Operator Overloading

Gary Powell, Doug Gregor, Jaakko Järvi

Project: Programming Language C++, Evolution Working Group
Reply-to: Gary Powell <powellg@amazon.com>, Doug Gregor <dgregor@cs.indiana.edu>, Jaakko Järvi <jarvi@cs.tamu.edu>

Abstract

There are a few C++ operators which cannot be overloaded. To better support Smart Pointers, and to remove inconsistencies in the language definition, we are proposing that these limitions be removed.

1. Background

The ability to overload the cast operators has been long a desired request of the library group. There have been other papers (N1870) which suggest that a library solution is possible. However we believe that a core langauge change is best and will make it easier for new users and library writers using smart pointers to avoid pitfalls and unnecessary meta programming when casting.

Especially in light of all the metaprograming and template work that has been done, there are now stronger cases to allow overloading of the rest of the operators. With one notable exception for the two dot operators which we currently believe cause more trouble than they fix. The newer libraries have shown us that the addition of overloading the cast operators could benefit all users of C++, by allowing the creation of smart pointers classes that behave in a more natural way. The ability to overload the cast operators on a smart pointer class will give it more of a look and feel of a raw pointer type. In addition there are some member operators that should also be allowed to be global operators as well. That these operators are required to be only member operators is an unnecessary hindrance to library writers and users which have overloaded every other operator appropriately.

The ability to overload the cast operators allows us to give a smart pointer access to the inherited data easily Overloading the cast operators allows the controlled casting away of const'ness similarly as const_cast<> does now for regular C++ pointers. Overloading the dynamic_cast<> operator allows the library writer to do the right thing, and maintain the correct reference count with a controled cast of the templated type.

- Overloading operators can make the intention of the user of the smart pointer more understandable.

In addition, casting of a smart pointer correctly can be difficult and often yields results which are unexpected. And at best results in a copy of the smart pointer, and at worst destruction of the base class inappropriately. This paper proposes that we fix these language features in way that makes the usage of smart pointer classes easier and more natural.

2. Problems to be Solved

The fact that the cast operators cannot be overloaded may be attributed to their being added to the language late in the process. Now that we have had over 15 years experience with these operators and smart pointers it has become obvious that overloading the cast operators would make smart pointers just that much smarter.

2.1. Allowing overloading of the cast operators.

The next example uses the SmartPointer class to show how the overloading of the cast operators makes the code easier to read and use. The goal is to allow the creator of the SmartPointer to give controlled access to the underlying data via a regular C++ cast operator. By overloading the cast operators, the SmartPointer can be made to look more like a regular pointer.

EXAMPLE

```
struct T {
   void f();
   void f() const;
};

template<typename T, typename S>
SmartPointer<T> const_cast<T *>(SmartPointer<S const> rhs)
{ return SmartPointer<T>(rhs); } // overly simplistic conversion.

SmartPointer<T const> cspt new(T);

const_cast<T *>(cspt)->f(); // calls T::f()
```

Note that the user of the SmartPointer when using a cast operator, treats it the same as when using a C++ style pointer. This is the part of the design goal of a SmartPointer, and that is to make it act like a native type.

Similar examples can be made for the 2 other cast operators.

```
struct A : public T {
virtual ˜A(){};
void f();
void f() const;
};

template<typename T, typename S>
SmartPointer<T> static_cast<T *>(SmartPointer<S> rhs)
{ return SmartPointer<T>(rhs)); } // overly simplistic conversion.

template<typename T, typename S>
SmartPointer<T> dynamic_cast<T *>(SmartPointer<S> rhs)
{ return SmartPointer<T>(rhs)); } // overly simplistic conversion.

SmartPointer<A> spa new(A);

static_cast<T *>(spa)->f();      // calls T::f()
dynamic_cast<T *>(spa)->f();     // calls T::f()
```

Allowing reinterpret_cast<>() to be overloaded appears to us as to just cause trouble, as it’s meaning is to say, I know what these bits really are, just do it. Overloading this operator doesn’t seem a good idea at this time.

2.2. Allowing global overloading of all operators.

There are four operators which are only allowed to be overloaded as members of a class, operator[](), operator()(), operator=() and operator->(). There is no particularly good reason for this restriction and there are some benefi ts to allowing overloading that will be shown. Therefore the restriction should be removed in the same language wart removal sweep that the other operators are being fi xed.

EXAMPLE

```
struct A { T m_t[10]; };

T & operator[](A &a, int i)
{ return a.m_t[i]; }

A a;
T t = a[1];   // calls global operator[](A &, int)

struct B {
T x;
T y;
};

B &operator=(B &b, pair<T,T> const &p)
{
  b.x = p.first;
  b.y = p.second;
  return b;
}

B b;
pair<T,T> p;

b = p;    /// calls global operator=(B&, pair<T,T> const &)
```

Clearly overloading a global operator=() is only useful if there is a way to transfer the data from one object to another, via friend, struct, or accessor functions. There are some issues with the implicitly-declared copy-assignment operator creating ambiguous lookups, so that overloading operator= from outside the class becomes brittle. Therefore global operator=() should be restricted to not collide with the implicitly-defined copy assignment operator. For example, it might make sense to allow global overloads of operator=() so long as the right-hand side is not T& or const T&.

```
struct C { void f(); }
void operator()(C &c)
{ c.f(); }

C c;
c();    // calls global void operator()(C &)

struct D {
T * f();
}

T *operator->(D &d)
{ return d.f(); }

d->f();   // calls T::f() after calling D::f() through operator->(D &)
```

The restriction that operator->() must return a pointer should be kept for all the same reasons that member T::operator->() must return a pointer. The reason is that cascading of the pointer is expected behaviour and that the compiler does not pass along the function offset as an argument.

The overriding reason for enabling global operators is consistency and removing language warts. In addition there are some active benefits for modifing the behavior of a class to which you do not own the source code. As usual with overriding of any other operator, care is required to do it well.

2.3. Overloading global new and delete for user defined types

We can override for a given user defined type the new and delete operators, but we must have access to the source code of that class to do it. There are occasionally times when a library writer would like to override the default storage for a given user defined type where the user of the class does not own the source code. The easiest way to easily manage the free store for this class would be to override new and delete as global operators. The current member functions, new and delete are basically static member functions of the class, the signature to the compiler is nearly the same for matching purposes.

EXAMPLE

```
#include "VendorClass"

void * operator new (VendorClass *v, size_t s)
{ return mySpecialAllocator(s); }

void operator delete (VendorClass *v, void *vp, size_t s)
{ mySpecialDeallocator(vp,s)

VendorClass *vc (new VendorClass);  // calls global operator new (VendorClass *, size_t)
delete vc;                          // calls global operator delete(VendorClass *, void *, size_t)
```

All of the variants of the new and delete operators should have this restriction removed. Note that the signature of global new and delete operators mimic those of a static member functions of the same name. However the first argument will always be null.

3. Proposal

3.1. Extensions

We are proposing a relatively minor extension to the language definition that has major implications, and yet has major benefits to writers and users of proxy classes. The proposal is that the 3 cast operators operator static_cast<>(), operator const_cast<>(), and operator dynamic_cast<>(), be allowed to be overloaded. In addition that the member operators: operator[], operator->, operator=(), and operator()() have global counterparts the same as the rest of the member operators, i.e. operator+=() et.al. And lastly that the free store operators new and delete have non member functions with a signature matching that of the static member free store operators.

3.2. Standardese Changes

INSERT

3.7.3.1.1 (Allocation Functions)

or the first parameter shall have type class *T and be a null pointer, and the second parameter shall have type size_t.

3.7.3.2.2 (Deallocation Functions)

or the first parameter shall have type class *T and be a null pointer, and the second parameter shall have type void *.

13.5.1

To the table of overloadable operators, add ".", ".*", "static_cast<>" "const_cast<>", and "dynamic_cast<>"

13.5.3.1 (Assignment)

or by a non-member function with two parameters, the fi rst parameter is of type class T, and the second shall not be of type T& or T const &.

13.5.4.1 (Function Call)

or by a non-member function with at least one parameter of type class T.

13.5.5.1 (Subscripting)

or by a non-member function with two parameters, the fi rst paramter is of type class T.

4. Summary

The argument for allowing overloading of the three cast operators is to make all smart pointers just that much smarter and easier to use. The removal of the constraint of operator()(), operator[](), operator->(), and the free store operators new and delete from being only member functions is to clean up a wart in the language and to make it easier to manage the free store for classes to which you do not own the source code. There is now suffi cient history and usage of forwarding pointer classes to demonstrate the need for smart pointers that behave as users expect.

There is also enough experience in writing complex classes by experienced software engineers to demonstrate that they are capable of making intelligent design choices given the powerful tools already at their hand.

5. Acknowledgements

We have had many private conversations with individuals who wish to remain anonymous. We thank them anyway. All errors and omissions are of course ours.

Bibliography

"Design and Evolution of C++" by Bjarne Stroustrup ISBN 0-201-54330-3

G. Powell, Doug Gregor, J.Järvi . ’WG21/N1671’Operator.() & Operator.*() ’J16/04-0111’

Thorsten Ottoseni ’WG21/N1870 and J16/05-0130 14 crazy ideas for the standard library in C++0x’ August 24,2005