Restoring Your Sanity

An Approach to Dealing with Reference Types in the Generic Programming Paradigm

What Is the Purpose of This Talk?

- Examine the methodology of Generic Programming
- Attempt to establish sensible options when using reference types as arguments to generic code that expects *Regular* types
- Identify common missteps
- Get into a holy war

People Should Be Active in This Discussion

- Shout at me if you disagree
- Don’t bother raising your hand
- There is no obvious “right” answer to most of these problems

Regular Types

Regular Type

A type `T` *whose construction, copy, and destruction semantics are the same as those of a built-in integer type for valid values of that type.*

- Used when developing the STL
- Formally introduced by Stepanov in the early 2000s
- Others have since augmented the definition, usually including move semantics

Regular Type Requirements

Given a type `T` and instances of `T` named `a` and `b`.

Operation	Syntactic Requirements
Default Constructor	`T c;`
Equality	`a == b`
Copy Constructor	`T d(a)`
Assignment	`a = b`
Ordering	`a < b`
Destructor	`a.~T()`

Semantics of Default Construction

A constructor that places the object into a partially-formed state.

- An object that is partially-formed may be assigned-to or destroyed.
  - Anything else has undefined behavior.

Additional Syntactic Requirement of Equality

Syntactic Requirement	Same Semantics As
`a != b`	`!(b == a)`

Semantics of Equality

A procedure that takes two objects of the same type and returns true if and only if the object states are equal. Inequality is always defined and returns the negation of equality.

Semantics of Copy-Construction and Assignment

A procedure that takes two objects of the same type and makes the first object equal to the second without modifying the second.

- The meaning of assignment does not depend on the initial value of the first object.
- The two objects are *disjoint*.

Semantics of Ordering

Ordering of Regular types is a total ordering.

Reflexivity	`a <= a`
Antisymmetry	`a <= b && b <= a` *implies* `a == b`
Transitivity	`a <= b and b <= c` *implies* `a <= c`
Trichotomy Law	`a <= b` *or* `b <= a`

Additional Syntactic Requirements of Ordering

In addition to <, the following operations must also be defined.

Syntactic Requirement	Same Semantics As
`a > b`	`b < a`
`a >= b`	`!(a < b)`
`a <= b`	`!(b < a)`

Semantics of Destruction

A procedure ending the object's existence.

- After a destructor is called, no further procedure can be applied to the object and resources may be reclaimed.
- Destructor invocation is usually implicit.

Standard Types that Are Not Regular (Syntactic)

- C-style array types
  - int[5]
- const-qualified types (generally speaking)
  - const int
- Move-only types
  - std::unique_ptr<T>
- Non-copyable types
  - std::mutex
  - std::atomic<int>

Standard Types that Are Not Regular

- Reference types
  - T&
- Proxy types
  - std::vector<bool>::reference
  - std::tuple<T&>
- Things in-between
  - std::reference_wrapper<T>
  - std::tuple<T, U&, V>
  - std::string_view

Are Types That Are Not Regular Bad?

Some people do take that stance.

- A little too idealistic
- Subsetting Regular is commonly considered fine
- Refernce and proxy types are reasonable, but cannot generally be used in the same generic code as Regular types

Defining Generic Algorithms and Datastructures

Generic Datastructures

- Parameterized on types, values, and/or templates
  - Specify *syntactic* and *semantic* requirements for parameters
- *Semantics* of associated functions do not change based on the parameters
  - Be generic as opposed to being vague
- Specializations of generic datastructures may *refine* but not change existing meaning

Start with Concrete Implementations

1) Start with existing, concrete implementations of the same algorithm or datastructure

```cpp
int sum(int* array, std::ptrdiff_t n) {
  int result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

```cpp
float sum(float* array, std::ptrdiff_t n) {
  float result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Lifting

2) *Lift* the algorithm to create a generic form.

```
int sum(int* array, std::ptrdiff_t n) {
  int result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

```
float sum(float* array, std::ptrdiff_t n) {
  float result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

```
template<class T>
T sum(T* array, std::ptrdiff_t n) {
  T result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Lifting (continued)

3) Identify operations in the body that are *dependent* on the generic parameters.

```
template<class T>
T sum(T* array, std::ptrdiff_t n) {
  T result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Lifting (continued)

4) Identify the *syntactic and semantic* requirements of the dependent operations such that the algorithm/datastructure has consistent meaning.
Operations that exist for *Regular* types require the same semantic meaning.

```
template<class T>
T sum(T* array, std::ptrdiff_t n) {
  T result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Lifting (continued)

5) Repeat the process of refining the generic definition through comparisons with concrete implementations.

```
template<class T>
T sum(T* array, std::ptrdiff_t n) {
  T result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

```
std::string sum(std::string* array,
                std::ptrdiff_t n) {
  std::string result = "";
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

Lifting (continued)

Provide a unified syntax when the same operation has a different syntax for certain concrete implementations.

Initialization for Arithmetic Types	Initialization for `std::string`
`T result = 0;`	`std::string result = "";`

Options:

Value-Initialization	`T()`
Traits	`Additive<T>::identity()`

Lifting (continued)

```
// Assume defined for built-in arithmetic types
// Assume works for const-qualified T
// This trait is a concept-map for the Additive concept
template<class T>
struct Additive;

template<>
struct Additive<std::string> {
  // By value to be concise
  static std::string add(std::string lhs, std::string rhs) {
    return lhs + rhs;
  }

  static std::string identity() { return {}; }
};
```

Lifting (continued)

```
template<class T>
T sum(T* array, std::ptrdiff_t n) {
  T result = 0;
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

```
std::string sum(std::string* array,
                std::ptrdiff_t n) {
  std::string result = "";
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = result + array[i];
  return result;
}
```

```
template<class T>
T sum(T* array, std::ptrdiff_t n) {
  T result = Additive<T>::identity();
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = Additive<T>::add(result,
                              array[i]);
  return result;
}
```

Lifting (continued)

Document requirements.

```
// Requirements:
//   T must be Additive
//   T must be MoveAssignable
//   T must be MoveConstructible
//   T must be Destructible
template<class T>
T sum(T* array, std::ptrdiff_t n) {
  T result = Additive<T>::identity();
  for(std::ptrdiff_t i = 0; i < n; ++i)
    result = Additive<T>::add(result,
                              array[i]);
  return result;
}
```

Notes on Datastructure Constraints

Constraints may be coarse-grained or fine-grained.

- Coarse-grained
  - All concept requirements for all associated functions are applied to the top-level template.
  - Any generic code that takes an instance of the datastructure is able to call all associated functions.
- Fine-grained
  - Concept requirements may be specified per associated function of the template.
  - Allows more types to be used with the template.
  - Any generic code that takes an instance of the datastructure must still verify that it meets any additional requirements of associated functions before use.

Standard Library Approach

The standard library is generally fine-grained.

- Move-only types may be used with std::vector
  - Attempting to copy has additional requirements
- Relational operators don’t need to be defined unless used

Designing a Variant

What Is a Variant?

Description: *A Sum Type containing the state of exactly one of N alternatives.*

Example: std::variant<H, T...>

A Concrete Case

Member functions left out for simplicity.

```
struct variant_ {
  union data_t {
    int a;
    std::string b;
  } data;
  enum class which_t { a, b } which;
};
```

Variant Copy-Assignment

Starting with Regular types, assuming no exceptions for simplicity:

- Destroy the active lhs
- Copy-construct the active rhs element into the lhs storage
- Update the lhs index

A Concrete Case

```
variant_& variant_::operator =(const variant_& other) {
  data.destroy();  // Destroys the active alternative

  if(other.which == which_t.a)
    new (&data) int(other.data.a);
  else
    new (&data) std::string(other.data.b);

  which = other.which;

  return *this;
}
```

Optimizing the Concrete Case

What do we depend on for this optimization to make sense?

```
variant_& variant_::operator =(const variant_& other) {
  if(which != other.which) {
    data.destroy();  // Destroys the active alternative
    if(other.which == which_t.a)
      new (&data) int(other.data.a);
    else
      new (&data) std::string(other.data.b);
  } else if(which == which_t.a)
      data.a = other.data.a;
    else
      data.b = other.data.b;

  which = other.which;
  return *this;
}
```

Skip a Few Steps... Variant Template

Arrive at variadic variant template (coarse-grained requirements).

```
// Requires:
//   Each H and T model Regular
template<class H, class...T>
class variant;
```

What Does Assignment Look Like

```
variant_& variant_::operator =(const variant_& other) {
  if(which != other.which) {
    data.destroy();  // Destroys the active alternative
    if(other.which == which_t.a)
      new (&data) int(other.data.a);
    else
      new (&data) std::string(other.data.b);
  } else if(which == which_t.a)
      data.a = other.data.a;
    else
      data.b = other.data.b;

  which = other.which;
  return *this;
}
```

```
variant_& variant_::operator =(const variant_& other) {
  if(which != other.which) {
    data.destroy();
    raw_copy_construct(other);
    which = other.which;
  } else {
    copy_assign(other);  // As-if with the = operator
  }

  return *this;
}
```

A Feature Request...

Feature Request

Request: *I want to be able to make a variant containing a reference type*.

```
template<class Fun>
variant<empty, std::invoke_result_t<Fun>>
maybe_call(Fun&& fun, bool call) {
  return call ? std::invoke(std::move(fun))
              : empty{};
}

template<class Fun>
void keep_calling(Fun&& fun) {
  while(true) {
    auto res = maybe_call(fun, user_input());
    output_result(res);
  }
}
```

Feature Request

Note: A reference type is not Regular

Options:

1. Flag as NAD, move on with your life
2. Sanction everything that would compile
3. Create fine-grained requirements per function
4. Something else???

Regularity and Reference Types

	L-Value Reference Types (Meets Syntactic Requirements)	L-Value Reference Types (Meets Semantic Requirements)
Default Constructor	No	No
Equality	Iff target does	No
Copy Constructor	Yes	Yes
Assignment	Iff target does	No
Ordering	Iff target does	No
Destructor	No	Yes?

Naive: Do What References Do

```
variant_& variant_::operator =(const variant_& other) {
  if(which != other.which) {
    data.destroy();
    raw_copy_construct(other);
     which = other.which;
  } else {
    copy_assign(other);  // As-if with the = operator
  }

  return *this;
}
```

Feature Applied

User likes the feature!

```
template<class Fun>
variant<empty, std::invoke_result_t<Fun>>
maybe_call(Fun&& fun, bool call) {
  return call ? std::invoke(std::move(fun))
              : empty{};
}

template<class Fun>
void keep_calling(Fun&& fun) {
  while(true) {
    auto res = maybe_call(fun, user_input());
    output_result(res);
  }
}
```

Development Continues...

Let’s pull the variant out of the loop...

```
template<class Fun>
variant<empty, std::invoke_result_t<Fun>>
maybe_call(Fun&& fun, bool call) {
  return call ? std::invoke(std::move(fun))
              : empty{};
}

template<class Fun>
void keep_calling(Fun&& fun) {
  variant<empty, std::invoke_result_t<Fun>> res;
  while(true) {
    res = maybe_call(fun, user_input());
    output_result(res);
  }
}
```

What Went Wrong?

We ignored the semantic requirements of assignment.

What should we have done?

A) Not supported assignment for this operation
B) Gone through the lifting process again

Observation

What did the user expect to happen?

```
template<class Fun>
variant<empty, std::invoke_result_t<Fun>>
maybe_call(Fun&& fun, bool call) {
  return call ? std::invoke(std::move(fun))
              : empty{};
}

template<class Fun>
void keep_calling(Fun&& fun) {
  variant<empty, std::invoke_result_t<Fun>> res;
  while(true) {
    res = maybe_call(fun, user_input());
    output_result(res);
  }
}
```

Looking Back at the Implementation

What do we depend on for this optimization to make sense?

```
variant_& variant_::operator =(const variant_& other) {
  if(which != other.which) {
    data.destroy();
    raw_copy_construct(other);
    which = other.which;
  } else {
    copy_assign(other);  // As-if with the = operator
  }

  return *this;
}
```

If a type doesn't meet the requirements for an optimization, then don't do the optimization

Observation

Is there anything wrong with this?

```
using Reference = std::vector<bool>::reference;
std::vector<Reference> woops;
```

What happens when you use vector assignment?
What happens when you use algorithms?
What happens if you put it in a set?

Observation

Our `variant<T, empty>` is analogous to:

- `optional<T>`
- Container of max-size 1

Updating the Implementation

This is a hack… really we should use traits.

```
variant_& variant_::operator =(const variant_& other) {
  if(which != other.which || is_a_reference()) {
    data.destroy();
    raw_copy_construct(other);
    which = other.which;
  } else {
    copy_assign(other);  // As-if with the = operator
  }

  return *this;
}
```

Observation

We cannot simply use traits for assignment because there is no way to rebind a reference.

Potential alternative?

When creating storage for the type, use a trait for a “storage” type.

```
// Identity for non-reference types.
// Meets all syntactic and semantic requirements of Regular.
// Contains a pointer for reference types.
// Exposes access to the logically contained T via traits.
template<class T>
using value_storage = unspecified;
```

What Are the Drawbacks to This Approach?

- Complicates implementation
  - Simply not supporting irregular types is easiest
- May be surprising to users
  - Likely *less* surprising than semantics that vary at run-time and with no known use-cases
- People interacting with the object still need to be aware of the subtleties when interacting in non-trivial ways
  - In the container case, algorithms wouldn't behave as expected unless they were aware of the abstraction

Why Does Wrapping Work?

- Instead of considering reference types to not be Regular, it considers them to be Regular but *without* a known mapping to the Regular concept
- Our wrapper and the associated traits act as a concept map
  - It provides all operations with semantics that are consistent with the Regular concept

Alternative

Manual wrapping.

- If the user wishes to use references, they can manually wrap *before* passing the type to the template
- User manually unwraps at every access

Drawbacks to Manual Wrapping/Unwrapping

- Requires the user to do some metaprogramming or always go through traits in generic code

```cpp
template<class Fun>
variant<empty, std::invoke_result_t<Fun>>
maybe_call(Fun&& fun, bool call) {
  return call ? std::invoke(std::move(fun))
              : empty{};
}

template<class Fun>
void keep_calling(Fun&& fun) {
  while(true) {
    auto res = maybe_call(fun, user_input());
    output_result(res);
  }
}
```

```cpp
template<class Fun>
variant<empty, wrap_if_ref<std::invoke_result_t<Fun>>>
maybe_call(Fun&& fun, bool call) {
  return call ? wrap_ref(std::invoke(std::move(fun)))
              : empty{};
}

template<class Fun>
void keep_calling(Fun&& fun) {
  while(true) {
    auto res = maybe_call(fun, user_input());
    output_result(res);  // Internally unwraps
  }
}
```

Summary of Options

Support for references is *not* crazy, however there are only a few reasonable options:

a) Don’t support them at all
b) Have fine-grained constrain
c) Internally wrap
d) Have users externally wrap and unwrap

Which option is picked is somewhat subjective, but usually “sanction whatever compiles” is never a good choice.

What Happened with std::variant and std::optional?

Support for references was heavily discussed.

- Reference support for optional was cut fairly early on
- Most people advocated a “do-what-references-do” approach to assignment
  - No use-cases were ever presented
- Subset wanted to either rebind or not support assignment or comparisons
  - Several use-cases were presented
- Members are actively pursuing adding in some kind of limited support for references in C++20

Examining std::tuple

- What does copying mean for all Regular T?
- How does it treat reference types and why?
- Does `std::tie` need to return a tuple as opposed to something else?
- What are the various uses of `std::forward_as_tuple`?
  - What semantics might you expect in the various uses?

Core Semantics of std::tuple Are Parametric

- This makes it difficult to reason about a tuple in generic code
- Having a type be a *subset* of Regular is a bit better, but operations that are defined should not have conflicting semantics with Regular

What Might Be Another Approach?

- std::tie could return something other than a std::tuple
- std::forward_as_tuple *may* be rationalized to return a tuple for certain uses
  - Contained references would have semantics consistent with the value_storage wrapper
- If a tuple supports references at all, they probably should rebind

Questions/Discussion