Vocabulary Types for Composite Class Design

Jonathan B. Coe, Antony Peacock & Sean Parent

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P3019r1

Kona Hawaii 2023-11-8

Introduction

We propose adding two new class templates to the C++ Standard Library:

```
template <typename T, typename Allocator = std::allocator<T>>
class indirect;
```

```
template <typename T, typename Allocator = std::allocator<T>>
class polymorphic;
```

These fill a gap in the suite of existing standard library vocabulary types.

Vocabulary Types

We refer to standard library types such as `std::array`, `std::map`, `std::optional`, `std::variant` and `std::vector` as *vocabulary types*.

We postulate that an arbitrary piece of C++ library or application code would make use of some of these types.

Standardizing vocabulary types is important as it allows different libraries to easily interoperate and for users to build applications.

The standard library contains other, non-vocabulary types to do specific jobs such as interacting with the filesystem, formatting text for output, or dealing with concurrency.

We probably want to standardize both sorts of library type.

Composite classes

We define composite classes as classes with other class instances as member data.

Vocabulary types can be used to express common idioms.

Idiom	Type
An instance of an object `T`	`T`
A nullable instance of an object `T`	`std::optional<T>`
An instance of one of a closed-set of types `Ts...`	`std::variant<Ts...>`
`N` instances of a type `T`	`std::array<T, N>`
Variable-count, multiple instances of a type `T`	`std::vector<T>`
Unique, variable-count, instances of a type `T`	`std::set<T>`
Key-accessed, instances of a type `T`	`std::map<Key, T>`

Special member functions

The compiler can generate special member functions. Each special member function can be compiler-generated if it is supported by all component objects.

Special member(s)	Signature(s)
Default constructor	`T();`
Destructor	`~T();`
Copy constructor/assignment	`T(const T&);` `T& operator=(const T&);`
Move constructor/assignment	`T(T&&);` `T& operator=(T&&);`

Const propagation

The compiler will only allow const-qualified member functions to be called on components when they are accessed through a const-access path.

We call this *const-propagation*.

Standard library vocabulary types provide const-qualified and non-const-qualified overloads of accessors to owned objects to enforce const propagation:

```
const T& std::vector<T>::operator[](size_t index) const;
T& std::vector<T>::operator[](size_t index);
```

Requirements for vocabulary types

Composite classes built with vocabulary types should be composable:

- const propagates.
- The compiler can generate all special member functions where they are defined for an owned type `T`.

These requirements make working with C++ easier - classes should do business logic or resource management, not both.

Combining vocabulary types

We can combine vocabulary types to express combined idioms:

Idiom	Type
Variable-count, multiple, nullable instances of a type `T`	`std::vector<std::optional<T>>`
Key-accessed, instances of one of a closed-set of types `Ts...`	`std::map<Key, std::variant<Ts...>>`

Incomplete types

We may want a composite to contain an instance of an incomplete type, either directly or through the use of existing vocabulary types.

Incomplete type support is needed for defining recursive data structures, supporting open-set polymorphism, and hiding implementation detail.

Incomplete types are supported by storing the object of the incomplete type outside of the composite object.

Storing an object outside of a composite type can also be used to optimize use of cache (hot/cold splitting).

Incomplete types using pointers

Since we want to store the object of incomplete type outside of our composite type, pointers are the first thing we may reach for:

```
class Composite {
    T* ptr_;
    // ...
}
```

Pointers are a poor fit for owned data as we need to implement all special member functions and manually check const-qualified methods (const does not propagate to a pointee when the pointer is accessed through a const-access path).

Incomplete types using `std::unique_ptr<T>`

Unique pointers are a little better:

```
class Composite {
    std::unique_ptr<T> ptr_;
    // ...
}
```

We do not need to implement move construction, move assignment or destruction. The compiler will implicitly delete the copy constructor and copy assignment operator; either the composite is non-copyable or we implement the copy constructor and copy assignment operator ourselves.

Const does not propagate through `std::unique_ptr`, we must manually check const-qualified methods for correctness.

Incomplete types using `std::shared_ptr<T>`

Shared pointers do not model the right thing:

```
class Composite {
    std::shared_ptr<T> ptr_;
    // ...
}
```

The compiler-generated copy constructor and assignment operator will give rise to multiple composite objects that share the same components: that is not ownership.

Const does not propagate through `std::shared_ptr<T>`, we must manually check const-qualified methods for correctness.

Incomplete types using `std::shared_ptr<const T>`

Shared pointers to const might work:

```
class Composite {
    std::shared_ptr<const T> ptr_;
    // ...
}
```

Again, the compiler-generated copy constructor and assignment operator will give rise to multiple composite objects that share the same components, *but* the shared components are immutable.

We cannot call non-const qualified member functions accessed though `ptr_`, part of our composite is immutable.

New vocabulary types (post P3019)

Idiom	Type
An instance of an object of an incomplete type `T`	`indirect<T>`
An instance of an object from an open set of types derived from a base type `T`	`polymorphic<T>`

Required properties of our new vocabulary types

Any composable vocabulary type needs:

- Resource ownership (destruction and move).
- Deep copies.
- const-propagation.

Both of the proposed types, `indirect` and `polymorphic`, need:

- Indirectly allocated object storage.

Further details

- Our proposal, with design discussion and proposed formal wording:
  https://wg21.link/p3019r1
- Our (header-only, C++20) reference implementation with tests and benchmarks:
  https://github.com/jbcoe/value_types

Acknowledgements

Thanks to @nbx8 and @Ukilele for short-notice review of this material.