Toward Opaque `typedef`s in C++0X

Document #: WG21/N1706 = J16/04-0146
Date: September 10, 2004
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

One's mind, once stretched by a new idea, never regains its original dimensions.

— Oliver Wendell Holmes

1 Introduction

The notion of "opaque `typedef`s" is a recurring entry ([SV03, item ES072], [SV04, item ES072], [VSS04, item 65]) on the Evolution Working Group's issues lists. Each recurrence includes the annotation, "There have been explicit requests for such a mechanism for user defined types. . . ."

In this paper, we present a preliminary exploration of this issue.

2 Transparent types

It is a common programming practice to use one data type directly as an implementation technique for another. This is facilitated by the C++ `typedef` facility, allowing the programmer to provide a synonym or *alias* (the name of the desired new type) for the existing type that is being used as the underlying implementation. In the standard library, for example, `size_t` is an alias for a native `unsigned` integral type; this provides a convenient and portable means for user programs to make use of an implementation-selected type that may vary across platforms.

In traditional C and C++, `typedef` can be characterized as a *transparent* type facility. That is, while a new type name is introduced by a `typedef` declaration, is does not denote a new type. In particular, instances of the ostensibly new type can be freely substituted for instances of the underlying implementation type, *and vice versa*.

This *de facto* type identity has significant drawbacks. In particular, because the types are freely interchangeable (*mutually substitutable*), functions may be applied to arguments of either type even where it is conceptually inappropriate to do so. The following example provides a framework to illustrate such generally undesirable behavior. The new typenames make clear the intent: to `penalize` a given `game_score` and to ask for the `next_id` following a given `serial_number`.

```
//                                Listing 1
typedef  unsigned  game_score;
game_score  penalize( game_score n )
{
  return (n > 5u) ? n - 5u
                  : 0u;
}

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

Intentions to the contrary notwithstanding, the use of `typedef`s 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 `game_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 substituted, deliberately or accidentally, for an instance of either of the other types.

We see the results of such type confusion even among 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 a `typedef` 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 when a different compiler and library are installed, because the replacement library uses as its `iterator` implementation some sort of `struct`, a choice generally incompatible with the user's now-hardcoded pointer type.

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 `typedef`'d 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 `typedef`'d to `double` and so indistinguishable from each other. 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.

In short, what is often wanted is a variant of `typedef` that:

- introduces a genuinely new type as well as a new type name, and
- allows an existing type's operations to be applied to instances of the new type, but

- does not allow mutual substitutability.

3 Opaque types

To help address this situation, we now explore *opaque* types for possible introduction into C++0x.

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.

3.1 `typedef public`

This first form of opaque `typedef` is modeled after the behavior of two features of today's C++: `enum`s and `public` inheritance. Both these features induce a relationship between two designated types. Generally termed the *is-a* relationship, its primary characteristic of interest (for the moment) is the substitutability that it permits among instances of the types.

In the case of `public` inheritance, instances of the *derived* type may stand in (*be substituted*) where instances of the *base* type are expected. Similar behavior manifests in the relationship between an `enum` type and its integral underlying type: via an *integral promotion*, a value of `enum` type may stand in for an instance of the underlying type.

We propose to generalize such relationships, using a small addition to existing `typedef` syntax as illustrated below:

```
//                                Listing 2
typedef public  unsigned  game_score;
game_score  accumulate( game_score g1, game_score g2 )
{
  return g1 + g2;
}

typedef public  unsigned  serial_number;
serial_number  next_id( serial_number n )
{
  return static_cast<serial_number>(n + 1u);
}
```

The semantics of the proposed *public typedef* would permit substitutability in that instances of the newly declared type may stand in for instances of the original type, but would not permit instances of distinct types to substitute for each other, even when both are public typedefs for the same underlying type. (Copying operation, of course, would continue to obey rules akin to today's: copying a derived to a base is okay, but not conversely.) Thus, continuing the above example, no `serial_number` may stand in for an expected `game_score`, nor conversely. However, instances of either may stand in for an expected `unsigned`, since that is their common underlying type.

Operations defined on the underlying type would be directly applicable to instances of the newly declared type provided all operands matched in type. When operand mismatches arise, a common type is sought, according to the following rule sketch, from the intersection of the operand types and their underlying types (and the underlying types, if any, of the underlying types, *etc.*):

- If the intersection is empty, the expression is ill-formed.

- Otherwise, each type in the intersection is assigned a score by counting and summing the number of substitutions required to obtain it from each operand type. The single type, if any, that has the lowest score is used as the common type, and any operators are applied as if to this common type. If there is no single type with the lowest score, the expression is ill-formed.

Application of this algorithm is illustrated in the expressions that constitute the bodies of the two functions above:

- Summing two `game_score`s, the operator is applied directly to the common type.
- Incrementing a `serial_number` by an `unsigned` amount is done at the level of the common type, `unsigned`.

However, determining the result type of such operations can be subtle when the operands are `typedef`s for native types. For example, when subtracting values of identical numeric types, the result type matches the common operand type. However, when subtracting values of identical pointer types, the result type must be a `ptrdiff_t`, an alias for an implementation-defined signed integer type.

We would propose to apply a rule akin to the existing rules governing the type substitution that takes place after function template argument type deduction has succeeded: In that context, all uses of the deducible type, whether or not in a deducible context, are replaced by the deduced type. Our variant, in our context, would replace all uses of the operand type, including the return type if applicable, by the common type determined as outlined above.

3.2 `typedef private`

This second form of `typedef` is modeled on the behavior of `private` inheritance in today's C++ and is intended to fill the perceived need for a non-substitutable type. As before, operations defined on the underlying type would be directly applicable to instances of the newly declared type provided all operands matched in type. However, should operand mismatches arise, no common type is sought. Thus, types declared via a *private typedef* may not interact with their underlying types. The following code is illustrative:

```
//                                Listing 3
unsigned  accumulate( unsigned u1, unsigned u2 )        // (1)
{
  return u1 + u2;
}

typedef private  unsigned  game_score;
game_score  accumulate( game_score g1, game_score g2 )  // (2)
{
  return g1 + g2;
}

unsigned    u;
game_score  s;

accumulate( u, u );  // ok; calls (1)
accumulate( s, s );  // ok; calls (2)
accumulate( s, u );  // error
accumulate( u, s );  // error
```

If, in this illustration, the `typedef`s had been `public` rather than `private`, both erroneous calls would have resolved to function (1) since a `game_score` instance could have been supplied where an `unsigned` was expected, but not conversely.

4 A brief analysis of an early example

In each of the previously-cited issues lists, we find the following notional code (slightly reformatted from the original), illustrating some of the *desiderata*, as well as some of the perceived issues, regarding opaque `typedef`s:

```
//                              Listing 4
real typedef  int  A;
real typedef  int  B;

A  a, a2;
B  b;

a = b;       // error
a = a2;      // ok
a = a + b;   // ???
a = a + a2;  // ???
```

We now briefly analyze slightly extended versions of this example in light of our proposed `typedef` variants.

Treating the opaque (`real typedef`) declaration as a public typedef produces:

```
//                              Listing 5
typedef public  int  A;
typedef public  int  B;

A    a, a2;
B    b;

a = b;          // error:  the rhs B can't substitute for the lhs A
a = a2;         // ok
a = a + b;      // error:  the rhs int can't substitute for the lhs A
a = a + a2;     // ok

int  i;

i = a;          // ok
i = b;          // ok
a = i;          // error:  the rhs int can't substitute for the lhs A
b = i;          // error:  the rhs int can't substitute for the lhs B
```

However, treating the opaque (`real`) `typedef` as a private typedef produces:

```
//                                  Listing 6
typedef private  int  A;
typedef private  int  B;

A    a, a2;
B    b;

a = b;          // error:  the rhs B can't substitute for the lhs A
a = a2;         // ok
a = a + b;      // error:  an A and a B share no common type to sum
a = a + a2;     // ok

int  i;

i = a;          // error:  the rhs A can't substitute for the lhs i
i = b;          // error:  the rhs B can't substitute for the lhs i

a = i;          // error:  the rhs int can't substitute for the lhs A
b = i;          // error:  the rhs int can't substitute for the lhs B
```

Both our proposals are thus consistent with the explicitly-desired behavior, while producing slightly different results in the scenarios whose behavior had been left open. In each case, explicit casting operations may be inserted to affect the validity of the code. Rules for such casts remain an open issue at the moment, but it seems that an explicit `static_cast` should be permitted to "downcast"; that is, to cast from an underlying type to a "derived" opaque type.

5 A concluding example

The following code sample exemplifies some of the expected behaviors when multiple levels of the proposed `typedef`s are involved:

```
//                                  Listing 7
typedef private double                    mass1_leng2_per_time2;

typedef public  mass1_leng2_per_time2   energy;

typedef public  energy                    kinetic_energy;
typedef public  energy                    potential_energy;
typedef public  energy                    heat_energy;

double            d;
energy            e;
kinetic_energy    k;
potential_energy  p, q;

e = k + p;         // ok
p = p + q;         // ok
k = p + q;         // error
d = std::sqrt(e);  // error
```

6 Summary and conclusion

This paper has opened discussions on the topic of opaque `typedef`s for C++0x. It has presented the outline of a mechanism that encompasses and unifies existing C++ behaviors, namely `enum`s and inheritance, and has extended these notions in desirable directions along two axes, herein known as *public typedefs* and *private typedefs*. We would be pleased to receive feedback regarding these draft proposals in order to determine whether the outlined directions meet the perceived *desiderata* underlying the historical and on-going requests for an opaque `typedef` facility in C++0x.

7 Acknowledgments

I am pleased to acknowledge, with deep appreciation, my Fermilab colleague and friend Marc Paterno. His steadfast adherence to important principles even in the face of expediency has been a constant reminder that there are still people in the world who believe in doing a job the right way.

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. Finally, many thanks are also due to each of the reviewers (you know who you are!) for due vigilance and diligence under extreme time pressure.

Bibliography

[SV03] Bjarne Stroustrup and David Vandevoorde. Evolution WG issues list. Paper N1512, JTC1-SC22/WG21, September 18 2003. Online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1512.html`; same as ANSI NCITS/J16 03-0095.

[SV04] Bjarne Stroustrup and David Vandevoorde. Evolution WG issues list. Paper N1598, JTC1-SC22/WG21, February 14 2004. Online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1598.html`; same as ANSI NCITS/J16 04-0038.

[VSS04] Daveed Vandevoorde, Bjarne Stroustrup, and Herb Sutter. C++ evolution working group — active proposals, revision 1. Paper N1650, JTC1-SC22/WG21, April 15 2004. Online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1650.html`; same as ANSI NCITS/J16 04-0090.