p1412 - On user-declared and user-defined special member functions in safety-critical code

Peter Sommerlad

2019-01-21

Document Number:	p1412
Date:	2019-01-21
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 to avoid safety risks, especially undefined behavior, are one of the required practices.

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 class and even proposing to use features of the language or library that are not yet standardized or even implemented ([**?**]). Other relevant guidelines that are industry standards stem from a past and provide a style that is considered no longer recommendable.

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. This paper explains the situation by repeating Howard Hinnant's classical special member functions table and suggest a set of special member function declaration/definition combinations the author recommends. The underlying philosophy, such as categorizing class types into values, managing types, and referring types, is explained below together with other attributes the author believes are important for modern high-quality C++ code.

Promote value types/regular types.

UB

level of developer expertise

context of audience.

2 history

rule of zero and related

Properties to be discussed.

Potential dangers.

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 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)

4 Howard Hinnant's special member function overview

Table 1 — Howard Hinnant's special member functions table

	What the compiler provides for class X						
user declares	`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`	`=default`	not decl	not decl	**BAD**
`X(X const&)`	not decl	`=default`	*declared*	`=default`	not decl	not decl	**BAD**
`=(X const&)`	`=default`	`=default`	`=default`	*declared*	not decl	not decl	**BAD**
`X(X&&)`	not decl	`=default`	`=delete`	`=delete`	*declared*	not decl	**BAD**
`=(X&&)`	`=default`	`=default`	`=delete`	`=delete`	not decl	*declared*	(BAD)

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

	declared or provided						
type category	`X()`	`~X()`	`X(X const&)`	`=(X const&)`	`X(X &&)`	`=(X &&)`	
value/aggregate	`=default`	`=default`	`=default`	`=default`	`=default`	`=default`	OK
value	not decl	`=default`	`=default`	`=default`	`=default`	`=default`	OK
`X()`	*declared*	`=default`	`=default`	`=default`	`=default`	`=default`	(OK)
OO	`=default`	*declared*	`=default`	`=default`	not decl	not decl	**BAD**
`X(X const&)`	not decl	`=default`	*declared*	`=default`	not decl	not decl	**BAD**
`=(X const&)`	`=default`	`=default`	`=default`	*declared*	not decl	not decl	**BAD**
`X(X&&)`	not decl	`=default`	`=delete`	`=delete`	*declared*	not decl	**BAD**
`=(X&&)`	`=default`	`=default`	`=delete`	`=delete`	not decl	*declared*	(BAD)

4.1 Items to be discussed

Things I am unsure

— Are there further useful and safe exceptions?

5 Categories of safe user-defined classes

5.1 Plain Value Types

5.2 Monomorphic Object Types (better name) - Encapsulation Types

5.3 Polymorphic Object Types - Class Hierarchies

5.4 Resource Managing Types

6 Bibliography

Core Guidelines

MISRA

Rule of Zero

Howard's table