Progress toward Opaque Typedefs for C++0X

Document #: WG21/N1891 = J16/05-0151
Date: 2005-10-18
Revises: None
Project: Programming Language C++
Reference: ISO/IEC IS 14882:2003(E)
Reply to: Walter E. Brown<wb@fnal.gov>
CEPA Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

Contents

Important among [the qualities needed to do scientific research] is the judgment of what problems are ripe for solution: exactly when does it become profitable to look again over old ground, to rediscuss problems that once seemed too hard.

— SIR FRED HOYLE

1 Introduction

This paper continues the discussion begun in N1706 [Bro04]. That paper presented introductory rationale, exposition, and examples as a preliminary exploration on the subject of an *opaque typedef*. In particular, we envisioned to combine the classical `typedef` with the C++ concepts of `public` and `private` inheritance, thereby producing two new constructs that would jointly address the oft-requested *opaque typedef* feature.

That early work was presented to the C++ standards bodies on October 20, 2004, at their meeting in Redmond, Washington, USA. The presentation resulted in very strong encouragement

to continue development of the *opaque typedef*. The present paper presents results of a deeper study of this topic, using nomenclature defined in Appendix A herein.

2 The motivating requirement: overloading

From extended conversations with prospective users, it has become clear that the characteristic feature desired of an *opaque typedef* is the ability to overload functions and operators based on one or more newly-defined *opaque-type*s. For example, we would wish to overload the constructors in a `PhysicsVector` class such that each constructor corresponds to a distinct coordinate system. Temporarily using a notional `opaque` keyword, we might code this as:

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

5  class PhysicsVector
6  {
7  public:
8    PhysicsVector(X, Y, Z);
9    PhysicsVector(Rho, Theta, Phi);
10   ...
11 };  // PhysicsVector
```

In this way, a compiler would be able to diagnose usages that accidentally provided coordinates in an unsupported order or in an unsupported mixture of coordinate systems. While this can be accomplished in C++03 by inventing classes for each of the coordinates, this is generally viewed as a fairly heavy burden: the above code would require six near-identical classes, each wrapping a single value in the same way, differing only by name.

As a natural consequence of this required overloading capability, we propose that an *underlying-type* `UT` shall meet all requirements of TR1 [Aus05, §4] such that `is_convertible<OT,UT>::value` is well-formed for any *opaque-type* `OT` for which `UT` serves as *underlying-type*. (This requires, for example, that `UT` be non-`void`, and that it not be an incomplete type.)

In addition, we propose that an *underlying-type* `UT` shall not be cv-qualified. This restriction is consistent with two important C++03 precedents: (1) The underlying type of an `enum` has no provision that permits its underlying type to be cv-qualified. (2) Application of inheritance makes no provision that permits a base class to be cv-qualified.

3 Some basic properties

To complement the above overloading *desideratum*, it is convenient to express in terms of TR1's type traits [Aus05, §4] many of the basic properties desired of an *opaque-type* `OT` and of the relationship between `OT` and its *underlying-type* `UT`. In the following, we use a function-style notation to gain economy of expression without loss of clarity:

1. How shall `OT`'s unary type traits be defined?
   Proposed answer: for each category and property defined by TR1, `is_category(OT) == is_category(UT)` shall be `true`, and
   `has_property(OT) == has_property(UT)` shall also be `true`.

2. Are `OT` and `UT` the same type?
   Proposed answer: `is_same(UT,OT)` shall be `false`.

3. Are `OT` and `UT` related by inheritance?
   Proposed answer: `is_base_of(UT,OT)` shall be `false`, and `is_base_of(UT,OT)` shall also be `false`.

4. Given two template instantiations, one with `OT` as a template argument and the other with `UT` as the corresponding argument, how are the instantiations related?
   Proposed answer: the two instantiations are unrelated.

5. Can instances of `UT` be explicitly converted to instances of `OT`?
   Proposed answer: yes.

6. Can instances of `UT` be implicitly converted to instances of `OT`?
   Proposed answer: no.

7. Can instances of `OT` be explicitly converted to instances of `UT`?
   Proposed answer: yes.

The next section will address the final question: can instances of `OT` be implicitly converted to instances of `UT`?

4 Two kinds of *opaque typedef*

We now come to a particularly important issue: Can instances of `OT` be implicitly converted to instances of `UT`?

As we wrote in N1706,

> Guided by well-understood *substitutability* principles as embodied in today's C++, we believe there is value in proposing to extend classical transparent `typedef`s with two forms of opacity. We have designated these new forms, respectively, as `public` and `private`. The former would permit *substitutability* in one (consistently specified) direction, the latter would permit no *substitutability* at all, while classical `typedef`s would continue to permit mutual *substitutability*.

Our proposed answer thus depends on the kind of *opaque typedef* with which `OT` was declared.

4.1 `typedef public`

This first kind of *opaque typedef* is modeled after the behavior of two features of today's C++: `enum`s and `public` inheritance. These features' common characteristic of interest is the one-way *substitutability* that they induce among instances of the types involved.

In brief, the semantics of the proposed `public typedef` would permit similar *substitutability* in that instances of a newly declared type (the *opaque-type*) may be used wherever an instance of the original type (the *underlying-type*) is expected. Unlike the mutual *substitutability* induced by a classical `typedef`, an instance of an *underlying-type* may not stand in where an instance of the *opaque-type* is expected. Further, an instance of a `public typedef` may never stand in for an instance of a second *opaque-type*, even when both have the identical *underlying-type*.

As one consequence of this proposal, the *underlying-type* of a `public typedef` shall be *reference-related* to its *opaque-type* (*i.e.*, to its *public-type*) per [ISO03, 8.5.3/4]. Further exposition and discussion of *substitutability* is provided in §5.

4.2 `typedef private`

This second kind of *opaque typedef* is modeled on the behavior of `private` inheritance in today's C++ and was intended to fill an anticipated need for a non-*substitutable* type. However, we have to date found no killer example for such a facility, and therefore will give less attention to `private typedef`s in the remainder of this paper. We invite interested readers to formulate and contribute such an example; failing any, we will likely discard this kind of *opaque typedef* from future consideration.

5 *Substitutability*

N1706 relied heavily on *substitutability* as a significant defining characteristic for classical as well as for *opaque typedef*s. In this section, we will formalize these notions.

5.1 Definition of *substitutable*

A type `B` is said to be *substitutable* for a type `A` if and only if there exists an implicit conversion from `B` to `A`.

Equivalently (per [ISO03, §4/3]), if type `B` is *substitutable* for type `A`, then for an arbitrary expression `b` of type `B` and for some invented temporary variable `__a`, the declaration `A __a = b;` must be well-formed. If that declaration is ill-formed, then type `B` is not *substitutable* for type `A`, and conversely.

Note that the above definition is similar, but not identical, to that of TR1's `is_convertible` type trait. The latter is defined to exclude certain "[s]pecial conversions" and "adjustments" that our definition of *substitutability* does include.

5.2 Notation: the *substitutability predicate*

As a convenient shorthand in the nature of a type trait, we introduce the *substitutability predicate* notation `is_subst(B,A)`, defined such that `is_subst(B,A)` is true if type `B` is *substitutable* for type `A`, and is false otherwise.

Future developments may suggest the extension of this predicate into a fully-featured C++0X type trait; if so, the form `is_subst<B,A>::value` will likely be more appropriate in that context. We will herein continue to use the functional style, reserving for the future any discussion of such a trait's usefulness in programming, of its name, or of the order of its parameters.

5.3 *Substitutability* in C++03

We note that the above definition of *substitutable* captures a relationship defined, induced, or required by several extant C++ constructs. For example:

1. Given a class `D` that publicly inherits from a class `B`, we have `is_subst(D,B)` (commonly known in this context as the *is-a* relationship) as well as `is_subst(D*,B*)`, `is_subst(D,B&)`, *etc.*
2. We also have `is_subst(E,I)` where `E` is an `enum` type and `I` is the integral type used to encode `E`'s enumerators.
3. For arbitrary type `T`, we generally have the trivial `is_subst(T,T)` as well as the more interesting `is_subst(T,T const)`, `is_subst(T[...],T*)`, `is_subst(T*,void *)`, *etc.*
4. A valid function call requires `is_subst(Ai,Pi)` where `Ai` denotes the type of the i^{th} argument in the call and `Pi` denotes the type of the i^{th} parameter of the function being called.

5. Among numeric types in the core language, we have such relationships as `is_subst(int,long)`, `is_subst(float,double)`, *etc.*

5.4 `typedef`-induced *substitutability*

Let `N` denote the type identified by a (classical or opaque) `typedef`'s *declarator-id*, and let `U` denote that *declarator-id*'s *underlying-type*. We then distinguish the following different forms of `typedef` based on the *substitutability* relationships that each induces:

Kind of **`typedef`**	**`is_subst(N,U)`**	**`is_subst(U,N)`**
classical (transparent)	true	true
public (opaque)	true	false
private (opaque)	false	false

Further let `N2` denote the type named by the *declarator-id* of a second `typedef` of the same kind (*i.e.*, classical, public, or private) and with the same *underlying-type* `UT`. We then have the following additional *substitutability* relationships between `N` and `N2`:

Kind of **`typedef`**	**`is_subst(N,N2)`**	**`is_subst(N2,N)`**
classical	true	true
public	false	false
private	false	false

5.5 Transitivity

The *substitutability* induced by classical `typedef`s is clearly transitive: if `is_subst(A,B)` and `is_subst(B,C)`, then `is_subst(A,C)`. However, this transitivity is a consequence of the classical `typedef`'s transparent nature. As the following counter-example demonstrates, transitivity is not a general property of the *substitutability* relationship:

```
1  //                                  Listing 2
2  struct A {       };
3  struct B { B(A); };
4  struct C { C(B); };
5
6  A a;
7  B ba = a;   // okay:  is_subst(A,B)
8  C cb = ba;  // okay:  is_subst(B,C)
9  C ca = a;   // oops:  ! is_subst(A,C) per [ISO03, §13.3.3.1.2/1]
```

We intend that the *substitutability* induced by `public typedef`s, like that induced by classical `typedef`s, be transitive:

```
1  //                                  Listing 3
2  struct A {};
3  typedef public A B;
4  typedef public B C;
5
6  A a;
7  B ba = a;   // okay:  is_subst(A,B)
8  C cb = ba;  // okay:  is_subst(B,C)
9  C ca = a;   // okay:  is_subst(A,C)
```

5.6 Other factors influencing *substitutability*

On reflection, *substitutability* is not an automatic consequence of a classical `typedef`. Indeed, not even type identity is a sufficient condition to ensure *substitutability*: The presence and the accessibility of a suitable copy constructor are important factors.

For example, *substitutability* issues surrounding `std::auto_ptr<>` are materially affected by the `auto_ptr`'s `const`ness:

```
1 //                                  Listing 4
2 typedef  std::auto_ptr<int>  AP;
3 AP p1(new int);
4 AP p2 = p1;  // okay:  is_subst(AP,AP)

6 typedef  const std::auto_ptr<int>  CAP;
7 CAP p3(new int);
8 CAP p4 = p3;  // oops:  ! is_subst(CAP,CAP)
9 AP  p5 = p3;  // oops:  ! is_subst(CAP,AP)
```

This result follows from the lack of any `auto_ptr` copy constructor that can bind to a `const auto_ptr`. (The example's `typedef`s neither aid nor hinder these semantics.)

While not always intuitive, this state of affairs is a direct consequence of our definition of *substitutability*. We are not proposing any change to the current behavior. Therefore, our proposal to introduce a C++ *opaque typedef* can be viewed as a new opportunity for *substitutability*, but one that is subject to existing semantic constraints.

6 Overload resolution

To permit the above-described overloading behavior based on `public typedef`, we propose to introduce into C++0X's implicit conversion sequences the notion of a *substitution Conversion*.

Much like the *derived-to-base Conversion* described in [ISO03, §13.3.3.1/6], a *substitution Conversion* "exists only in the description of implicit conversion sequences." We propose the following wording be added to this paragraph: "When the parameter has an *underlying-type* and the argument expression has a *public-type*, the implicit conversion sequence is a *substitution Conversion* from the *public-type* to the *underlying-type*. A *substitution Conversion* has Conversion rank."

Additionally, we propose to augment the second and the third bullets of [ISO03, §13.3.3.2/4], which paragraph provides the rules to distinguish two conversion sequences with the same rank.

- Revise the second bullet so as to read: "If class `B` is derived directly or indirectly from class `A`, or if type `B` is a *public-type* whose *underlying-type* is `A`, . . . ".
- Similarly revise the third bullet to become: "If class `B` is derived directly or indirectly from class `A` and class `C` is derived directly or indirectly from class `B`, or if type `B` is a *public-type* whose *underlying-type* is `A` and type `C` is a *public-type* whose *underlying-type* is `B`, . . . ".

Alternatively, we could duplicate the entirety of bullets two and three as new bullets four and five, and make substitutions in the introductory clauses of the new bullet items instead of the additions proposed above. In any event, we respectfully recommend that all future additions and revisions to [ISO03] avoid use of anonymous bullets in order to simplify future references.

7 The *return type* issue

One of the consistent stumbling blocks in the design of an *opaque typedef* facility for C++0X has involved the *return type* of a function[1] selected via overload resolution in the presence of an argument whose type is declared via an *opaque typedef*. While we had previously proposed that the *return type* be determined via a form of type substitution, we now believe that this is not a viable approach.

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*, and sometimes a distinct third type is the desired *return type*. Indeed, sometimes the operation should be disallowed, and so there is no correct *return type* at all.

Since only the provider of the *opaque-type* is in a position to know the desired behavior, we now propose that the function's result in every case be returned as the type originally specified by the function selected by overload resolution, and that any other desired *return type* be specifically provided by a suitably overloaded version of the function.

This is also consistent with the behavior of `typedef private`. Since in this case there is no *substitutability* with respect to the *underlying-type*, the user must in all cases provide overloaded functions to obtain whatever behavior is desired in the user context.

8 Expressions

8.1 Overloaded operators

We propose to augment [ISO03, §5/2] so as to read:

> Operators can be overloaded, that is, given meaning when applied to expressions of class type (clause 9) or enumeration type (7.2) or *opaque-type*.

8.2 Static casting

Analogous to [ISO03, §5.2.9/7], we propose to permit explicit static casts by adding:

> A value of a type `T` can be explicitly converted to an *opaque-type* for which `T` serves as a direct or indirect *underlying-type*. The value is unchanged.

8.3 Dynamic casting

We also propose to permit dynamic casting, where applicable. The second sentence of [ISO03, §5.2.7/1] would be augmented so as to read:

> `T` shall be a pointer or reference to a complete class type or to an *opaque-type* whose direct or indirect *underlying-type* is a complete class type, or shall be "pointer to *cv* `void`".

Further, [ISO03, §5.2.7/2] would now begin:

[1] For purposes of this discussion, we distinguish neither between functions and operators, nor between a function's arguments and an operator's operands.

> If `T` is a pointer type, `v` shall be an rvalue of a pointer to complete class type or to an *opaque-type* whose direct or indirect *underlying-type* is a complete class type, and the result is an rvalue of type `T`. If `T` is a reference type, `v` shall be an lvalue of a complete class type or of an *opaque-type* whose direct or indirect *underlying-type* is a complete class type, and the result is . . . .

To describe the augmented behavior of the `dynamic_cast` operator, we propose to insert the following new paragraph into [ISO03, §5.2.7]:

> If `T` is an *opaque-type* `OT` or is a pointer or reference to an *opaque-type* `OT` such that `OT`'s *underlying-type* `UT` is a complete class type, then the `dynamic_cast` shall be carried out pursuant to this description as if `UT` had been specified in place of `OT`. and then that result `static_cast` to the required result type.

9 Summary and conclusion

This paper has continued discussions on the topic of *opaque typedef*s for C++0X. It has presented a number of important details regarding the behavior of a mechanism, including preliminary wording. We would be pleased to receive feedback regarding this proposal in order to determine whether these directions meet the perceived *desiderata* underlying the historical and continuing requests for an *opaque typedef* facility in C++0X. Assuming the proposed directions are of continued interest, we would ask for additional feedback in the form of specific citations to paragraphs of [ISO03] in need of updating to accommodate the new facility.

10 Acknowledgments

I am pleased to acknowledge my Fermilab colleagues Mark Fischler, Jim Kowalkowski, and Marc Paterno for significant ongoing discussions on this topic, and for their comments on early drafts of this paper. Additionally, Richard Brown's very careful proofreading and helpful suggestions materially improved the clarity of this paper's presentation.

I also wish to thank the Fermi National Accelerator Laboratory's Computing Division, sponsor of our participation in the C++ standards effort, for its past and continuing support of our efforts to improve C++ for all our user communities.

A Nomenclature describing `typedef` declarations

In describing the syntax and semantics of `typedef`s and other declarations, C++03 [ISO03] uses the following terminology:

declarator — That part of a declaration responsible for declaring "a single object, function, or type." In general, a single declaration may incorporate one or more *declarator*s.

type-specifier — That part of a declaration denoting a type common to (*i.e.*, shared by) all *declarator*s within that declaration.

declarator-id — A name introduced by a declaration, specified as part of a *declarator*. A valid `typedef` declaration must include at least one *declarator-id*, else the declaration would be pointless.

operator — A symbol or sequence of symbols used (singly or in appropriate combination) to adjust ("modify") a declaration's *type-specifier*. Representative examples include `&`, `* const`, and `[...]`.

For descriptive purposes, we augment this standard nomenclature with the term:

underlying-type — The synthesized type that is obtained from the type denoted by a *type-specifier* after adjusting ("modifying") that type with all *operator*s associated with a *declarator-id*.

The above nomenclature is applicable to all declarations. For example, in the C++03 declaration:

```
1 typedef  int  I,
2               * P,
3               A[5],
4               * const CP;
```

`int` fills the role of *type-specifier*; it is shared by four *declarator*s, each consisting of a distinct *declarator-id* and an associated *underlying-type*:

declarator	***declarator-id***	***operator***	***underlying-type***
`I`	`I`	none	`int`
`* P`	`P`	`*`	`int *`
`A[5]`	`A`	`[5]`	`int [5]`
`* const CP`	`CP`	`* const`	`int * const`

Finally, we introduce the following terms specifically for use in connection with *opaque typedef*s:

opaque-type — A type that is distinct from, yet isomorphic to, a declaration's *underlying-type*, originating via an *opaque typedef* and associated with a specific *declarator-id* within that `typedef`. The properties of an *opaque-type* depend on the kind of *opaque typedef* with which it was declared, as detailed throughout this paper.

public-type — An *opaque-type* declared via a `public typedef` declaration.

private-type — An *opaque-type* declared via a `private typedef` declaration.

Bibliography

[Aus05] Matt Austern. (Draft) technical report on standard library extensions. Paper N1836, JTC1-SC22/WG21, June 24 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf; same as ANSI NCITS/J16 05-0096.

[Bro04] Walter E. Brown. Toward opaque typedefs in C++0x. Paper N1706, JTC1-SC22/WG21, September 10 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1706.pdf; same as ANSI NCITS/J16 04-0146.

[ISO98] *Programming Languages — C++, International Standard ISO/IEC 14882:1998(E).* International Organization for Standardization, Geneva, Switzerland, 1998. 732 pp. Known informally as C++98.

[ISO03] *Programming Languages — C++, International Standard ISO/IEC 14882:2003(E).* International Organization for Standardization, Geneva, Switzerland, 2003. 757 pp. Known informally as C++03; a revision of [ISO98].