# Function Aliases + Extended Inheritance = Opaque Typedefs

*Document #:* WG21/P0109R0
*Date:* 2015-09-25
*Revises:* N3741, N3515
*Project:* JTC1.22.32 Programming Language C++: EWG
*Reply to:* Walter E. Brown <webrown.cpp@gmail.com>

## Contents



*The doctor should be opaque to his patients and, like a mirror, should show them nothing but what is shown to him.*

— SIGMUND (né SIGISMUND) FREUD

**Abstract**

This paper proposes two core language additions to C++: (a) function aliases and (b) extended inheritance. While individually useful, the combination of these two features provides the functionality of an *opaque typedef*, a feature that has long been requested for C++.

## 1 Background

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*.[1] 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.[2] Alas, the press of other obligations did not permit us to resume our explorations until 2013.

With C++11 as a basis, those resumed explorations were published as N3515 and N3741. Where our earlier thinking and nomenclature seemed still valid, we repeated and amplified our earlier exposition; where we had new insights, we followed our revised thinking and presented for



[1] "I am not inventing a need for strong typedefs. We already have such a need. As evidenced by people constantly asking for them" — Nicol Bolas, "Re: [std-proposals] Thoughts on N4542 std::variant," 2015-09-22.

[2] A later paper by Alisdair Meredith, 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."

EWG discussion a high-level proposal for a C++1Y language feature to be known as an *opaque alias*.[3] Those papers were presented to and discussed by EWG at the Chicago meeting on 2013-09-26.[4] This presentation again resulted in strong encouragement to continue this feature's development. Taking EWG's guidance into account, the present paper recapitulates and augments those earlier explorations. New in this revision are:

- an updated title that is more descriptive of our current thinking,
- this expanded Introduction,
- discussion (in §5) of additional prior art that attempted to address the fundamental issues under discussion,
- revisions and expansions to treat the requested feature as akin to inheritance[5] rather than as akin to aliasing,
- a proposal (§10) for a general C++ *function alias* language feature that appears useful in its own right and that also appears useful as a strategy for declaring *trampolines* (defined in §7 below), and
- a proposal (§11) for an extension of traditional inheritance to allow native object types to serve as base classes.

## 2 Motivation

It is a very 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 an application-specific 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 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.[6] 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 very modest C++11 examples provide a framework to illustrate such generally undesirable behavior:

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

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

[3] Citations that look [like.this] refer to subclauses of C++ draft N4527. We will generally omit cross-references from quoted text.

[4] The notes of that discussion are available at http://wiki.edg.com/twiki/bin/view/Wg21chicago2013/EvolutionWorkingGroup.

[5] "I disagree that it's an alias. It's definitely not a synonym. I can see this as a different sort of inheritance, but not an alias or a typedef. . . . [H]ow is it not inheritance? It feels very much like inheritance." — David Vandevoorde, during the Chicago review, *ibid.*

[6] "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).

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, 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 a result, the programmer's intentions are unenforceable. As pointed out in a WG21 reflector message, "The **typedef** problem is one that we know badly bites us ever so often . . . vis-a-vis overloading."[7] 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 **sizetype**. 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,[8] 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 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.

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

4  class PhysicsVector {
5  public:
6    PhysicsVector(X, Y, Z);
7    PhysicsVector(Rho, Theta, Phi);
8    ...
9  };  // 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

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

[8] 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.

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[9] 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, the principal utility of an opaque alias is to define a new type that is distinct from and distinguishable from its underlying type, yet retaining layout compatibility[10] with its underlying type. The intent is to allow a programmer to control:

1. substitutability of an opaque alias for its underlying type, and
2. overloading (including operator overloading) based on any parameter whose type is or otherwise involves an opaque alias.

Unlike the traditional relationship of a derived class to its underlying base class, we desire 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*`)`**.
- For each primary or composite type trait[11] **`is_x`**, **`is_x<`***opaque-type*`>::value == is_x<`***underlying-type*`>::value`**.
- Consistent with restrictions imposed on analogous relationships such as base classes underlying derived classes and integer types underlying enums, an underlying type should be (1) complete and (2) not cv-qualified. We also do not require that any enum type, reference type, array type, function type, or pointer-to-member type 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.
- 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.[12]
- 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*[13] 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.

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

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

[11] These traits are defined in [meta.unary.cat] and [meta.unary.comp], respectively.

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

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

## 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.[14] 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

We have become aware of several attempts to provide our desired feature via a C++ library. As described below, these efforts are remarkably similar in their goals, their implementation techniques, and their perceived weaknesses. This list is intended to be indicative, not comprehensive; we are aware of additional attempts that share essential characteristics with one or more of the following.

**Boost.** A macro implementing a kind of opaque alias has been distributed for over a decade as part of the Boost serialization library by Robert Ramey. Internal documentation summarizes its behavior as "**`BOOST_STRONG_TYPEDEF(T,D)`** creates a new type named **`D`** that operates as a type **`T`**."[15] 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.

With the benefit of considerable hindsight, Ramey has posted an evaluation of his experience in creating and using the macro. He writes in significant part:[16]

> 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

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

[15] In header **`boost/strong_typedef.hpp`**, © 2002 by Robert Ramey.

[16] Robert Ramey: "[std-proposals] Re: Any plans for strong typedef." . 2013-01-11 Lightly reformatted and with some typos corrected.

> 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.

**`true_typedef`.** In 2003, Matthew Wilson published an article describing **`true_typedef`**, "A template wrapper that provides type uniqueness for otherwise synonymous types" [Wil03]. He summarizes the problem being addressed:

> One of the few legitimate criticisms of C++ is the fact that typedefs are always weak. One can define two types from the same base type and, without constraint (or even a compiler warning!), mix these types. Not only can this lead to problems in erroneous assignment of one type to another, or between a typedef and its base type, but it also precludes the use of overloaded functions based on such types.

The current version of the code is available at http://www.stlsoft.org/. While a template rather than a macro, code inspection reveals that, like the Boost library described above, this project also creates a class named for the opaque type, wrapping an instance of the underlying type and providing a fixed set of functionality. While the set seems considerably larger than that provided by Boost, there is again no mechanism for adjusting this set of operations.

**"strong typedef for integer/floating point types."** In a brief blog post, Akira Takahashi presents class templates **`tagged_real`** and **`tagged_int`** as his solution to the need for opaque aliases when the underlying type is an arithmetic type [Tak12]. Each template again wraps an underlying type, with yet another fixed set of functionality. Template instantiations are distinguished via the use of a tag type.

**DESALT_NEWTYPE.** Finally, DESALT_NEWTYPE, by Oyama Koichi, seems to be the most recent library attempt that features a macro-based approach to opaque aliases.[17] Unfortunately, internal documentation indicates that the library is incomplete and we are informed that the project has been abandoned due to lack of motivation.[18] However, in two private emails, a translator has communicated Koichi's thoughts as to the project's scope and remaining weaknesses:[19]

> Suppose, there are string objects of plain text and encrypted text. I don't want to mix up these objects. I want to make these mixed up code into an error at compile time. So I wanted a different type with same interfaces and behaviors. But it's tedious to wrap original type and write forwarding functions for all members.
>
> So, initially, I wrote some simple thing that does "derive by private and offers all base class members as derived class's public member". But when I wrote that, I realized it's

[17] See header **`newtype.cpp`** at https://github.com/dechimal/desalt as of 2013-09-07.

[18] Ryoe Ezoe: "Prior art of N3741 opaque alias" (personal correspondence), 2013-09-07.

[19] Ryoe Ezoe: "Re: Prior art of N3741 opaque alias" (personal correspondence), 2013-09-07 and 2013-09-08. Translation of private remarks by Oyama Koichi. Lightly reformatted and with some errors corrected. Italized remarks were inserted by the translator.

> incomplete for member functions which have original type as a parameter or return type, or non-member functions which is associated to the original type by ADL.
> So my code got additional features like automatically wrapping the parameters and return types *he means defining the forwarding function that replace the occurrence of original type in parameter and return type by new type*, or forwarding by explicitly specifying the signature. But even if it automatically wraps **`std::vector<int>`**, it doesn't take care of **`std::vector<int>::iterator`**, so in reality, it's still inconvenient.
>
> ---
>
> It is a serious flaw that it can't automatically take care of the container's member function which takes iterator as an argument. As I wrote earlier, it makes it hard to create a new type like **`encrypted_string`** from **`std::string`**.

## 6 A hypothetical *opaque alias* syntax

For the sake of discussion, let us assume an *opaque alias* facility via the following variation of *alias-declaration* syntax:

```
1 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.

Note that the *access-specifier* is not optional. We make this recommendation because (1) none of the *access-specifier*s is an obvious default and (2) the presence of an *access-specifier* serves as a syntactic marker to distinguish an opaque alias from a traditional type 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,[20] declared for 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 an 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 operand.

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 no single approach to the return type issue will consistently meet expectations under all circumstances:

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

- 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.[21]

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:[22]

```
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,[23] 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 **`= 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.[24]

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.

[21] 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.

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

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

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

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 consider the analogous issue for member functions.[25] 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.[26]

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 return 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

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

[26] 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.

(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 (a) a brace-enclosed body[27] or (b) 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.[28] 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&, T     ) = default;
  energy  operator* (energy , energy) = delete;
  energy  operator* (energy , T     ) = default;
  energy  operator* (T      , 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

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
```

[27] 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**.

[28] Specified in [temp.alias].

## 10 Introducing function aliases

Following the precedent set by namespace aliases and type aliases, we propose to introduce a *function alias* to C++.

The idea behind a *function alias* is far from new. Stroustrup described it in his D&E book [Str94, §12.8, pp. 273-5] as a "renaming" feature in the context of resolving name clashes due to multiple inheritance: "The semantics of this concept are simple, and the implementation is trivial; the problem seems to be to find a suitable syntax." He states that such a proposal "was presented at the standards meeting in Seattle in 1990" and that, although there was initially "a massive majority," the feature was ultimately not adopted: "At the next meeting, . . . we agreed that such name clashes were unlikely to be common enough to warrant a separate language feature."

We agree with Stroustrup's conclusion: "Synonyms can be useful and occasionally essential. However, their use should be minimized to maintain clarity and commonality of code used in different contexts." We now (re)raise the proposal[29] because, in addition to the above name clash issue, we have found at least two other use cases for such a feature.

Our first additional use case arises from algorthms that "have two versions: one that takes a function object of type **Compare** and one that uses an **operator<**" [alg.sorting]/1. Via use of a *function alias* and other modern C++ features, we believe we can and should unify these two versions. The following example illustrates the basic idea:

```
template< class RA, class R = std::less<> >
void
  sort( RA b, RA const e, R lt = {} )
{
  using operator<() = lt;  // operator < has type R
  // Remaining code in this scope uses infix < in place of calls to lt().
  // (A future proposal may suggest synthesis of other relational
  // operators from an operator< declared in this fashion.)
}
```

More important to this paper's goals, however, is that function aliases present a viable mechanism for specifying the trampolines as defined in §7. This would make it clear that aliasing is involved, rather than any artificial forwarding function initially suggested above.

Syntactically, we propose that a *function alias* nominally consist of a **using** keyword followed by a *function-definition* [dcl.fct.def.general]/1 in which the *function-body* consists of an equal sign followed by a (possibly qualified) function name and a terminating semicolon. Optionally, the *declarator* may consist solely of a *declarator-id*, in which case the associated return type and parameters shall implicitly be those of the *function-body*. We also envision a possible additional shorter form, for programmer convenience, that takes advantage of implicit type adjustment such as that permitted by **public** inheritance.

Semantically, we propose that the usual name lookup rules apply to each call of a *function alias*. If lookup finds a funcion alias, a second lookup, this time of the aliased function, is carried out. Modulo overload resolution (whose rules need to be adjusted slightly to take this new alias into account), the program is ill-formed if (a) this second lookup is unsuccessful, (b) any argument can not be treated as being of the corresponding parameter type in the aliased function, or (c) the aliased function's return type can not be treated as being of the return type declared in the alias. Of course, further lookups would be required if the aliased function were itself discovered to be another alias.

[29] Anyone who has copies of the relevant 1990-vintage committee papers is requested to share them with the author.

It is an open question whether we wish to permit a free function to alias a member function, or conversely.

## 11 Extended inheritance

As Stroustrup wrote, "In the absence of virtual functions, a user could use objects of a derived class and treat base classes as implementation details" [Str94, §2.9.1, p. 49]. This is precisely the situation giving rise to the common request for an *opaque typedef*. The only part that seems to be missing is the ability to inherit from a native object type, e.g., from an `int`. During the Chicago review, Stroustrup commented, " I've used inheritance to fake this, which I can't do . . . for integers for historical reasons, but one way to look at it would be 'what would you get if we allowed inheritance from `int`'?"

We believe that this capability has been lacking because it has not been clear how to provide operations on such a derived type. Function aliases as proposed above seem to fill that role, and therefore we propose an extension to the rules of inheritance such that any native object type (not an array, not cv-qualified) can serve as a base class. Such a derived class needs no member functions, but does need trampoline-like capability; we believe that function aliases as proposed above can serve that role.

## 12 Summary and conclusion

This paper has outlined a new solution 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 the proposed language extensions to be viable new tools, complementing traditional type and other aliases, that allow programmers to address practical problems. We invite feedback from WG21 participants and other knowledgeable parties. We especially invite implementors to collaborate with us in order to experiment and gain experience with these proposed new language features.

## 13 Acknowledgments

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

## 14 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.

[N3515] Brown, Walter E.: "Toward Opaque Typedefs for C++1Y." ISO/IEC JTC1/SC22/WG21 document N3515 (mid Portland/Bristol mailing), 2013-01-11. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3515.pdf.

[N3741] Brown, Walter E.: "Toward Opaque Typedefs for C++1Y, v2." ISO/IEC JTC1/SC22/WG21 document N3741 (pre-Chicago mailing), 2013-05-16. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3741.pdf.

[N4527] Smith, Richard: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4521 (post-Lenexa mailing), 2015-05-22. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4527.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.

[Str94] Stroustrup, Bjarne: *The design and evolution of C++*. Addison-Wesley, 1994. ISBN: 0-201-54330-3.

[Tak12] Takahashi, Akira: "strong typedef for integer/floating point types." In *Japanese C++ Programmers Activity*, 2012-11-27. Online: http://cppjp.blogspot.com/2012/11/strong-typedef-for-integerfloating.html.

[Wil03] Wilson, Matthew: "True typedefs." Dr. Dobb's, 2013-05-16. Online: http://www.drdobbs.com/true-typedefs/184401633.

## 15 Revision history

| Version | Date | Changes |
|---|---|---|
| 1 | 2013-01-11 | • Published as N3515. |
| 2 | 2013-08-30 | • Fixed copy/paste errors in example code. • Added bullet and footnote re type traits (§3). • Tweaked formatting, punctuation, and grammar. • Published as N3741. |
| 3 | 2015-09-25 | • Retitled to reflect latest thoughts. • Updated the introduction (§1). • Discussed Wilson's, Takahashi's, and Koichi's projects (§5). • New: Abstract, §10, and §11. • Numerous editorial tweaks. • Published as P0109R0. |