Nxxxx - How many smart pointer types could exist?

Peter Sommerlad

2014-10-10

Document Number: Nxxxx	
Date:	2014-10-010
Project:	Programming Language C++

1 Introduction

This paper tries to map the landscape of possible and useful smart pointer classes to foster discussion which parts of the map should be provided by the standard library in addition to the existing shared_ptr, weak_ptr and unique_ptr classes. My goal is to limit the usage of plain pointers in C++ user-level (as opposed to library) code to avoid the inherent ambiguities present when a plain pointer variable or parameter is present in such code. In the same sense also plain arrays should be banned, at least when the degenerate to pointers in passing as parameters.

The reason for this is, that multiple different smart pointers have been proposed in the past to better document pointer usage or provide additional functionality, but none of them seem to have passed the hurdle of the committee (N3515-N3740-N3840, N3339, N3974. I guess, one of the reasons, was that either of them was looked at individually and either its use case or its specification wasn't up to the committee's liking, or the authors went on to other more important issues than updating a only mildly acknowledged paper (my pure speculation).

Some currently proposed or already accepted classes also fall in the realm of the solution space, such as optional¡T¿, std::string, std::array, string_view, array_view, or using plain references or values of a type.

This paper is not about "rich" smart pointers as have been proposed by N3340. Also what is not addressed are the atomicity of smart pointers as given in N4058.

2 Acknowledgements

3 Motivation and Scope

3.1 Dimensions of Smart Pointer Services

Most of the dimensions are yes/no binary decisions, but for some it makes sense to consider multiple exclusive options. I use the terms pointer and pointee to denote the handle and the object, even if the handle type is not a (smart) pointer, such as an lvalue reference.

I will consider the following primary dimensions that drive the design. Note that plain pointers are used in today's code for almost all of these options (except shared, if correct and cloning):

- assignability of pointer, rebinding (no for references)
- pointer can be empty (nullptr) (no for references)
- ownership of pointee/resource: none, unique, shared
- single pointee vs. array, array of pointees means iteration
- polymorphic pointee (none, to base class, generic)
- polymorphic copy-ability of pointee (clone_ptr)

A naive view would give us 144 possible design locations and almost as many possible smart pointer like classes. Fortunately, not all make sense and many of the design locations are already handled by existing library and language features, but still an enormous amount of possible specific combinations of requirements remain. To make the design space smaller, I stop considering references as an possible solution and only consider the design where a pointer can actually be rebound and be empty or equal to nullptr, leaving 36 combinations.

The following dimensions are more-or-less derived from the designs of the primary dimensions.

- copy-ability of pointer (shallow copy) vs move-ability
- type erasure for cleanup and other special functionality, such as copying,
- pointer can be kept in standard container
- single allocation optimization for type erasure and

Table 1: default

own	[n]	poly	clone	solution	alt	comment
no	no	no	no	`exempt_ptr<T>`	T *	
no	no	no	yes	`exempt_ptr<T>`	T*	T copyable
no	no	bas	no	`exempt_ptr<T>`	T *	base with virtual members
no	no	bas	yes	non_owning_clone_ptr ?	`clone_ptr<T>`	or base virtual clone
no	no	gen	no	?		useful? non-owning variant/any
no	no	gen	yes	non_owning_clone_ptr?	void *	non-owning variant/any ?
no	yes	no	no	`array_view<T>`	T *	
no	yes	no	yes	`array_view<T>`	T*	T copyable
no	yes	bas	no		T**	array of pointers?
no	yes	bas	yes		T**	with base virtual clone
no	yes	gen	no	?		useful? array of non-owning variant/any
no	yes	gen	yes		void *	non-owning variant/any ?
un	no	no	no	`unique_ptr<T>`	`optional<T>`	
un	no	no	yes	`unique_ptr<T>`	`optional<T>`	T copyable
un	no	bas	no	`unique_ptr<T>`	T *	base with virtual dtor
un	no	bas	yes	`clone_ptr<T>`		base virtual clone and dtor
un	no	gen	no	`variant<>`	`any`	
un	no	gen	yes	`variant<>`	`any`	active type copyable
un	yes	no	no	`unique_ptr<T[]>`	T *	
un	yes	no	yes	`unique_ptr<T[]>`	T*	T copyable
un	yes	bas	no	`vector<unique_ptr<T>>`	T**	array of pointers
un	yes	bas	yes	`vector<unique_ptr<T>>`	`vector<clone_ptr<T>>`	with base virtual clone
sh	yes	gen	no	?		useful ?
sh	yes	gen	yes	?		useful ?
sh	no	no	no	`shared_ptr<T>`	`optional<T>`	
sh	no	no	yes	`shared_ptr<T>`	`optional<T>`	T copyable
sh	no	bas	no	`shared_ptr<T>`	T *	base with virtual dtor
sh	no	bas	yes	`clone_ptr<T>`		base virtual clone and dtor
sh	no	gen	no	`variant<>`	`any`	
sh	no	gen	yes	`variant<>`	`any`	active type copyable
sh	yes	no	no	`shared_ptr<T[]>`	T *	
sh	yes	no	yes	`shared_ptr<T[]>`	T*	T copyable
sh	yes	bas	no	`vector<shared_ptr<T>>`	T**	array of pointers
sh	yes	bas	yes	`vector<shared_ptr<T>>`	`vector<clone_ptr<T>>`	with base virtual clone
sh	yes	gen	no	?		useful ?
sh	yes	gen	yes	?		useful ?

4 Impact on the Standard

5 Design Decisions

5.1 General Principles

5.2 Prior Implementations

5.3 Open Issues to be Discussed

- Should we make the regular constructors private and friend the factory functions only?
- Should we provide a factory for type-erasing the deleter/exit_function using std::function?

6 Technical Specifications