Smart Pointers in C++

Lukas A. Schwoebel

***Abstract*—Pointers in C++ are lightweight and efficient, but due to their limited capabilities they come with some problems. It is easy possible that something goes wrong when copying resources or deleting resources that are pointed to by regular pointers. Smart pointers in C++ belong to a concept that tries to face these issues, while still simulating the behaviour of simple pointers and making it easy to exchange them without much effort. This paper discusses smart pointers, introducing their aspects and behaviour and outlining common errors while implementing them as well as problems that might occur.**

I. Introduction

Normal pointers in C++ and C are simple, yet efficient. The problem is, that resource handling is difficult and it takes a lot of effort to keep the memory consistent. Even a simple problem like deleting a resource can load to an inconsistent state, if the pointer was copied – how to determine if a resource was deleted already, how does a pointer even being aware that the resource it points to is also pointed to by other pointers?

Letting the user manage the pointers and the memory resources that are pointed to means freedom for the programmers but also leads to the danger of misuse – most smart pointer implementation perform useful tasks that prevent most misuses, like sophisticated memory management, locks at the resources like semaphores, automatically freeing of resources that are not used any more. The applications therefore do not need to observe and careful manage the lifetime of the objects they point to.

Yet, smart pointers are not a free lunch, as they come with some drawbacks. There are different type of smart pointers and it highly depends on the specific use case which type is the best.

In this paper we will first look at the common foundations of smart pointers including a simple implementation in Section II, then we examine different ownership management strategies in Section III. Section IV covers the problems with smart pointers concerning standard conversions. In Section V a short glance is taken concerning existing smart pointer implementations and the standard pointers. We will discuss the advantages and disadvantages of smart pointers among with common mistakes and misuses, most importantly problems with conversions. Section VII will briefly introduce *accessors* that are somehow related to smart pointers, yet differ in syntax and behaviour. Finally we take a look at the smart pointer concept compared to other approaches from other programming languages, first and foremost the memory management in Objective C(?).

II. Smart Pointer Foundations

What is a smart pointer in general? In short terms, a smart pointer is a class object that manages a specific resource and tries to emulate the behaviour of native raw pointers [2], [8]. In this context we consider a *raw pointer* a pointer that is directly supported by the compiler[1]. In general the object to which a pointer points to, be it a raw or smart pointer, is called *pointee*.

The goal with smart pointers is to have all the functionality of regular *raw* pointers and additional logic that can vary for the specific purpose. Some examples could be:

- garbage collection,
- reference counting,
- measuring of code,
- ownership management,
- or others

In order to emulate the behaviour of raw pointers the indirection operators * and -> are overloaded, so they can be used with normal pointer syntax. It is possible to initialise smart pointers with raw pointers and they offer various types of conversions. Most importantly and interesting for programmers are implicit type conversions, hence conversions that are done *automatically* by the compiler. With raw pointers this behaviour is part of the standard but smart pointers lack this behaviour.

Table I lists a set of implicit conversions that can be performed with smart pointers implicitly and are interesting to be implemented by a smart pointer class. Not all of these can or *should* be provided, though, which will be discussed in section VI.

TABLE I
SUMMARY OF IMPLICIT TYPE CONVERSIONS [2]

From	To	Comment
T	T&	object to reference
T&	T	reference to object
T[]	T*	array to pointer
T(args)	T(*)(args)	function to pointer
T	const T	type to const type
T	volatile T	type to volatile type
T*	const T*	pointer to pointer to const
T*	volatile T*	pointer to pointer to volatile
0	T*	NULL to pointer
Derived*	Base*	if base is accessible and derived isn't const or volatile
Derived&	Base&	if base is accessible and derived isn't const or volatile
T[]	T*	array to pointer to first element
T*	void*	if T is not const or volatile

In the following a non-complete example implementation in C++ is provided for a smart pointer that encapsulates the

[1]Even though the auto_ptr is now part of the standard, it's obviously not meant as a *raw pointer* in this context

behaviour of a raw pointer by overloading the default syntax [1]

```
template <class T>
class SmartPtr
{
public:
 explicit SmartPtr(T* pointee) :
    pointee_(pointee);

 SmartPtr& operator=(SmartPtr& other)
 {
  ...
  return *this;
 }

 ~SmartPtr();

 T& operator*() const
 {
  ...
  return *pointee_;
 }

 T* operator->() const
 {
  ...
  return pointee_;
 }

private:
 T* pointee_;
 ...
};
```

So this smart pointer is a templated class, it aggregates a pointer to a object of the generic type T in its member variable *pointee_* and offers in this version four methods:

- `operator=` is the encapsulation of an assignment operator, so another smart pointer can be assigned to this one like
  ```
  SmartPointer<Widget> sp(new Widget);
  sp = new Widget;
  ```
- `~SmartPtr()` is the destructor operator, in this case it is not defined.
- `operator*` returns a direct reference to the encapsulated pointee object
- `operator->` returns the pointee object itself

With smart pointers, three (potentially) distinct types need to be distinguished [1]:

- The *storage type*, which is the type of the object that is pointed to, hence the type of pointee_
- The *pointer type*, thus the type of the object that is returned by operator->. This does not necessarily need to be equal to the storage type, it could be just a proxy object that is returned.
- The *reference type*, thus the type returned by operator*

III. Ownership Management

A big issue with smart pointers is the topic of *Ownership Management*. As smart pointers "own" the objects to which they point to, they are responsible for them. Most importantly they are responsible for deleting them and freeing resources, once it is time. The *time* of deletion is important here, because this varies, depending on the clients[2] intention. As tracking the pointee means tracking overhead, it depends on the application which type of ownership management should be applied, to prevent unnecessary tracking operations. In the following the five most common types are presented:

A. Deep Copy [1]

This is the simplest strategy of ownership management: Whenever the smart pointer is copied, the pointee object is copied, too. Therefore there always exists distinct versions that are completely independent. As this strategy is so simple, it does not really make sense to use rather than raw pointers - in normal cases it produces only overhead without any added value to standard C++ semantics[3]. There is a use case where it makes sense, though: Deep Copy enables support for *polymorphism*. If a pointer points to an object that is derived from a base class, during the copy-procedure only the base part would be copied which is called *slicing*. A solution is to implement a virtual Clone method in all derived classes like this:

```
class AbstractBase
{
  ...
  virtual Base* Clone() = 0;
};

class Concrete : public AbstractBase
{
  ...
  virtual Base* Clone()
  {
   return new Concrete(*this);
  }
};
```

The smart pointer calls the Clone() method every time a object needs to be copied.

B. Copy on Write [1]

The *Copy on Write* strategy is more resource-efficient than *Deep Copy*. Rather than create a copy of an object every time it is accessed, in this strategy the object is shared by several pointers until an attempt is made to modify it, then it is finally copied (using a deep copy for polymorphism support). The problem is, that a smart pointer is too low level to be able to differentiate between two function calls.

Given the following example of a class which is contained in a smart pointer:

[2]Hence the class/object that uses the smart pointer to access the pointee
[3]beside a use case like performance/code measuring

```
class Bar {
 public:
  int FunctionNonConst();
  int FunctionConst() const;
}
...
SmartPtr<Bar> sp;
...
sp->FunctionNonConst();
sp->FunctionConst();
```

The `operator->` cannot distinguish between the both function calls, if they are const or not. Therefore, smart pointers are not the best place to implement this strategy but rather full-qualified classes.

C. Reference Counting [1]

The *Reference Counting* strategy is for objects that are pointed to by several smart pointers in order to determine the right moment to delete the object. There is a counter that contains the number of smart pointers that are maintaining the object. When a new smart pointer wants to use the object, the counter is increased, and when a smart pointer doesn't need the object any more, the counter is decreased. Once the last smart pointer doesn't need the object any more and the *reference counter* goes to zero, the object can safely be deleted. That is, of course, only if no *raw pointers* are used at the same time, because they have no influence to the reference counter.

[1] presents three different ways to maintain the reference counter. Figure 1 shows the first way, saving the reference counter on the heap. Here, every smart pointer needs to pointers, one to the object (pointee) and one pointer to the heap where the reference counter is saved. The drawback with this method is, that it doubles the size of each pointer because two pointers are needed.

Another way is shown in Figure 2, instead of having all smart pointers pointing to the heap, all pointers point to one intermediate pointer object that maintains a pointer to the pointee and the reference pointer as class variable. This, however, reduces the access speed as for every access to the pointee another level of indirection needs to be surpassed.

This problem could be solved with a structure like in Figure 3, the reference counter is saved in the pointee itself. This is an efficient way to implement reference counting, but this also means, that the pointee class needs to be designed to use the smart pointer. Instances of classes that do not support this way of reference counting, cannot be used with this method.

D. Reference Linking [1]

Instead of counting the number of "users" for objects, it would be more efficient and reliable to keep track of the moment the number goes down to zero, only. Therefore, every smart pointers keeps a double linked list to its next and previous element with a wraparound at both ends. This way, a smart pointer can efficiently be removed or added to the list. Once a smart pointer wants to remove itself from the list and

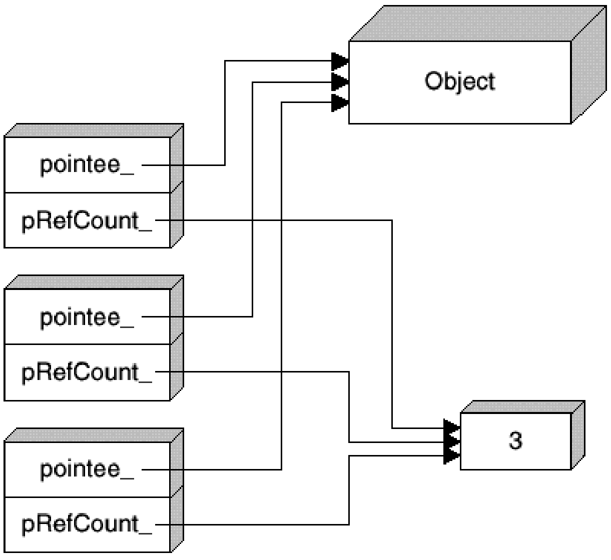

Fig. 1. Method 1: Save the reference counter on the heap [1]

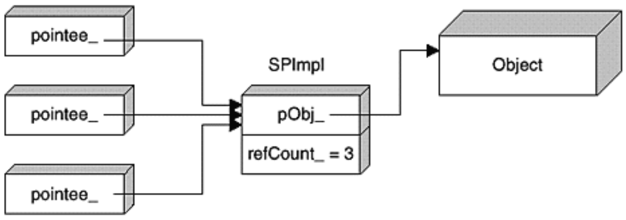

Fig. 2. Method 2: Save the reference counter at an intermediate pointer-object [1]

finds that it is its own predecessor and successor, the object can safely be deleted. Of course there are three pointers needed for this strategy instead of only one, as can be seen in Figure 4.

Reference linking is more reliable and does not need any space on the heap, however the smart pointers are bigger due to two extra pointers for the list, also adding and removing objects from the list is more time-consuming than just incrementing or decrementing an integer.

A disadvantage of both strategies is the lack of cyclic management detection. An object A with a smart pointer to B which itself holds a smart pointer to A would lead to a cyclic reference. Even though no object uses neither of A and B they use each other and therefore would not be deleted.

E. Destructive Copy [1]

This is a rather simple strategy that is also used by the standard smart pointer `std::auto_ptr`[4]. Once a smart pointer is copied, the original is destroyed, or rather set to 0.

[4]As discussed in Section V this is one of four standard pointers at the moment, even though auto_ptr is set deprecated (C++11)

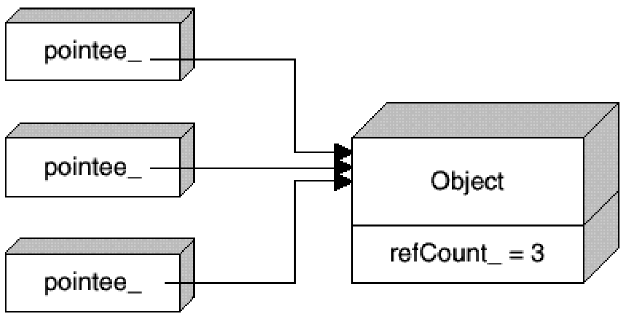

Fig. 3. Method 3: Save the reference counter at the pointee itself [1]

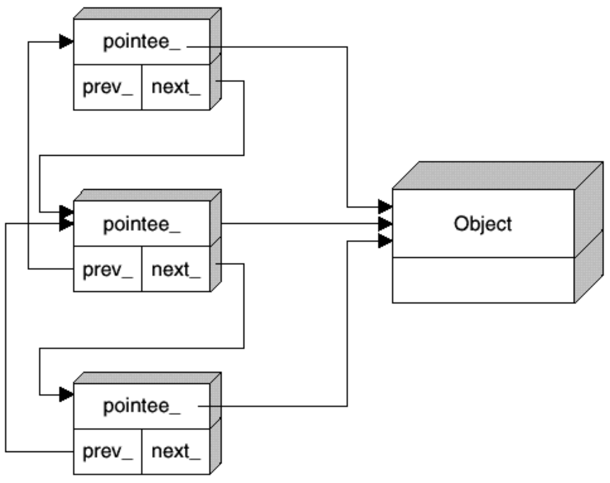

Fig. 4. Reference linking, keep all smart pointers in a doubled linked list [1]

Using this type of ownership strategy, for example by applying the standard smart pointer `auto_ptr` can lead to negative effects for the program data and correctness if not carefully used. For instance, a method that uses a smart pointer *by value* would destroy the original smart pointer:

```
int FuncByValue(SmartPtr<T> sp);
...
auto_ptr<FooBar> ap(new FooBar);
FuncByValue(ap);
```

By calling the method in this example, the value of ap would be 0 after the method returns, because a copy was created for the method call. As smart pointers with *Destructive Copy* strategies do not support value semantics, they cannot or rather *should not* be used in containers [10, Item 21], [5, Item 28], such as vectors or maps – which seems like a big drawback.

However, this strategy has some advantages, too:

- Almost no overhead
- when used as return value for functions, it is made sure that the pointee object is destroyed when the caller doesn't use it
- As it is one of the very few smart pointer implementation that the standard provides, programmers are forced to get used to its behaviour

IV. Standard Conversions

V. Common Implementations and Standard

As briefly described in Section III-E already, there are currently only three smart pointer implementations in the standard C++11, the `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. The so called `auto_ptr` was the first and for a long time the only smart pointer in the C++ standard, but in the last years it was set to deprecated and replaced with `std::unique_ptr` [6, p. 1233].

In the following all common implementations from the Boost library [3, Part 1] of smart pointers are described, at first the three smart pointers in the C++11 standard.

A. *auto_ptr and unique_ptr* (*in std::*)

In specific situations the auto_ptr can be very useful [5, Item 10], [9] while on the other hand using it with containers like vector or map can be dangerous [10, Item 21], [5, Item 28].

As described the auto_ptr applies the *Destructive Copy* from Section III-E ownership strategy which leads to leaving right-hand smart pointers empty on copying or assigning two auto_ptr instances. Therefore the *copy* semantics is wrong [7] which lead to the thought to replace it. Due to backwards compatibility reasons as the copy and assignment operators of auto_ptr are already defined, in C++11 a new smart pointer was introduced: unique_ptr.

A unique_ptr solves the problem of misuse by explicitly deleting the copy and assignment operators, instead the `std::move()` operation has to be called which then copies the smart pointer destructively.

B. *shared_ptr* (*in std::*)

shared_ptr implements the ownership management strategy *Reference Counting* from Section III-C. Different from auto_ptr or unique_ptr it can be used in containers, but just like auto_ptr [11] it also can make use of sources and sinks [9]. In other words: An ownership can also be "passed", when a shared_ptr source does not need the object any more it "frees" it by decrementing the reference counter - the sink however wants to retain the object and increments it in turn.

C. *weak_ptr* (*in std::*)

This smart pointer is often listed together with shared_ptr because it is used in conjunction with shared_ptr only. Once an object is pointed to by a shared_ptr, a weak_ptr can point to it, too. In general, the weak_ptr is much like a *raw pointer*, is does not "own" the object but only points to it - once the last shared_ptr goes out of scope, it deletes the object, regardless if and how many weak_ptrs are still pointing to it. The difference to a *raw pointer* however is, that the shared_ptr sets all weak_ptrs to null [11] - while a raw pointer still would point to a invalid location in the memory. Using a weak_ptr therefore prevents from having a dangling pointer.

D. *shared_array*

This is the same as shared_ptr, but it owns an array instead of a single object. But different from shared_ptr, it is not part of the standard.

E. *scoped_ptr and scoped_array*

These two types of smart pointers try to be the behaviour that auto_ptr might be to have been developed for [11]: To automatically delete the object once it goes out of scope. It was designed to make sure dynamically allocated objects are properly deleted and has similar characteristics to auto_ptr. The main difference is, scoped_ptr cannot be moved, copied, or assigned. Therefore it is a pointer to an object with the single purpose to delete the object after it is not used any more. scoped_array is the same again, just for arrays.

F. *intrusive_ptr*

This smart pointer is like shared_ptr applying the *Reference Counter* strategy. But instead of applying the method depicted in Figure 1 with holding the actual reference counter on the heap, the reference counter variable is part of the pointee object, as shown in Figure 3. As mentioned before, this requires the class that is to be managed to support intrusive reference counting.

VI. PROBLEMS (IMPLICIT CONVERSION ET AL.)

- pointer leakage, accessors solve this problem
- implicit conversions with smart pointers:
  - user-defined conversions can't be implicitly chained together – no indirect base conversion
  - ambigious calls if indirect conversions are user-defined (hierarchy C –> B –> A, conversion in C for B and A)
  - try to support preference to convert to direct base over a indirect base
  - supporting const
  - support for multiple inheritance

VII. OAUTH ACCESSORS

Accessors in OAUTH are described to be an alternative to smart pointers - the main difference is that while smart pointers overload the operators and ideally the object maintained by a smart pointer can be accessed as if it was just a smart pointer, accessors duplicate all public member methods of the application class. Listing VII [2] shows an example of an accessor implementation and usage:

```
// application class
// target of the accessor
class Targetclass {
 public:
  int foo;
  Targetclass(int init) : foo(init) { }
  void set(int foobar) { foo = foobar; }
  int get() { return foo; }
 ...
};

// accessor for Targetclass
class TargetclassA {
 private:
  Targetclass * ptr;
 public:
  TargetclassA() : ptr(0) { }

  void make(int init) {
   ptr = new Targetclass(i);
  }
  void set(int i) {
   ptr->set(i);
  }
  int get() {
   return ptr->get();
  }
 ...
};

TargetclassA tca = new TargetclassA;
tca::make(5);
tca::set(4);
int foovar = 5;
foovar = tca::get(); // foovar set to 4
```

So from the users point of view an accessor does the same as an smart pointer, but instead of using $->$ to access the pointee functionality, :: is used. From the technical point of view, an accessor are much more complex and more difficult to declare, because *every* member function needs to be wrapped by an accessor class. Therefore it is hard to template accessors and reuse them with other classes.

The big advantage of accessors compared to smart pointers is the fact, that with accessors the pointee is not leaked as described in the previous section. It is not possible to directly manipulate the pointee, therefore accessors are more safe and reliable.

VIII. ONGOING RESEARCH AND OTHER LANGUAGES

A. *Objective C [4, Chapter 17]*

- In Objective C smart pointers are part of the language, every variable is pointed to by a smart pointer using the reference counting strategy to automatically releasing it.
- Autorelease pool concept

- What about Java?

IX. SUMMARY AND OUTLOOK

REFERENCES

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Professional, 2001. II, II, III-A, III-B, III-C, III-D, 1, 2, III-E, 3, 4

[2] Daniel R. Edelson. *Smart Pointers: They're Smart, but They're Not Pointers.* Proceedings of the 1992 Usenix C++ Conference, 1992. II, I, VII

[3] Bjoern Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost.* Addision Wesley Professional, 2005. V

[4] Steven G. Kochan. *Programming in Objective C 2.0*. Addision-Wesley, 2009. VIII-A

[5] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996. III-E, V-A

[6] open std.org. Working Draft, Standard for Programming Language C++ N3092, Accessed on 28 May 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf, 2010. V

[7] Software Engineering Institute CERT. Don't use auto_ptr where copy semantics might be expected, Accessed on 28 May 2012. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=28180606, 2011. V-A

[8] Bjarne Stroustrup. *The C++ Programming Language, second edition*. Addision-Wesley, 1991. II

[9] Herb Sutter. *Using auto_ptr Effectively*. C/C++ Users Journal, 1999. V-A, V-B

[10] Herb Sutter. *More Exceptional C++*. Addison-Wesley, 2002. III-E, V-A

[11] Herb Sutter. The New C++: Smart(er) Pointers, Accessed on 28 May 2012. http://www.drdobbs.com/184403837, 2002. V-B, V-C, V-E