QUIP: 18 Title: Implementing Value Types in Qt Author: Giuseppe D'Angelo <giuseppe.dangelo@kdab.com> Status: Active Type: Implementation Created: 2020-11-09 Post-History: https://lists.qt-project.org/pipermail/development/YYYY-Month/nnnnnn.html

Overview

This document defines how to design value types in Qt public APIs. The aim is offering a practical checklist for type authors, and streamlining code reviews for value types.

In this document we are going to use terminology from [1] and [2], which are mandatory material for any value type author; the reader is expected to have familiarity with them. This document complements the Qt Coding Conventions [3].

The style of this document deliberately follows the style of the C++ Core Guidelines [4]. In particular:

- Each section is numbered, so it can be cross-referenced;
- The numbering is stable over time: new rules only get added with new numbers, and rules that are removed do not renumber the following ones.
- The numbering is **not** in any specific order (and in particular it's not in some order of importance); all the rules apply all the time.
- Most of these rules can be enforced by tooling (and some already are by Clazy, see [5]).

It's worth noting that a lot of existing code may violate some of these guidelines (mostly for historical reasons).

Value classes (VALUE)

This section contains design rules that apply to any value class.

This also applies to any *helper* datatype defined for value classes: inner enumerations, inner classes, and so on.

(VALUE.1) Value classes must be default constructible, and initialize their members

Although not Modern C++ design, this rule stems from being consistent with all the other value classes in Qt. (Also, historically, Qt containers required default constructability of the contained types. This may still be the case for some Qt container).

Consider adding `isNull` or `isValid` methods if default initialization leaves your class in a non-valid state.

(VALUE.2) Value classes must be publicly copiable, movable, destructible

Should go without saying. If a class isn't copiable then it's not a value class. No class should ever lack move operations (even if they effectively mean copying), so do not write a class in a way that it's not movable (see also THICK.3).

(VALUE.3) Declare `QTypeInfo`

Always use `Q_DECLARE_TYPEINFO` to correctly classify the class. Most of the time (> 99%) a Qt value class is relocatable, and sometimes primitive. Complex value classes are not commonly found.

Note that in Qt 6 trivial classes are automatically considered primitive types by QTypeInfo. We don't have that many trivial classes anyhow, and still it wouldn't hurt to be explicit.

(VALUE.4) Allow storing in QVariant

Any value class should have either a built-in constructor for QVariant or have `Q_DECLARE_METATYPE` applied to them. We do not actually go for `qRegisterMetaType`; it's fine if the user does it as well (calling it multiple times is a no-op).

(VALUE.5) Define debug streaming operators

Debug meaning `qDebug`. We do not offer built-in streaming into `std::ostream` objects.

(VALUE.6) Define QDataStream streaming operators

Although usage of QDataStream is frowned upon (in favor of standardized formats such as CBOR or JSON), the streaming operators for QDataStream must be defined for consistency.

In Qt 6 QMetaType picks up the streaming operations automatically, allowing QVariant objects that hold your type to be streamed. This, in turn, unlocks e.g. QSettings, DND with item views, and similar.

(VALUE.7) If a class is comparable for equality, then it must also offer a `qHash` overload

Rationale: if you can compare for equality, then your type should be usable as a key in QHash. Strongly consider using `qHashMulti` or `QHashCombine` in your implementation.

We still lack a policy regarding offering `std::hash` support.

(VALUE.8) Do not use public inheritance

A `Point3D` is not a `Point2D` with the addition of a Z value. Due do how inheritance works in C++, and in particular how it interferes with value semantics, one should never have value classes inherit from each other. Strongly prefer composition instead.

Non-public inheritance (to share implementation) is fine.

Note that we do not enforce this rule in code towards client code: value classes **must not** be marked as `final`.

(VALUE.9) The moved-from state is *valid but unspecified*

This means that users can call any function without preconditions on a moved-from instance, and the instance must still have valid class invariants. Pay extreme attention at what this implies for a class.

(VALUE.10) Default constructors should be `noexcept`

This includes *not allocating any memory*. Pimpled types must work with a null d-pointer anyhow because of VALUE.9 + VALUE.12, hence the default constructor can simply set the d-pointer to `nullptr` and still be `noexcept`. (Mut. mut., they can set the d-pointer to a `sharedNull` object).

(VALUE.11) Default constructors should be `constexpr`

This allows to use the type e.g. as a global object without risking a static initialization order fiasco and the overhead associated with the workarounds (such as `Q_GLOBAL_STATIC`). It makes thin abstractions with trivial destructor usable as `constexpr` objects (they're literal types), and thick abstractions usable as `constinit`.

(VALUE.12) Move operations should be `noexcept`

Should go without saying.

(VALUE.13) Copy operations should be `noexcept`

Qt uses implicit sharing. Hence, copies are cheap, and they don't allocate memory; copy operations should therefore be `noexcept`. The risk of overflowing the reference counter is practically non-existent.

(VALUE.14) Provide relational comparison operators as hidden friends

So they reduce the search space when finding overloads, and avoid triggering unwanted conversions. See also [6].

(VALUE.15) Build relational operators in terms of the underlying operations

Don't give the built-in operators fuzzy or fancy semantics. This is surprising for end-users, ends up corrupting the API (one must use `isStrictlyEquals` or some other strange name, rather than the built-in == operator), and causes API flaws (the fuzzy comparisons in QPointF / QSizeF / etc. make it impossible for them to be hashable).

(VALUE.16) Value classes are only reentrant by contract

Unless extra guarantees are offered by the class author, you must not protect anything from concurrent access.

However, many users expect `const` access to fall into the reentrancy contract (technically, it does not) by making `const` access thread safe. Therefore any internal side-effect of `const` functions that could result in a data race (such as caching, JIT compilation, etc.) when called on the same instance must be mutex protected. Otherwise: there should be big warnings in the documentation.

Thin abstractions (THIN)

(THIN.1) Thin abstractions should be fully inline, non-exported classes

Do not export thin abstractions. Define them fully inline. (Marking a fully inline class as exported might still output unnecessary symbols on certain platforms, so don't do it.)

Individual functions can be exported if it makes sense (too complex to have in a header; may benefit from out-of-line changes; etc.). Always provide arguments in code comments as of the why a certain function is exported.

(THIN.2) Honor the rule of zero

Do not even *declare* the destructor; the move operations; or the copy operations. (Let the compiler do its job.) If you like you can be explicit in the code by leaving a comment like:

```
// compiler-generated special member functions are fine!
```

(THIN.3) Thin classes should have trivial destructors

Which, combined with VALUE.11, makes them *literal types*, thus suitable for being used as `constexpr` (global) objects.

Note that "trivial destructor" does not mean "compiler-generated destructor"; it means (roughly) "no code is run to destroy this class". A class that contains a QVector does not have a trivial destructor!

Corollary: a class that cannot have a trivial destructor is extremely likely *not* a thin abstraction, but a thick one. A non trivial destructor has implications on the moved-from state reached through a compiler-generated move constructor (and it may easily require customization of the moves). Once one goes for the rule of five, we are in thick abstraction territory.

(THIN.4) Prefer using `class` and accessors over `struct`

The overwhelming majority of value classes in Qt have accessors and `private` data members, even when the classes themselves don't have invariants (e.g. QPoint). This is frowned upon in Modern C++, but it's an established pattern in Qt. One should deviate from it only for good reasons.

Thick abstractions (THICK)

(THICK.1) Always have a d-pointer

Thick abstractions should be pimpled. Even if right now one doesn't see the need for expansion, always leave a `class Private *d = nullptr;` to allow for future additions, and an out-of-line destructor to be able to free it (the day it actually gets used) without breaking ABI.

(THICK.2) Honor the rule of five

Because THICK.1 imposes an out-of-line destructor, make sure you declare all five special member functions:

1. copy constructor
2. copy assignment operator
3. move constructor
4. move assignment operator
5. destructor

Implement them as instructed below (THICK.3, THICK.4, THICK.5).

(THICK.3) Implement the move constructor idiomatically

The move constructor must be inline (and likely `= default`, see below). In case it can't be defaulted, it still must be defined inline.

Since thick abstractions are typically pimpled, this means

1. resetting the moved-from instance's d-pointer to `nullptr`, **and being ready to deal with a null d-pointer in all codepaths**;
2. or resetting the d-pointer to point to a statically allocated special instance (`sharedNull` or similar).

Use `std::exchange` to implement the body of the move constructor and reset the moved-from to a *valid but unspecified state* (cf. VALUE.9); do not hand-roll `std::exchange` in terms of `std::move` plus a reset.

For pimpled types that use `Q(Explicitly)SharedDataPointer` (cf. THICK.8), provide an inline defaulted move constructor, and use the `QT_DECLARE_QESDP_SPECIALIZATION_DTOR` and `QT_DEFINE_QESDP_SPECIALIZATION_DTOR` family of macros to make it work (grep in qtbase for usage examples).

(THICK.4) Implement the move assignment operator idiomatically

Don't DIY, use the convenience macros for this (private APIs):

- If a class uses memory and only memory as its resources (no file handles, no user-defined datatypes, etc.) then the move operations are pure swap, so use `QT_MOVE_ASSIGNMENT_OPERATOR_IMPL_VIA_PURE_SWAP`
    - A possible exception are types that allocate arbitrary (huge) amounts of memory and one wants to be certain not to keep allocated for too long, like QImage/QPixmap. Be sure to document the different strategy via code comments.
- Otherwise, use `QT_MOVE_ASSIGNMENT_OPERATOR_IMPL_VIA_MOVE_AND_SWAP`

(THICK.5) Implement the copy operations idiomatically

The copy constructor should be out-of-line, and likely = *default*.

The copy assignment operator should be

- inline
- implemented as copy-and-swap. **Any alternative implementation must be justified with a comment**.

Both should be `noexcept` (VALUE.13).

(THICK.6) Implement a member `swap` and a free `swap` overload

Thick classes can usually be swapped faster than `std::swap` can do, so they must provide a free `swap` function. Implement that function in terms of a member `swap`. (Technically, the member isn't required, and the free could be a friend; but stick to this pattern for consistency with existing code.)

The free `swap` overload is automatically provided by `Q_DECLARE_SHARED`.

(THICK.7) Use reference counting / implicit sharing

Qt thick value types are normally reference counted. If you have a type that is not, it **must** be justified with comments in code and carefully explained in the documentation. (Rationale: users may write getters that return objects of the type by value or similar.)

(THICK.8) Use *managed* reference counting

In order to implement reference counting **avoid the NIH syndrome** by employing a hand-rolled implementation. Instead, one of the established solutions, most likely (at the time of this writing) `QExplicitlySharedDataPointer` + `QSharedData`.

Do not use `QSharedDataPointer`; it is error prone and expensive (as it detaches on any non-const access); working around its API is cumbersome. Exception: use it if your API does not allow for any mutation (e.g. a "result object" from some query, with data that you can only read, so no detach is ever necessary).

Use `Q_DECLARE_SHARED` to correctly declare typeinfo (VALUE.3) and `swap` (THICK.6).

Implement a member `detach()` (as public API) to signal that your type is implicitly shared, and to implement detaching correctly (incl. allocating a d-pointer if you don't have one, because default-constructed or moved-from).

Acknowledgments

Thanks to KDAB, the Qt, C++ and OpenGL Experts.

Thanks to Marc Mutz for tirelessly bringing these efforts forward along the years. Thanks to Sérgio Martins for creating and maintaining Clazy.

References

1 QtWS15 - Qt Value Class Design, Marc Mutz, KDAB https://www.youtube.com/watch?v=Y3YMA1Ip3ek

2 Designing value classes for modern C++ - Mark Mutz @ Meeting C++ 2014 https://www.youtube.com/watch?v=_cwVvBsZE6M

3 Coding Conventions https://wiki.qt.io/Coding_Conventions

4 C++ Core Guidelines https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

5 Clazy List of checks https://github.com/KDE/clazy#list-of-checks

6 The Power of Hidden Friends in C++ https://www.justsoftwaresolutions.co.uk/cplusplus/hidden-friends.html