Details Matter

Alisdair Meredith
BloombergLP
(Recently retired) C++ Library Working Group Chair

Topics Today

- Collected folk wisdom of a decade on standard committee, and occasional library implementer
- *not* a study in template metaprogramming - you get that elsewhere
- *not* going to make continous reference to allocators
  - best way to learn template metaprogramming trickery is to implement `std::allocator_traits`

Documentation

- Most important thing that developers like to neglect
- Document the specification/contract, not the implementation
- Good naming of classes/functions/objects makes readable code, but is not a substitute for documentation
- Good documentation does not excuse bad naming!

Documentation

- Most important thing that developers like to neglect
- Document the specification/contract, not the implementation
- Good naming of classes/functions/objects makes readable code, but is not a substitute for documentation
- Good documentation does not excuse bad naming!

Documentation is a Contract

- Tells the user their responsibilities in calling a function correctly
- Describes the library's responsibility for a correctly called function
- Makes clear whether a bug is due to the caller, the library, or both.
    - Potentially the bug is in the documentation, unclear contracts abound!

What Happens if a Contract is Violated?

- Undefined behavior?
- throw an exception?
- terminate the program?
- log and return?

What Happens if a Contract is Violated?

- Undefined behavior?
- throw an exception?
- terminate the program?
- log and return?
- Ultimately, the library author chooses how to support their customers
- Personal preference: use assertions with customizable handler function

Narrow vs. Wide Contract

- A wide contract has no preconditions, and so accepts all input
- A narrow contract is one with *any* preconditions, that the user is responsible for validating before making a call
  - Such validation may be working with known inputs rather than strictly writing checking code before each call
  - Often, pre-conditions bubble up through layers of contracts

Narrow vs. `noexcept`

- Design rule for std library *specifications*
  - `noexcept` only on wide contracts
  - conditional `noexcept` only when `noexcept` is vital, e.g., move or swap
- *Implementations* have freedom to strengthen specification
  - and always have, since 1998 standard
- Preserves implementors ability to respond to out-of-contract calls

Narrow or Wide?

```
auto sum(int a, int b, int c) -> int;
    // return the mathematical sum of a, b and c.
```

- Is this contract narrow or wide?

Narrow or Wide?

```
auto sum(int a, int b, int c) -> int;
   // return the mathematical sum of a, b and c.  The behavior
   // is undefined unless the sum can be represented by an ‘int’.
```

- Is this contract narrow or wide?
  - It is poorly documented, and implicitly narrow

Narrow or Wide?

```
auto sum(int a, int b, int c) -> int;
    // return the mathematical sum of a, b and c.  The behavior
    // is undefined unless the sum can be represented by an ‘int’.
{
    return a + b + c;
}
```

- Is this contract narrow or wide?
    - It is poorly documented, and implicitly narrow
- Is this implementation correct?

Narrow or Wide?

```
auto sum(int a, int b, int c) -> int;
   // return the mathematical sum of a, b and c.  The behavior
   // is undefined unless the sum can be represented by an ‘int’.
{
   return a + b + c;
}
```

- Is this contract narrow or wide?
  - It is poorly documented, and implicitly narrow
- Is this implementation correct?
  - `sum(INT_MAX, 2, -3)`

Bug in Contract or Implementation?

```
auto sum(int a, int b, int c) -> int
    // return the mathematical sum of a, b and c.  The behavior
    // is undefined unless the sum can be represented by an ‘int’.
{
    if (a < 0 != b <= 0) {
        int tmp = a + b;
        if (tmp <= 0 and c >= INT_MIN - tmp) { return tmp + c; }
        if (tmp >= 0 and c <= INT_MAX - tmp) { return tmp + c; }
    }
```

```
    // CONTRACT VIOLATION
}
```

Bug in Contract or Implementation?

```
auto sum(int a, int b, int c) -> int
   // return the mathematical sum of a, b and c.  The behavior
   // is undefined unless the sum can be represented by an ‘int’.
{
   if (a < 0 != b <= 0) {
      int tmp = a + b;
      if (tmp <= 0 and c >= INT_MIN - tmp) { return tmp + c; }
      if (tmp >= 0 and c <= INT_MAX - tmp) { return tmp + c; }
   }
   else if (a < 0 != c <= 0) {
      int tmp = a + c;
      if (tmp <= 0 and b >= INT_MIN - tmp) { return tmp + b; }
      if (tmp >= 0 and b <= INT_MAX - tmp) { return tmp + b; }
   }
   else if (b < 0 != c <= 0) {
      int tmp = b + c;
      if (tmp <= 0 and a >= INT_MIN - tmp) { return tmp + a; }
      if (tmp >= 0 and a <= INT_MAX - tmp) { return tmp + a; }
   }

   // CONTRACT VIOLATION
}
```

Bug in Contract or Implementation?

```
auto sum(int a, int b, int c) -> int
   // return the mathematical sum of a, b and c.  The behavior
   // is undefined unless the sum can be represented by an ‘int’.
{
   if (a < 0 != b <= 0) {
      int tmp = a + b;
      if (tmp <= 0 and c >= INT_MIN - tmp) { return tmp + c; }
      if (tmp >= 0 and c <= INT_MAX - tmp) { return tmp + c; }
   }
   else if (a < 0 != c <= 0) {
      int tmp = a + c;
      if (tmp <= 0 and b >= INT_MIN - tmp) { return tmp + b; }
      if (tmp >= 0 and b <= INT_MAX - tmp) { return tmp + b; }
   }
   else if (b < 0 != c <= 0) {
      int tmp = b + c;
      if (tmp <= 0 and a >= INT_MIN - tmp) { return tmp + a; }
      if (tmp >= 0 and a <= INT_MAX - tmp) { return tmp + a; }
   }
   // a,b and c all have the same sign
   if (a >= 0) {
      if (a <= INT_MAX - b) {
         int tmp = a + b;
         if (c <= INT_MAX - tmp) { return tmp + c; }
      }
   }
   else if (a > INT_MIN - b) {
      int tmp = a + b;
      if (c >= INT_MIN - tmp) { return tmp + c; }
   }

   // CONTRACT VIOLATION
}
```

Bug in Contract or Implementation?

```
auto sum(int a, int b, int c) -> int
   // return the mathematical sum of a, b and c.  The behavior
   // is undefined unless the sum can be represented by an ‘int’.
{
   if (a < 0 != b <= 0) {
      int tmp = a + b;
      if (tmp <= 0 and c >= INT_MIN - tmp) { return tmp + c; }
      if (tmp >= 0 and c <= INT_MAX - tmp) { return tmp + c; }
   }
   else if (a < 0 != c <= 0) {
      int tmp = a + c;
      if (tmp <= 0 and b >= INT_MIN - tmp) { return tmp + b; }
      if (tmp >= 0 and b <= INT_MAX - tmp) { return tmp + b; }
   }
   else if (b < 0 != c <= 0) {
      int tmp = b + c;
      if (tmp <= 0 and a >= INT_MIN - tmp) { return tmp + a; }
      if (tmp >= 0 and a <= INT_MAX - tmp) { return tmp + a; }
   }
   // a,b and c all have the same sign
   if (a >= 0) {
      if (a <= INT_MAX - b) {
         int tmp = a + b;
         if (c <= INT_MAX - tmp) { return tmp + c; }
      }
   }
   else if (a > INT_MIN - b) {
      int tmp = a + b;
      if (c >= INT_MIN - tmp) { return tmp + c; }
   }

   // CONTRACT VIOLATION ?
}
```

Bug in Contract or Implementation?

```
auto sum(int a, int b, int c) -> int
   // return the mathematical sum of a, b and c.  The behavior
   // is undefined unless the sum can be represented by an ‘int’.
{
   if (a < 0 != b <= 0) {
      int tmp = a + b;
      if (tmp <= 0 and c >= INT_MIN - tmp) { return tmp + c; }
      if (tmp >= 0 and c <= INT_MAX - tmp) { return tmp + c; }
   }
   else if (a < 0 != c <= 0) {
      int tmp = a + c;
      if (tmp <= 0 and b >= INT_MIN - tmp) { return tmp + b; }
      if (tmp >= 0 and b <= INT_MAX - tmp) { return tmp + b; }
   }
   else if (b < 0 != c <= 0) {
      int tmp = b + c;
      if (tmp <= 0 and a >= INT_MIN - tmp) { return tmp + a; }
      if (tmp >= 0 and a <= INT_MAX - tmp) { return tmp + a; }
   }
   // a,b and c all have the same sign
   if (a >= 0) {
      if (a <= INT_MAX - b) {
         int tmp = a + b;
         if (c <= INT_MAX - tmp) { return tmp + c; }
      }
   }
   else if (a > INT_MIN - b) {
      int tmp = a + b;
      if (c >= INT_MIN - tmp) { return tmp + c; }
   }

   // CONTRACT VIOLATION ?   sum(0, -2, -3)
}
```

Narrow or Wide?

```
auto sum(int a, int b, int c) -> int;
    // return the mathematical sum of a, b and c.  The behavior
    // is undefined unless the sum can be represented by an ‘int’,
    // and the sum ‘a + b’ can be represented by an ‘int’.
{
    return (a + b) + c;
}
```

- Is this contract narrow or wide?
  - It is poorly documented, and implicitly narrow
- Is this implementation correct?
  - `sum(INT_MAX, 2, -3) : out of contract`

Is Narrow or Wide Better?

- Narrower contracts are often easier to implement
- Wider contracts are often easier to use
  - Unless they result in a larger set of failure post-conditions to test
- Exercise engineering judgement

A Simple Correction

```
auto sum(int a, int b, int c) -> int;
    // return the mathematical sum of a, b and c.  The behavior
    // is undefined unless the sum can be represented by an ‘int’.
{

    long long tmp = a;
    tmp += b;
    tmp += c;
    if (numeric_limits<int>::min() <= tmp and
        numeric_limits<int>::max() >= tmp) {
        return static_cast<int>(tmp); // cast to avoid warnings
    }
    // OUT OF CONTRACT
}
```

A Portable Correction

```
auto sum(int a, int b, int c) -> int;
    // return the mathematical sum of a, b and c.  The behavior
    // is undefined unless the sum can be represented by an ‘int’.
{
    static_assert(sizeof(long long) > sizeof(int), “Cannot be same”);
    long long tmp = a;
    tmp += b;
    tmp += c;
    if (numeric_limits<int>::min() <= tmp and
        numeric_limits<int>::max() >= tmp) {
        return static_cast<int>(tmp); // cast to avoid warnings
    }
    // OUT OF CONTRACT
}
```

Simple vs. Complex Constraints

- Design a class `integer_range`
  - Represents a sequence of consecutive integers
    - e.g. {1,2,3,4,5} or {18,19,20}
      - Starting from a `first` number
- How do we represent this in the interface?

Simple vs. Complex Constraints

- Design a class `integer_range`
  - Represents a sequence of consecutive integers
    - e.g. {1,2,3,4,5} or {18,19,20}
      - Starting from a `first` number
- How do we represent this in the interface?
  - `first` / `last`
  - `first` / `length`

Design Questions for `integer_range`

- Do we support both increasing and decreasing sequences?
- Do we need to constrain `last` to be representable as an `int`?
- What are the most important operations on our range?
  - test for membership
  - translate
  - combine

Design Questions for `integer_range`

- Do we support both increasing and decreasing sequences?
  Increasing only for this example
- Do we need to constrain `last` to be representable as an `int`?
  Useful, but perhaps not necessary
- What are the most important operations on our range?
  - test for membership — Is a number/subrange in the range?
  - translate — Move range higher/lower
  - combine — Union of two ranges, including range between

Design A

- `first` is `int`, `length` is `unsigned` - unidirectional list
- precondition calling `last()` that it is representable
- `first` and `length` can be set independently
  - translate, and grow/expand/shrink
- combine/join has a precondition, that final `length` is representable

Design B

- `first` and `last` are `int`, *complex* constraint: `first <= last`
- `first`/`last`/`length` queries have wide contracts
- `first` and `last` must always be set as a pair
  - otherwise, may pass through invalid state
- combine/join has a wide contract

Is Design A or B Better?

- A is often simpler to work with, as it has *simple* constraints between attributes
    - Simply constrained attributes can be manipulated independently
    - Complex constraints require all mutations to go through functions that establish invariants
- B is closer to a typical mental model
    - In particular, B is easier to iterate, a likely use case
- A can represent more valid ranges than B, but are the additional ranges useful?

What is a moved-from state?

```
auto        x = “Hello world!”s;
std::string y{ std::move(x) };
```

What is a moved-from state?

```
auto        x = “Hello world!”s;
std::string y{ std::move(x) };
```

- What is state of x after y is initialized?
- a) empty
- b) “Hello World!”
- c) I don’t know
- d) Undefined behavior to ask

What is a moved-from state?

```
auto        x = “Hello world!”s;
std::string y{ std::move(x) };
```

- What *should be* state of x after y is initialized?
- a) empty
- b) “Hello World!”
- c) I don’t know
- d) Undefined behavior to ask

What is a moved-from state?

```
auto        x = “Hello world!”s;
std::string y{ std::move(x) };
```

- What **is** state of x after y is initialized?
- a) empty
- b) “Hello World!”
- c) I don’t know
- d) Undefined behavior to ask

What is a moved-from state?

```
auto        x = “Hello world!”s;
std::string y{ std::move(x) };
```

- What is state of x after y is initialized?
- a) empty
- b) “Hello World!”
- c) I don’t know
- d) Undefined behavior to ask

What is a moved-from state?

A moved-from state is “valid but unspecified”.

It is safe to call functions without preconditions (wide contracts), and you can trust the results. You may learn enough about the current state to safely call (some) functions with narrow contracts too.

What is a moved-from state?

A moved-from state is “valid but unspecified”.

It is safe to call functions without preconditions (wide contracts), and you can trust the results. You may learn enough about the current state to safely call (some) functions with narrow contracts too.

```
if (!v.empty()) { use(v.front()); }
```

Templates are a frequent source of bad contracts

- Document exactly what is required of each template parameter
- Do not rely in the implementation on anything not explicitly required
  - unquestioned assumptions abound…
- Collect common requirements as named Concepts

What makes a Bad Contract

- Testing requires many awkward types may indicate overly general contracts
  - difficult to implement
  - difficult to test
  - tricky for user extension, especially as a concept
  - overly general support may impact performance, e.g., `std::addressof` vs `operator&`
- Too restrictive, and contracts become tricky to read and use

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`
  - or a type convertible to `bool`

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`
  - or a type convertible to `bool`
- or may be a function pointer (or function reference)

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`
  - or a type convertible to `bool`
- or may be a function pointer (or function reference)
- parameters may pass by `const &`

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`
  - or a type convertible to `bool`
- or may be a function pointer (or function reference)
- parameters may pass by `const &`
  - or by non-const &

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`
  - or a type convertible to `bool`
- or may be a function pointer (or function reference)
- parameters may pass by `const &`
  - or by non-const &
  - or by value

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`
  - or a type convertible to `bool`
- or may be a function pointer (or function reference)
- parameters may pass by `const &`
  - or by non-const &
  - or by value
  - or may be two different types convertible to T

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`
  - or a type convertible to `bool`
- or may be a function pointer (or function reference)
- parameters may pass by `const &`
  - or by non-const &
  - or by value
  - or may be two different types convertible to `T`
    - or convertible to a (`const`) reference to `T`

E.g., What is a Predicate?

- function taking two parameters of type `T`, returning a `bool`
  - or a type convertible to `bool`
- or may be a function pointer (or function reference)
- parameters may pass by `const &`
  - or by non-const &
  - or by value
  - or may be two different types convertible to `T`
    - or convertible to a (`const`) reference to `T`
  - or deduce from a forwarding reference

What is a Predicate?

- or may be a class overloading `bool operator()(T, T) const` as above
  - or just `bool operator()(T, T)`
  - or maybe a function template that deduces `T` (and `U`)
    - or taking more than 2 parameters, where extra have default arguments

What is a Predicate?

- or may be a class overloading `bool operator()(T, T) const` as above
  - or just `bool operator()(T, T)`
  - or maybe a function template that deduces `T` (and `U`)
    - or taking more than 2 parameters, where extra have default arguments
- or may be a class convertible to such a function pointer (or reference)

What is a Predicate?

- or may be a class overloading `bool operator()(T, T) const` as above
  - or just `bool operator()(T, T)`
  - or maybe a function template that deduces `T` (and `U`)
    - or taking more than 2 parameters, where extra have default arguments
- or may be a class convertible to such a function pointer (or reference)
- or the result of a (generic) lambda expression *[subset already covered?]*

What is a Predicate?

- or may be a class overloading `bool operator()(T, T) const` as above
  - or just `bool operator()(T, T)`
  - or maybe a function template that deduces `T` (and `U`)
    - or taking more than 2 parameters, where extra have default arguments
- or may be a class convertible to such a function pointer (or reference)
- or the result of a (generic) lambda expression *[subset already covered?]*
- remember classes may be unions or final classes

What is a Predicate?

- or may be a class overloading `bool operator()(T, T) const` as above
  - or just `bool operator()(T, T)`
  - or maybe a function template that deduces `T` (and `U`)
    - or taking more than 2 parameters, where extra have default arguments
- or may be a class convertible to such a function pointer (or reference)
- or the result of a (generic) lambda expression *[subset already covered?]*
- remember classes may be unions or final classes
  - and might not be default constructible

What is a Predicate?

- result type that is convertible-to-bool
  - might have an `explicit` conversion operator
  - might be move-only
  - might not be default constructible
  - might overload `operator&&`, breaking short-circuit evaluation
  - might overload `operator||`, breaking short-circuit evaluation
  - might overload `operator&`
  - might overload `operator,`
  - might overload `operator!`
  - might overload `operator==` and/or `operator!=`

Examples from the Standard Library

Exception Classes

- classes designed for use reporting exceptional conditions
- any class allowed, some work better than others
- all std exceptions derive from `std::exception`
  - `virtual char const * what() const noexcept;`
- Often designed as a type hierarchy, where type connotes some category of failure
  - `std::runtime_error`, `std::logic_error`, `std::system_error`

Exceptions should not throw

- What happens when we throw an exception?
  - system makes a copy in a special memory location
- What happens when we catch by-value?
  - system make another copy
- What happens when a copy constructor throws?
  - it depends…

Exception thrown by copy

- If the initial *exception object* is being constructed, the new exception is thrown instead

- If copying from the currently active *exception object*, `std::terminate` is called

- Classes intended for use as exceptions in `std` all have non-throwing, `noexcept` copy constructors

Standard Exception Classes

- `std::exception`, `std::bad_alloc`, `std::bad_cast` etc.
  - hold just a pointer to a string literal
- `std::runtime_error`, `std::logic_error` constructors are passed a string
  - copy string in user-called constructor - may throw
  - internally stored as immutable reference-counted string - copies will not throw
  - C++11: reference count needs to be thread safe

Broken Symmetry

- `vector<bool>` : proxy container failure
- `basic_string` : CoW and short string optimizations
- `shared_ptr<T[]>` : removes idiomatic methods
  (Library Fundamentals TS)
- `array<T,N>` : no allocator support for elements
  - `array<T, 0>` : special implementation, one interface

Array<T,0>

Why does `array<T,0>` have `front` and `back` functions that are always out of contract if called?

Array<T,0>

Why does `array<T,0>` have `front` and `back` functions that are always out of contract if called?

```
template <typename Container>
auto address_of_first(Container &c) {
    return c.empty()
           ? nullptr
           : std::addressof(c.front());
}
```

Follow the Design Language

- Containers, Iterators and Algorithms
- Traits

Type Traits

- Naming: is_/add_/remove_
- Nested ::type member
- Derive from std::integral_constant

Type Traits

- Evolve with the language
- C++14: _t for alias template
  - easier user experience without typename
- Library Fundamentals: _v for variable template
  - simpler programming model that templates

Containers

- Manage a collection of elements
- Expose a sequence through iterators
    - begin/end support ‘for’ loop
    - c prefix to return a const_iterator
    - r prefix to return a reverse_iterator
- well documented iterator and reference invalidation guarantees
    - swap never invalidates
    - insert depends on container
    - iterators/references may have different guarantees e.g., unordered containers
- May offer optimized interface
    - ‘find’ for (unordered) associative containers
    - ‘sort’ for list
- Allocators allow customization of memory management

Container Members

- Common typedefs allow for adapters
- optimized swap
- empty
- insert/emplace (C++11 : take a const_iterator for reference into same container)
- front/back for sequence containers
- constant time ‘size’

Iterators

- Form a conceptual hierarchy
  - Input iterator
  - Forward iterator (stable iteration, return a reference)
  - Bidirectional iterator
  - Random access iterator (also supports operator[])
  - Contiguous iterator (C++17)

Iterator Traits

- Common interface
  - operators: ++ == * ->
- Common types accessed through `iterator_traits`
  - `value_type`
  - `iterator_category`
  - `difference_type`
  - `reference`
  - `pointer`

Output iterators

- Oddball, no meaningful traits
- Common `operator++` interface
- Does not even require `operator==`
- Forward iterators are both Input and Output iterators
  - yet you cannot write through a `const_iterator`

Working with Iterators

- `iterator_traits` transparently supports pointers as iterators : algorithms should always use traits
- user defined iterators should supply typedefs in iterator type, and let the traits deduce : simplest implementation
- C++17: traits typedefs not present if cannot deduce : supports SFINAE detection of traits

Tag dispatch

- Some algorithms have optimized implementations taking advantage of iterator category
  - `distance`, `find` subsequence, `partition`
- Technique : dispatch to a second implementation based on `iterator_category`
  - Refined: use SFINAE to avoid unused argument

Tag Dispatch

```
template <typename ForwardIterator>
auto distance( ForwardIterator first
             , ForwardIterator last
             )
  -> typename iterator_traits<FrowardIterator>::difference_type;
```

Tag Dispatch: 1

```
template <typename ForwardIterator>
auto distance( ForwardIterator first, ForwardIterator last)
  -> enable_if_t<
       is_same_v< random_access_iterator_tag,
typename iterator_traits<ForwardIterator>::iterator_category>
   , typename iterator_traits<ForwardIterator>::difference_type> {
   return last - first;
}

template <typename ForwardIterator>
auto distance( ForwardIterator first, ForwardIterator last)
  -> enable_if_t<
      !is_same_v< random_access_iterator_tag,
typename iterator_traits<ForwardIterator>::iterator_category>
   , typename iterator_traits<ForwardIterator>::difference_type> {

   typename iterator_traits<ForwardIterator>::difference_type result{0};
   while (last != first) { ++first; ++result; }
   return result;
}
```

Tag Dispatch: 2

```
template <typename ForwardIterator>
auto distance( ForwardIterator first, ForwardIterator last)
  -> typename iterator_traits<ForwardIterator>::difference_type> {
   return impl_distance(first, last,
          typename iterator_traits<ForwardIterator>::iterator_category{});
}

template <typename ForwardIterator>
auto impl_distance( ForwardIterator first, ForwardIterator last
                                         , forward_iterator_tag)
  -> typename iterator_traits<ForwardIterator>::difference_type> {
   typename iterator_traits<ForwardIterator>::difference_type result = 0;
   while (last != first) { ++first; ++result; }
   return result;
}

template <typename ForwardIterator>
auto impl_distance( ForwardIterator first, ForwardIterator last
                                         , random_access_iterator_tag)
  -> typename iterator_traits<ForwardIterator>::difference_type> {
   return last - first;
}
```

Tag Dispatch: fail

```
template <typename ForwardIterator>
auto distance( ForwardIterator first, ForwardIterator last)
  -> typename iterator_traits<ForwardIterator>::difference_type> {
   if(is_same_v< random_access_iterator_tag,
            typename iterator_traits<ForwardIterator>::iterator_category>)
   {
      return last - first;
   }

   typename iterator_traits<ForwardIterator>::difference_type result{0};
   while (last != first) { ++first; ++result; }
   return result;
}
```

Implementing Templates

- use the traits, they catch oft-forgotten corners
  - add_lvalue_reference vs. T&?
  - add_const vs. T const vs. const T?
  - should array and function types decay?

Implementing Templates

- use the traits, they catch oft-forgotten corners
  - add_lvalue_reference vs. T&?
    (cv-qualified) void
  - add_const vs. T const vs. const T?
    function types
  - should array and function types decay?
    decay_t<TYPE> if intended

Testing

- Recurring theme, especially for templates
- Black box testing against the documented contract
- White box testing against the implementation and its optimizations
- Expect considerably more test code than library implementation code

Testing Templates

- Test against minimal documented requirements
- Create a battery of common test types
- Test your implementation, not the compiler
  - (mostly)

Testing Algorithms

- use iterator adaptors to test with strictest iterator
- special iterators might test empty ranges
- remember to test awkward value types
  - e.g., overload `operator&`, `operator,`
  - non-default/copy/move constructible
- awkward iterators have `operator==` return a `bool`-like type, that breaks short circuit evaluation

Types to Stress Templates

- not default constructible
- move-only
- immutable
- explicit copy/move/default constructors
- overload awkward operators: & , && ||
- Convert to/from anything
- `union`, or C++11 `final` class
- use reference qualifiers on key methods, such as operators
- awkward in multiple dimensions

What I learned from tuple

```
template <typename ...TYPES>
struct tuple : TYPES... {
   tuple();
   tuple(TYPES const& ...args);

   // … more stuff
};
```

What I learned from tuple

```
template <typename ...TYPES>
struct tuple : TYPES... {
    tuple();
    tuple(TYPES const& ...args);

    // … more stuff
};
```

What if TYPES contains:

- fundamental type
- enum
- pointer or reference
- union

What I learned from tuple

```
template <typename ...TYPES>
struct tuple : Wrap<TYPES>... {
    tuple();
    tuple(TYPES const& ...args);

    // … more stuff
};
```

What if TYPES contains:

- fundamental type
- enum
- pointer or reference
- union

What I learned from tuple

```
template <typename ...TYPES>
struct tuple : Wrap<TYPES>... {
   tuple();
   tuple(TYPES const& ...args);

   // … more stuff
};
```

What if TYPES contains:

- fundamental type
- enum
- pointer or reference
- union
- duplicate types

What I learned from tuple

```
template <typename ...TYPES>
struct tuple : Wrap<N, TYPES>... {
    tuple();
    tuple(TYPES const& ...args);

    // … more stuff
};
```

What if TYPES contains:

- fundamental type
- enum
- pointer or reference
- union
- duplicate types

What I learned from tuple

```
template <typename ...TYPES>
struct tuple : Wrap<N, TYPES>... {
    tuple();
    tuple(TYPES const& ...args);

    // … more stuff
};
```

What if TYPES contains:

- fundamental type
- enum
- pointer or reference
- union
- duplicate types

How will we manufacture N?

What I learned from tuple

```
template <typename INDEXER, typename ...TYPES>
struct tuple_impl;  // primary template is not defined

template <size_t ...INDEX, typename ...TYPES>
struct tuple_impl<index_sequence<INDEX...>, TYPES...>
      : Wrap<INDEX, TYPES>... {
    tuple_impl();
    tuple_impl(TYPES const& ...args);

    // … more stuff
};
```

What I learned from tuple

```
template <typename INDEXER, typename ...TYPES>
struct tuple_impl;  // primary template is not defined

template <size_t ...INDEX, typename ...TYPES>
struct tuple_impl<index_sequence<INDEX...>, TYPES...>
     : Wrap<INDEX, TYPES>... {
    tuple_impl();
    tuple_impl(TYPES const& ...args);

    // … more stuff
};


template <typename ...TYPES>
using tuple = tuple_impl<index_sequence_for<TYPES...>, TYPES...>;
```

What I learned from tuple

```
template <typename INDEXER, typename ...TYPES>
struct tuple_impl;  // primary template is not defined

template <size_t ...INDEX, typename ...TYPES>
struct tuple_impl<index_sequence<INDEX...>, TYPES...>
     : Wrap<INDEX, TYPES>... {
    tuple_impl();
    tuple_impl(TYPES const& ...args);

    // … more stuff
};
```

~~template <typename ...TYPES>~~
~~using tuple = tuple_impl<index_sequence_for<TYPES...>, TYPES...>;~~

What I learned from tuple

```
template <typename INDEXER, typename ...TYPES>
struct tuple_impl;  // primary template is not defined

template <size_t ...INDEX, typename ...TYPES>
struct tuple_impl<index_sequence<INDEX...>, TYPES...>
     : Wrap<INDEX, TYPES>... {
    tuple_impl();
    tuple_impl(TYPES const& ...args);

    // … more stuff
};


template <typename ...TYPES>
struct tuple : tuple_impl<index_sequence_for<TYPES...>, TYPES...>{
    using tuple_iml::tuple_impl;
};
```

Another Awkward Case

```
namespace my {

struct stuff {
   int data;
};

auto operator<<(std::ostream &os, stuff const & x)
  -> std::ostream & {
   return os << "stuff{" << x.data << "}";
}

} // namespace my
```

Another Awkward Case

```
namespace my {

struct stuff {
   int data;
};

auto operator<<(std::ostream &os, stuff const & x)
  -> std::ostream & {
   return os << "stuff{" << x.data << "}";
}

} // namespace my


int main() {
   std::tuple<int, my::stuff> y{1, 2};
   std::cout << y;
}
```

Argument Dependent Lookup

- The associated set of namespaces for a class template instantiation are:
  - The (innermost) namespace the class template itself lives in
  - The (innermost) namespace of each (direct and indirect) base class
  - (nested classes only) The innermost namespace of the class it is a member of (and of its direct and indirect base classes)(recursively for multiple levels of nesting)
  - The set of associated namespaces for each template type parameter
  - The innermost namespace of any template template parameters
  - Plus inline namespace associations

What I learned from tuple

```
template <typename INDEXER, typename ...TYPES>
struct tuple_impl;  // primary template is not defined

template <size_t ...INDEX, typename ...TYPES>
struct tuple_impl<index_sequence<INDEX...>, TYPES...>
     : Wrap<INDEX, TYPES>... {
    tuple_impl();
    tuple_impl(TYPES const& ...args);

    // … more stuff
};


template <typename ...TYPES>
struct tuple : tuple_impl<index_sequence_for<TYPES...>, TYPES...>{
    using tuple_iml::tuple_impl;
};
```

What I learned from tuple

```
template <typename INDEXER, typename ...TYPES>
struct tuple_impl;  // primary template is not defined

template <size_t ...INDEX, typename ...TYPES>
struct tuple_impl<index_sequence<INDEX...>, TYPES...>
     : Wrap<INDEX, TYPES>... {
    tuple_impl();
    tuple_impl(TYPES const& ...args);

    // … more stuff
};


template <typename ...TYPES>
struct tuple {
    tuple_iml<TYPES...> data;
};
```

Squeezing Unused Bytes

- Why pay for what you don’t use?
- What is size of std::less, or std::allocator?
  - all objects require a unique address, so at least size 1
  - Empty base-class sub-objects can share the address of their derived object, and so effectively have 0-size when derived from

Deploying Empty Base

- If a class is empty, derive from it
- Otherwise make it a data member
- Have a member function return a reference to the object, whether it be a base or member
- Implement container using that by-reference function

```
template < class Key
         , class Compare   = less<Key>
         , class Allocator = allocator<Key>
         >
class set {
private:
   NodeType *root;
   Compare   comp;
   Allocator alloc;

   Allocator &       allocator()       noexcept { return alloc; }
   Allocator const & allocator() const noexcept { return alloc; }
   Compare &         comparator()      noexcept { return comp; }
   Compare   const & comparator() const noexcept { return comp; }

public:
   typedef Compare       key_compare;
   typedef Compare       value_compare;
   typedef Allocator     allocator_type;

   allocator_type get_allocator() const noexcept {
      return allocator();
   }

// observers:
   key_compare   key_comp()   const { return comparator(); }
   value_compare value_comp() const { return comparator(); }
};
```

```
template < class Key
         , class Compare   = less<Key>
         , class Allocator = allocator<Key>
         >
class set : private Compare, Allocator {
private:
   NodeType *root;



   Allocator &       allocator()       noexcept { return *this; }
   Allocator const & allocator() const noexcept { return *this; }
   Compare &         comparator()      noexcept { return *this; }
   Compare   const & comparator() const noexcept { return *this; }

public:
   typedef Compare       key_compare;
   typedef Compare       value_compare;
   typedef Allocator     allocator_type;

   allocator_type get_allocator() const noexcept {
      return allocator();
   }

// observers:
   key_compare   key_comp()   const { return comparator(); }
   value_compare value_comp() const { return comparator(); }
};
```

Awkward Users

- What about using the same class for predicate / hash function / allocator?
  - Unlikely, but would would give an illegal duplicate base (depending on our implementation/wrapping)
  - What if user provides a function pointer?
    - Cannot derive
- … or a union, or a final class?
  - see above

Another Awkward Case

- What if the user decides to instrument their allocator, to count allocations?

```
namespace my_stuff {

template <typename T>
struct allocator {
   int allocations;
   // ... rest of allocator interface ...
};

std::ostream& operator<<(std::ostream &os, allocator<T> const & a) {
   return os << "allocations: " << a.allocations;
}

}
```

```
template < class Key
         , class Compare   = less<Key>
         , class Allocator = allocator<Key>
         >
class set : private Compare, Allocator {
private:
   NodeType *root;



   Allocator &       allocator()       noexcept { return *this; }
   Allocator const & allocator() const noexcept { return *this; }
   Compare &         comparator()      noexcept { return *this; }
   Compare   const & comparator() const noexcept { return *this; }

public:
   typedef Compare       key_compare;
   typedef Compare       value_compare;
   typedef Allocator     allocator_type;

   allocator_type get_allocator() const noexcept {
      return allocator();
   }

// observers:
   key_compare   key_comp()   const { return comparator(); }
   value_compare value_comp() const { return comparator(); }
};
```

```
template < class Key
         , class Compare   = less<Key>
         , class Allocator = allocator<Key>
         >
class set {
private:
   tuple<Compare, Allocator, NodeType *> members;


   Allocator &       allocator()       noexcept { return get<1>(members); }
   Allocator const & allocator() const noexcept { return get<1>(members); }
   Compare &         comparator()      noexcept { return get<0>(members); }
   Compare   const & comparator() const noexcept { return get<0>(members); }

public:
   typedef Compare       key_compare;
   typedef Compare       value_compare;
   typedef Allocator     allocator_type;

   allocator_type get_allocator() const noexcept {
      return allocator();
   }

// observers:
   key_compare   key_comp()   const { return comparator(); }
   value_compare value_comp() const { return comparator(); }
};
```

What I learned from tuple

```
template <typename INDEXER, typename ...TYPES>
struct tuple_impl;  // primary template is not defined

template <size_t ...INDEX, typename ...TYPES>
struct tuple_impl<index_sequence<INDEX...>, TYPES...>
      : Wrap<INDEX, TYPES>... {
   tuple_impl();
   tuple_impl(TYPES const& ...args);

   // … more stuff
};


template <typename ...TYPES>
struct tuple {
   tuple_iml<TYPES...> data;
};
```

Making Empty Base Work

```
template <size_t N, typename TYPE>
struct Wrap {
  TYPE d_data;

  auto data()       -> TYPE       & { return d_data; }
  auto data() const -> TYPE const & { return d_data; }
};
```

Making Empty Base Work

```
template <typename T>
constexpr const bool is_valid_base_v =
                is_class_v<T> && !is_union_v<T> && !is_final<T>_v;

template <size_t N, typename TYPE, bool = is_valid_base_v<TYPE>>
struct Wrap {
  TYPE d_data;

  auto data()       -> TYPE       & { return d_data; }
  auto data() const -> TYPE const & { return d_data; }
};
```

Making Empty Base Work

```
template <typename T>
constexpr const bool is_valid_base_v =
                is_class_v<T> && !is_union_v<T> && !is_final<T>_v;

template <size_t N, typename TYPE, bool = is_valid_base_v<TYPE>>
struct Wrap {
  TYPE d_data;

  auto data()       -> TYPE       & { return d_data; }
  auto data() const -> TYPE const & { return d_data; }
};


template <size_t N, typename TYPE>
struct Wrap<N, TYPE, true> : TYPE {
  using TYPE::TYPE;

  auto data()       -> TYPE       & { return *this; }
  auto data() const -> TYPE const & { return *this; }
};
```

What I learned from tuple

```
template <typename ...TYPES>
struct tuple {
   // constructors
   tuple();
   tuple(TYPES const& ...args);

   // … more stuff
private:
   tuple_impl<TYPES...> d_data;
};
```

What I learned from tuple

```
template <typename ...TYPES>
struct tuple {
    // constructors
    tuple();
    tuple(TYPES const& ...args);

    // … more stuff
private:
    tuple_impl<TYPES...> d_data;
};

tuple<> x{};
```

What I learned from tuple

```
template <typename ...TYPES>
struct tuple {
    // constructors
    tuple();
    tuple(TYPES const& ...args);

    // … more stuff
private:
    tuple_impl<TYPES...> d_data;
};


template <>
struct tuple<> {
    // empty tuple specialization is needed
    // to avoid ambiguous default constructor
};
```

Consider tuple_size

```
template <typename TYPE>
struct tuple_size {
   // no members is SFINAE friendly, C++17
};

template <typename ...TYPES>
struct tuple_size<tuple<TYPES...>>
     : integeral_constant<size_t, sizeof...(TYPES)> {};

template <typename TYPE, size_t N>
struct tuple_size<array<TYPE, N>>
     : integeral_constant<size_t, N> {};

template <typename TYPE_A, typename TYPE_B>
struct tuple_size<pair<TYPE_A, TYPE_B>>
     : integeral_constant<size_t, 2> {};
```

Consider tuple_size

```
template <typename TYPE>
struct tuple_size {
   // no members is SFINAE friendly, C++17
};

template <typename ...TYPES>
struct tuple_size<tuple<TYPES…>>
   : integeral_constant<size_t, sizeof...(TYPES) {};
```

What about cv-qualified tuples?

Consider tuple_size

```
template <typename TYPE>
struct tuple_size {
   // no members is SFINAE friendly, C++17
};

template <typename TYPE>
struct tuple_size<TYPE const> : tuple_size<TYPE> {
};

template <typename TYPE>
struct tuple_size<TYPE volatile> : tuple_size<TYPE> {
};

template <typename const TYPE>
struct tuple_size<TYPE const volatile> : tuple_size<TYPE> {
};

template <typename ...TYPES>
struct tuple_size<tuple<TYPES…>>
     : integeral_constant<size_t, sizeof...(TYPES) {};
```

```
template <>
struct tuple {

};

test< tuple<> >();
```

```
    template <typename T>
    void test() {
       T x1;              //
       T x2();            //
       T x3{};            //
       T x4 =  {};        //
       T x5 = T();        //
       T x6 = T{};        //

       T y1=x3;           //
       T y2(x3);          //
       T y3{x3};          //
       T y4 =  { x3 }; //
       T y5 = T( x3 ); //
       T y6 = T{ x3 }; //
    }
```

Test the library, not the compiler

```
template <>
struct tuple {

};

test< tuple<> >();
```

```
    template <typename T>
    void test() {
       T x1;              //
       T x2();            //
       T x3{};            //
       T x4 =  {};        //
       T x5 = T();        //
       T x6 = T{};        //

       T y1=x3;           //
       T y2(x3);          //
       T y3{x3};          //
       T y4 =  { x3 }; //
       T y5 = T( x3 ); //
       T y6 = T{ x3 }; //
    }
```

```
template <>
struct tuple {

};

test< tuple<> >();
```

```
    template <typename T>
    void test() {
        T x1;           //
        T x2();         // actually, function declaration
        T x3{};         //
        T x4 =  {};     //
        T x5 = T();     //
        T x6 = T{};     //

        T y1=x3;        //
        T y2(x3);       //
        T y3{x3};       //
        T y4 =  { x3 }; //
        T y5 = T( x3 ); //
        T y6 = T{ x3 }; //
    }
```

```
template <>
struct tuple {

};

test< tuple<> >();
```

```
    template <typename T>
    void test() {
        T x1;               //
        T x2();             // actually, function declaration
        T x3{};             //
        T x4 =  {};         //
        T x5 = T();         //
        T x6 = T{};         //

        T y1=x3;            //
        T y2(x3);           //
        T y3{x3};           // T is not a member of the aggregate
        T y4 =  { x3 }; //
        T y5 = T( x3 ); // T is not a member of the aggregate
        T y6 = T{ x3 }; // T is not a member of the aggregate
    }
```

```
template <>
struct tuple {

};

test< tuple<> const>();
```

```
    template <typename T>
    void test() {
        T x1;               //
        T x2();             // actually, function declaration
        T x3{};             //
        T x4 =  {};         //
        T x5 = T();         //
        T x6 = T{};         //

        T y1=x3;            //
        T y2(x3);           //
        T y3{x3};           //
        T y4 =  { x3 }; //
        T y5 = T( x3 ); //
        T y6 = T{ x3 }; //
    }
```

```
template <>
struct tuple {

};

test< tuple<> const>();
```

```
    template <typename T>
    void test() {
        T x1;               // const value must be initialized
        T x2();             // actually, function declaration
        T x3{};             //
        T x4 =  {};         //
        T x5 = T();         //
        T x6 = T{};         //

        T y1=x3;            //
        T y2(x3);           //
        T y3{x3};           // T is not a member of the aggregate
        T y4 =  { x3 }; //
        T y5 = T( x3 ); // T is not a member of the aggregate
        T y6 = T{ x3 }; // T is not a member of the aggregate
    }
```

```
template <>
struct tuple {
   // Can we fix it???
};

test< tuple<> const>();
```

```
   template <typename T>
   void test() {
      T x1;              // const value must be initialized
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          // T is not a member of the aggregate
      T y4 =  { x3 }; //
      T y5 = T( x3 ); // T is not a member of the aggregate
      T y6 = T{ x3 }; // T is not a member of the aggregate
   }
```

```
template <>
struct tuple {
   tuple() {}  // Yes we can!
};

test< tuple<> const>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
test<int>();
```

```
template <typename T>
void test() {
   T x1;              //
   T x2();            // actually, function declaration
   T x3{};            //
   T x4 =  {};        //
   T x5 = T();        //
   T x6 = T{};        //

   T y1=x3;           //
   T y2(x3);          //
   T y3{x3};          //
   T y4 =  { x3 }; //
   T y5 = T( x3 ); //
   T y6 = T{ x3 }; //
}
```

```
test<int>();
```

```
template <typename T>
void test() {
   T x1;              //
   T x2();            // actually, function declaration
   T x3{};            //
   T x4 =  {};        //
   T x5 = T();        //
   T x6 = T{};        //

   T y1=x3;           //
   T y2(x3);          //
   T y3{x3};          //
   T y4 =  { x3 }; //
   T y5 = T( x3 ); //
   T y6 = T{ x3 }; //
}
```

```
test<int const>();

    template <typename T>
    void test() {
       T x1;              //
       T x2();            // actually, function declaration
       T x3{};            //
       T x4 =  {};        //
       T x5 = T();        //
       T x6 = T{};        //

       T y1=x3;           //
       T y2(x3);          //
       T y3{x3};          //
       T y4 =  { x3 }; //
       T y5 = T( x3 ); //
       T y6 = T{ x3 }; //
    }
```

```
test<int const>();

    template <typename T>
    void test() {
       T x1;              // const int must be initialized
       T x2();            // actually, function declaration
       T x3{};            //
       T x4 =  {};        //
       T x5 = T();        //
       T x6 = T{};        //

       T y1=x3;           //
       T y2(x3);          //
       T y3{x3};          //
       T y4 =  { x3 };    //
       T y5 = T( x3 );    //
       T y6 = T{ x3 };    //
    }
```

```
struct explicit_default {
   explicit explicit_default() {}

};

test<explicit_default>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct explicit_default {
   explicit explicit_default() {}

};

test<explicit_default>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        // copy-list-initialization
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct explicit_default {
   explicit explicit_default() {}

};

test<explicit_default const>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct explicit_default {
   explicit explicit_default() {}

};

test<explicit_default const>();   // no change for const
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        // copy-list-initialization
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct aggregate {
   explicit_default member;

};

test<aggregate>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct aggregate {
   explicit_default member;

};

test<aggregate>();        // clang 3.6
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            // member cannot be initialized
      T x4 =  {};        // member cannot be initialized
      T x5 = T();        //
      T x6 = T{};        // member cannot be initialized

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 };    //
      T y5 = T( x3 );    //
      T y6 = T{ x3 };    //
   }
```

```
struct aggregate {
   explicit_default member;

};

test<aggregate>();        // gcc 4.9.3
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          // aggregate is not a member of itself
      T y4 =  { x3 }; // aggregate is not a member of itself
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; // aggregate is not a member of itself
   }
```

```
struct aggregate {
   explicit_default member;

};

test<aggregate const>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct aggregate {
   explicit_default member;

};

test<aggregate const>();
```

```
   template <typename T>
   void test() {
      T x1;              // clang
      T x2();            // actually, function declaration
      T x3{};            // clang
      T x4 =  {};        // clang
      T x5 = T();        //
      T x6 = T{};        // clang

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          // gcc
      T y4 =  { x3 }; // gcc
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; // gcc
   }
```

```
struct no_default {
   no_default(std::initializer_list<int>) {}

};

test<no_default>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct no_default {
   no_default(std::initializer_list<int>) {}

};

test<no_default>();
```

```
   template <typename T>
   void test() {
      T x1;              // no default constructor
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        // no default constructor
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct no_default {
   no_default(std::initializer_list<int>) {}

};

test<no_default const>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct no_default {
   no_default(std::initializer_list<int>) {}

};

test<no_default const>();   // no change for const

   template <typename T>
   void test() {
      T x1;              // no default constructor
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        // no default constructor
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 };    //
      T y5 = T( x3 );    //
      T y6 = T{ x3 };    //
   }
```

```
struct explicit_list {
   explicit explicit_list(std::initializer_list<int>) {}

};

test<explicit_list>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct explicit_list {
   explicit explicit_list(std::initializer_list<int>) {}

};

test<explicit_list>();
```

```
   template <typename T>
   void test() {
      T x1;              // no list-initialization
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        // copy-list-initialization
      T x5 = T();        // no list-initialization
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct explicit_list {
   explicit explicit_list(std::initializer_list<int>) {}

};

test<explicit_list const>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct explicit_list {
   explicit explicit_list(std::initializer_list<int>) {}

};

test<explicit_list>();   // no change for const
```

```
   template <typename T>
   void test() {
      T x1;              // no list-initialization
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        // copy-list-initialization
      T x5 = T();        // no list-initialization
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct evil_init {
   evil_init(std::initializer_list<int>) {}
   template <typename T> evil_init(std::initializer_list<T>) = delete;
};

test<evil_init>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct evil_init {
   evil_init(std::initializer_list<int>) {}
   template <typename T> evil_init(std::initializer_list<T>) = delete;
};

test<evil_init>();
```

```
   template <typename T>
   void test() {
      T x1;              // no initializer list
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        // no initializer list
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 };    //
      T y5 = T( x3 );    //
      T y6 = T{ x3 };    //
   }
```

```
struct evil_init {
   evil_init(std::initializer_list<int>) {}
   template <typename T> evil_init(std::initializer_list<T>) = delete;
};

test<evil_init>();       // Clang 3.6
```

```
   template <typename T>
   void test() {
      T x1;              // no initializer list
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        // no initializer list
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct evil_init {
   evil_init(std::initializer_list<int>) {}
   template <typename T> evil_init(std::initializer_list<T>) = delete;
};

test<evil_init>();        // gcc 4.9.3
```

```
   template <typename T>
   void test() {
      T x1;              // no initializer list
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        // no initializer list
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          // does not match list
      T y4 =  { x3 };    // does not match list
      T y5 = T( x3 );    //
      T y6 = T{ x3 };    // does not match list
   }
```

```
struct evil_init {
   evil_init(std::initializer_list<int>) {}
   template <typename T> evil_init(std::initializer_list<T>) = delete;
};

test<evil_init const>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct evil_init {
   evil_init(std::initializer_list<int>) {}
   template <typename T> evil_init(std::initializer_list<T>) = delete;
};

test<evil_init>();          // no change for const
```

```
   template <typename T>
   void test() {
      T x1;               // no initializer list
      T x2();             // actually, function declaration
      T x3{};             //
      T x4 =  {};         //
      T x5 = T();         // no initializer list
      T x6 = T{};         //

      T y1=x3;            //
      T y2(x3);           //
      T y3{x3};           // gcc 4.9.3
      T y4 =  { x3 }; // gcc 4.9.3
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; // gcc 4.9.3
   }
```

```
struct explicit_copy {
   explicit_copy() = default;
   explicit explicit_copy(explicit_copy &) = default;
};

test<explicit_copy>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct explicit_copy {
   explicit_copy() = default;
   explicit explicit_copy(explicit_copy &) = default;
};

test<explicit_copy>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        // no copy initialization
      T x6 = T{};        // no copy initialization

      T y1=x3;           // no copy initialization
      T y2(x3);          //
      T y3{x3};          // gcc 4.9.3:  still working on it
      T y4 =  { x3 }; // no copy initialization
      T y5 = T( x3 ); // no copy initialization
      T y6 = T{ x3 }; // no copy initialization
   }
```

```
struct explicit_copy {
   explicit_copy() = default;
   explicit explicit_copy(explicit_copy &) = default;
};

test<explicit_copy const>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct explicit_copy {
   explicit_copy() = default;
   explicit explicit_copy(explicit_copy &) = default;
};

test<explicit_copy const>();
```

```
   template <typename T>
   void test() {
      T x1;              // clang 3.6:  trivial type
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        // no copy initialization
      T x6 = T{};        // no copy initialization

      T y1=x3;           // no copy initialization
      T y2(x3);          // oops!  Not a const & copy
      T y3{x3};          // oops!  Not a const & copy
      T y4 =  { x3 }; // no copy initialization
      T y5 = T( x3 ); // no copy initialization
      T y6 = T{ x3 }; // no copy initialization
   }
```

```
struct explicit_copy {
   explicit_copy() = default;
   explicit explicit_copy(explicit_copy const &) = default;
};

test<explicit_copy const>();
```

```
   template <typename T>
   void test() {
      T x1;              // clang 3.6:  trivial type
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        // no copy initialization
      T x6 = T{};        // no copy initialization

      T y1=x3;           // no copy initialization
      T y2(x3);          //
      T y3{x3};          // gcc 4.9.3:  still working on it
      T y4 =  { x3 }; // no copy initialization
      T y5 = T( x3 ); // no copy initialization
      T y6 = T{ x3 }; // no copy initialization
   }
```

```
struct stolen_copy {
   stolen_copy() = default;
   template <typename T> stolen_copy(T &) = delete;
};

test<stolen_copy>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct stolen_copy {
   stolen_copy() = default;
   template <typename T> stolen_copy(T &) = delete;
};

test<stolen_copy>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           // constructor is deleted
      T y2(x3);          // constructor is deleted
      T y3{x3};          // constructor is deleted
      T y4 =  { x3 }; // constructor is deleted
      T y5 = T( x3 ); // constructor is deleted
      T y6 = T{ x3 }; // constructor is deleted
   }
```

```
struct stolen_copy {
   stolen_copy() = default;
   template <typename T> stolen_copy(T &) = delete;
};

test<stolen_copy const>();
```

```
   template <typename T>
   void test() {
      T x1;              //
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 }; //
      T y5 = T( x3 ); //
      T y6 = T{ x3 }; //
   }
```

```
struct stolen_copy {
   stolen_copy() = default;
   template <typename T> stolen_copy(T &) = delete;
};

test<stolen_copy const>();
```

```
   template <typename T>
   void test() {
      T x1;              // clang 3.6:  trivial type
      T x2();            // actually, function declaration
      T x3{};            //
      T x4 =  {};        //
      T x5 = T();        //
      T x6 = T{};        //

      T y1=x3;           //
      T y2(x3);          //
      T y3{x3};          //
      T y4 =  { x3 };    //
      T y5 = T( x3 );    //
      T y6 = T{ x3 };    //
   }
```

```
struct immovable {
    immovable() = default;
    immovable(immovable&&) = delete;
};

test<immovable>();
```

```
    template <typename T>
    void test() {
        T x1;               //
        T x2();             // actually, function declaration
        T x3{};             //
        T x4 =  {};         //
        T x5 = T();         //
        T x6 = T{};         //

        T y1=x3;            //
        T y2(x3);           //
        T y3{x3};           //
        T y4 =  { x3 }; //
        T y5 = T( x3 ); //
        T y6 = T{ x3 }; //
    }
```

```
struct immovable {
    immovable() = default;
    immovable(immovable&&) = delete;
};

test<immovable>();
```

```
    template <typename T>
    void test() {
        T x1;              //
        T x2();            // actually, function declaration
        T x3{};            //
        T x4 =  {};        //
        T x5 = T();        // no copy initialization
        T x6 = T{};        // no copy initialization

        T y1=x3;           // no copy at all
        T y2(x3);          // no copy at all
        T y3{x3};          // no copy at all
        T y4 =  { x3 }; // no copy at all
        T y5 = T( x3 ); // no copy at all
        T y6 = T{ x3 }; // no copy at all
    }
```

```
struct immovable {
    immovable() = default;
    immovable(immovable&&) = delete;
};

test<immovable const>();
```

```
    template <typename T>
    void test() {
        T x1;               //
        T x2();             // actually, function declaration
        T x3{};             //
        T x4 =  {};         //
        T x5 = T();         //
        T x6 = T{};         //

        T y1=x3;            //
        T y2(x3);           //
        T y3{x3};           //
        T y4 =  { x3 }; //
        T y5 = T( x3 ); //
        T y6 = T{ x3 }; //
    }
```

```
struct immovable {
   immovable() = default;
   immovable(immovable&&) = delete;
};

test<immovable>();
```

```
   template <typename T>
   void test() {
      T x1;             // clang 3.6:  trivial type
      T x2();           // actually, function declaration
      T x3{};           //
      T x4 =  {};       //
      T x5 = T();       // no copy initialization
      T x6 = T{};       // no copy initialization

      T y1=x3;          // no copy at all
      T y2(x3);         // no copy at all
      T y3{x3};         // no copy at all
      T y4 =  { x3 }; // no copy at all
      T y5 = T( x3 ); // no copy at all
      T y6 = T{ x3 }; // no copy at all
}
```

What have we learned?

```
template <typename T>
void test() {
   T x1;              // trivial, init-list
   T x2();            // actually, function declaration
   T x3{};            // best bet
   T x4 =  {};        // explicit default
   T x5 = T();        // init-list, accessible copy
   T x6 = T{};        // explicit/accessible copy

   T y1=x3;           // explicit copy
   T y2(x3);          //
   T y3{x3};          // explicit copy, aggregates
   T y4 =  { x3 }; // explicit copy, aggregates
   T y5 = T( x3 ); // explicit copy
   T y6 = T{ x3 }; // explicit copy, aggregates
}
```

What have we learned?

Run again with volatile - all copies fail (but works for int)

```
template <typename T>
void test() {
   T x1;              // trivial, init-list
   T x2();            // actually, function declaration
   T x3{};            // best bet
   T x4 =  {};        // explicit default
   T x5 = T();        // volatile, init-list, accessible copy
   T x6 = T{};        // volatile, explicit/accessible copy

   T y1=x3;           // volatile, explicit copy
   T y2(x3);          // volatile
   T y3{x3};          // volatile, explicit copy, aggregates
   T y4 =  { x3 }; // volatile, explicit copy, aggregates
   T y5 = T( x3 ); // volatile, explicit copy
   T y6 = T{ x3 }; // volatile, explicit copy, aggregates
}
```

Library Evolution

Avoid Breaking Changes

- If your next release is not compatible with the previous releases, users need to audit their whole code base before they can safely update
- If you must break something, prefer a breakage that is loud and clear, so is found easily and early
  - API breakage - fails to compile
  - ABI breakage - fails to link
  - Silent behavior change - end users report bugs, oops!
    - behavior change is undefined behavior - all bets are off

C++11 Memory Model

- Making concurrency a well specified concern for the language introduced some compatibility concerns
- Some compiler optimizations became illegal
- basic_string banned CoW, enabled short string
  - ‘begin’ no longer invalidates iterators
  - ‘swap’ can now invalidate iterators

Enriching Exceptions

- Potentially unsafe idiom from C++11 onwards
- `catch(Type& exception)`, update the exception object
  - C++11: Same exception object can unwind in several threads in parallel
    - Modifications are potential race conditions
  - Preferred idiom: always catch by `const &`, use `nested_exception` or similar to throw an enriched exception if necessary

Racing Exceptions

- If we control whole code path, we know whether we risk joining the thread that might throw - low risk (code changes under maintenance)
- If we call user code, then we have no control over which exceptions might throw
  - function pointer
  - virtual function call
  - template specialization
  - user function found via ADL
- Can limit damage by writing a contract that disallows exceptions from other threads - but any bugs from violations will be subtle.

Summary

Some takeaways

- Document contracts clearly; they define your library
- Testing will reveal incomplete contracts as well as broken implementations
- It is generally easier to fix an implementation than a contract on a live system

Some takeaways

- Exception classes should have no throw copies
  - e.g., std exceptions may be immutable ref-counted strings
- Class templates frequently store data in tuples
- Beware of (unexpected) ADL
- Learn the safe idioms, especially for generic code
  - T var{}; // default a value this way
    T var1(var2); // copy an object this way