Toward Opaque Typedefs for C++1Y

Document #:	WG21/N3515
Date:	2013-01-11
Revises:	None
Project:	Programming Language C++
Reply to:	Walter E. Brown <webrown.cpp@gmail.com>

Contents

1 Introduction

Although this paper is self-contained, it logically follows our discussion, begun several years ago in N1706 and continued in N1891, of a feature oft-requested for C++: an *opaque typedef*, sometimes termed a *strong typedef*. The earlier of those works was presented to WG21 on 2004-10-20 during the Redmond meeting, and the later work was presented during the Berlin meeting on 2005-04-06. Both presentations resulted in very strong encouragement to continue development of such a language feature.[1] Alas, the press of other obligations has until recently not permitted us to resume our explorations.

Now with C++11 as a basis, we return to the topic. Where our earlier thinking and nomenclature seem still valid, we will repeat and amplify our exposition; where we have new insights, we will follow our revised thinking and present for EWG discussion a high-level proposal for a C++1Y language feature to be known as an *opaque alias*.[2]

2 Motivation

It is a common programming practice to use one data type directly as an implementation technique for another. This is facilitated by the traditional **`typedef`** facility: it permits a programmer to provide a synonym or *alias* for the existing type that is being used as the underlying implementation. In the standard library, for example, **`size_t`** is a useful alias for a native unsigned

[1] A later paper, N2141, very briefly explored the use of forwarding constructors as an implementation technique to achieve a strong typedef, and listed several "Issues still to be addressed."

[2] Citations that look [like.this] refer to subclauses of C++ draft N3485.

integer type; this provides a convenient and portable means for user programs to make use of an implementation-selected type that may vary across platforms.

We characterize the classical typedef (even if expressed as a C++11 *alias-declaration*) as a *transparent* type facility: such a declaration introduces a new type name, but not a new type.[3] In particular, variables declared to have the newly-introduced alias type can just as easily be variables declared to have the aliased type, and *vice versa*, with not the slightest change in behavior.

This *de facto* type identity can have significant drawbacks in some scenarios. In particular, because the types are freely interchangeable (implicitly *mutually substitutable*), functions may be applied to arguments of either type even where it is conceptually inappropriate to do so. The following C++11 example provides a framework to illustrate such generally undesirable behavior:

```
using score = unsigned;
score  penalize( score n )  { return n > 5u ? n - 5u : score{0u}; }

using serial_number = unsigned;
serial_number  next_id( serial_number n )  { return n + 1u; }
```

The new aliases make clear the intent: to **`penalize`** a given **`score`** and to ask for the **`next_id`** following a given **`serial_number`**. However, these intentions are unenforceable, as the use of type aliases in the above code have made it possible, without compiler complaint, to **`penalize`** a **`serial_number`**, as well as to ask for the **`next_id`** of a **`score`**. One could equally easily **`penalize`** an ordinary **`unsigned`**, or ask for its **`next_id`**, since all three apparent types (**`unsigned`**, **`next_id`**, and **`serial_number`**) are really only three names for a single type. Therefore, they are all freely interchangeable: an instance of any one of these can be deliberately or accidentally substituted for an instance of either of the other types. As pointed out in a recent WG21 reflector message, "The **`typedef`** problem is one that we know badly bites us ever so often . . . vis-a-vis overloading."[4]

We see the results of such *type confusion* among even moderately experienced users of the standard library. For example, each container template provides a number of associated types such as **`iterator`** and **`size_type`**. In some library implementations, **`iterator`** is merely an alias for an underlying pointer type. While this is, of course, a conforming technique, we have all too often seen programmers treating **`iterator`**s as interchangeable with pointers. With their then-current compiler and library version, their code "works" because the **`iterator`** is implemented via a typedef to pointer. However, their code later breaks because an updated or replacement library uses some sort of struct as its **`iterator`** implementation, a choice generally incompatible with the user's now-hardcoded pointer type.

Even when there is no type confusion, a classical typedef can still permit inapplicable functions to be called. For example, it probably is reasonable to add two **`score`**s, to double a **`score`**, or to take the ratio (quotient) of two **`score`**s. However, it seems meaningless to allow the product of two **`score`**s,[5] yet nothing in the classical typedef interface could prevent such multiplication.

A final example comes from application domains that require representation of coordinate systems. Three-dimensional rectangular coordinates are composed of three values, not logically interchangeable, yet each aliased to **`double`** and so substitutable without compiler complaint. Worse, applications may need such rectangular coordinates to coexist with spherical and/or

[3] "A *typedef-name* is thus a synonym for another type. A *typedef-name* does not introduce a new type . . . . A *typedef-name* can also be introduced by an *alias-declaration*. . . . It has the same semantics as if it were introduced by the **`typedef`** specifier. In particular, it does not define a new type. . . " ([dcl.typedef]/1–2).

[4] Gabriel Dos Reis, WG21 reflector message c++std-sci-52, 2013-01-10. Lightly reformatted.

[5] The pattern in this example follows that of the customary rules of *commensuration* as summarized in The International System of Units (SI). Per SI, for example, two lengths can be summed to produce another length, but the product of two lengths produces a length-squared (*i.e.*, an area), not a length. Applying this principle to our **`score`** example, the product of two **`score`**s should yield a **`score`**-squared. In the absence of such a type, the operation should be disallowed.

cylindrical coordinates, each composed of three values each of which is commonly aliased to **`double`** and so indistinguishable from each other. As shown in the example below, such a large number of substitutable types effectively serves to defeat the type system: an ordinary **`double`** is substitutable for any component of any of the three coordinate systems, permitting, for example, a **`double`** intended to denote an angle to be used in place of a **`double`** intended to denote a radius.

```
typedef  double  X, Y, Z;           // Cartesian 3D coordinate types
typedef  double  Rho, Theta, Phi;   // spherical 3D coordinate types

class PhysicsVector {
public:
  PhysicsVector(X, Y, Z);
  PhysicsVector(Rho, Theta, Phi);
  ...
};  // PhysicsVector
```

If the above **`typedef`**s were opaque rather than traditional, we would expect a compiler to diagnose calls that accidentally provided coordinates in an unsupported order, in an unknown coordinate system, or in an unsupported mixture of coordinate systems. While this could be accomplished by inventing classes for each of the coordinates, this seems a fairly heavy burden. The above code, for example, would require six near-identical classes, each wrapping a single value[6] in the same way, differing only by name.

3 Desiderata

From extended conversations with numerous prospective users, including WG21 members, we have distilled the key characteristics that should distinguish between a classical typedef and what we will term an *opaque alias* feature.

In brief, an opaque alias is to define a new opaque type, treated as a new type that is distinct from and distinguishable from its underlying type, yet retaining layout compatibility[7] with its underlying type. The intent is to allow programmer control (1) over substitutability between an opaque alias and its underlying type, and (2) over overloading (including operator overloading) based on parameters whose type is or involves an opaque alias. Unlike the similar relationship of a derived class to its underlying base class, it is desired that both class and (perhaps especially) non-class types be usable as underlying types in an opaque alias.

Some consequences and clarifications, in no particular order:

- **`is_same<`***opaque-type***`,`***underlying-type***`>::value == false`**.
- **`typeid(`***opaque-type***`) != typeid(`***underlying-type***`)`**.
- **`sizeof`** *opaque-type* **`==`** **`sizeof`** *underlying-type*.
- Consistent with restrictions imposed on related features such as base classes and integer types underlying enums, an underlying type (1) should be complete and (2) should not be cv-qualified. We also do not require that reference types, array types, function types, or pointer-to-member types be allowed as an underlying type.
- Overload resolution should follow existing language rules, with the clarification that a parameter of an opaque type is a better match for an argument of an opaque type than is a parameter of its underlying type.

[6] A typical C++11 implementation of **`std::duration<>`** exemplifies a family of such wrappers around a single value.

[7] Specified in [basic.types]/11, [dcl.enum]/8, and [class.mem]/16–17.

- Mutual substitutability should be always permitted by explicit request, using either constructor notation or a suitable cast notation, *e.g.*, **`reinterpret_cast`**. Such a *type adjustment* conversion between an opaque type and its underlying type (in either direction) is expected to have no run-time cost.[8]
- A type adjustment conversion should never cast away constness.
- Pointers/references to an opaque type are to be explicitly convertible, via a type adjustment, to pointers/references to the underlying type, and conversely. This may imply that an underlying type should be considered *reference-related*[9] to its opaque type, and conversely,
- A template instantiation based on an opaque type as the template argument is distinct from an instantiation based on the underlying type as the argument. No relationship between such instantiations is induced; in particular, neither is an opaque type for the other.

4 Implicit type adjustment

We have found it both convenient and useful, when defining certain kinds of opaque types, to allow that type to model the *is-a* relationship with its underlying type in the same way that public inheritance models it with its base class. Therefore, in addition to the explicit substitutability described in the previous section, we propose controlled implicit unidirectional substitutability.

When permitted by the programmer, an instance of the opaque type can be implicitly used as an instance of its underlying type.[10] Such implicit type adjustment is expected to have the same run-time cost (*i.e.*, none) as the explicit type adjustment that is always permitted.

We have found three levels of implicit type adjustment permissions to suffice, and propose to identify them via the conventional access-specifiers:

- **`private`**: permits no implicit type adjustment.
- **`public`**: modelling *is-a*, permits implicit type adjustment everywhere.
- **`protected`**: modelling *is-implemented-as*, permits implicit type adjustment only as part of the opaque type's definition.

Even if modelling *is-a*, an opaque alias induces no inheritance relationship. In particular, **`is_base_of<`***opaque-type*, *underlying-type***`>::value`** and **`is_base_of<`***underlying-type*, *opaque-type***`>::value`** are each **`false`**. Classes marked **`final`** can thus serve as underlying types.

5 Prior art

A macro implementing a kind of opaque alias has been distributed with Boost's serialization library for over a decade. Internal documentation[11] summarizes its behavior as "**`BOOST_STRONG_TYPEDEF(T,D)`** creates a new type named **`D`** that operates as a type **`T`**." Using this paper's nomenclature, we would say that **`D`** denotes an opaque type whose underlying type is denoted by **`T`**.

The macro's code is relatively short and straightforward. In essence, it creates a class named for the opaque type, wrapping an instance of the underlying type and providing a fixed set of basic functionality for construction, copying, conversion, and comparison. There is no mechanism for adjusting this set of operations.

[8] "No temporary is created, no copy is made, and constructors ... or conversion functions ... are not called" [expr.reinterpret.cast]/11.

[9] Specified in [dcl.init.ref]/4.

[10] Implicit type adjustment in the other direction is never permitted, so some degree of opacity will always be present.

[11] Found in header **`boost/strong_typedef.hpp`**, © 2002 by Robert Ramey.

With the benefit of hindsight, the macro's author has very recently posted[12] an evaluation of his experience in creating and using the macro. He writes in significant part (lightly reformatted and with some typos corrected):

> Here's the case with **BOOST_STRONG_TYPEDEF**. I have a "special" kind of integer. For example a class version number. This is a number well modeled by an integer. But I want to maintain it as a separate type so that overload resolution and specialization can depend on the type. I needed this in a number of instances and so rather than re-implementing it every time I made **BOOST_STRONG_TYPEDEF**. This leveraged on another boost library which implemented all the arithmetic operations so I could derive from this. So it was only a few lines of macro code and imported all the "numeric" capabilities via inheritance. Just perfect.
>
> But in time I came to appreciate that the things I was using it for weren't really normal numbers. It makes sense to increment a class version number — but it doesn't make any sense to multiply two class version numbers. Does it make sense to automatically convert a class version number to an unsigned int? Hmmmm — it seemed like a good idea at the time, but it introduced some very sticky errors in the binary archive. So I realized that what I really needed was more specific control over the numeric operations rather than just inheriting the whole set for integers. So I evolved away from **BOOST_STRONG_TYPEDEF**.
>
> No question that **BOOST_STRONG_TYPEDEF** has value and is useful. But it's also not the whole story. It's more of a stepping stone along the way to more perfect code.

6 A start on *opaque alias* syntax

We propose that C++1Y complement traditional type aliases by providing an *opaque alias* facility, and nominate the following variation of *alias-declaration* syntax:

```
using identifier = access-specifier type-id opaque-definition
```

Much like a classical typedef, such a declaration introduces a new name (the *identifier*) for an opaque type that implicitly shares the definition of the underlying type named by the *type-id*. Thus, every opaque alias constitutes a definition; there are no forward declarations of an opaque type. However, as illustrated below, we will allow an opaque type to serve as the underlying type in a subsequent opaque alias.

The *opaque-definition* syntax will be clarified via the examples in subsequent sections.

7 The return type issue

Consider the following near-trivial example, sufficient to illustrate what we refer to as the *return type issue*:

```
1 using opaq = public int;
2 opaq o1{16};
3 auto o2{+o1};   // what's the type of o2?
```

As the comment indicates, the issue is to decide the type of the variable **o2** at line 3.

Since we have not (yet) provided any functions with **opaq** parameters, we appeal to the substitutability (type adjustment) property described above and find a built-in function,[13] declared for

[12] Robert Ramey: "[std-proposals] Re: Any plans for strong typedef." 2013-01-11.

[13] Specified in [over.built]/9.

overload resolution purposes as **`int operator+(int);`**. This is the function to call in evaluating the example's initializer expression. Accordingly, the expression's result type is **`int`**, leading to variable **`o2`** being deduced as **`int`**.

But this is probably not the intended outcome, and certainly not the expected outcome. After all, unary **`operator+`** is in essence an identity operation; it certainly seems jarring that it should suddenly produce a result whose type is different from that of its argument.

This issue has been one of the consistent stumbling blocks in the design of an *opaque typedef* facility. In particular, we have come to realize that there is no one consistent approach to the return type issue such that it will meet all expectations under all circumstances:

- Sometimes the *underlying-type* is the desired return type.
- Sometimes the *opaque-type* is the desired return type.
- Sometimes a distinct third type, as declared in the underlying function, is the desired return type.
- Occasionally, a fourth type, distinct from the above three, is the desired return type.
- Indeed, sometimes the operation should be disallowed, and so there is no correct return type at all.

Thus, we must allow a programmer to exercise control over the return type. Further, by analogous reasoning, we must allow a programmer to exercise control over the parameters' types.[14]

Returning to our example, what can a programmer do to obtain the expected result type of **`opaq`** instead of the underlying **`int`** type? Since we allow overloading on opaque types, the programmer can introduce a forwarding function into the example:[15]

```
using opaq = public int {
  opaq operator+(opaq o) { return opaq{ +int{o} }; }
};
opaq o1{16};
auto o2{+o1};  // type of o2 is now opaq
```

As shown above, the purpose of such a forwarding function (which we will term a *trampoline* in this context) is to adjust the type(s) of the argument(s) prior to calling the underlying type's version of the same function, and to adjust the type of the result when that call returns.

While it is a common pattern for the trampoline's return type to be the opaque type, we note that this need not hold in general. A trampoline can easily use the result of the underlying function call to produce a value of any type to which it is convertible. Indeed, under certain common circumstances, calls to trampolines can be elided by the compiler.

Each of the trampolines we have written during our explorations follows a common pattern, namely:

- Adjust the type or otherwise convert the opaque-type argument(s) to have the underlying type, and analogously for arguments whose types involve the opaque type.
- Then call the corresponding underlying function,[16] passing the adjusted argument(s).
- Finally, adjust the type or otherwise convert that call's result to a corresponding value of the specified result type.

Because of its frequency, we propose a shorthand to ease programmer burden in producing such trampolines: a function taking one or more parameters of an opaque type may be defined via

[14] Such granularity becomes especially important when there are at least two parameters, and (as in the earlier example of **`score`** multiplication) not all combinations of {*opaque-type*, *underlying-type*} are to be supported as parameter types.

[15] We use constructor syntax for brevity, but **`reinterpret_cast`** would seem more descriptive of the actual effects.

[16] If there is no corresponding underlying function to be called, the program is of course ill-formed.

`= default`, thereby instructing the compiler to generate the boilerplate forwarding code for us. Moreover, as suggested above, we expect a compiler to take advantage of its aliasing knowledge to elide the trampoline in all such cases, instead calling the corresponding underlying function and type-adjusting the return type as specified.[17]

As a final convenience to the programmer, we propose that the compiler be always permitted to generate constructors and assignment operators for copying and moving whenever the underlying type supports such functionality and the programmer has not provided his own versions. Should the programmer wish the opaque type to be not copyable, he can define his own version with **`= delete`**. The programmer can similarly define a trampoline with **`= delete`** whenever a particular combination of parameter types ought be disallowed.

The following example employs many of these proposed features:

```
using energy = protected double {
  energy  operator+ (energy , energy) = default;
  energy& operator*=(energy&, double) = default;
  energy  operator* (energy , energy) = delete;
  energy  operator* (energy , double) = default;
  energy  operator* (double , energy) = default;
};

energy e{1.23};  // okay; explicit
double d{e};  // okay; explicit
d = e;  // error; protected disallows implicit type adjustment here

e = e + e;  // okay; sum has type energy
e = e * e;  // error; call to deleted function
e *= 2.71828;  // okay

using thermal = public energy;
using kinetic = public energy;

thermal t{···};  kinetic k{···};
e = t;  e = k;  // both okay; public allows type adjustment
t = e;  t = k;  // both in error; the adjustment is only unidirectional

t = t + t;  k = k + k;  // okay; see next section
e = t + k;  // okay; calls the underlying trampoline
```

8 Opaque `class` types

We described and illustrated above how to address the return type issue for free functions. When the underlying type is a class, we additionally have the analogous issue for member functions.[18] It seems clear that each accessible member function ought have a corresponding trampoline as a member of the opaque type. Since all member functions of the underlying type are known to the compiler, we can take advantage of this and therefore propose that the compiler, by default, generate default versions of these trampolines.[19]

[17] When such elision takes place, the address of the trampoline (if taken) would be the same as the address of the underlying function.

[18] Trampolines for non-member functions can continue to be handled as described in the previous section.

[19] This behavior is also possible for an opaque type whose underlying type is itself an opaque type because all the underlying type's trampolines are known. Such circumstances give rise to default-generated trampolines that forward to other trampolines, with transitive elision encouraged where feasible.

Each such default-generated member trampoline will:

- Adjust the type of each argument of opaque type, including the invoking object, to the underlying type, and analogously for arguments whose types involve the opaque type.
- Then call the underlying type's corresponding member function, passing the type-adjusted argument(s).
- Finally, if the underlying function's return type is the underlying type, adjust the call's result so as to have the opaque type; otherwise returns the call's result unchanged.

This behavior means that the type of each default-generated member trampoline is isomorphic to that of the corresponding underlying member function, with each occurrence of the underlying type replaced by the opaque type. In case this is not what is wanted, the programmer may (as always) write his own member trampoline, thereby inhibiting the default generation of that trampoline and of any overloads of that trampoline. We propose the following syntax:

```
using opaque-type = access-specifier underlying-class {
  desired-member-trampoline-signature = default;
  friend desired-nonmember-trampoline-signature = default;
};
```

A member trampoline may not be thusly defined unless it has an accessible corresponding underlying member function. Two or more distinct trampolines may forward to the same underlying function. If the **= default** behavior is not what is desired, the programmer may instead supply (1) a brace-enclosed body[20] or (2) an **= delete** definition.

As is the case for non-member trampolines, each member trampoline is treated as an overload of the underlying function to which it forwards.

If an opaque type has an underlying type that directly inherits from a base class, we propose that **is_base_of<***base-type*, *opaque-type***>::value** be **true**. Note that, as proposed earlier, **is_base_of<***opaque-type*, *underlying-type***>::value** remains **false**. The effects of dynamic dispatch involving an opaque type are as yet unclear.

9 Opaque template aliases

There is no conceptual problem in extending opaque aliases in the direction of a C++11 alias-template.[21] Here is our example from an earlier section, rewritten in template form:

```
template <class T = double>  using energy = protected double {
  energy  operator+ (energy , energy) = default;
  energy& operator*=(energy&, double) = default;
  energy  operator* (energy , energy) = delete;
  energy  operator* (energy , double) = default;
  energy  operator* (double , energy) = default;
};

energy<> e{1.23};  // okay; explicit
double d{e};  // okay; explicit
d = e;  // error; protected disallows implicit type adjustment here

e = e + e;  // okay; sum has type energy<>
es = e * e;  // error; call to deleted function
```

[20] For example, such type adjustments as **std::shared_pointer<***opaque-type***>** to or from **std::shared_pointer<***underlying-type***>** are likely more complicated than a compiler should be asked to handle via **= default**.

[21] Specified in [temp.alias].

```
e *= 2.71828;  // okay

template <class T = double>  using thermal = public energy<T>;
template <class T = double>  using kinetic = public energy<T>;

thermal<> t{···};  kinetic<> k{···};
e = t;  e = k;  // both okay; public allows type adjustment
t = e;  t = k;  // both in error; the adjustment is only unidirectional

t = t + t;  k = k + k;  // okay; each calls a default-generated trampoline
e = t + k;  // okay; calls the underlying trampoline
```

10 Summary and conclusion

This paper has outlined a new solution, known as *opaque aliases*, to accommodate the long-standing desire for opaque types in C++. We have identified a number of *desiderata* and fundamental properties of such a feature, and provided realistic examples of its application.

We believe opaque aliases to be a viable new tool, complementing traditional type aliases, that allows programmers to address practical problems. We invite feedback from WG21 participants and other knowledgeable parties, and especially invite implementors to collaborate with us in order to experiment and gain experience with this proposed new language feature.

11 Acknowledgments

Many thanks to the readers of early drafts of this paper for numerous helpful discussions and insightful reviews of this work and its predecessors.

12 Bibliography

[N1706] Brown, Walter E.: “Toward Opaque **typedef**s in C++0X.”
ISO/IEC JTC1/SC22/WG21 document N1706 (pre-Redmond mailing), 2004-09-10.
Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1706.pdf.

[N1891] Brown, Walter E.: “Progress toward Opaque **typedef**s for C++0X.”
ISO/IEC JTC1/SC22/WG21 document N1891 (post-Tremblant mailing), 2005-10-18.
Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1891.pdf.

[N2141] Meredith, Alisdair: “Strong Typedefs in C++09 (Revisited).”
ISO/IEC JTC1/SC22/WG21 document N2141 (post-Portland mailing), 2006-11-06.
Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2141.html.

[N3485] Du Toit, Stefanus: “Working Draft, Standard for Programming Language C++.”
ISO/IEC JTC1/SC22/WG21 document N3485 (post-Portland mailing), 2012-11-02.
Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf.

[SI] Bureau International des Poids et Mesures: “The International System of Units (SI).”
8th edition, 2006.
Online: http://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf.
Also available in French.