D1412r0 - On user-declared and user-defined special member functions in safety-critical code

Peter Sommerlad

2019-01-22

Document Number:	D1412r0
Date:	2019-01-22
Project:	Programming Language C++ Programming Language Vulnerabilities C++
Audience:	SG12 / ISO SC22 WG23, SG20, Misra C++

1 Introduction

C++ is a complex language providing a lot of flexibility in its use. However, not all programs are valid or behave as a developer thinks. Because of its deterministic resource management C++ is used in many systems that have high quality and safety-related requirements. While high software quality does not make a system safe or fault-tolerant, it can be an enabling factor to achieve them. Therefore, in safety-critical environments adhering to programming guidelines restricting the language use are one of the required practices to avoid safety risks, especially undefined behavior.

Over the decades of C++ evolution, best practices and what is considered good style changed a lot. Typical systems built with the language often outlive the style they were written in. With the introduction of a three years release cycle and the frequent innovation and updates it becomes hard to keep up. Some style guides try to achieve that by taking a looking glass and even proposing to use features of the language or library that are not yet standardized or even implemented ([SS18]). Other relevant guidelines that are industry standards stem from a past and provide a style that is considered no longer recommendable by C++ experts.

One area, where classic and more modern C++ versions differ are the recommended practices of declaring and defining special member functions in a user-defined class type. The introduction of move operations with C++11 and the guaranteed copy-elision of C++17 are game changers to the rules viable for C++03, and the world has not become simpler for novices due to that. This paper explains the situation by repeating Howard Hinnant's classical special member functions table[Hin] and suggest a set of special member function declaration/definition combinations the author recommends. The underlying philosophy, such as categorizing class types into value types, empty types, managing types, pointing types, and object-oriented polymorphic types is explained

below together with other attributes the author believes are important for modern high-quality C++ code.

2 History

With pre-C++11, class type usage often is taught to introduce object-oriented programming with virtual functions and inheritance. The rule-of-three stems from that era and usage[Mey05]. I have seen a lot of C++ software employing dynamic polymorphism where static polymorphism would have been sufficient. This is often due to the design and language know-how available, e.g., where developers were originally trained in Java or C#,. For example, applying the Bridge Design Pattern [GJHV95] to achieve portability across operating systems, where never two operating systems would be supported simultaneously. This practice might stem from times before policy-based design using template parameters was well known, or because the used compilers did not support templates appropriately. Another reason might be the "fear of template error messages" that lead developers shy away from them. We and modern C++ safety programming guidelines should explain that much better and provide more guidance to appropriately use the language features for software design.

2.1 The Rule of Three for C++03

TODO [?]

2.2 Rule of Zero

The easiest solution to potential confusion of what special member functions to declare or define is to write class types by carefully selecting member variable types in a way that the compiler-provided default implementations are automatically correct. For example, choosing only value types as member variable types and providing member initializers for their default values, will give a compiler to provide a defaulted default constructor as well as copy and move operations and a defaulted destructor, without the need to add any user code.

Code that is not written is generally also not incorrect.[1] In addition code that is not there does not add to the cognitive burden of understanding code in reviews or when evolving it. One pre-requisite that makes that relieve is to understand and internalize compiler defaults.

TODO: Properties to be discussed.

TODO: Potential dangers.

2.3 Howard Hinnant's special member function overview

At ACCU 2014 Howard Hinnant [Hin] explained move operations of C++11 and showed and explained the Table 1 in his famous keynote.

3 Forces for Safety in Source Code

As a pattern (book) author I would like to introduce so-called "forces" that are used in a pattern's problem description to denote design constraints that influence the pattern's solution. Often such

1) I know there are exceptions to that rule.

Table 1 — Howard Hinnant's special member functions table

user declares	What the compiler provides for class X						
	`X()`	`~X()`	`X(X const&)`	`=(X const&)`	`X(X &&)`	`=(X &&)`	OK?
nothing	`=default`	`=default`	`=default`	`=default`	`=default`	`=default`	OK
`X(T)`	not decl	`=default`	`=default`	`=default`	`=default`	`=default`	OK
`X()`	*declared*	`=default`	`=default`	`=default`	`=default`	`=default`	(OK)
`~X()`	`=default`	*declared*	`=default`[1]	`=default`[1]	not decl	not decl	**BAD**[2]
`X(X const&)`	not decl	`=default`	*declared*	`=default`[1]	not decl	not decl	**BAD**[3]
`=(X const&)`	`=default`	`=default`	`=default`[1]	*declared*	not decl	not decl	**BAD**[3]
`X(X&&)`	not decl	`=default`	`=delete`	`=delete`	*declared*	not decl	**BAD**[4]
`=(X&&)`	`=default`	`=default`	`=delete`	`=delete`	not decl	*declared*[5]	BAD[4]

[1] generating copy operations in case of a user-defined destructor or other copy operation is considered a bug in the standard, but can not be easily changed due to backward compatibility.
[2] Defining a destructor while allowing default copying is almost always an error.
[3] Inconsistent copy operations (one user-defined, one compiler-provided) are almost always wrong.
[4] Inconsistent move operations (one user-declared/defined, the other deleted) are often wrong.
[5] Declaring a deleted move-assignment operator can help to suppress compiler-provided copy operations in the case of a user-declared virtual destructor without influencing the provisioning of a default constructor.

forces are not absolute and a pattern make conscious trade-offs. That is also a reason, why often conflicting patterns for a problem exist that resolve to different solutions.

Here I collect forces that in my observation have influenced existing programming guidelines.

— Simplicity

— Familiarity

— Code Evolution (aka Maintenance)

TODO.

4 Class type categories

I observed that programming guidelines tend to shy away from helping developers think about their design. This is a situation I'd like to change. One of the contributions I'd like to make is give guidance on roles of types, here class types. While a user-defined class type can be defined in a very flexible way, mixing these roles unconsciously can lead to serious problems. I distinguish the following major categories:

— value types

— empty types

— managing types

— pointing types

— object-oriented polymorphic types

4.1 plain categories

This section introduces the different core type categories and their implication on special member functions. Later on, some useful combinations are considered.

4.1.1 value types

C++'s built-in mechanics all directly support value types. Initialization, assignment and comparison. Scott Myers explains it, "when in doubt, do as the `int`s do!". While I have some problem with the built-in integral types as well, this covers the intention well. If you do not define any special member function and your member variable types are also value types, then your class type will work as a value type without thinking.

If a value type supports equality comparison it is supporting the concept Regular. Regular means, default construction, copying and assignment are well defined and equality comparison works as expected. Without comparison, the concept is Semiregular. A plain C-like struct with only value-type data members is automatically Semiregular. Such an aggregate, a struct with public value type data members, automatically obtains all compiler-provided special member functions (default constructor, destructor, copy- and move-operations).

Plain value types will automatically obtain correct compiler-provided special member functions.

4.1.2 empty types

It might sound counter-intuitive to define class types without any non-static data members. There can be only one value in such a type, since there are no bits to distinguish different values. However, empty types in a strong type system, like C++, can serve a variety of purposes:

tags Used as a function parameter type, such empty types can help to distinguish function overloads. For example, the iterator tags obtainable from an iterator type help library algorithms to select better performing implementations for "stronger" iterator.

mix-ins Empty types can contain additional member or regular (friend) functions that can be mixed into the scope of a class deriving from such a mix-in type. This can be used to ease introducing generic (operator-)function overloads for a class type without overhead. C++'s Empty Base-class Optimization (EBO) or the new C++20 attribute `[no_unique_address]` help with employing functionality of empty types.

traits Empty types are often used as template meta functions that allow compile-time type and value computations. For example, `std::is_same<U,V>` is a binary type trait comparing two types for equality. Such a type trait class does not contain any non-static data members or non-static member functions. Another use of such traits is to implement values or value sequences as types, such as std::true_type, std::integral_number, or std::index_sequence. Newer C++ versions introduced template aliases with the suffix `_t` for type computing type traits and variable templates with the suffix `_v` for type conditions that ease the use of the standard type traits.

Since there is nothing that can go wrong with a type that has only one value, the rule-of-zero applies here as well. There is no invariant to guarantee and no cleanup to care for. In addition there is no difference between copy or move operations and the compiler provided ones are perfectly safe and sane.

Empty types will automatically obtain correct and trivial compiler-provided special member functions.

4.1.3 managing types

Managing types have a destructor that has non-trivial behavior. Typical managing types hold on to a resource that is either acquired or given to it at construction time and release that resource in the destructor. C++ RAII (resource acquisition is initialization) principle often as SBRM (scope bound resource management) using local variables is one of the cornerstones. The need for a destructor also means that copying and moving must be taken care for. Either of these operations require careful consideration.

Managing types of member variables make a containing class also a managing type, even so it does not itself define a destructor.

A simple managing type is usually not copyable and encapsulates its managed state. If that state can not be empty or requires user arguments to create it will also not provide a default constructor. So in the simple SBRM case, a destructor is defined and copying can be prohibited by defining the move-assignment operator as deleted.

In case the managing object needs to be returned from a factory function pre-C++17, a move constructor is necessary to guarantee a unique managing object per allocated resource. This means, that in a moved-from state needs to be representable to avoid double-release of the managed resource. This can mean overhead, such as using a `std::optional<Resource>` data member instead of representing the `Resource` type directly. In case of pointer types as resources, such as std::unique_ptr manages, the unique value for invalid/released pointers `nullptr` is always available.

For more complex managing types that also take one or more of the other type categories (value, OO), expert knowledge for design and implementation is required. For example, implementing a container class, like std::vector that manages multiple objects, while itself is a Regular type, if the elements' type is, requires really world-class expertise to do well. So better use what is available than re-invent the wheel. Only in the case of a proven need, e.g., via profiling and test cases, a special-purpose solution should be considered.

For managing multiple resource object members, it is better to wrap each resource in its own manger type, so that construction failures half way through the object construction are automatically cleaned up correctly by the compiler-provided mechanics rather than trying to do that by hand. To make the latter work well, all resource manager member variables must be initialized in all constructor's member initializer list or via member initializers. This wisdom was a tough learning from specifying and implementing the proposed generic `std::unique_resource` class. The latter as well as several resource management helper types of the standard library, like `std::unique_ptr`, containers like `std::vector`, `std::string`, etc. should be your go-to building blocks to implement your own managing types on top.

Managing types will typically user-define one or more constructors. It depends on the members if a user-defined destructor is necessary. A plain managing type will inhibit copying, often by allowing

move operations, at least move construction for returning from a factory function. If no such factory facility is required or at least C++17 is used, a managing type can also define move-assignment as deleted to inhibit all copy- and move operations. Richard Corden calls such a type "monomorphic" object type, because it encapsulates functionality, it is not a value type, and thus its object identity is an important factor.

4.1.4 pointing types

One potentially dangerous area of C++'s types are *pointing types*[2]. The language allows constructing references to objects (`T&` or `T&&`) that must be initialized and eliminate assignment and copy-operations if used as members and it allows to construct pointers to objects (`T*`) that need to be initialized to avoid invalid references to be used. One can think of `T*` as if it be an `std::optional<T&>` that is unfortunately not allowed (yet?). Beside the plain reference or pointer types constructed by the language, all library container's iterator types fall in the category of pointing types. Also, less obvious, `std::string_view`, `std::span`, or `std::reference_wrapper` are pointing types.

Pointing types carry the danger of *undefined behavior* if the *pointee object*[3] no longer exists when accessed via the pointer object. At run-time such a situation can often not easily be detected. At least in the case of pointers to objects, such a situation can be made signalled, by setting the pointer object to `nullptr` when its pointee object is about to die, because `nullptr` is a detectable invalid state. But this requires that the code that ends the pointee object's lifetime to know about the pointing object, so it can adjust it. Therefore, lifetime management of pointee objects is critical and must be a very conscious design consideration. For example, memorizing an iterator obtained from a container object, e.g., as the result of a `find_if()` algorithm, must take into account the iterator validity guarantees of the underlying container, if such container changes subsequently. In C++ standard library containers are unaware of existing external iterator values. So they can not automatically invalidate iterators or the iterator objects can not detect a changed container. The best solution is to avoid keeping pointing objects longer than their pointee's lifetime, e.g., by only passing them down the call chain, from where a pointee object lives.[4]

As with managing types, pointing types as members are "contagious", so any class type with a member of pointing type is also in the category of being a pointing type with the burden to carefully design or manage the lifetime of the pointee object.

As said above, one means to sane and safe lifetime management of pointee objects is to allocate those high on the call tree and pass down their references as function arguments down the call tree. That way, the lifetime of the object higher on the call chain. This simple mechanic breaks down, when references are shared across threads, or references need to be returned from functions. The necessary details of sharing objects across threads is a separate topic and beyond the scope of this paper. Returning pointing objects from functions requires a clear contract about the pointee's lifetime and adherence.

---

2) This document uses terms references and pointers for objects of *pointing types* interchangeably, unless specified.

3) This document uses the term *pointee object* to denote an object or memory that is referred to by an object of a pointing type.

4) Using objects with static or global storage duration as pointee objects, might seem to solve that problem, but raises multiple others, such as testability, composability, synchronization, or even introduce subtle initialization-order bugs. Global variables as an approach should only be taken for very very tiny closed solutions that are heavily scrutinized, especially if re-used or evolved.

In C, plain pointers are often used as a marker for an optional return value. In case of a failure, such functions return NULL and any other value is assumed to be a valid pointer to the result. Then one of the hard parts of C's function design happens, depending on the function called, the caller might be responsible for the memory returned as a pointer, or not. In the latter case functions often return the address of a variable with static lifetime, making them non-thread safe and often also non-reentrant. Both cases are often addressed in C, by versions of the functions obtaining a pointer to memory provided by the caller, e.g., as the address of a local variable. In C++, even with pointing types, such error prone mechanics are not necessary and usually verboten by programming guidelines.

The following options for optional returning are better than using a pointer return type:

return-by-value Even for larger objects, return by value is efficient in modern C++ through move operations and guaranteed return-value-optimization (RVO) even for factory functions. Failure can be denoted either by a special value of the type, if available, or by throwing an exception.

return optional<T> If the type T does not have a distinguished error value, and throwing an exception is also inappropriate, use the option std::optional<T>. In situations where a C function would return a NULL pointer when there is an error this is a better means to return a value and indicate a failure. The value return gives now doubt about janitorial duties of the caller, there aren't any.

You can not live without pointing types, but you should reduce their use to an absolute minimum. Use value types whenever you can, unless copying the values really becomes a burden. If you really need pointing types and need efficiency, keeping a manager object for all pointees high up in the call tree that manages pointee objects' lifetimes can help to avoid dangling pointing objects. That manager object itself or one of its managed objects must then be passed down the call tree by reference to be useful.

You rarely will define your own class types that are pointing types. If so, they should act like pointers and be value types, see below.

If you put pointing types as members in a class that is not itself a pointing type, such a class is a manager type and the guidelines given above apply.

4.1.5 object-oriented polymorphic types

As said above, in some older code bases, object-oriented types with inheritance are often overused. However, if you have the problem of run-time flexibility and a good abstraction (=base class type) to specify the common behavior, an object-oriented class hierarchy can still be a viable solution. While not always needed, you might want to allocate such objects on the heap. Today this should be done using std::make_unique() to automatically have a manager object for each heap-allocated polymorphic object, so your programs do not leak.

A key indicator of a polymorphic type is the use of the `virtual` keyword with member functions to denote the hook methods that derived classes can override. Whenever, such objects get heap allocated and stay referred only through base-class (unique) pointers, the base-class' destructor

should be declared `virtual`.[5] From the table explained in the next chapter, we learn that this automatically suppresses compiler-provided move operations, but still allows copying.

For object-oriented programming the object identity is an important factor and copying objects by the default standard mechanism of C++ can introduce subtle problems. For example an object instance of a derived class that is only known through its base-class reference, should not be copied into a base-class variable, because such a copy will slice the original object, and lose any information from the more derived class' members. It is best, to suppress copying for polymorphic hierarchies as well, to avoid this unintended object slicing. Often developers from other, reference-based languages (Java, C#) use assignment of a subclass' object to a base class variable unconsciously and thus slice.

Dynamic polymorphism with multiple base classes with virtual member functions is most often a design mistake and should be avoided. So most safety guidelines prohibit the use of virtual base classes that easily are needed in such cases. Unfortunately, many "natural" categorization examples used in object-oriented introductory courses tended to show diamond inheritance. If you see that in a design, refactor it or run away :-) Therefore, dynamic polymorphism with multiple (virtual) base classes is not further considered. Note, this does not inhibit multiple inheritance in general, e.g., for using empty base classes to mix-in functionality.

An object-oriented base class that declares virtual member functions and that has no further public base classes will define a defaulted virtual destructor. This already inhibits compiler-provided move operations, but unfortunately would still generate copy operations. To inhibit this with the minimal amount of user-written code, this paper suggests to define a move assignment operator as deleted. This has the benefit of inhibiting the compiler to provide copy special member functions without influencing the provisioning of a default constructor. Other guidelines suggest to be explicit in this case, requiring to define copy constructor and copy-assignment as deleted. This however, requires the "resurrection" of the then inhibited default constructor. So instead of defining an additional two to three special member functions, a single deleted move assignment operator suffices. As Björn Fahller says

> "But, don't fall into the trap of confusing unfamiliarity with difficulty. It's OK to have to learn something new to understand a solution."

Therefore, I suggest something unfamiliar that requires less code to write and also is easy to spot.

4.2 Category Combinations

Combining different categories in a single type often requires expert-level C++ knowledge. But this easily happens unconsciously, for example, by obtaining copy-operations automatically for a polymorphic class hierarchy using virtual member functions leading to unintended object slicing. This section tries to demonstrate some of the viable combinations and gives some hints on what to look out. Further combinations, not mentioned here, are either a design error, or if done consciously, very tricky to implement and use correctly. In a safety critical systems such combinations should be avoided or at least heavily scrutinized and proven that no simpler approach is feasible.

5) The exclusive use of `shared_ptr` and `make_shared<derived>()` for all heap allocations can make that work without a `virtual` destructor in the base class, but comes at a high run-time cost.

It does not make sense to combine empty types with any of the others, because they no longer would be empty.

4.2.1 Managing Types as Value Types

The standard containers are a classic example of a managing type that also works as a value type. However, implementing them correctly requires first class expertise, especially since they have to take into account almost arbitrary element types. In a less generic sense, one might be able to have a simpler live, by not only implementing a destructor, but also sane copy and move operations. It heavily depends on the kind of managed resource if copying makes sense or not. For example, the standard library's `fstream` classes manage a buffer object with a file handle used for I/O and you can not copy such stream class, because I/O via a single file handle from two different sources, easily mixes up output, since the operating system usually keeps a single current I/O position for a file. It would be ridiculous to copy such a stream object.

If you do not need your own managing types to behave as values and the underlying abstraction does not have a direct value representation, do not make them values. The implementation burden and the risk of bugs seems to high. Just make sure, that copy operations are suppressed.

4.2.2 Pointing Types as Value Types

Plain pointers and (most of the) standard iterators in C++ are Regular types. You can assign a value to another variable and both refer to the same pointee. You can even compare two pointers for equality if they are from the same underlying range of objects or have the `nullptr` value. Copying pointing objects is cheap in general but increases the risk of dangling, when no care is taken how many pointing objects to a given pointee are around.

The class `std::shared_ptr` tries to keep track of existing pointer objects to a given pointee and manages the lifetime of the pointee. But this comes at an increased cost for each copy, due to synchronization of the atomic counter. In addition sharing the pointers across threads is well defined, but the pointees must take synchronization seriously if not immutable. If possible use `std::unique_ptr` and pass references to the pointees instead. For creating simple object graphs unique_ptr might be sufficient, but if there are more interesting toplogies, plain pointers might be needed in addition. Do always encapsulate such a situation with a corresponding managing type and use that to create and navigate the underlying object graph.

4.2.3 Managing Types and Pointing Types

This is often a natural combination, since managing types often need to employ pointing types for member variables to keep track of the managed stuff. Using "nullable" pointing types also provides a simple to identify up moved-from state, where the corresponding member variable gets assigned nullptr if the managing object gets into a moved-from state, i.e., by being passed as an argument to a move constructor.

4.2.4 Value Types as dynamic polymorphic Types

Sean Parent gave a nice talk [6] on how to get the benefits of dynamic polymorphism with the ease of use of value types. His approach nicely encapsulates virtual members in a common and

6) https://sean-parent.stlab.cc/papers-and-presentations/#better-code-runtime-polymorphism

generic handle type and even allows the different types to be used polymorphically to be plain value types completely unrelated by inheritance. Sean Paraent's "trick" thus allows retrofitting dynamic polymorphism and encapsulates all uses of virtual, which is nice. However, it might also hide the fact that some seemingly unrelated classes implement a common abstraction.

4.2.5 O-O polymorphic Types as Value Types

Another approach is fostered by a paper https://wg21.link/p0201 currently considered for standardization where a `polymorphic_value` wrapper type adds value semantics to a class hierarchy, where copying through `polymorphic_value<base>` that actually hold an object of type `derived` will not slice. This requires heap allocation and defined copy operations for all classes in the class hierarchy, either by providing copy constructors (with the risk of slicing) or by parameterizing polymorphic_value's constructors with a non-default copy operation function object. While useful in cases where such polymorphic class hierarchies already exist or are an appropriate design choice, I would not consider such a design at first, unless proven necessary.

4.3 Outlier Types

One might argue that the above categories do not cover all useful C++ types. For example, in my teaching I use a tracer class that generates output on construction and destruction to demonstrated the deterministic lifetime of objects in C++. This tracer class violates the rules set out in this paper, by defining a destructor with a side effect and still allow copying and in some cases even declares a move operation. But such a class and corresponding mechanisms (e.g., side effects in destructors that might fail) is not something you should use to model a production quality safety critical software system.

A second use case for really strange behaving types are test cases for generic code. For example, if you need to make sure no resource leaks on exceptions for generic code, you will have to create classes that throw on move or copy operations deliberately to see if the implement mechanism fulfills its exception safety guarantee promises. Again such strangely behaving types will never be used in production. Developers doing so, should get their compiler license withdrawn.

4.4 Items to be discussed

Things I am unsure

— Are there further useful and safe exceptions?

5 Bibliography

Bibliography

[GJHV95] Erich Gamma, Ralph Johnson, Richard Helm, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[Hin] Howard Hinnant. Everything you ever wanted to know about move semantics (and then some).

Table 2 — Safe and Sane combinations of Special Member Functions (TODO)

	some ctor	default ctor	destructor	copy ctor	copy assign	move ctor	move assign
Aggregate	none	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Values	yes	none / `=default`	defaulted	defaulted	defaulted	defaulted	defaulted
Manager	typical	none / `=default`	`=default` to expert level	(=delete)	(=delete)	`=default` (=delete)	`=default` `=delete`
RAII	yes	none / `=default`	user declared	(=delete)	(=delete)	`=default` (=delete)	(=delete) `=delete`
OO - base	maybe	may be	`virtual` `=default`	(=delete)	(=delete)	(=delete)	`=delete`
Manager as Value	typical	`=default`	expert level	expert level	expert level	expert level	expert level
polymorph from value (S. Parent)			expert level to `=default`	expert level	expert level	expert level	expert level

[Mey05] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs.* Pearson Education, 2005.

[SS18] Bjarne Stroustrup and Herb Sutter. C++ core guidelines, 12 2018.