

Special types

g++-10	Clang-12	MSVC v19.30

```
auto v = std::variant<std::monostate, int>{};

auto a = std::any{};

auto o = std::optional<int>{};
```

```
is<empty>(v);     is<int>(v);     is(v, 42);

is<empty>(a);     is<int>(a);     is(a, 42);      ...

is<empty>(o);     is<int>(o);     is(o, 42);
```

is<T>(v)

g++-10	Clang-12	MSVC v19.29

```
template<typename T, typename... Ts>
auto is( std::variant<Ts...> const& x ) {
  if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 0>(x)), T >) { if (x.index() == 0) return true; }
  if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 1>(x)), T >) { if (x.index() == 1) return true; }
  if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 2>(x)), T >) { if (x.index() == 2) return true; }
/*
  checks for: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
*/
  if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<19>(x)), T >) { if (x.index() == 19) return true; }

  if constexpr (std::is_same_v< T, empty > ) {
    if (x.valueless_by_exception()) return true;
    // Need to guard this with is_any otherwise the get_if is illegal
    if constexpr (is_any<std::monostate, Ts...>) return std::get_if<std::monostate>(&x) != nullptr;
  }
  return false;
}
```

```
if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< N>(x)), T >) {
  if (x.index() == N) return true;
}
```

Variant type index

n-th variant type

type_find_if

g++-10	Clang-12	MSVC v19.30

```
template <typename... Ts>
auto type_find_if() {
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    std::cout << sizeof...(Is) << std::endl;
  }(std::index_sequence_for<Ts...>());
}

int main() {
  type_find_if<int, long, double>();
}
```

Output

```
3
```

type_find_if

g++-10	Clang-12	MSVC v19.30

```
template <typename... Ts, typename F>
auto type_find_if(F fun) {
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    fun(sizeof...(Is));
  }(std::index_sequence_for<Ts...>());
}

int main() {
  type_find_if<int, long, double>([](auto I){
    std::cout << I << std::endl;
  });
}
```

Output

```
3
```

type_find_if

g++-10	Clang-12	MSVC v19.30

```
template <typename... Ts, typename F>
auto type_find_if(F fun) {
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    (fun(Is), ...);
  }(std::index_sequence_for<Ts...>());
}

int main() {
  type_find_if<int, long, double>([](auto I){
    std::cout << I << std::endl;
  });
}
```

Output

```
0
1
2
```

type_find_if

g++-10	clang-13	MSVC v19.30

```
template <typename... Ts, typename F>
auto type_find_if(F fun) {
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    (fun.template operator()<Is>(), ...);
  }(std::index_sequence_for<Ts...>());
}

int main() {
  type_find_if<int, long, double>([]<auto I>(){
    std::cout << I << std::endl;
  });
}
```

Output

```
0
1
2
```

type_find_if

g++-10	Clang-13	MSVC v19.30

```
template <typename... Ts, typename F>
auto type_find_if(F fun) {
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    (fun.template operator()<Ts, Is>(), ...);
  }(std::index_sequence_for<Ts...>());
}

int main() {
  type_find_if<int, long, double>(
    []<typename T, auto I>(){
    std::cout << typeid(T).name()
              << ":" << I << std::endl;
  });
}
```

Output

```
i:0
l:1
d:2
```

type_find_if

g++-10	Clang-13	MSVC v19.30

```cpp
template <typename... Ts, typename F>
auto type_find_if(F fun) {
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    (fun.template operator()<Ts, Is>() || ...);
  }(std::index_sequence_for<Ts...>());
}

int main() {
  type_find_if<int, long, double>(
    []<typename T, auto I>(){
      std::cout << typeid(T).name()
                << ":" << I << std::endl;
    return I == 1;
  });
}
```

Output

```
i:0
l:1
```

type_find_if

g++-10	Clang-13	MSVC v19.30

```
template <typename... Ts, typename F>
auto type_find_if(F fun) {
  std::size_t idx = -1;
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    ((fun.template operator()<Ts, Is>() && (idx = Is, true) ) || ...);
  }(std::index_sequence_for<Ts...>());
  return idx;
}

int main() {
  auto i = type_find_if<int, long, double>(
    []<typename T, auto I>(){
      std::cout << typeid(T).name()
                << ":" << I << std::endl;
    return I == 1;
  });
}
```

Output

```
i:0
l:1
```

type_find_if

g++-10	Clang-13	MSVC v19.30

```cpp
template <typename... Ts, typename F>
auto type_find_if(F fun) {
  std::size_t idx = std::variant_npos;
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    ((fun.template operator()<Ts, Is>() && (idx = Is, true) ) || ...);
  }(std::index_sequence_for<Ts...>());
  return idx;
}

int main() {
  auto found = type_find_if<int, long, double>(
    []<typename T, auto I>(){
      std::cout << typeid(T).name()
                << ":" << I << std::endl;
      return I == 1;
  }) != std::variant_npos;
}
```

Output

```
i:0
l:1
```

is<T>(v) – v2

g++-10	clang-14	MSVC v19.30

```
template<typename T, typename... Ts>
auto is( std::variant<Ts...> const& x ) {
  return type_find_if<Ts...>([&]<typename X, std::size_t I>() -> bool {
    if constexpr (std::is_same_v< X, T >) { return x.index() == I; }
    return false;
  }) != std::variant_npos;
}

template<std::same_as<empty> T, typename... Ts>
auto is( std::variant<Ts...> const& x ) {
  if (x.valueless_by_exception()) return true;
  if constexpr (is_any<std::monostate, Ts...>)
     return std::get_if<std::monostate>(&x) != nullptr;
  return false;
}
```

is<T>(v) – v2.1

g++-10	Clang-12	MSVC v19.30

```
template <std::size_t Index, typename T>
struct type_it {
  using type = T;
  inline static const std::size_t index = Index;
};

template <typename... Ts, typename F>
constexpr auto type_find_if(F fun) {
  std::size_t found = std::variant_npos;
  [&]<std::size_t... Is>(std::index_sequence<Is...>){
    ((fun(type_it<Is, Ts>{}) && (found = Is, true)) || ...);
  }(std::index_sequence_for<Ts...>());
  return found;
}
```

is<T>(v) – v2.1

g++-10	Clang-12	MSVC v19.30

```
template<typename T, typename... Ts>
auto is( std::variant<Ts...> const& x ) {
  return type_find_if<Ts...>([&]<typename It>(It const&) {
    if constexpr (std::is_same_v< typename It::type, T >) {
      return x.index() == It::index; } else { return false; }
  }) != std::variant_npos;
}
```

is<T>(v) – v1 & v2.1

```cpp
template<typename T, typename... Ts>
auto is( std::variant<Ts...> const& x ) {
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 0>(x)), T >) { if (x.index() ==  0) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 1>(x)), T >) { if (x.index() ==  1) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 2>(x)), T >) { if (x.index() ==  2) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 3>(x)), T >) { if (x.index() ==  3) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 4>(x)), T >) { if (x.index() ==  4) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 5>(x)), T >) { if (x.index() ==  5) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 6>(x)), T >) { if (x.index() ==  6) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 7>(x)), T >) { if (x.index() ==  7) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 8>(x)), T >) { if (x.index() ==  8) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as< 9>(x)), T >) { if (x.index() ==  9) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<10>(x)), T >) { if (x.index() == 10) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<11>(x)), T >) { if (x.index() == 11) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<12>(x)), T >) { if (x.index() == 12) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<13>(x)), T >) { if (x.index() == 13) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<14>(x)), T >) { if (x.index() == 14) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<15>(x)), T >) { if (x.index() == 15) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<16>(x)), T >) { if (x.index() == 16) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<17>(x)), T >) { if (x.index() == 17) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<18>(x)), T >) { if (x.index() == 18) return true; }
    if constexpr (std::is_same_v< CPP2_TYPEOF(operator_as<19>(x)), T >) { if (x.index() == 19) return true; }
    if constexpr (std::is_same_v< T, empty > ) {
        if (x.valueless_by_exception()) return true;
        //  Need to guard this with is_any otherwise the get_if is illegal
        if constexpr (is_any<std::monostate, Ts...>) return std::get_if<std::monostate>(&x) != nullptr;
    }
    return false;
}
```

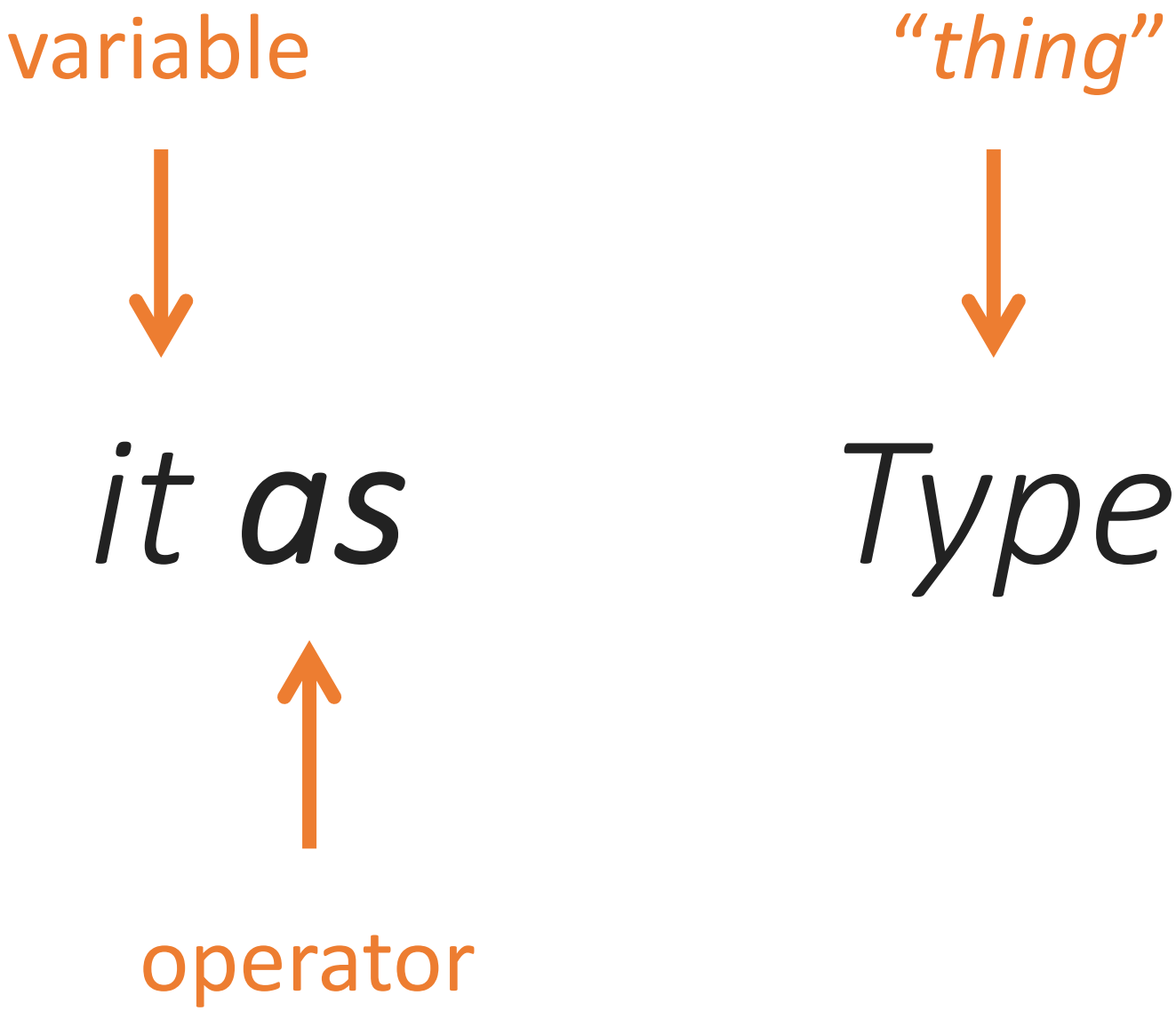
```cpp
template <std::size_t Index, typename T>
struct type_it {
    using type = T;
    inline static const std::size_t index = Index;
};

template <typename... Ts, typename F>
constexpr auto type_find_if(F fun)
{
    std::size_t idx = std::variant_npos;
    [&]<std::size_t... Is>(std::index_sequence<Is...>){
        if constexpr ((requires { {fun(type_it<Is, Ts>{})} -> boolean_testable;} && ...)) {
            ((fun(type_it<Is, Ts>{}) && (idx = Is, true)) || ...);
        }
    }(std::index_sequence_for<Ts...>());
    return idx;
}
```

```cpp
template<typename T, typename... Ts>
auto is( std::variant<Ts...> const& x ) {
    return type_find_if<Ts...>([&]<typename It>(It const&) -> bool {
        if constexpr (std::is_same_v< typename It::type, T >) { return x.index() == It::index; } else { return false; }
    }) != std::variant_npos;
}
```

```cpp
template<std::same_as<empty> T, typename... Ts>
auto is( std::variant<Ts...> const& x ) {
    if (x.valueless_by_exception()) return true;
    if constexpr (is_any<std::monostate, Ts...>) return std::get_if<std::monostate>(&x) != nullptr;
    return false;
}
```

is - takeaways

- You can implement an is operator as a **set of overloaded functions**,
- **Compiler Explorer helps** in checking overload matches,
- Concepts are powerful!
  - Learn them,
  - Remember about Compiler Explorer,
- If something is not working as expected -> Compiler Explorer
- Try to keep the **fuction signatures similar**,
- **Lambdas** can solve your challenging issues,


variable
"thing"
it as
Type
operator

it ***as*** *T*

bound

safe cast

potentially unsafe cast

nonesuch

X as T

g++-10	Clang-12	MSVC v19.29

```
template< typename C, typename X>
auto as( X const& ) -> auto {
  return nonesuch;
}


int i = 42;
as<long>(i); // returns nonesuch
```

X as T (static_assert)

g++-10	Clang-12	MSVC v19.29

```
template <typename... Ts>
inline constexpr auto program_violates_type_safety_guarantee
  = sizeof...(Ts) < 0;

template< typename C, typename X>
auto as( X const& ) -> auto {
  static_assert(program_violates_type_safety_guarantee<C,X>);
  return nonesuch;
}

as<long>(42); // triggers static assert
as<int>(42);  // works
```

X as T (static_assert issue)

g++-10	Clang-12	MSVC v19.29

```
void fun(auto x) {
  if constexpr (requires { as<long>(x); }) {
    std::cout << as<long>(x) << std::endl;
  } else {
    std::cout << "type not supported!" << std::endl;
  }
}
```

```
fun(42); // triggers static assert
```

X as T (solution)

g++-10	Clang-12	MSVC v19.29

```
template< typename C, typename X>
auto as( X const& ) -> auto { return nonesuch; }

void fun(auto x) {
  if constexpr (requires {
    {as<long>(x)} -> std::same_as<nonesuch_>;
  }) {
    std::cout << "type not supported!" << std::endl;
  } else {
    std::cout << as<long>(x) << std::endl;
  }
}

fun(42);  // prints type not supported
fun(42L); // prints 42
```

X as T (solution with static_assert)

g++-10	Clang-12	MSVC v19.29

```
template< typename C, typename X>
auto as_( X const& x ) -> decltype(auto) {
  if constexpr (requires {
    {as<C>(x)} -> std::same_as<nonesuch_>;
  }) {
    static_assert(program_violates_type_safety_guarantee<C,X>);
  } else {
    return as<C>(x);
  }
}

  as_<long>(42);  // triggers static assert
  as<long>(42);   // returns nonesuch
```

8

static_assert is not kind for your requires expression

If you want to use your function in requires clause you cannot trigger static asserts

same_as<T> as T

g++-10	Clang-12	MSVC v19.29

```
template< typename C, std::same_as<C> X>
constexpr auto as( X const& x ) -> decltype(auto) {
  return x;
}

template< typename C, std::same_as<C> X>
constexpr auto as( X& x ) -> decltype(auto) {
  return x;
}

int i = 42;
as<int>(i);  // returns int& = 42
```

same_as<T> as T

g++-10	Clang-12	MSVC v19.29

```
int i = 42;
as<int>(i);  // returns int& = 42
as<long>(i); // returns nonesuch

std::string s = "string";

as<std::string>(s); // returns std::string&
as<std::string>(std::as_const(s));
                    // returns std::string const&
```

literal as T

g++-10	Clang-12	MSVC v19.29

```
template< typename C, arithmetic auto x >
requires castable<C, x>
      || castable<typename C::value_type, x>
inline constexpr auto as() -> auto
{
  return static_cast<C>(x);
}

as<int, 12L>();
as<double, 12>();
as<std::int8_t, -1>();
as<std::uint8_t, -1>(); // failure non-castable
```

integral as integral (no explicit)

g++-10	Clang-12	MSVC v19.29

```
template< std::integral C, std::integral X >
  requires not_valid_convertible_to<X, C>
inline constexpr auto as(X const& x) -> auto
{
  const C c = static_cast<C>(x);
  assert( static_cast<CPP2_TYPEOF(x)>(c) == x
          && (c < C{}) == (x < CPP2_TYPEOF(x){}) );
  return c;
}

as<std::uint8_t>(std::int16_t{ 42});
as<std::uint8_t>(std::int16_t{-42}); // terminate!
```

variant as T

g++-10	Clang-12	MSVC v19.29

```
template<typename T, typename... Ts>
auto as( std::variant<Ts...> && x ) -> decltype(auto) {
  T* ptr = nullptr;
  type_find_if<Ts...>([&]<typename It>(It const&) -> bool {
    if constexpr (std::is_same_v< typename It::type, T >) {
      if (x.index() == It::index) {
        ptr = &std::get<It::index>(x); return true;
      }
    }; return false;
  });
  if (!ptr) { throw std::bad_variant_access(); }
  return *ptr;
}
```

variant as T

g++-10	Clang-12	MSVC v19.29

```
template<typename T, typename... Ts>
auto as( std::variant<Ts...> & x ) -> decltype(auto) {
  T* ptr = nullptr; /* ... */
}
template<typename T, typename... Ts>
auto as( std::variant<Ts...> const & x ) -> decltype(auto) {
  const T* ptr = nullptr; /* ... */
}

std::variant<int, long, float, double> v = 3;
std::cout << as<int>(v) << std::endl;
std::cout << as<int>(std::as_const(v)) << std::endl;
std::cout << as<int>(std::move(v)) << std::endl;
```

variant as T (we can do better)

g++-10	Clang-12	MSVC v19.29

```
template<typename T, spec_of_template<std::variant> X>
auto as( X && x ) -> decltype(auto) {
  constness_like<T, X> * ptr = nullptr;
  type_find_if(x, [&]<typename It>(It const&) -> bool {
    if constexpr (std::is_same_v< typename It::type, T >) {
      if (x.index() == It::index) {
        ptr = &std::get<It::index>(x); return true;
      }
    }; return false;
  });
  if (!ptr) { throw std::bad_variant_access(); }
  return *ptr;
}
```

variant as T (we can do better)

g++-10	Clang-12	MSVC v19.29

```
template <template<typename...> class C, typename... Ts>
constexpr auto type_find_if(C<Ts...>, auto&& fun)
{
  return type_find_if<Ts...>(CPP2_FORWARD(fun));
}

template <typename X, template<typename...> class C>
concept spec_of_template = requires (X x) {
  { is<C>(x) } -> std::same_as<std::true_type>;
};
```

variant as T (we can do better)

g++-10	Clang-12	MSVC v19.29

```
template <typename C, typename X>
using constness_like =
  std::conditional_t<
    std::is_const_v<
      std::remove_pointer_t<
        std::remove_reference_t<X>
      >
    >,
    std::add_const_t<C>,
    std::remove_const_t<C>
  >;
```

polymorphic as *polymorphic*

g++-10	Clang-12	MSVC v19.29

```
template <typename X>
concept polymorphic = std::is_polymorphic_v<X>;

template< polymorphic C, polymorphic X >
auto as( X& x ) -> C& {
  return dynamic_cast<C&>(x);
}

Derived d; Base1& b1 = d; Base2& b2 = d;
as<Base2>(b1).non_virtual();   // prints Base2
as<Base1>(b2).non_virtual();   // prints Base1
as<Derived>(b1).non_virtual(); // prints Derived
```

polymorphic as *polymorphic*

g++-10	Clang-12	MSVC v19.29

```
template <typename X> concept polymorphic
  = std::is_polymorphic_v<std::remove_cvref_t<X>>;

template< polymorphic C, polymorphic X >
auto as( X&& x ) -> decltype(auto) {
  return dynamic_cast<constness_like<C, X> &>(
    std::forward<X>(x)
  );
}

Derived d; Base1 const& b1 = d; Base2 const& b2 = d;
as<Base2>(b1).const_non_virtual();   // prints Base2
as<Base1>(b2).const_non_virtual();   // prints Base1
as<Derived>(b1).const_non_virtual(); // prints Derived
```

pointer_like<X> as X

g++-10	Clang-12	MSVC v19.29

```
template < typename C, pointer_like X>
  requires std::same_as<pointee_t<X>, C>
auto as (X const & x) -> decltype(auto) {
  if (x) {
    return *x;
  }
  throw std::runtime_error("'as' cast failed");
}

auto sp = std::make_shared<int>(24);
as<int>(sp); // returns int& = 24
sp.reset();
as<int>(sp); // throws exception
```

as - takeaways

- It is possible to introduce a **common way of casting in C++20**,
- *static_asserts* are not kind for *requires*,
- With proper type constraint function (concepts & subsumption rules), you can add **new overloads without touching old ones**,
- **Lambda explicit template argument list** is excellent!
- **Perfect forwarding** may save you from writing multiple versions of the same function,

Unified Function Call Syntax (`UFCS`)

```
s: std::string = "Fred";
myfile := fopen("xyzzy", "w");
myfile.fprintf( "Hello %s with UFCS!", s.c_str() );
myfile.fclose();
```

```
std::string s {"Fred"};
auto myfile {fopen("xyzzy", "w")};
CPP2_UFCS(fprintf, myfile, "Hello %s with UFCS!”
                         , CPP2_UFCS_0(c_str, std::move(s)));
CPP2_UFCS_0(fclose, std::move(myfile));
```

UFCS - the beginning

```
#define CPP2_UFCS(FUNCNAME,PARAM1,...) \
[](auto&& obj, auto&& ...params) { \
  if constexpr (requires{ \
    std::forward<decltype(obj)>(obj).FUNCNAME(std::forward<decltype(params)>(params)...); \
  }) { \
    return std::forward<decltype(obj)>(obj).FUNCNAME(std::forward<decltype(params)>(params)...); \
  } else { \
    return FUNCNAME(std::forward<decltype(obj)>(obj), std::forward<decltype(params)>(params)...); \
  } \
}(PARAM1, __VA_ARGS__)

#define CPP2_UFCS_0(FUNCNAME,PARAM1) \
[](auto&& obj) { \
  if constexpr (requires{ std::forward<decltype(obj)>(obj).FUNCNAME(); }) { \
    return std::forward<decltype(obj)>(obj).FUNCNAME(); \
  } else { \
    return FUNCNAME(std::forward<decltype(obj)>(obj)); \
  } \
}(PARAM1)
```

UFCS - now… oh my!

```
#define CPP2_UFCS(FUNCNAME,PARAM1,...) \
[](auto&& obj, auto&& ...params) { \
if constexpr (requires{ std::forward<decltype(obj)>(obj).FUNCNAME(std::forward<decltype(params)>(params)...); }) { \
return std::forward<decltype(obj)>(obj).FUNCNAME(std::forward<decltype(params)>(params)...); \
} else { \
return FUNCNAME(std::forward<decltype(obj)>(obj), std::forward<decltype(params)>(params)...); \
} \
}(PARAM1, __VA_ARGS__)

#define CPP2_UFCS_0(FUNCNAME,PARAM1) \
[](auto&& obj) { \
if constexpr (requires{ std::forward<decltype(obj)>(obj).FUNCNAME(); }) { \
return std::forward<decltype(obj)>(obj).FUNCNAME(); \
} else { \
return FUNCNAME(std::forward<decltype(obj)>(obj)); \
} \
}(PARAM1)
```

- Force inline,
- No discard,
- Workarounds for bugs,
- Handle templates with multiple parameters,
- UFCS chaining

```
#if defined(_MSC_VER) && !defined(__clang_major__)
#define CPP2_FORCE_INLINE __forceinline
#define CPP2_FORCE_INLINE_LAMBDA [[msvc::forceinline]]
#define CPP2_LAMBDA_NO_DISCARD
#else
#define CPP2_FORCE_INLINE __attribute__((always_inline))
#define CPP2_FORCE_INLINE_LAMBDA __attribute__((always_inline))

#if defined(__clang_major__)
// Also check __cplusplus, only to satisfy Clang -pedantic-errors
#if __cplusplus >= 202302L && (__clang_major__ > 13 || (__clang_major__ == 13 && __clang_minor__ >= 2))
#define CPP2_LAMBDA_NO_DISCARD [[nodiscard]]
#else
#define CPP2_LAMBDA_NO_DISCARD
#endif
#elif defined(__GNUC__)
#if __GNUC__ >= 9
#define CPP2_LAMBDA_NO_DISCARD [[nodiscard]]
#else
#define CPP2_LAMBDA_NO_DISCARD
#endif
#if (__GNUC__ * 100) + __GNUC_MINOR__ < 1003
// GCC-10.2 doesn't support this feature (10.3 is fine)
#undef CPP2_FORCE_INLINE_LAMBDA
#define CPP2_FORCE_INLINE_LAMBDA
#endif
#else
#define CPP2_LAMBDA_NO_DISCARD
#endif
#endif


// Note: [&] is because a nested UFCS might be viewed as trying to capture 'this'

#define CPP2_UFCS(FUNCNAME,PARAM1,...) \
[&] CPP2_LAMBDA_NO_DISCARD (auto&& obj, auto&& ...params) CPP2_FORCE_INLINE_LAMBDA -> decltype(auto) { \
if constexpr (requires{ CPP2_FORWARD(obj).FUNCNAME(CPP2_FORWARD(params)...); }) { \
return CPP2_FORWARD(obj).FUNCNAME(CPP2_FORWARD(params)...); \
} else { \
return FUNCNAME(CPP2_FORWARD(obj), CPP2_FORWARD(params)...); \
} \
}(PARAM1, __VA_ARGS__)

#define CPP2_UFCS_0(FUNCNAME,PARAM1) \
[&] CPP2_LAMBDA_NO_DISCARD (auto&& obj) CPP2_FORCE_INLINE_LAMBDA -> decltype(auto) { \
if constexpr (requires{ CPP2_FORWARD(obj).FUNCNAME(); }) { \
return CPP2_FORWARD(obj).FUNCNAME(); \
} else { \
return FUNCNAME(CPP2_FORWARD(obj)); \
} \
}(PARAM1)

#define CPP2_UFCS_REMPARENS(...) __VA_ARGS__

#define CPP2_UFCS_TEMPLATE(FUNCNAME,TEMPARGS,PARAM1,...) \
[&] CPP2_LAMBDA_NO_DISCARD (auto&& obj, auto&& ...params) CPP2_FORCE_INLINE_LAMBDA -> decltype(auto) { \
if constexpr (requires{ CPP2_FORWARD(obj).template FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (CPP2_FORWARD(params)...); }) { \
return CPP2_FORWARD(obj).template FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (CPP2_FORWARD(params)...); \
} else { \
return FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (CPP2_FORWARD(obj), CPP2_FORWARD(params)...); \
} \
}(PARAM1, __VA_ARGS__)

#define CPP2_UFCS_TEMPLATE_0(FUNCNAME,TEMPARGS,PARAM1) \
[&] CPP2_LAMBDA_NO_DISCARD (auto&& obj) CPP2_FORCE_INLINE_LAMBDA -> decltype(auto) { \
if constexpr (requires{ CPP2_FORWARD(obj).template FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (); }) { \
return CPP2_FORWARD(obj).template FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (); \
} else { \
return FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (CPP2_FORWARD(obj)); \
} \
}(PARAM1)

// But for non-local lambdas [&] is not allowed

#define CPP2_UFCS_NONLOCAL(FUNCNAME,PARAM1,...) \
[] CPP2_LAMBDA_NO_DISCARD (auto&& obj, auto&& ...params) CPP2_FORCE_INLINE_LAMBDA -> decltype(auto) { \
if constexpr (requires{ CPP2_FORWARD(obj).FUNCNAME(CPP2_FORWARD(params)...); }) { \
return CPP2_FORWARD(obj).FUNCNAME(CPP2_FORWARD(params)...); \
} else { \
return FUNCNAME(CPP2_FORWARD(obj), CPP2_FORWARD(params)...); \
} \
}(PARAM1, __VA_ARGS__)

#define CPP2_UFCS_0_NONLOCAL(FUNCNAME,PARAM1) \
[] CPP2_LAMBDA_NO_DISCARD (auto&& obj) CPP2_FORCE_INLINE_LAMBDA -> decltype(auto) { \
if constexpr (requires{ CPP2_FORWARD(obj).FUNCNAME(); }) { \
return CPP2_FORWARD(obj).FUNCNAME(); \
} else { \
return FUNCNAME(CPP2_FORWARD(obj)); \
} \
}(PARAM1)

#define CPP2_UFCS_TEMPLATE_NONLOCAL(FUNCNAME,TEMPARGS,PARAM1,...) \
[] CPP2_LAMBDA_NO_DISCARD (auto&& obj, auto&& ...params) CPP2_FORCE_INLINE_LAMBDA -> decltype(auto) { \
if constexpr (requires{ CPP2_FORWARD(obj).template FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (CPP2_FORWARD(params)...); }) { \
return CPP2_FORWARD(obj).template FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (CPP2_FORWARD(params)...); \
} else { \
return FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (CPP2_FORWARD(obj), CPP2_FORWARD(params)...); \
} \
}(PARAM1, __VA_ARGS__)

#define CPP2_UFCS_TEMPLATE_0_NONLOCAL(FUNCNAME,TEMPARGS,PARAM1) \
[] CPP2_LAMBDA_NO_DISCARD (auto&& obj) CPP2_FORCE_INLINE_LAMBDA -> decltype(auto) { \
if constexpr (requires{ CPP2_FORWARD(obj).template FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (); }) { \
return CPP2_FORWARD(obj).template FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (); \
} else { \
return FUNCNAME CPP2_UFCS_REMPARENS TEMPARGS (CPP2_FORWARD(obj)); \
} \
}(PARAM1)
```

UFCS - takeaways

- **Lambdas always help** you solve tricky issues,
- Macros are powerful, but you don’t want to write and use them,
  - **Commas are always tricky**, but there are solutions!
  - There are **Variadic Macros** – search for __VA_OPT__
- In **C++23 lambdas will have attributes** – support for [[nodiscard]],
- Current macros make it impractical to write them by hand – it is meant for generation,

9 Preparing presentation will help you find better ways of doing things

While I have been preparing for this talk
I have reworked most of the
implementation of is and as in cppfront

Rework of `is` and `as` that adds new functionalities or simplify implementation #701

Edit Code

Open **filipsajdak** wants to merge 33 commits into `hsutter:main` from `filipsajdak:fsajdak-refactor-of-is`

Conversation 42 Commits 33 Checks 0 Files changed 42 +2,093 −503

filipsajdak commented 2 weeks ago • edited Contributor

This change reworks a big part of the code for `is` and `as`.

- added concepts for simplify and utilize concept subsumption rules,
  - inspired by **@JohelEGP** - I have learned more here: https://andreasfertig.blog/2020/09/cpp20-concepts-subsumption-rules/
- Added `type_find_if` to iterate over `variant` types to simplify `is` / `as` for `variant`,
- rewrite `is` and `as` overloads to use `concepts` instead of `type_traits` (to utilize concepts subsumption rules),
- rewrite `narrowing`, `castable` as concepts,
- `castable` returns `true` for types that could be directly cast with `C{x}`,
- remove `operator_is`, and `operator_as`, implementation of `is` / `as` for variants now use `type_find_if`,
- in static asserts that use `program_violates_type_safety_guarantee`, I have replaced `CPP2_TYPEOF` with `decltype` to get the type with qualifiers - needed in debugging overloads,
- `is` that is not using argument `x` for inspection returns `std::true_type` or `std::false_type`
- add missing `as` overload for handling polymorphic pointer to const,

Closes #701
Closes #689

`to_string`

Reviewers
JohelEGP

Still in progress? Convert to draft

Assignees
No one assigned

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully linked these issues.

Let's learn from
Each other!