P2070R0: A case for object_ptr and optional reference

Date:	2020-01-13 (Prague)
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG12, WG21, WG23(C++)

Authors:	Peter Sommerlad, Michael Wong, Jan Babst, Anthony Williams
Contributors:	MISRA C++
Emails:	peter.cpp@sommerlad.ch

Revision History

Revision	Description
P2070R0	Inital version based on Oct 17 MISRA C++ meeting

Introduction

In C and in many classical C++ code, plain pointers serve multiple purposes. So using a pointer variable or a function taking a pointer or returning a pointer value, often requires extra documentation or understanding of the local convention used, not to misuse the interface or misunderstand the code when changing it. We can not get rid of pointers as an implementation detail, but safety critical code must not rely on a developer guessing meaning of every use of a pointer type. This paper maps potential alternatives to previous pointer use, often through existing language or standard library mechanisms, that first, document the intent and second, guarantee more correct usage through the type system.

Note to Reviewers

The MISRA C++ group will meet between the pre-Prague mailing deadline and the Prague meeting. So we expect to have a revision of this paper available by the Prague meeting. Please check the link [4] for any updates when you read this.

Motivation

In many existing and older C++ code, pointers carry a leading role in the definition and implementation of interfaces. However, pointers carry inherent risks, such as: being nullptr, referring to already destructed objects (dangling), being uninitialized, question of ownership, question of single object or start of array, and even more uses and potential misuses.

Safety Critical Case

Guidelines for programming safety critical systems contain rules that often strive to reduce the amount of potential misunderstandings and try to reduce error prone constructs. History shows that raw pointers are such an error prone construct. In addition C++' pointers inherited from C many operations that make each use of a pointer parameter or argument a guesswork to figure out what that pointer actually means.

The C++ type system and current C++ standard library, on the other hand, provide often much safer and better intent communicating replacements to old pointer usage. The goal of C++ safety guidelines must be to foster those better replacements and encapsulate plain pointer usage where it is really needed for implementation.

Wider C++ Case

Everything said above about safety critical code is also true, while not as pronounced in regular code. (TBD)

Let us look at the role pointers can play in such code and then look at the potential alternative representations for the pointer to single object use cases.

Pointers to ranges

For the use cases where pointers represent ranges of values of identical types, such as when an array is passed to a function, the standard library provides the alternatives: std::array,

std::vector, iterators, std::span, std::string, std::stringview, and ranges. While some of these alternatives shared the risk of referring to no-longer valid ranges, other deficiencies of using pointers in that context are eliminated.

Pointers to single objects

This case is the topic of this paper. To categorize the potential usage of pointers in C++ code, we distinguish the following categories:

- Ownership: pointer refers to owned object (lifetime needs to be managed via the pointer) vs. pointer refers to object not owned (potentially dangling, lifetime of object needs to be longer than pointer value usage)
- Nullable: pointer might be nullptr (requires checking on access) or always refers to an object

With these established categorizations the following replacement strategies for replacing pointers T* (as variables, parameters or return types) are possible:

	Owning vs Referring	**Nullable?**	**Purpose**	**Replacement strategy for T***
0	Owning	non-null	Value representation	T
1		nullable		std::optional<T>
2		non-null	Heap allocation required	unique_ptr<T> const [1]
3		nullable		unique_ptr<T>
4	Referring	non-null	Fixed binding	T&
5		non-null	rebindable	std::reference_wrapper<T>
6		nullable	Fixed binding	--- missing --- *optional<T&>* *object_ptr<T> const*
7		nullable	rebindable	std::optional<reference_wrapper<T>> *object_ptr<T>*

While using values is clearly preferable, there are situations where a referring type is needed and where rebinding can happen or should not happen, to avoid dangling, for example. These

[1] Always initialized from make_unique() to a valid non-nullptr value.

are the cases, where the standard library does not yet provide a replacement for T*, at least to our understanding.

The C++ Core Guidelines (R.2, R.3), for example, suggest to use T* for referring to non-owned single objects. In our opinion, this advice is unsafe, because it is a convention only that can not be checked by a compiler and as a convention does not prevent the potential misuse discussed above. In addition, using a raw pointer in the sense of the Core Guidelines rules R.2/R.3 is indistinguishable from other uses in legacy code, so every time, one encounters such a pointer, one needs to justify if it is just following the guidelines, or is code that is not yet following the guidelines.

Suggestion 1: Allow std::optional<T&>

Introducing optional<T&> as a replacement in case 6 of the table above requires a design of optional<T&> which forbids rebinding, i.e. assignment from T& or optional<T&>. Interestingly, this is the same design decision as taken in https://wg21.link/p1175, albeit for different reasons. The paper P1175R0 suggests to extend the current standard library by allowing optional<T&> as a nullable, non-rebindable referring type. We completely follow its argumentation and strongly support its inclusion into the standard library.

Suggestion 2: Adopt a modified std::ext::observer_ptr<T>

While we believe that std::optional<T&> should be used in modern code denoting a nullable-reference, we also understand that it might be too much of a burden to retrofit it to code currently using raw pointers. Despite its cumbersome name, observer_ptr<T> lacks some features that would make it an ideal candidate. We need a pointer type that does not allow pointer arithmetic to designate it is referring to a single object (which observer_ptr does) and ideally one that easily converts from other smart pointers, so they can be used without breaking their encapsulation (which observer_ptr does not yet). The role this new smart pointer type can play, is the one of T* as parameter type as suggested by the C++ core guidelines rules R.2/R.3. Fortunately, Anthony Williams already implemented such a better observer_ptr under the name of jss::object_ptr<T> [3]. We think ptr<T> might be a viable alternative name for the standard, attractive due to its shortness.

From [3]:

> “Differences to std::experimental::observer_ptr
>
> The most obvious change is the name: observer_ptr is a bad name, because it conjures up the idea of the Observer pattern, but it doesn't really "observe" anything. I believe object_ptr is better: it is a pointer to an object, so doesn't have any array-related functionality such as pointer arithmetic, but it doesn't tell you anything about ownership.

> The most important change to functionality is that it has implicit conversions from raw pointers, std::shared_ptr<T> and std::unique_ptr<T>. This facilitates the use of jss::object_ptr<T> as a drop-in replacement for T* in function parameters. There is nothing you can do with a jss::object_ptr<T> that you can't do with a T*, and in fact there is considerably less that you can do. The same applies with std::shared_ptr<T> and std::unique_ptr<T>: you are reducing functionality, so this is safe, and reducing typing for safe operations is a good thing.
>
> The other change is the removal of the release() member function. An object_ptr doesn't own anything, so it can't release ownership. To clear it, call reset(); if you want the wrapped pointer, call get() first."

Impact on the Standard

This will be a pure Library feature and as such will require LEWG and LWG input.

Design Proposals

Adopt P1175.
Add ptr<T> with the features of jss::object_ptr<T> [3]. Wording TBD.

Are we getting too many smart pointers?

While we understand the reluctance to add more smart pointers to the C++ standard library, we see definitely a need for the suggested object_ptr<T> to referring a single object as an important vocabulary types to reduce the use of error-prone raw pointers. While Andrej Alexandrescu showed in his Modern C++ Design that there might be more than 200 potential smart pointer categories and some shy away from adding such a volume to the standard library, we believe object_ptr<T> is an important feature to better employ the C++ type system and make code much safer and expressive in situations where T* would currently be used.

From our analysis, we believe there is no need standardizing further smart pointers. The case for representing sequence of objects will be resolved through stringview, span<T>, ranges and further view types for multi-dimensional representations.

Acknowledgements

Anthony Williams for his work on object_ptr<T>.

JeanHeyd Meneide for proposing P1175.

This paper is the result of the discussions of MISRA C++ WG.

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada.

References

[0] C++ Core Guidelines: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
[1] A simple and practical optional reference for C++: https://wg21.link/p1175
[2] Discussion of P1175: http://wiki.edg.com/bin/view/Wg21sandiego2018/P1175
jss::object_ptr<T>
[3] https://github.com/anthonywilliams/object_ptr
[4] The most current draft of this document D2070Rx is available at:
https://docs.google.com/document/d/1e4UcZ_udhiieAXy8U6QqPkxPxzT8261RGNSQpHidQqc/edit?usp=sharing