Smart Pointers: They're Smart, but They're Not Pointers

Daniel R. Edelson*

UCSC–CRL–92–27
10 June 1992

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

*This research was performed while the author was a visiting researcher at *INRIA Project SOR, BP 105 Rocquencourt, 78153 Le Chesnay CEDEX, FRANCE.*

This work was supported in part by Esprit project 5279 *Harness.*

This paper originally appeared in *Proc. of the 1992 Usenix C++ Conference*, Usenix Association, 1992, pp 1–19.

1. Introduction

The ability to substitute user-defined code for pointers is a very powerful programming mechanism. It facilitates using C++ in domains for which the language is not specialized. For example, smart pointers [Str87] or variations thereof can be used to support distributed systems [SDP92, SMC92], persistent object systems [MIKC92, SGH$^+$89, Str91, pg. 244], to provide reference counting (e.g. the *ObjectStars* of [MIKC92] or the *counted pointers* idiom of [Cop92]) or garbage collection [Ken92, Ede92].

In this paradigm, a smart pointer encapsulates some kind of raw pointer or complex handle. The smart pointer overloads the indirection operators in order to be usable with normal pointer syntax. For example, code that accesses both transient and persistent objects can be written to perform its manipulations through smart pointers. These pointers would be able to refer to either normal transient objects, or to objects that reside in persistent storage. When an object in persistent storage is referenced through the smart pointer, a copy is loaded into memory. The smart pointers should even be able to enforce a consistency protocol if the object is replicated or loaded into shared memory.

In analyzing how effective a pointer substitute is, we consider two criteria: (1) how run-time efficient it is, and (2), how it impacts the code in terms of programming style. Smart pointers with a lot of functionality could be quite inefficient; it is also possible to write very lightweight smart pointers. We do not concentrate on run-time efficiency because that is entirely determined by the specific implementation. Rather, we focus on the second issue: how the use of smart pointers impacts the client code.

This paper shows how the behavior of smart pointers diverges from that of raw pointers in certain common C++ constructs. Given this, we conclude that the C++ programming language does not support seamless smart pointers: smart pointers cannot transparently replace raw pointers in all ways except declaration syntax. We show that this conclusion also applies to *accessors* [Ken92].

The organization of this paper is as follows: Section 1 very briefly summarizes the behavior of raw pointers that smart pointers try to emulate, particularly in terms of the standard type conversions. Then, Sect. 2 presents several ways of implementing smart pointers, and for each, shows limitations and problems with it. Section 3 shows why these results apply equally to accessors, after which the last section concludes the paper.

Raw Pointer Behavior

In order to evaluate the effectiveness of a pointer substitute, it is necessary to have a baseline for comparison. That baseline is, of course, the *raw pointer*.[1] The semantics of raw pointers are too complex to list exhaustively. The most important aspect of their behavior for this discussion is how they undergo implicit type conversions. The problem is to design user-defined pointers that will behave nearly the same as raw pointers, in terms of implicit type conversions, in all interesting cases.

Table 1.1 summarizes the conversions that take place on function arguments and in expressions such as assignment. All of these type conversions may be performed *implicitly* by the compiler. We are not interested in *explicit* type coercions.

[1]We use *raw pointer* to mean the pointer type that is directly supported by the compiler.

Table 1.1: Summary of implicit type conversions

The conversion classes are listed in order of precedence. The conversions within a group are of approximately the same precedence.

Class 0: Trivial Conversions

	From	**To**	**Notes**
1.	T	T&	*object* ⇒ *reference*
2.	T&	T	*reference* ⇒ *object*
3.	T[]	T∗	*array* ⇒ *pointer*
4.	T(args)	T(∗)(args)	*function* ⇒ *pointer*
5.	T	const T	*type* ⇒ *const type*
6.	T	volatile T	*type* ⇒ *volatile type*
7.	T∗	const T∗	*pointer* ⇒ *pointer to const*
8.	T∗	volatile T∗	*pointer* ⇒ *pointer to volatile*

Class 1: Standard Conversions

	From	**To**	**Notes**
9.	0	T∗	*the* NULL *pointer conversion*
10.	Derived∗	Base∗	*if* base *is accessible and* derived *isn't* const *or* volatile
11.	Derived&	Base&	*if* base *is accessible and* derived *isn't* const *or* volatile
12.	T[]	T∗	*array* ⇒ *pointer to first element*
13.	T(args)	T(∗)(args)	*except following* & *or before* ()
14.	T∗	void∗	*provided* T *is not const or volatile*
15.	T(∗)(args)	void∗	*provided sufficient bits are available*[ANS93, §4.6, line 6]

Class 2: User-defined Conversions

16.	*conversion by constructor*
17.	*conversion by conversion operator*

2. Smart Pointers

Smart pointers are class objects that behave like raw pointers [Str87, Str91]. The smart pointers overload the indirection operators (∗ and ->) in order to be usable with normal pointer syntax. They have constructors that permit them to be initialized with raw pointers such as `new` returns. Smart pointers may supply a conversion to `void*` in order to be usable directly used in control statements, e.g. `if (ptr)` and `while (ptr)`. The conversion to `void*` may also be seen as undesirable [Gau92], in which case all testing is explicit using overloaded comparison operators. Smart pointers may optionally supply a conversion to the corresponding raw pointer types.

Our goal in manipulating smart pointers is to have all the functionality of regular pointers *and then some.* For example, the 'and then some' might be:

- tracing garbage collection [Ede92],
- reference counting [Ken92, Mae92, MIKC92, Cop92],
- convenient access to persistent objects [SGH$^+$89, Str91, HM90, SGM89, MIKC92],
- uniform access to distributed objects [SDP92, Gro92, SMC92],
- instrumenting (measuring) the code,
- or others.

To accomplish this, the smart pointers should look and feel, to the greatest extent possible, like raw pointers. Achieving the ideal, i.e. making the smart pointer semantics a superset of raw pointer semantics, is impossible (as we will show). The next best thing is to see how close the code can come to making the smart pointers perfect substitutes for raw pointers in all ways except declaration syntax.

Raw pointers support numerous conversions, for example, conversion of `T*` to `void*`, of `T*` to `const T*`, and of `derived*` to `base*`. There are two ways to define smart pointers that can allow them to emulate these conversions:

1. the smart pointer classes can use user-defined type conversions to emulate the standard conversions, or,
2. the smart pointer classes can be related in an inheritance hierarchy.

In this paper we will consider both these possibilities, sub-possibilities of each, and combinations thereof.

2.1 Supporting Class Hierarchies

Pointers in a class hierarchy undergo a very important set of conversions. In particular a derived class pointer can be implicitly converted to a base class pointer for an accessible base class. This conversion, along with `virtual` functions, is how C++ supports polymorphism. To be general, the smart pointer classes must emulate this type conversion.

2.1.1 User-Defined Conversions

First we consider the case where the smart pointer classes do not have any subclass relations, even though the referenced classes may derive from each other. In this case, the standard conversions of raw pointers must be emulated with user-defined conversions. In

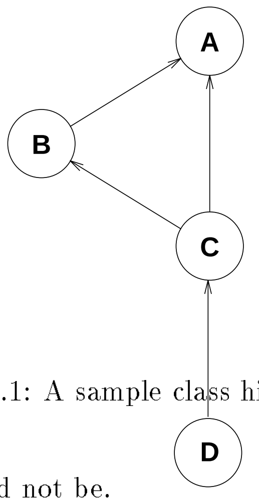


Figure 2.1: A sample class hierarchy

This hierarchy is rooted, but it need not be.
Sections 2.1.1 through 2.1.3 require that class D be in the hierarchy. However, figures later in the paper will only include classes A, B and C.

---

particular, we are concerned with line 10 in Table 1.1: the derived class pointer to base class pointer conversion.

Let us assume that the class hierarchy of user objects is as shown in Fig. 2.1. There are four client classes: A, B, C, and D. Since there are four client classes we also require four smart pointer classes. We call the pointer classes Pa, Pb, Pc, and Pd. With standard conversions and raw pointers, the following implicit conversions are available:

$$\begin{array}{ll} \mathsf{B}* \Rightarrow \mathsf{A}* & \mathsf{C}* \Rightarrow \mathsf{A}* \\ \mathsf{D}* \Rightarrow \mathsf{A}* & \mathsf{D}* \Rightarrow \mathsf{B}* \\ \mathsf{C}* \Rightarrow \mathsf{B}* & \mathsf{D}* \Rightarrow \mathsf{C}* \end{array}$$

The goal is to implement these same conversions among the smart pointer classes. Using user-defined conversions, there are two possibilities:

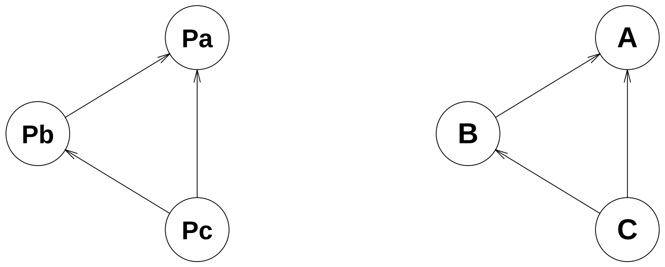
1. every smart pointer class provides a user-defined conversion to the smart pointer types that correspond to its referent type's direct bases, or,
2. every smart pointer class provides a user-defined conversion corresponding to every base class, whether direct or indirect.

2.1.2 Conversion to Direct Bases

Suppose every smart pointer class supplies a user-defined conversion to the smart pointer classes for direct base classes of the referent type. In our current example, this would provide the following user-defined conversions:

$$\begin{array}{ll} \mathsf{Pb} \Rightarrow \mathsf{Pa} & \mathsf{Pc} \Rightarrow \mathsf{Pb} \\ \mathsf{Pc} \Rightarrow \mathsf{Pa} & \mathsf{Pd} \Rightarrow \mathsf{Pc} \end{array}$$

Under this scheme, there is no implicit conversion from Pd to Pa. This is because user-defined conversions can't be implicitly chained together. By contrast, with raw pointers the corresponding conversion is available. The failure to support conversion to an indirect base pointer is a substantial shortcoming of this implementation.



A, B, C:	User classes
Pa, Pb, Pc:	Smart pointer classes for A, B, and C
$B \longrightarrow A$	Public virtual derivation of B from A

Figure 2.2: A pointer hierarchy for an object hierarchy

2.1.3 Conversion to All Bases

Instead of supplying user-defined conversions only to direct bases, we can instead provide conversions to all bases, direct and indirect. This scheme requires the following user-defined conversions:

$$\begin{array}{ll}
\mathsf{Pb} \Rightarrow \mathsf{Pa} & \mathsf{Pc} \Rightarrow \mathsf{Pb} \\
\mathsf{Pc} \Rightarrow \mathsf{Pa} & \mathsf{Pd} \Rightarrow \mathsf{Pa} \\
\mathsf{Pd} \Rightarrow \mathsf{Pb} & \mathsf{Pd} \Rightarrow \mathsf{Pc}
\end{array}$$

This supplies the conversion from Pd to Pa that was missing from the previous implementation. However, consider the following code:

```
void f(Pa);
void f(Pc);

int main(void) {
  Pd pd = new D;
  f(pd);
  return 0;
}
```

The call to f() is ambiguous. There are conversions to match both of the overloaded functions and there is no way to choose between them. In contrast, the equivalent code with raw pointers is unambiguous because, with raw pointers, conversion to a direct base is preferred over conversion to an indirect base, thus, f(C∗) would be called. This problem is less severe than the problem identified in §2.1.2; this is a more viable implementation.

2.2 Inheritance Hierarchy

In the previous section, we discussed emulating the standard pointer conversions with user-defined conversions. It is also possible to emulate them using the standard reference conversions [Ken92]. We arrange the smart pointer classes in a class hierarchy that parallels the object hierarchy. Figure 2.2 illustrates this.

Since class Pc derives from Pb, any instance of Pc can be converted to an instance of Pb through the standard Derived& to Base& conversion. This reference conversion from Pc to Pb can thus be used to emulate the corresponding standard pointer conversion from C∗ to B∗. The reference conversion has the same precedence as the pointer conversion, and also favors conversion to a direct base class over conversion to an indirect base class.

This scheme emulates the usual base class/derived class pointer conversions as follows. Assume that an instance of Pc (as shown in Fig. 2.2) must be converted to an instance of Pb, perhaps to initialize a temporary or to match a function parameter. Since class Pc is derived from class Pb, an instance of Pc contains an instance of Pb as a subobject. The standard conversion (Table 1.1, line 11) converts the Pc object to a Pb object by using the Pb subobject in place of the complete object. No user-defined code needs to be or may be provided to perform this conversion. The conversion simply changes the 'logical' address of the object from the beginning of the object to the beginning of the Pb subobject.

In an inheritance hierarchy of smart pointers, there is a choice to be made: What class defines the pointer instance data? Every class could potentially declare a pointer data member. Alternatively, either the root of the hierarchy or some other class can provide the data.

2.2.1 Replicated Data

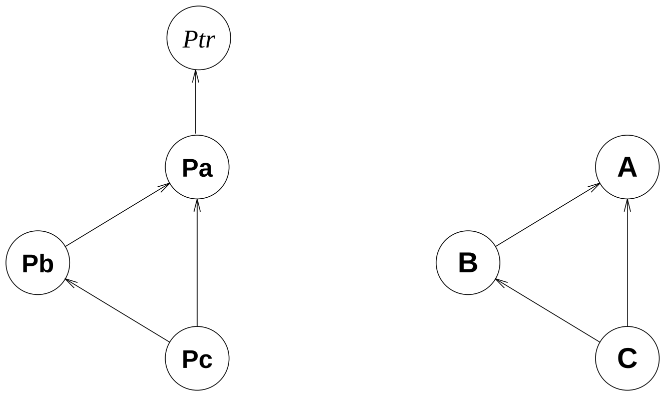
It is plausible for every smart pointer class in the smart pointer class hierarchy to define a new data member. Any derived class smart pointer then contains one pointer member added by the derived class, plus one pointer member for every direct or indirect base class. For example, suppose that Pb is a subclass of Pa, then a Pb contains a B∗ and the Pa subobject contains an A∗.

A derived class smart pointer contains a subobject for each of its base classes; converting a derived class smart pointer to a base type uses the corresponding base class subobject in place of the complete object. After a conversion, the overloaded operators (such as indirection) use the base class pointer member rather than the derived class pointer member. To correctly emulate raw pointers, these base class pointers must all point into the same object as the main derived pointer, which is also called the *most derived* pointer. Therefore, assigning to a smart pointer under this implementation must update all of the component pointers. Failure to do this results in a derived class smart pointer that cannot be correctly converted to a base class smart pointer.

This implementation does not require any explicit type conversions, and emulates the standard pointer conversions well: conversion of a smart pointer to a base class smart pointer favors conversion to a direct base over conversion to an indirect base; this eliminates the problem discussed in §2.1.3 in which a choice between converting to a direct base or an indirect base is ambiguous. It also works correctly in the presense of multiple inheritance. However, it's inefficient because updating a derived class smart pointer requires an operation per base class. In addition, this scheme permits an incorrect type conversion. The alternative described in the following subsection suffers from the same error, so we defer the discussion until §2.2.3.

2.2.2 Nonreplicated Data

To improve the efficiency of the previous organization, we make every smart pointer contain exactly one pointer as its instance data. This is done by defining an abstract



A, B, C:	User classes
Pa, Pb, Pc:	Smart pointer classes for A, B, and C
B $\longrightarrow$ A	Public virtual derivation of B from A
Ptr:	Base class to supply the pointer datum

Figure 2.3: A smart pointer hierarchy with an abstract base to supply the data

---

virtual base class that supplies the pointer datum; call this class *Ptr*. (Smart pointer classes that are indirectly derived from Ptr need not also be directly derived from it.) This way, each smart pointer class contains only one instance pointer. It also contains invisible pointers that implement the virtual derivation, but these pointers don't get modified during assignment. Figure 2.3 demonstrates this organization.

Since they use a virtual base class, under most C++ implementations, these smart pointers will have size larger than one word. Nonetheless, in contrast with the previous solution, assigning to one of these smart pointers only requires one indirect memory reference.

This organization supports conversion to a direct or indirect base class, and conversion to a direct base is preferred. Conversion to a base pointer is also preferred over conversion to void*. However, these smart pointers do not work with multiple inheritance.

Under multiple inheritance (and some implementations of single inheritance) a base class subobject may have a nonzero offset within a derived class object. With raw pointers, converting a derived pointer to a base pointer for such a base class adds the correct offset to the value of the pointer; this redirects the pointer from the beginning of the main object to the beginning of the base class subobject. For example, in Fig. 2.3, an object of class C contains a subobject of class B whose offset is probably nonzero. Converting a C* to a B* redirects the pointer from the beginning of the C object to the beginning of the B subobject by adding a positive offset to the pointer.

The corresponding conversion is performed on these smart pointers using the standard derived& to base& conversion shown on Line 11 of Table 1.1. The conversion causes the base smart pointer subobject to be used in place of the derived smart pointer object. This does not add the requisite offset to the value of the pointer. Instead, it simply reinterprets the same pointer value as a pointer of the base class type. Thus, these smart pointers cannot be converted to base class smart pointers for subobjects with nonzero offsets.

```
class BASE { ... };
class DER1 : public BASE { ... };
class DER2 : public BASE { ... };

void f(BASE** p1, BASE** p2) { *p1 = *p2; }

int main(void)
{
    DER1 * d1 = new DER1;
    DER2 * d2 = new DER2;

    f(&d1,&d2); // Illegal, but what if?
    return 0;
}
```

Figure 2.4: Why a derived∗∗ may not be converted to a base∗∗

If a derived∗∗ could be converted to a base∗∗, then this code would assign a DER1∗ to a DER2∗. However, there is no relationship between classes DER1 and DER2 that would justify such an assignment.

Expressed differently, the problem is that the operation derived::operator base&() cannot be overloaded. This is a built-in standard conversion that causes the compiler to substitute the base subobject in place of the derived object. If this operator could be overloaded such that it altered the value of the pointer, then the error could be avoided. Note, the current language definition does not explicitly forbid overloading this operator, nor does it explicitly permit it [ANS93], however, it seems inevitable that overloading this operator will eventually be prohibited.

2.2.3 Another Error with Pointer Hierarchies

There is one other error that the schemes presented in the last two subsections both share: they both permit an incorrect, implicit type conversion.

Every C++ programmer is familiar with the conversion from Derived∗ to Base∗. However, the conversion from Derived∗∗ to Base∗∗ is prohibited because it introduces a gaping hole in the otherwise (mostly) safe type system. Specifically, given two objects whose classes are different but have a common base, this conversion allows you to incorrectly compare or assign pointers to these objects [Sal92]. Figure 2.4 provides a example of how this conversion allows assignment between two incompatible pointer types.

With a class hierarchy of smart pointers, this conversion is not just between Derived∗∗ and Base∗∗; it is also between Derived∗ and Base∗ because the smart pointer classes are related through inheritance. The compiler permits the conversion because it uses the standard base class pointer conversion listed on Line 10 of Table 1.1. Figure 2.5 shows the same incorrect code using smart pointers. The difference is that the code using smart pointers compiles without error and crashes at runtime.

To show that this error also occurs with accessors, the code of Figure 2.6, written using OATH accessors and library classes, encounters this bug and dies with a segmentation

```
void f(PtrBASE* p1, PtrBASE* p2) { *p1 = *p2; }

int main(void)
{
   PtrDER1 d1 = new DER1;
   PtrDER2 d2 = new DER2;

   // Legal and wrong with a pointer hierarchy
   f(&d1,&d2);
   return 0;
}
```

Figure 2.5: The invalid conversion with smart pointers

Since the smart pointer classes are related through inheritance, the compiler permits the type conversion, even though this results in an assignment between incompatible types.

---

violation. This error exists because the pointer hierarchy provides the incorrect conversion of Derived** to Base**. (For those readers not acquainted with OATH accessors, there is a discussion of the differences between them and smart pointers in §3.)

```
#include <iostream.h>
#include "oath/minString.h"

void f(objA & a, objA & b) { a = b; }

int main(void)
{
    characterA ch = characterA::make('A');
    stringA str = minStringA::make();

    str << "hello\n";
    cout << str;
    f(str,ch);      // incompatible assignment
    cout << str;    // This causes a core dump.
    return 0;
}
```

Figure 2.6: How to misuse the conversion that smart pointer hierarchies permit

This example uses OATH accessors, which are discussed in §3.

2.2.4 Class Hierarchies Summary

We have presented four ways of organizing smart pointers to support class hierarchies. These ways include two that depend on user-defined conversions and two that use a parallel class hierarchy.

With user-defined conversions, it's best to supply conversions to both direct and indirect base classes. Given that, the problem is that the compiler can't choose between converting to a direct base and converting to an indirect base, nor between converting to a base class and converting to void*. Consequently, certain overloaded function invocations are ambiguous, whereas they are legal using raw pointers.

The alternative to user-defined conversions is to use a parallel class hierarchy of smart pointers. This uses standard reference conversions to convert a derived class smart pointer to a base class smart pointer. It is inefficient to replicate the pointer data in each class, so an abstract base class is used to supply a void* instance datum. However, this scheme does not support multiple inheritance, and it permits an incorrect pointer conversion.

Of the possibilities discussed, we suggest using user-defined type conversions to direct and indirect base classes. The programmer may need to disambiguate some overloaded function calls that would be legal using raw pointers.

2.3 Supporting const

Supporting the base class conversions is one problem. Supporting the conversion of T* to const T* is equally or more important because of the major role that const plays in documenting and structuring C++ programs.

Using raw pointers, there are two ways to modify a pointer declaration using const:

1. const T*	*The referent is* const.
2. T* const	*The pointer is* const.

These uses of const are not mutually exclusive, thus, const T* const is the type of a pointer for which both the referent and the value are const.

With smart pointers, on the other hand, const only can be used one way: const PtrT ptr;. This does not declare a smart pointer to a const object. Rather, this declares a smart pointer whose value may not change. The reader may argue that this discussion does not apply given templates because with templates we can declare both Ptr<T> and Ptr<const T>. However, these are two distinct types. This is the same as hand coding two classes: Ptr_T and Ptr_const_T. Being defined from the same template does not give the two classes any special relationship. In particular, there is no implicit type conversion from Ptr<T> to Ptr<const T>.

For this reason, one class of smart pointer cannot reference both const and mutable objects; instead, we need two smart-pointer classes. Let PtrT be the smart pointer class that replaces pointers of type T*, and let CPtrT be the smart pointer class that replaces pointers of type const T*. An overloaded indirection operator of CPtrT returns a const object; this allows the compiler to complain about attempts to modify an object through a CPtrT. For these smart pointers to resemble raw pointers, there must be a conversion from PtrT to CPtrT.

The conversion from PtrT to CPtrT can be implemented two ways: either there can be a user-defined conversion between them, or PtrT can be a derived class of CPtrT. The use of the user-defined conversion is self-explanatory. If the one is a derived class of the other, then the standard reference conversion can be used in place of the normal standard pointer conversion, as we have described previously.

Assume that the conversion between the two smart pointer classes is user-defined. Here are two classes of code that are affected:

1. The following works fine with raw pointers, but when the conversion from PtrT to CPtrT is user-defined, the code is illegal because it requires two user-defined conversions.
```
struct S {
    S(CPtrT);
};

S func(PtrT p) { return p; }
```

2. If a function is overloaded on types void∗ and CPtrT, it cannot be invoked with a PtrT because the call would be ambiguous. With raw pointers, the call would favor conversion to const T∗ over conversion to void∗.

A better way to implement the const pointer conversion is to make the class PtrT a derived class of CPtrT through public non-virtual derivation. They can share the same pointer data member so that instances of each class occupy only one word of storage. Figure 2.7 presents the basic structure of this organization. This uses a standard reference conversion to emulate the standard pointer conversion. The difference will be unnoticeable for most programs, except for the declaration syntax.

```
// smart pointer class to replace 'const T *'
class CPtrT {
protected:
  union {
    T * ptr;
    const T * cptr;
  } value;
public:
  ...
};

// smart pointer class to replace 'T *'
class PtrT : public CPtrT {
public:
  ...
};
```

Figure 2.7: A smart pointer hierarchy for const

2.4 Overall

We have identified 7 properties that a smart pointer organization should provide. They are (with keywords for future reference):

- **dir** implicit conversion to a direct base pointer;
- **indir** implicit conversion to an indirect base pointer;
- **prefer** a preference for converting to a direct base over an indirect base;
- **mult** support for multiple inheritance;
- **safe** no conversion from derived∗∗ to base∗∗;
- **const** the ability to reference normal and const objects, with compiler enforcement of the const attribute, and a conversion from non-const to const;
- **fast** the organization should be intrinsically efficient.

Table 2.1: Strengths and weaknesses of these methods

Method	**dir**	**indir**	**prefer**	**mult**	**safe**	**const**	**fast**
userdef direct (§2.1.2)		−	−	+	+		+
userdef all (§2.1.3)			−	+	+		+
hier replicated (§2.2.1)	+	+	+	+	−	+	−
hier abstract (§2.2.2)	+	+	+	−	−	+	+
recommended hybrid (§2.4)			−	+	+	+	+
OATH accessors (§3)	+	+	+	−	−	−	+

+	good behavior
[ ]	a user-defined conversion replaces a standard one
−	incorrect behavior

In general, any type conversion among the smart pointers should be the same precedence as the conversion to which it corresponds among raw pointers. For example, the conversion from derived∗ to base∗ is Class 1 (a *standard* conversion), as shown in Table 1.1. Therefore, it would be best for the corresponding conversion among smart pointers also to be Class 1. If this is done, the smart pointers closely resemble raw pointers in terms of overloaded function resolution and implicit conversions. Table 2.1 shows how well each organization that we've presented satisfies these goals.

As shown in Table 2.1, a class hierarchy of smart pointers emulates the derived class/base class conversion and the const pointer conversion well. However, it only supports inheritance when all subobjects have offset zero, and thus it fails to support multiple inheritance. In addition, it introduces the erroneous derived∗∗ to base∗∗ conversion. Therefore, a class hierarchy of smart pointers is good for implementing the const conversion, but not for implementing the base class conversions.

By contrast, user-defined conversions are less desirable in all cases because they replace a standard or trivial conversion with a user-defined conversion; this difference is noticeable in terms of overloaded function resolution and chaining of type conversions. In spite of that disadvantage, however, user-defined conversions allow the smart pointers to support the base class/derived class conversion, even under multiple inheritance, and don't permit the erroneous conversion.

These two observations lead to our recommended overall organization. We suggest using user-defined conversions to emulate the base class/derived class conversions because this is safe and correct. Simultaneously, the smart pointers should use a smart pointer inheritance hierarchy to emulate the const conversions.

A diagram of this organization is shown in Fig. 2.8. This shows an application class hierarchy and the corresponding smart pointer classes, including both the smart pointer classes for regular objects, and those for const pointers. For each of the application's classes there are two smart pointer classes, one that references mutable objects and one that references const objects. The smart pointer class that references mutable objects is a derived class of the one that references const objects. This supplies a standard conversion from *pointer to mutable* to *pointer to const*. In addition, the smart pointer classes for distinct application classes are related through user-defined type conversions. If class B is a

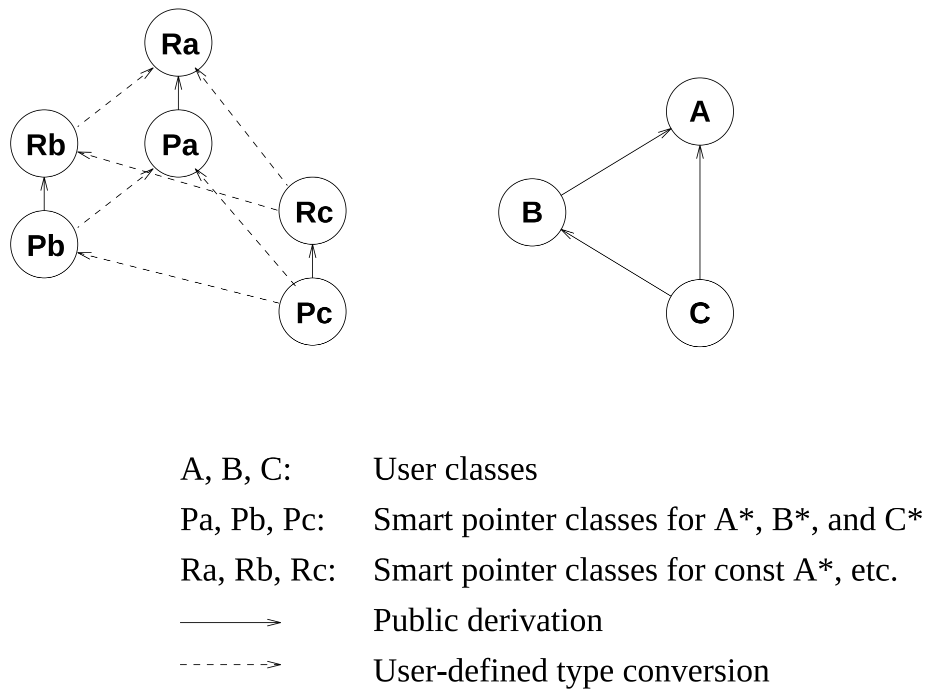

Ra
Rb
Pa
Pb
Rc
Pc
A
B
C
A, B, C: User classes
Pa, Pb, Pc: Smart pointer classes for A*, B*, and C*
Ra, Rb, Rc: Smart pointer classes for const A*, etc.
Public derivation
User-defined type conversion

Figure 2.8: The final smart pointer organization for the indicated object classes.

---

derived class of A, then Pb provides a user-defined type conversion to Pa, and CPb provides a user-defined type conversion to CPa. (CPb is the smart pointer class for const Bs.)

The use of user-defined conversions between distinct types Ptr$X$ and Ptr$Y$ supports multiple inheritance and avoids the erroneous conversion. The classes Ptr$X$ and CPtr$X$ are related by inheritance because it gives better behavior without allowing false conversions; the compiler can correctly enforce the const attribute of a referent of CPtr$X$.

2.4.1 A Unrooted Hierarchy

While we have only discussed using the smart pointers in a class hierarchy with a unique root, this does not make any difference in the implementation that has been suggested. Any type conversion that is legal among raw pointers can be implemented by the smart pointers by encapsulating the raw pointer conversion within a user-defined type conversion. Of course, as we have mentioned, whenever a user-defined conversion replaces a built-in conversion, some cases of overloading and chaining of conversions do not behave as desired.

2.5 Other Weaknesses

2.5.1 Pointers to volatile Objects

This paper has discussed const, but not volatile. Pointers to volatile objects must be supported in exactly the same way as pointers to const objects. In particular, for a single application class, distinct smart pointer classes are required to reference:

1. normal objects
2. const objects

Table 2.2: Some ways in which our smart pointers don't behave like raw pointers.

Case	**Raw Pointers**	**Smart Pointers**
Convert either to *pointer to direct base* or to *pointer to indirect base*	Convert to direct base	Ambiguous
Convert either to *pointer to base* or to void*	Convert to base	Ambiguous
Chain conversion to *pointer to base* with another user-defined type conversion	Legal	Illegal

3. volatile objects
4. const volatile objects

This plethora of classes adds a certain amount of notational complexity to the program.

2.5.2 Conversion Precedence

The proposed organization appears to be the best of the ones that have been considered because it is both safe and efficient. However, it emulates the standard derived* to base* conversions with user-defined type conversions. User-defined type conversions have lower precedence than the standard conversions. Therefore, there are many situations, primarily involving function overloading, in which these smart pointers do not behave the same as the corresponding raw pointers. Table 2.2 lists some of the cases in which these smart pointers behave differently than raw pointers.

2.5.3 Pointer Leakage

It is essentially impossible to prevent smart pointers from leaking raw pointers to the application (e.g. this pointers). In some cases, it is desirable to prevent this. For example, if smart pointers are used to implement copying garbage collection, then after a garbage collection, all dynamically allocated objects have been moved and any raw pointer no longer has the correct value.

As another example, [Ken92] discusses why the problem of raw pointer leakage makes smart pointers unsafe for reference counting. The basic idea is that the application can obtain reference counted pointer as a temporary expression, perhaps as the return value from a function. The application may then dereference the reference counted pointer by invoking the overloaded operator ->, which returns a raw pointer, which will in turn be dereferenced. Once the raw pointer is returned from the overloaded operator ->, the reference counted pointer has served its purpose and may be destroyed. However, destroying the reference counted pointer decrements the object's reference count and may cause the object to be deallocated. If the object is deallocated, then the raw pointer, which is about to be dereferenced, is a dangling reference.

In other cases, it is not critical that the application be prevented from obtaining raw pointers. For example, mark-and-sweep garbage collectors can normally tolerate the existence of raw pointers, provided the raw pointers point at objects that are *also* referenced by smart pointers [Ede92].

Smart pointers leak raw pointers because of the definition in C++ of the overloaded indirect member access operator, ->. When the compiler sees an expression of the form X->Y, where X is an expression of class type, the compiler evaluates X.operator->(). The language definition requires that this operator return a raw pointer.[1] This is a potential problem because if the smart pointer was a temporary object, the compiler may destroy it as soon as the raw pointer is obtained. However, as shown for the case of reference counting, for example, destroying the smart pointer may cause the raw pointer to become a dangling reference. This is the main problem that accessors solve.

[1]These operators may be chained together, but must eventually return a raw pointer.

3. Accessors

Kennedy describes accessors in OATH [Ken92] as an alternative to smart pointers. The central difference between accessors and smart pointers is that accessors don't overload the indirection operators; instead, like stubs [DMS92], they duplicate all the public member functions of the referent object and forward those calls through a pointer to the object. Accessors are somewhere in between smart pointers and *smart references*, because they implement pointer semantics, but use '.' rather than '->' to access the underlying object. Figure 3.1 gives the general idea behind how accessors work. This figure does not attempt to reproduce all the functionality described in [Ken92], instead, it just shows the relation between the application class and the accessor class.

Accessors are clearly superior to smart pointers because they prevent raw pointer leakage. However, they are difficult to declare because every member function of the application class must also be declared in the accessor class. Macros can abbreviate this, but the code looks significantly different from standard C++ class definitions and complex macros can hinder debugging.

```
// A sample application class.
class Thing {
friend class ThingA;
private:
   int value;
   Thing(int initial) : value(initial) { }
   void set(int val) { value = val; }
   int  get()        { return value; }
   ...
};

// A class for accessing Things.
class ThingA {
private:
   Thing * ptr;
public:
   ThingA() : ptr(0) { }
   void make(int i) { ptr = new Thing(i); }

   void set(int i)  { ptr->set(i); }
   int  get()       { return ptr->get(); }
   ...
};
```

Figure 3.1: An object class and an *accessor*-type reference class

The accessor class contains a raw pointer as its instance datum. All of the client class' member functions are duplicated in the accessor class and accessed with '.'. Therefore, the accessor class does not need to overload the indirection operators.

The accessors in OATH are organized into a class hierarchy that parallels the data object hierarchy. The reference conversions are used to convert one accessor class into a different one. The class hierarchy is rooted in the class oathCoreA; it is this class that supplies the pointer data member. This organization was discussed in Sect. 2.2.2. (Indeed, it was OATH that led us to consider this organization.)

The OATH class hierarchy uses only single inheritance; the class hierarchy, therefore, forms a tree. If it used multiple inheritance, then its implementation would suffer from the incorrect offset problem described in 2.2.2. In particular, for a pointer conversion that changes the value of the pointer, the corresponding reference conversion is incorrect because it changes the type of the accessor without changing the value of the pointer. Even using only single inheritance, this scheme permits the incorrect type conversion of derived** to base** that we discuss in 2.2.3 (see Fig. 2.6). Finally, the hierarchy of OATH uses a single accessor class per object class; therefore, it is unable to represent pointers to const objects (§2.3).

Accessors suffer from the same problems, with respect to type conversions, as smart pointers. However, the accessor model is safer than the smart pointer model. By not overloading ->, accessors avoid leaking raw pointers in a way that may result in dangling references if the compiler is aggressive in destroying temporary objects.

4. Conclusion

Pointer substitutes, whether smart pointers or accessors, are a powerful programming paradigm. C++ supports them, but not to the extent of allowing them to integrate seamlessly into a program. There are two main limitations: (1) supporting pointers to `const` objects, and (2) supporting the standard pointer conversions.

We have presented several possible implementations, and discussed how they address these two limitations. Supporting pointers to `const` objects requires two smart pointer classes per object class. The two smart pointer classes should be defined such that the class for the pointer to mutable is derived from the class for pointer to `const`. Supporting class hierarchies is more difficult. The best way appears to be to use user-defined type conversions between the pointer classes. The behavior under this organization diverges from that of raw pointers in some circumstances that involve function overloading or chaining user-defined conversions. However, this should present only a slight inconvenience, not a fatal handicap.

Changes to C++ could allow it to support smart pointers better. Some possible changes include allowing some user-defined conversions to chain, or permitting user-defined code to implement the `derived::operator base&()` conversion. However, smart pointers are useful enough that it's important to identify how best to implement them, given the current language definition. That's what this paper has done: We've shown how to make smart pointers closely emulate the standard pointer conversions for `const` and class hierarchies, while circumventing erroneous and incorrect type conversions.

Acknowledgements

I would like to thank: **Peter Dickman**, **David Plainfossé**, **Darrell Long** and **Marc Shapiro** for commenting on various versions of the paper, the referees of the 1992 Usenix C++ Conference for their relevant insightful comments, **Philipe Gautron** for some lively discussions and comments on the paper, and, **Marc Shapiro** (again) for supporting this work at INRIA.

References

[ANS93] Draft proposed international standard for information systems—Programming language C++, January 1993. ANSI document X3J16/93–0010, ISO document WG21/N0218.

[Cop92] James Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[DMS92] Peter Dickman, Messac Makpangou, and Marc Shapiro. Contrasting fragmented objects with uniform transparent object references for distributed programming. In *Proc. SIGOPS 1992 European Workshop on Models and Paradigms for Distributed Systems Structuring*, 1992.

[Ede92] Daniel R. Edelson. Precompiling C++ for garbage collection. In *Proc. International Workshop on Memory Management*, pages 299–314. Spring-Verlag, 1992. Lecture Notes in Computer Science Number 637.

[Gau92] Philippe Gautron. Don't convert smart pointers to void∗, 1992. Private communication.

[Gro92] Ed Grossman. Using smart pointers for transparent access to objects on disk or across a network, 1992. Private communication.

[HM90] Antony L. Hosking and J. Eliot B. Moss. Towards compile-time optimizations for persistence. In *Fourth International Workshop on Persistent Object Systems*, pages 17–27. Morgan Kaufman (1991), 1990.

[Ken92] Brian Kennedy. The features of the object-oriented abstract type hierarchy (OATH). In *Proc. Usenix C++ Technical Conference*, pages 41–50. Usenix Association, August 1992.

[Mae92] Roman E. Maeder. A provably correct reference count scheme for a symbolic computation system. In unpublished form, 1992.

[MIKC92] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell. Reification and reflection in C++: An operating systems perspective. Technical Report UIUCDCS–R–92–1736, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March 1992.

[Sal92] Hayssam Saleh. *Conception et réalisation d'un système pour la programmation d'applications objets concurrentes et réparties sur machines parallèles.* PhD thesis, Université de Paris VI, 1992.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proc. Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992. ACM.

[SGH$^+$89] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system—Assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[SGM89] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In Stephen Cook, editor, *Proc. Third European Conference on Object-Oriented Programming*, British Computer Society Workshop Series, pages 191–204, Nottingham (GB), July 1989. The British Computer Society, Cambridge University Society.

[SMC92] Marc Shapiro, Julien Maisonneuve, and Pierre Collet. Implementing references as chains of links. In *Proc. Workshop on Object Orientation in Operating Systems*, 1992.

[Str87] Bjarne Stroustrup. The evolution of C++ 1985 to 1987. In *Proc. Usenix C++ Workshop*, pages 1–22. Usenix Association, November 1987.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.