SAND2019-1604PE

Sandia National Laboratories

P1393R0: A General Property Customization Mechanism

David Hollman, Chris Kohlhoff, Bryce Lelbach,
Gordon Brown, Michał Dominiak

NNSA U.S. DEPARTMENT OF ENERGY

CONCEPT-DRIVEN DESIGN

CONCEPT-DRIVEN DESIGN

> *Concepts = Constraints + Axioms*
>
> *- A. Sutton and B. Stroustrup, SLE '11*

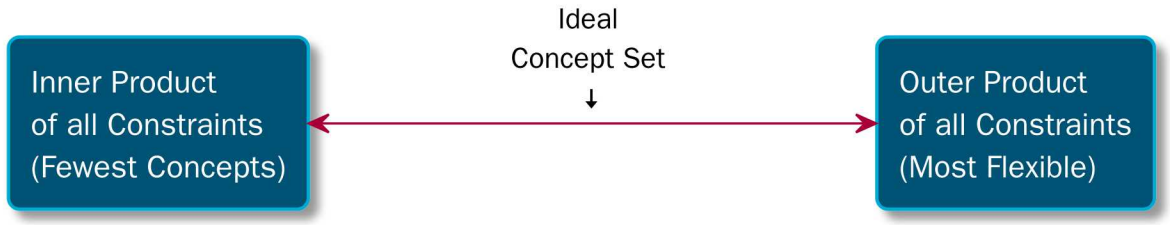
CONCEPT-DRIVEN DESIGN

> *Concepts = Constraints + Axioms*
>
> *- A. Sutton and B. Stroustrup, SLE '11*

> *Look at the algorithms!*
>
> *- E. Niebler, numerous hall conversations*

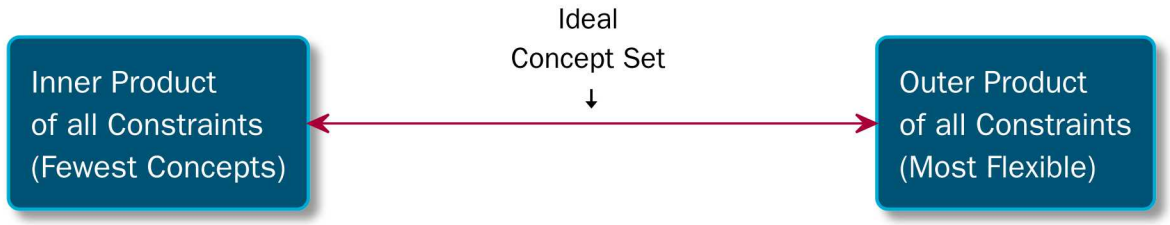
CONSTRAINT OPTIMIZATION

- Concept driven design is an *optimization problem* that balances minimization of constraints on a set of algorithms with the minimization of cognitive load on the user.



CONSTRAINT OPTIMIZATION

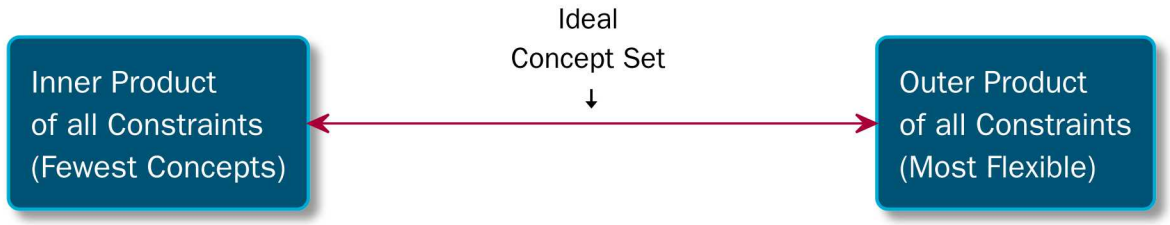
- Concept driven design is an *optimization problem* that balances minimization of constraints on a set of algorithms with the minimization of cognitive load on the user.



> *The design of a concept library is the result of two minimization problems: concept and constraint minimization.*
>
> *- A. Sutton and B. Stroustrup, SLE '11*

CONSTRAINT OPTIMIZATION

- Concept driven design is an *optimization problem* that balances minimization of constraints on a set of algorithms with the minimization of cognitive load on the user.

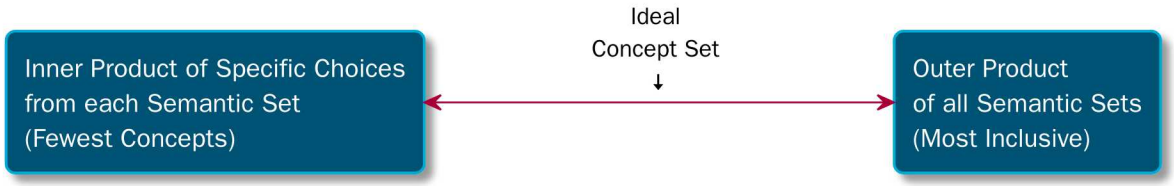


> *The design of a concept library is the result of two minimization problems: concept and constraint minimization.*
>
> *- A. Sutton and B. Stroustrup, SLE '11*

> *An effective specification of concepts is the product of an iterative process that minimizes the number of concepts while maintaining expressive and effective constraints.*
>
> *- A. Sutton and B. Stroustrup, SLE '11*

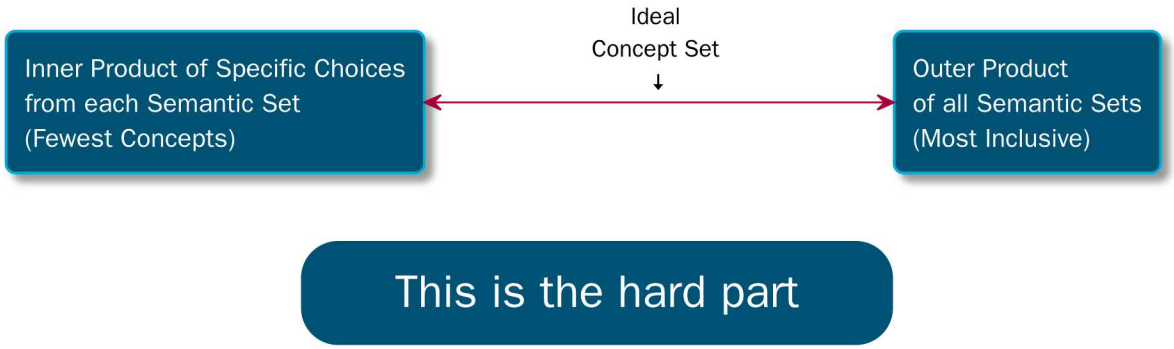
AXIOM SET OPTIMIZATION

- The analogous optimization problem for the semantic axioms looks like:



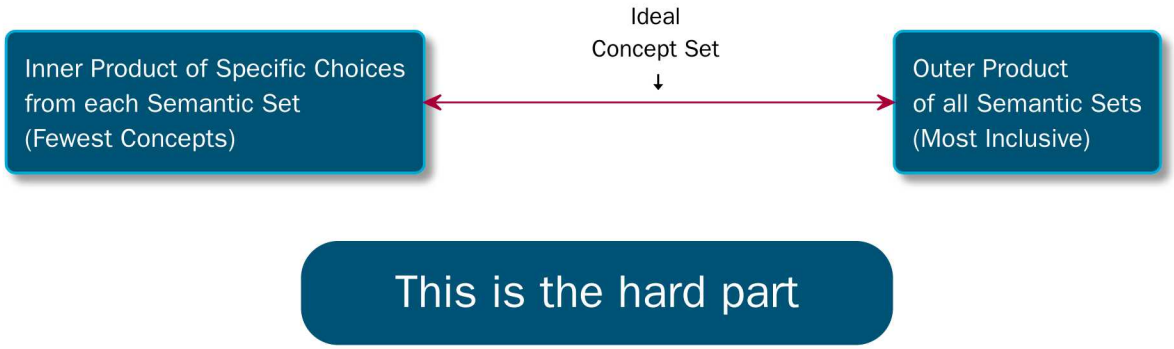
AXIOM SET OPTIMIZATION

- The analogous optimization problem for the semantic axioms looks like:



AXIOM SET OPTIMIZATION

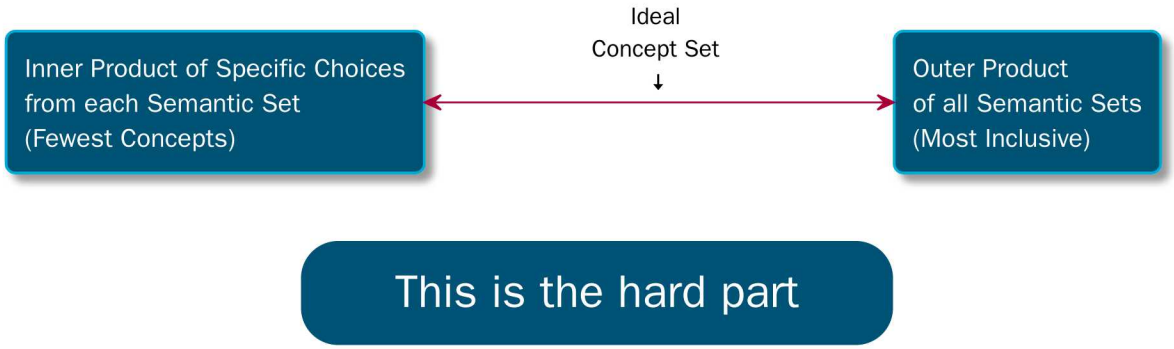
- The analogous optimization problem for the semantic axioms looks like:



- The most inclusive solution is to use all of the possible combinations of what everyone things the abstraction should do.

AXIOM SET OPTIMIZATION

- The analogous optimization problem for the semantic axioms looks like:



- The most inclusive solution is to use all of the possible combinations of what everyone things the abstraction should do.
- The fewest concepts solution (arguably the least cognitive load) chooses one semantic from each group of conflicting options.

AXIOM SET OPTIMIZATION

WHY IS THIS PART SO HARD?

AXIOM SET OPTIMIZATION

WHY IS THIS PART SO HARD?

- Not every user cares about every semantic property

AXIOM SET OPTIMIZATION

WHY IS THIS PART SO HARD?

- Not every user cares about every semantic property
  - But some users *really* care about some semantic properties

AXIOM SET OPTIMIZATION

WHY IS THIS PART SO HARD?

- Not every user cares about every semantic property
  - But some users *really* care about some semantic properties
  - Many concepts are *unusable* in certain domains without certain semantic properties

AXIOM SET OPTIMIZATION

WHY IS THIS PART SO HARD?

- Not every user cares about every semantic property
  - But some users *really* care about some semantic properties
  - Many concepts are *unusable* in certain domains without certain semantic properties
- Different domains have different axiom sets for different concepts

AXIOM SET OPTIMIZATION

WHY IS THIS PART SO HARD?

- Not every user cares about every semantic property
  - But some users *really* care about some semantic properties
  - Many concepts are *unusable* in certain domains without certain semantic properties
- Different domains have different axiom sets for different concepts
- Just meeting the semantic requirements of an algorithm may not be sufficiently consistent with the *zero-overhead principle* (or low-overhead principle)

THE NESTED CONCERNS PROBLEM

Consider two algorithms, `frobnicate_1` and `frobnicate_2`, that use the same concept, `Thing`:

```
template <Thing T, auto I1, auto I2>
void frobnicate_2(T t, I1 i, I2 j);

template <Thing T, ForwardRange R>
void frobnicate_1(T t, R r) {
  /* ... */
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(t, i, j);
    }
  }
  /* ... */
}
```

THE NESTED CONCERNS PROBLEM

Consider two algorithms, `frobnicate_1` and `frobnicate_2`, that use the same concept, `Thing`:

```
template <Thing T, auto I1, auto I2>
void frobnicate_2(T t, I1 i, I2 j);

template <Thing T, ForwardRange R>
void frobnicate_1(T t, R r) {
  /* ... */
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(t, i, j);
    }
  }
  /* ... */
}
```

- Both algorithms use the same constraints (yay!)

THE NESTED CONCERNS PROBLEM

Consider two algorithms, `frobnicate_1` and `frobnicate_2`, that use the same concept, `Thing`:

```
template <Thing T, auto I1, auto I2>
void frobnicate_2(T t, I1 i, I2 j);

template <Thing T, ForwardRange R>
void frobnicate_1(T t, R r) {
  /* ... */
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(t, i, j);
    }
  }
  /* ... */
}
```

- Both algorithms use the same constraints (yay!)
- But suppose `frobnicate_2()` (when used as inner-loop code), and won't run efficiently unless `Thing` has caching enabled.

THE NESTED CONCERNS PROBLEM

Consider two algorithms, `frobnicate_1` and `frobnicate_2`, that use the same concept, `Thing`:

```
template <Thing T, auto I1, auto I2>
void frobnicate_2(T t, I1 i, I2 j);

template <Thing T, ForwardRange R>
void frobnicate_1(T t, R r) {
  /* ... */
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(t, i, j);
    }
  }
  /* ... */
}
```

- Both algorithms use the same constraints (yay!)
- But suppose `frobnicate_2()` (when used as inner-loop code), and won't run efficiently unless `Thing` has caching enabled.
- Does the caller of `frobnicate_1()` need to know about caching?

THE NESTED CONCERNS PROBLEM

Consider two algorithms, `frobnicate_1` and `frobnicate_2`, that use the same concept, `Thing`:

```
template <Thing T, auto I1, auto I2>
void frobnicate_2(T t, I1 i, I2 j);

template <Thing T, ForwardRange R>
void frobnicate_1(T t, R r) {
  /* ... */
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(t, i, j);
    }
  }
  /* ... */
}
```

- Both algorithms use the same constraints (yay!)
- But suppose `frobnicate_2()` (when used as inner-loop code), and won't run efficiently unless `Thing` has caching enabled.
- Does the caller of `frobnicate_1()` need to know about caching?
- Current options:

THE NESTED CONCERNS PROBLEM

Consider two algorithms, `frobnicate_1` and `frobnicate_2`, that use the same concept, `Thing`:

```
template <Thing T, auto I1, auto I2>
void frobnicate_2(T t, I1 i, I2 j);

template <Thing T, ForwardRange R>
void frobnicate_1(T t, R r) {
  /* ... */
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(t, i, j);
    }
  }
  /* ... */
}
```

- Both algorithms use the same constraints (yay!)
- But suppose `frobnicate_2()` (when used as inner-loop code), and won't run efficiently unless `Thing` has caching enabled.
- Does the caller of `frobnicate_1()` need to know about caching?
- Current options:
  - Expand the concept to `Thing<CachingMode>`, and template `frobnicate_1()` on `CachingMode`

THE NESTED CONCERNS PROBLEM

Consider two algorithms, `frobnicate_1` and `frobnicate_2`, that use the same concept, `Thing`:

```
template <Thing T, auto I1, auto I2>
void frobnicate_2(T t, I1 i, I2 j);

template <Thing T, ForwardRange R>
void frobnicate_1(T t, R r) {
  /* ... */
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(t, i, j);
    }
  }
  /* ... */
}
```

- Both algorithms use the same constraints (yay!)
- But suppose `frobnicate_2()` (when used as inner-loop code), and won't run efficiently unless `Thing` has caching enabled.
- Does the caller of `frobnicate_1()` need to know about caching?
- Current options:
  - Expand the concept to `Thing<CachingMode>`, and template `frobnicate_1()` on `CachingMode`
  - Be okay with the efficiency loss 😱

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

Suppose we want to introduce a standard frobnication algorithm.

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

Presumably, if we put it in the standard, we want to do our best to avoid scenarios where users are forced to roll their on `apply_frobnication()`

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

A third-party library provides a `Frobnicator`.

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

Some user code wants to call the standard algorithm with the third party `Frobnicator`.

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

This issue is one of cross-cutting concerns

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

This issue is one of cross-cutting concerns

- `std::apply_standard_frobnication()` needs to communicate a cross-cutting concern to `TPLFrobnicator` without adding to the user's cognitive load for the `Frobnicator` concept.

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

This issue is one of cross-cutting concerns

- `std::apply_standard_frobnication()` needs to communicate a cross-cutting concern to `TPLFrobnicator` without adding to the user's cognitive load for the `Frobnicator` concept.
  - (The expression of this concern would need to be part of the standard library)

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

This issue is one of cross-cutting concerns

- `std::apply_standard_frobnication()` needs to communicate a cross-cutting concern to `TPLFrobnicator` without adding to the user's cognitive load for the `Frobnicator` concept.
  - (The expression of this concern would need to be part of the standard library)
- `my_frobnicate()` needs to communicate a cross-cutting concern (expressed as `MySpecialConcern`) to the third-party library, but that concern is too domain-specific to be in the standard library.

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

This issue is one of cross-cutting concerns

- `std::apply_standard_frobnication()` needs to communicate a cross-cutting concern to `TPLFrobnicator` without adding to the user's cognitive load for the `Frobnicator` concept.
  - (The expression of this concern would need to be part of the standard library)
- `my_frobnicate()` needs to communicate a cross-cutting concern (expressed as `MySpecialConcern`) to the third-party library, but that concern is too domain-specific to be in the standard library.
  - (But it would be nice for the third party library authors to be able to use the same mechanism to communicate this concern.)

NESTED CONCERNS: STANDARD LIBRARY

Why should the standard library care about this problem?

```
namespace std {
  template <Frobnicator F> void apply_standard_frobnication(F f);
} // namespace std
namespace generic_third_party_library {
  template <class...> struct TPLFrobnicator;
} // end generic_third_party_library
void my_frobnicate() {
  Frobnicator auto frob = TPLFrobnicator<MySpecialConcern>();
  /* ... */
  std::apply_standard_frobnication(frob);
}
```

This issue is one of cross-cutting concerns

- `std::apply_standard_frobnication()` needs to communicate a cross-cutting concern to `TPLFrobnicator` without adding to the user's cognitive load for the `Frobnicator` concept.
  - (The expression of this concern would need to be part of the standard library)
- `my_frobnicate()` needs to communicate a cross-cutting concern (expressed as `MySpecialConcern`) to the third-party library, but that concern is too domain-specific to be in the standard library.
  - (But it would be nice for the third party library authors to be able to use the same mechanism to communicate this concern.)
- The third-party library needs a way to communicate that the standard's cross-cutting concern is orthogonal (or not) to the user's cross-cutting concern.

A `String` CONCEPT?

Consider what it would take to make a `String` concept that makes everyone happy...

A `String` CONCEPT?

Consider what it would take to make a `String` concept that makes everyone happy...

Examples of `std::string`-like things in the wild:

- `llvm::SmallString`
- `folly::fbstring`
- Qt's `QString`
- Unreal Engine's `FString`

A `String` CONCEPT?

Consider what it would take to make a `String` concept that makes everyone happy...

Examples of `std::string`-like things in the wild:

- `llvm::SmallString`
- `folly::fbstring`
- Qt's `QString`
- Unreal Engine's `FString`

Properties algorithms may need to align with the zero-overhead principle:

- Unicode support
- Small buffer optimization
- Is the data contiguous?
- Is the data aligned?
- Is the data in network pinned memory?

A `String` CONCEPT?

Consider what it would take to make a `String` concept that makes everyone happy...

Examples of `std::string`-like things in the wild:

- `llvm::SmallString`
- `folly::fbstring`
- Qt's `QString`
- Unreal Engine's `FString`

Properties algorithms may need to align with the zero-overhead principle:

- Unicode support
- Small buffer optimization
- Is the data contiguous?
- Is the data aligned?
- Is the data in network pinned memory?

These can't be part of the concept (too much *cognitive* overhead), but they may be necessary for some algorithms to avoid unacceptable *performance* overhead.

FIXING THE `frobnicate` EXAMPLE

We can fix the `frobnicate` example with properties:

```
template <Thing T, auto Item>
void frobnicate_2(T t, Item i, Item j);

template <Thing T, Range R>
void frobnicate_1(T t, R r) {
  /* ... */
  auto my_t = prefer(t, caching);
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(my_t, i, j);
    }
  }
  /* ... */
}
```

FIXING THE `frobnicate` EXAMPLE

We can fix the `frobnicate` example with properties:

```
template <Thing T, auto Item>
void frobnicate_2(T t, Item i, Item j);

template <Thing T, Range R>
void frobnicate_1(T t, R r) {
  /* ... */
  auto my_t = prefer(t, caching);
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(my_t, i, j);
    }
  }
  /* ... */
}
```

Tell the `Thing` that it's about to be part of an inner loop and should cache things if it knows how.

FIXING THE `frobnicate` EXAMPLE

We can fix the `frobnicate` example with properties:

```
template <Thing T, auto Item>
void frobnicate_2(T t, Item i, Item j);

template <Thing T, Range R>
void frobnicate_1(T t, R r) {
  /* ... */
  auto my_t = prefer(t, caching);
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(my_t, i, j);
    }
  }
  /* ... */
}
```

FIXING THE `frobnicate` EXAMPLE

We can fix the `frobnicate` example with properties:

```
template <Thing T, auto Item>
void frobnicate_2(T t, Item i, Item j);

template <Thing T, Range R>
void frobnicate_1(T t, R r) {
  /* ... */
  auto my_t = prefer(t, caching);
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(my_t, i, j);
    }
  }
  /* ... */
}
```

- The `caching` property doesn't contribute to the cognitive load for most callers of `frobnicate_2()`.

FIXING THE `frobnicate` EXAMPLE

We can fix the `frobnicate` example with properties:

```
template <Thing T, auto Item>
void frobnicate_2(T t, Item i, Item j);

template <Thing T, Range R>
void frobnicate_1(T t, R r) {
  /* ... */
  auto my_t = prefer(t, caching);
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(my_t, i, j);
    }
  }
  /* ... */
}
```

- The `caching` property doesn't contribute to the cognitive load for most callers of `frobnicate_2()`.
- Without this property, `frobnicate_1()` might have to roll its own `frobnicate_2()`

FIXING THE `frobnicate` EXAMPLE

We can fix the `frobnicate` example with properties:

```
template <Thing T, auto Item>
void frobnicate_2(T t, Item i, Item j);

template <Thing T, Range R>
void frobnicate_1(T t, R r) {
  /* ... */
  auto my_t = prefer(t, caching);
  for(auto const& i : r) {
    for(auto const& j : r) {
      frobnicate_2(my_t, i, j);
    }
  }
  /* ... */
}
```

- The `caching` property doesn't contribute to the cognitive load for most callers of `frobnicate_2()`.
- Without this property, `frobnicate_1()` might have to roll its own `frobnicate_2()`
- A savvy `frobnicate_2()` author can even specialize for the `caching` case, if it matters.

HEADER <property> SYNOPSIS: CONCEPT-PRESERVING PROPERTIES

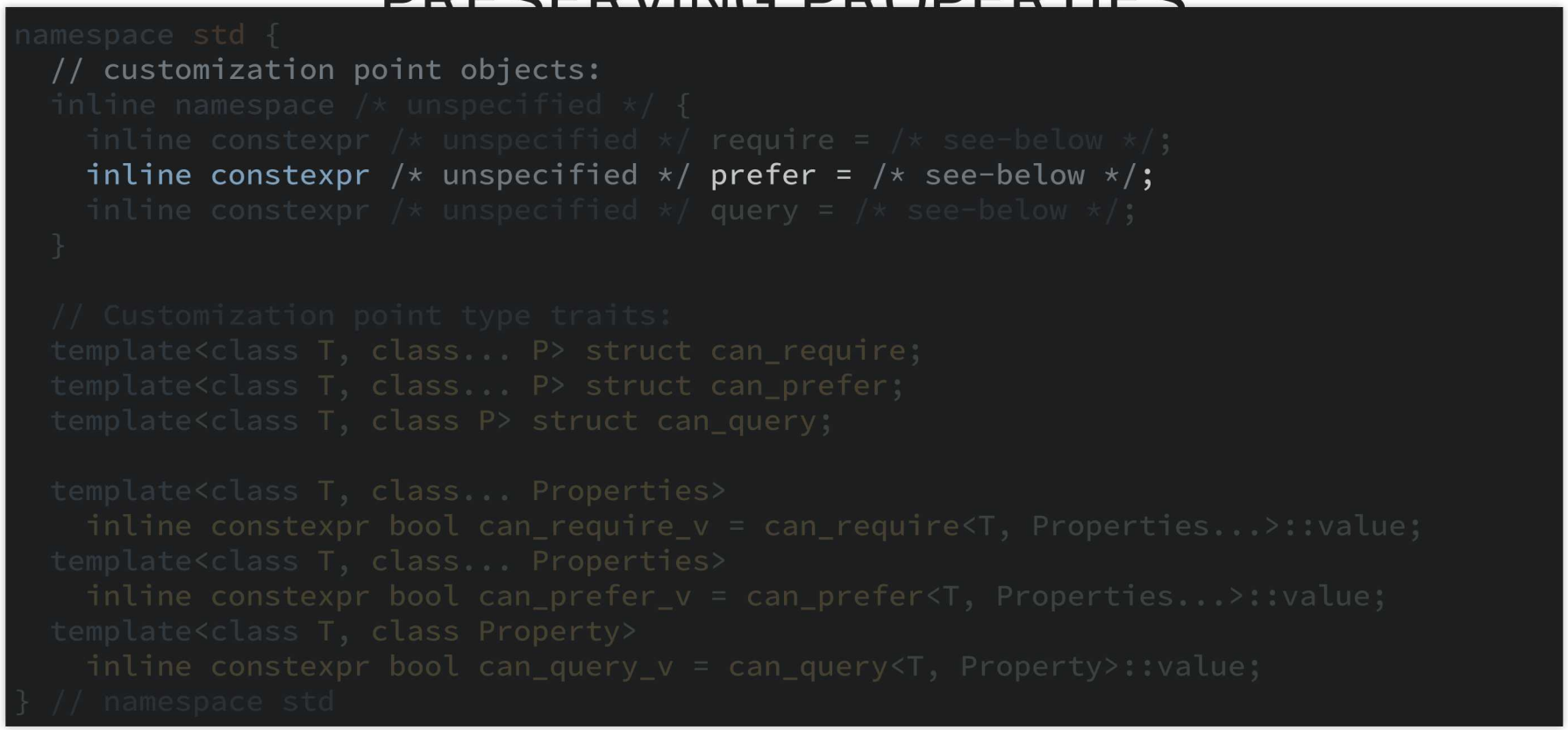
```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require = /* see-below */;
    inline constexpr /* unspecified */ prefer = /* see-below */;
    inline constexpr /* unspecified */ query = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class... P> struct can_require;
  template<class T, class... P> struct can_prefer;
  template<class T, class P> struct can_query;

  template<class T, class... Properties>
    inline constexpr bool can_require_v = can_require<T, Properties...>::value;
  template<class T, class... Properties>
    inline constexpr bool can_prefer_v = can_prefer<T, Properties...>::value;
  template<class T, class Property>
    inline constexpr bool can_query_v = can_query<T, Property>::value;
} // namespace std
```

HEADER <property> SYNOPSIS: CONCEPT-PRESERVING PROPERTIES

```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require = /* see-below */;
    inline constexpr /* unspecified */ prefer = /* see-below */;
    inline constexpr /* unspecified */ query = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class... P> struct can_require;
  template<class T, class... P> struct can_prefer;
  template<class T, class P> struct can_query;

  template<class T, class... Properties>
    inline constexpr bool can_require_v = can_require<T, Properties...>::value;
  template<class T, class... Properties>
    inline constexpr bool can_prefer_v = can_prefer<T, Properties...>::value;
  template<class T, class Property>
    inline constexpr bool can_query_v = can_query<T, Property>::value;
} // namespace std
```

A customization point object that returns an instance of the given object with the property. It can be the same instance if that object already has that property. Fails at compile time if it cannot require the property of the object given.

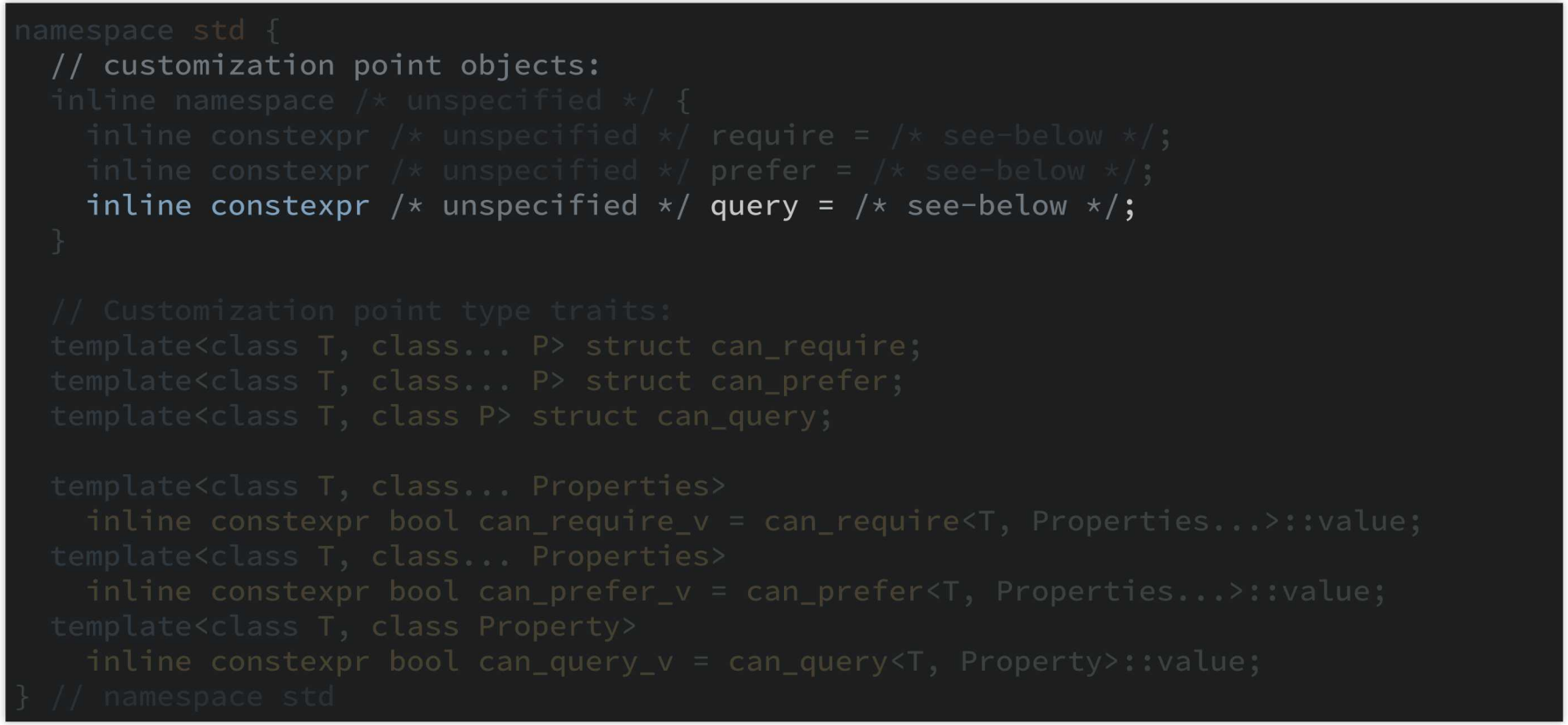
HEADER <property> SYNOPSIS: CONCEPT-PRESERVING PROPERTIES



```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require = /* see-below */;
    inline constexpr /* unspecified */ prefer = /* see-below */;
    inline constexpr /* unspecified */ query = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class... P> struct can_require;
  template<class T, class... P> struct can_prefer;
  template<class T, class P> struct can_query;

  template<class T, class... Properties>
    inline constexpr bool can_require_v = can_require<T, Properties...>::value;
  template<class T, class... Properties>
    inline constexpr bool can_prefer_v = can_prefer<T, Properties...>::value;
  template<class T, class Property>
    inline constexpr bool can_query_v = can_query<T, Property>::value;
} // namespace std
```

A customization point object that **`requires`** a property if it can, or returns the object as-is if it cannot.

HEADER <property> SYNOPSIS: CONCEPT-PRESERVING PROPERTIES



```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require = /* see-below */;
    inline constexpr /* unspecified */ prefer = /* see-below */;
    inline constexpr /* unspecified */ query = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class... P> struct can_require;
  template<class T, class... P> struct can_prefer;
  template<class T, class P> struct can_query;

  template<class T, class... Properties>
    inline constexpr bool can_require_v = can_require<T, Properties...>::value;
  template<class T, class... Properties>
    inline constexpr bool can_prefer_v = can_prefer<T, Properties...>::value;
  template<class T, class Property>
    inline constexpr bool can_query_v = can_query<T, Property>::value;
} // namespace std
```

A customization point object to query for the presence of a requirable property, or for the value of some other aspect of the object given, like its cache size or the alignment of its underlying data.

HEADER <property> SYNOPSIS: CONCEPT-PRESERVING PROPERTIES

```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require = /* see-below */;
    inline constexpr /* unspecified */ prefer = /* see-below */;
    inline constexpr /* unspecified */ query = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class... P> struct can_require;
  template<class T, class... P> struct can_prefer;
  template<class T, class P> struct can_query;

  template<class T, class... Properties>
    inline constexpr bool can_require_v = can_require<T, Properties...>::value;
  template<class T, class... Properties>
    inline constexpr bool can_prefer_v = can_prefer<T, Properties...>::value;
  template<class T, class Property>
    inline constexpr bool can_query_v = can_query<T, Property>::value;
} // namespace std
```

Type traits to determine the usability of the customization point objects statically.

HEADER <property> SYNOPSIS: CONCEPT-PRESERVING PROPERTIES

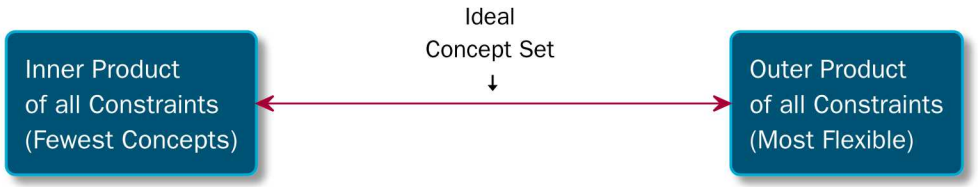
```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require = /* see-below */;
    inline constexpr /* unspecified */ prefer = /* see-below */;
    inline constexpr /* unspecified */ query = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class... P> struct can_require;
  template<class T, class... P> struct can_prefer;
  template<class T, class P> struct can_query;

  template<class T, class... Properties>
    inline constexpr bool can_require_v = can_require<T, Properties...>::value;
  template<class T, class... Properties>
    inline constexpr bool can_prefer_v = can_prefer<T, Properties...>::value;
  template<class T, class Property>
    inline constexpr bool can_query_v = can_query<T, Property>::value;
} // namespace std
```

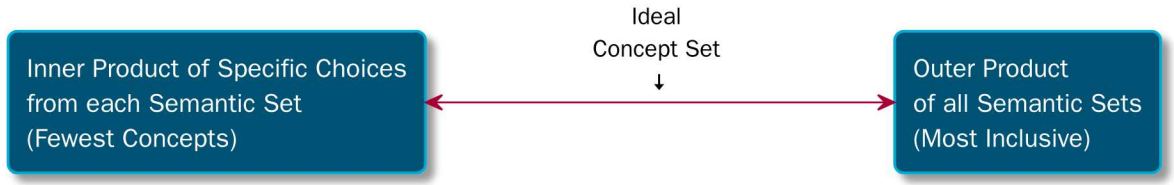
...and their _v versions, as usual.

HEADER <property> SYNOPSIS: CONCEPT-PRESERVING PROPERTIES

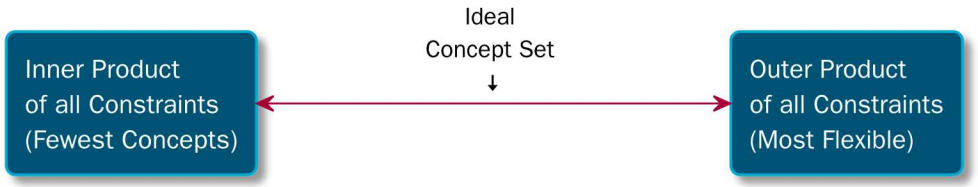
```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require = /* see-below */;
    inline constexpr /* unspecified */ prefer = /* see-below */;
    inline constexpr /* unspecified */ query = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class... P> struct can_require;
  template<class T, class... P> struct can_prefer;
  template<class T, class P> struct can_query;

  template<class T, class... Properties>
    inline constexpr bool can_require_v = can_require<T, Properties...>::value;
  template<class T, class... Properties>
    inline constexpr bool can_prefer_v = can_prefer<T, Properties...>::value;
  template<class T, class Property>
    inline constexpr bool can_query_v = can_query<T, Property>::value;
} // namespace std
```

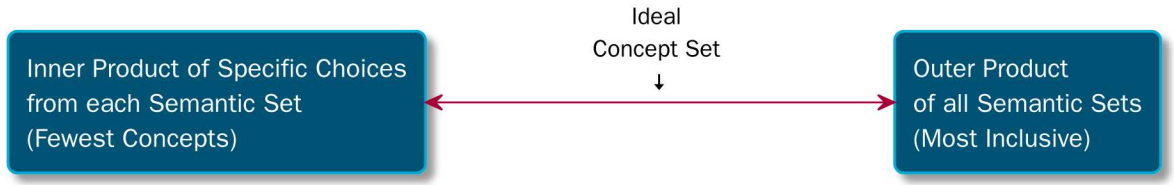
CONCEPT-ENFORCING PROPERTIES





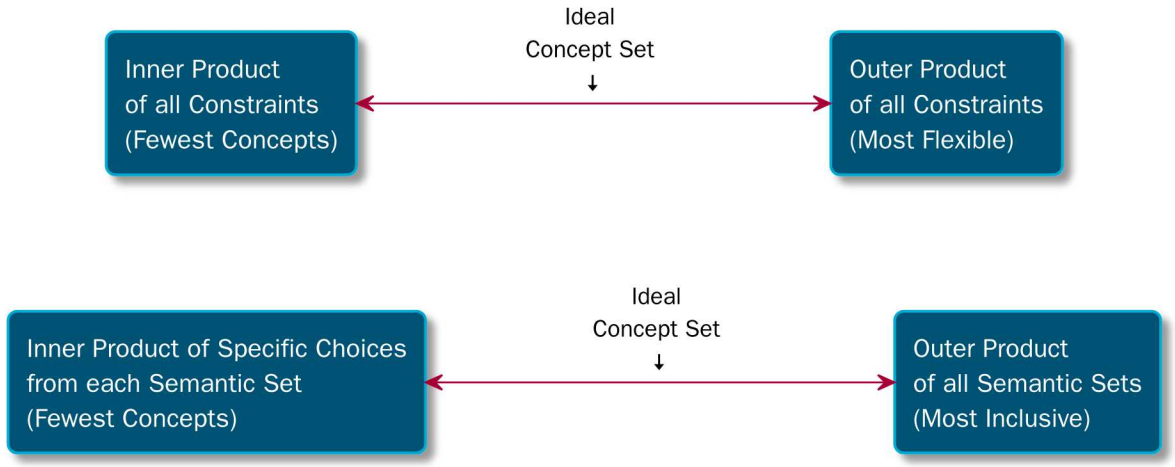
CONCEPT-ENFORCING PROPERTIES





Broadly speaking...

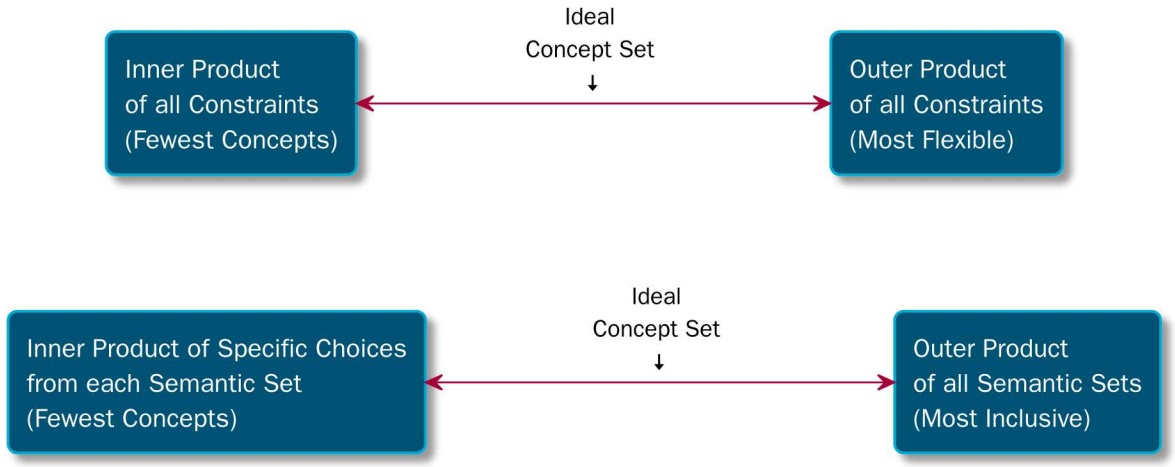
CONCEPT-ENFORCING PROPERTIES



Broadly speaking…

- *Concept-preserving* properties are useful when the cross-cutting concerns of a subset of the algorithms falls to the right (more flexible/inclusive side) of the otherwise ideal concept set.

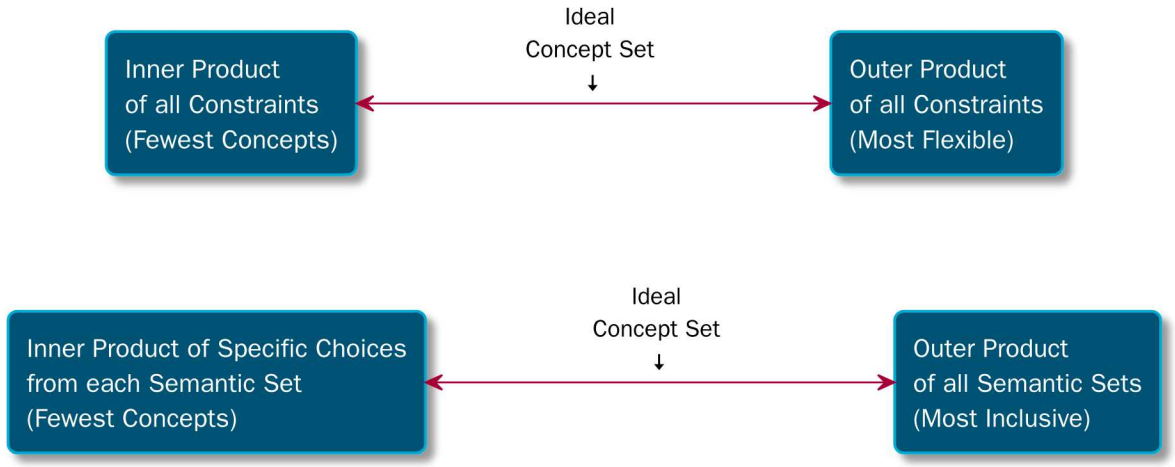
CONCEPT-ENFORCING PROPERTIES



Broadly speaking…

- *Concept-preserving* properties are useful when the cross-cutting concerns of a subset of the algorithms falls to the right (more flexible/inclusive side) of the otherwise ideal concept set.
- *Concept-enforcing* properties are useful when the needs of a subset of the algorithms falls to the left (fewer concepts) side of the otherwise ideal concept set.

CONCEPT-ENFORCING PROPERTIES



Broadly speaking…

- *Concept-preserving* properties are useful when the cross-cutting concerns of a subset of the algorithms falls to the right (more flexible/inclusive side) of the otherwise ideal concept set.
- *Concept-enforcing* properties are useful when the needs of a subset of the algorithms falls to the left (fewer concepts) side of the otherwise ideal concept set.
- When a particular cross-cutting concern is common to multiple concepts, there needs to be a way to obtain an object meeting the requirements of a different concept but that addresses the same cross-cutting concern.

HEADER <property> SYNOPSIS: CONCEPT-ENFORCING PROPERTIES

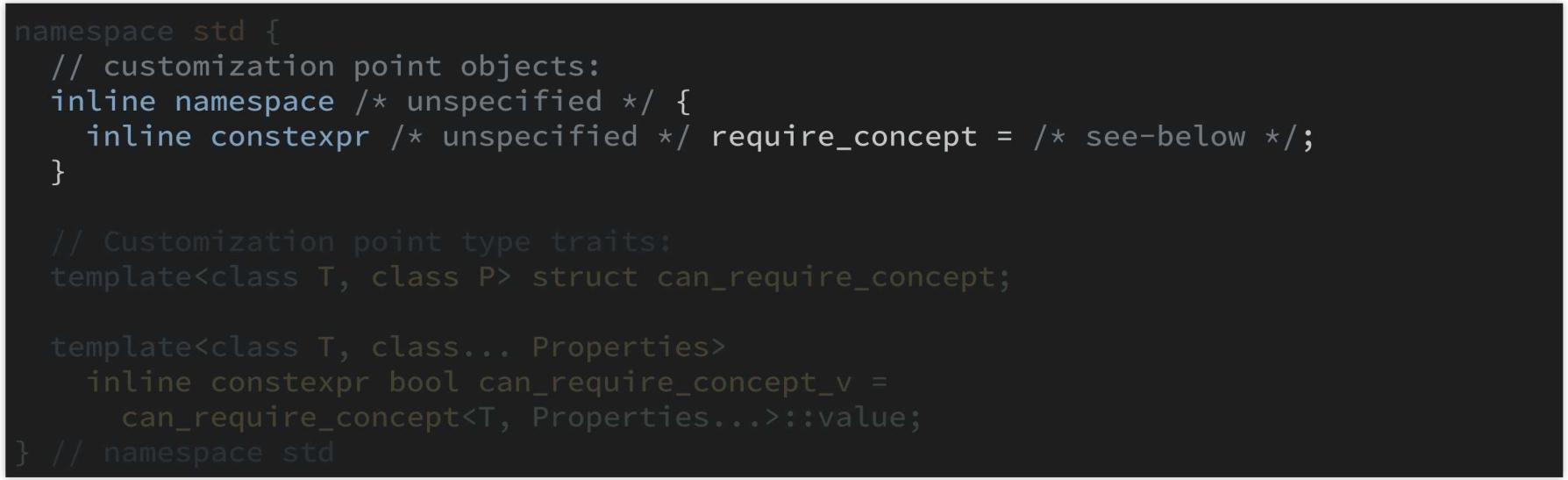
```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require_concept = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class P> struct can_require_concept;

  template<class T, class... Properties>
    inline constexpr bool can_require_concept_v =
      can_require_concept<T, Properties...>::value;
} // namespace std
```

HEADER <property> SYNOPSIS: CONCEPT-ENFORCING PROPERTIES

```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require_concept = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class P> struct can_require_concept;

  template<class T, class... Properties>
    inline constexpr bool can_require_concept_v =
      can_require_concept<T, Properties...>::value;
} // namespace std
```

A customization point object that returns an object with the same applicable concept-preserving properties but that meets the requirements of the concept associated with the given property.

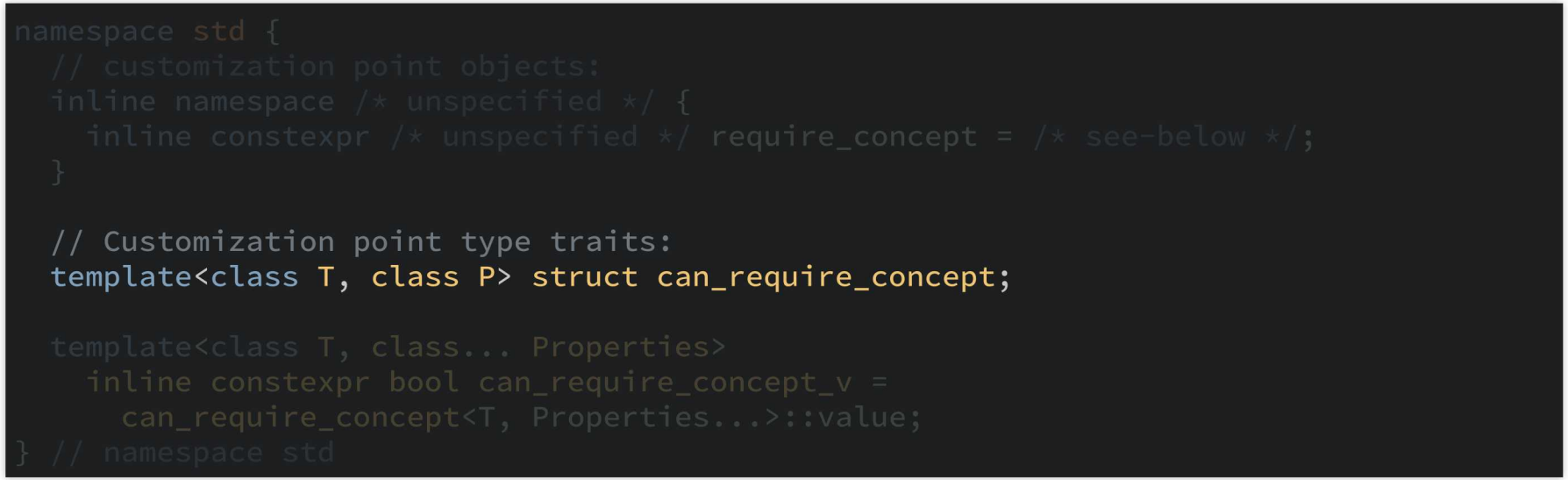
HEADER <property> SYNOPSIS: CONCEPT-ENFORCING PROPERTIES



```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require_concept = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class P> struct can_require_concept;

  template<class T, class... Properties>
    inline constexpr bool can_require_concept_v =
      can_require_concept<T, Properties...>::value;
} // namespace std
```

It can be the same instance if that object already meets the requirements of the associated concept.
Fails at compile time if no such object can be created.

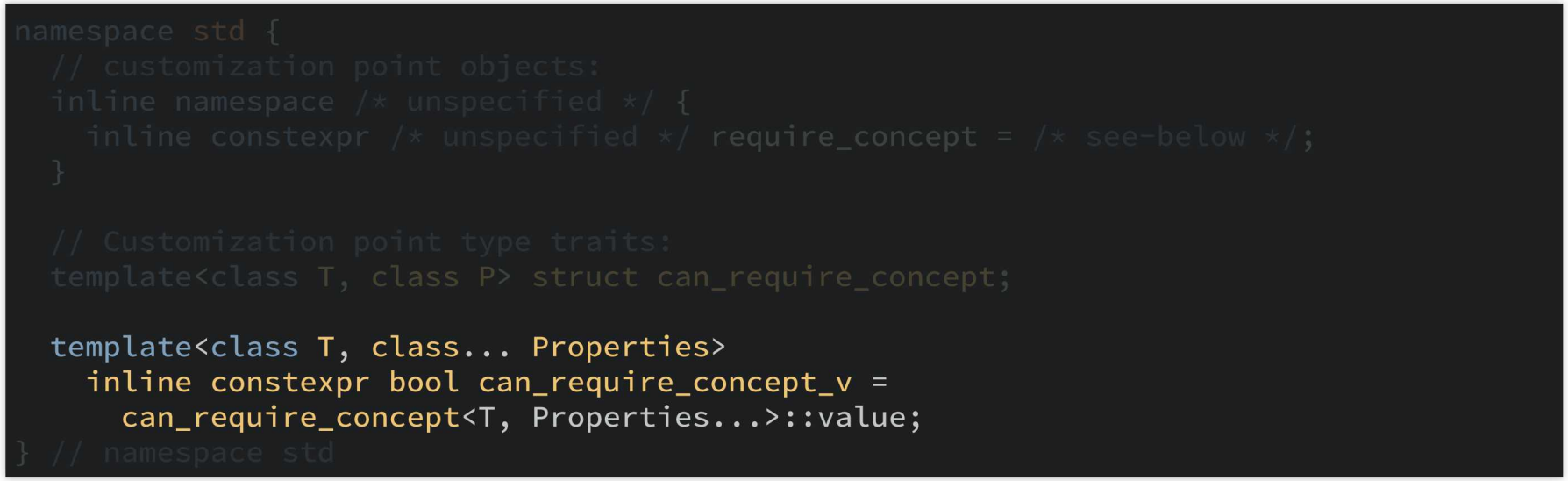
HEADER <property> SYNOPSIS: CONCEPT-ENFORCING PROPERTIES



```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require_concept = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class P> struct can_require_concept;

  template<class T, class... Properties>
    inline constexpr bool can_require_concept_v =
      can_require_concept<T, Properties...>::value;
} // namespace std
```

Type trait to determine the usability of the customization point object statically.

HEADER <property> SYNOPSIS: CONCEPT-ENFORCING PROPERTIES



```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require_concept = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class P> struct can_require_concept;

  template<class T, class... Properties>
    inline constexpr bool can_require_concept_v =
      can_require_concept<T, Properties...>::value;
} // namespace std
```

...and the _v version, as usual.

HEADER <property> SYNOPSIS: CONCEPT-ENFORCING PROPERTIES

```
namespace std {
  // customization point objects:
  inline namespace /* unspecified */ {
    inline constexpr /* unspecified */ require_concept = /* see-below */;
  }

  // Customization point type traits:
  template<class T, class P> struct can_require_concept;

  template<class T, class... Properties>
    inline constexpr bool can_require_concept_v =
      can_require_concept<T, Properties...>::value;
} // namespace std
```

PROPERTY APPLICABILITY TRAIT

PROPERTY APPLICABILITY TRAIT

- Concept-enforcing properties introduce a problem: how does one indicate that a concept-preserving property is "applicable" to a concept and must be preserved across a `require_concept()`?

PROPERTY APPLICABILITY TRAIT

- Concept-enforcing properties introduce a problem: how does one indicate that a concept-preserving property is "applicable" to a concept and must be preserved across a `require_concept()`?
- Simplest answer: specify that properties must declare the concept or concepts they apply to using a type trait.

PROPERTY APPLICABILITY TRAIT

- Concept-enforcing properties introduce a problem: how does one indicate that a concept-preserving property is "applicable" to a concept and must be preserved across a `require_concept()`?
- Simplest answer: specify that properties must declare the concept or concepts they apply to using a type trait.
  - This has the added benefit of restricting the cognitive effect of properties to the concept sets that they opt in to.

PROPERTY APPLICABILITY TRAIT

- Concept-enforcing properties introduce a problem: how does one indicate that a concept-preserving property is "applicable" to a concept and must be preserved across a `require_concept()`?
- Simplest answer: specify that properties must declare the concept or concepts they apply to using a type trait.
  - This has the added benefit of restricting the cognitive effect of properties to the concept sets that they opt in to.
  - A property intended for strings doesn't accidentally get applied to allocators, and allocator authors don't have to think about what happens if a string property is passed to their `require` implementation.

HEADER <property> SYNOPSIS: PROPERTY APPLICABILITY

```
namespace std {
  // Customization point type traits:
  template<class T, class P> struct is_applicable_property;

  template<class T, class... Properties>
    inline constexpr bool is_applicable_property_v =
      is_applicable_property<T, Properties...>::value;
} // namespace std
```

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the `Thing` concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the `Thing` concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

We've used the convention of naming the type of a property with an `_t` suffix.

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the **`Thing`** concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

This opts in to applicability to types meeting the requirements of **`Thing`**

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the **`Thing`** concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

This opts in to the **`require`** customization point

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the **`Thing`** concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

This opts in to the **`prefer`** customization point

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the `Thing` concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

Exposes the value of the query at compile time (if querying the property is a valid constant expression).

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the `Thing` concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

Properties must be equality comparable.

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the **`Thing`** concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

This is the type that a **`query`** on a polymorphic wrapper should return return when querying this property. Since querying **`caching`** just returns whether or not it is enabled, use **`bool`** here.

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the `Thing` concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

This will make more sense with non-boolean properties, but basically it allows generic code to check for a successful `prefer` of a property `p` using `std::query(std::prefer(x, p), p) == p.value()`

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the `Thing` concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

ANATOMY OF A PROPERTY

Let's implement the concept-preserving property (applicable to the `Thing` concept) from the earlier example.

```
struct caching_t {
  template <class T>
  static constexpr bool is_applicable_property_v = Thing<T>;
  static constexpr bool is_requirable = true;
  static constexpr bool is_preferable = true;
  template <class T>
  static constexpr bool static_query_v = std::query(T{}, caching_t{});
  constexpr auto operator<=>(caching_t) const noexcept = default;
  using polymorphic_query_result_type = bool;
  static constexpr bool value() { return true; }
};
inline constexpr caching_t caching = { };
```

(Note: this oversimplifies a bit; in reality, we'd want a pair of mutually exclusive properties that turn on and off caching, respectively.)

QUESTIONS?