Toward a Proposal for *Object Templates* in C++0x

Document number: WG21/N1785 = J16/05-0045
Date: 2005-04-11
Revises: N1586 = 04-0026 (but omitted from mailing)
Project: Programming Language C++
Reference: ISO/IEC IS 14882:2003(E)
Reply to: Walter E. Brown <wb@fnal.gov>
Mail Station 234
CEPA Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

Contents

1 Changes from N1586 **2**

2 Purpose, background, overview **2**

3 Motivating example **2**

4 Working syntax and notional semantics **4**

5 Are *Object Templates* worthwhile? **5**

A Function templates instead of *Object Templates* . . . 5

B But it's just syntactic sugar, right? . . . 5

C Do we need *Object Templates* if we have ICFs? . . . 6

6 Representative issues regarding the *template* part of *Object Templates* **6**

A Can an *Object Template* have more than one template parameter? . . . 6

B Can an *Object Template* be separately declared and defined? . . . 6

C Will *Object Templates* be exportable? . . . 6

D Can an *Object Template* be specialized? . . . 6

E Can we have partial specialization of *Object Templates*? . . . 7

F Are *Object Template* parameters limited to type parameters? . . . 7

G Do *Object Templates* participate in template argument deduction? . . . 7

7 Representative issues regarding the *object* part of *Object Templates* **7**

A What is the lifetime of an *Object Template* instantiation? . . . 7

B Can we have *Object Template* instantiations that are mutable? . . . 7

C How is linkage determined? . . . 7

D What about initialization semantics? . . . 8

8 Conclusion **8**

9 Acknowledgments **8**

Bibliography **8**

1 Changes from N1586

N1586 was originally written 2004-03-05, but was accidentally omitted from mailings. It has been issued a new document number, but aside from the addition of this note and some slight reformatting, this paper is identical to the original.

2 Purpose, background, overview

This paper, aimed at C++ 0x, is intended to motivate and articulate a proposed language feature we have termed *Object Templates*. Informally, we view *Object Templates* as a syntactic generalization of static data members in the context of a class template.

Based on user feedback, several years of experimental effort to achieve the desired effect via current language syntax have produced generally unacceptable results. This has led us to conclude that it is now necessary to explore core language enhancement as a means of obtaining the desired functionality via a straightforward syntax. We believe, however, that existing core language compiler technology is very likely sufficient to implement *Object Templates*.

To begin, we present an extended motivating example in §3. Along the way, we describe a well-known programming idiom whose benefits are essentially universally acknowledged, that is nearly ubiquitous in use, and that can be cleanly generalized via *Object Templates*.

We will then exhibit, in §4, an important candidate syntax and describe its intended notional semantics via an *as-if* approach. Section 5 will discuss the perceived value of the *Object Template* feature as (at least) a notation that provides considerable convenience, economy, and clarity of expression.

In the two sections that then follow, we address (at varying levels of detail) a representative sampling of technical issues related to *Object Templates*: §6 is about issues primarily concerned with the *object* part of *Object Templates*, while §7 is devoted to topics mainly centered on the *template* part of *Object Templates*. Sections 8 and 9 complete the paper with concluding remarks and acknowledgments, respectively.

This paper is generally intended as an exploratory document. It is certainly not a comprehensive technical treatise, nor is it a formal proposal. Our purpose is solely to inaugurate and stimulate discussion exploring interest in and feasibility of *Object Templates*. We therefore respectfully request that our readers provide us their feedback in order that we may determine how next to proceed.

3 Motivating example

According to an oft-quoted early Fortran manual for Xerox computers:

> The primary purpose of the `DATA` statement is to give names to constants; instead of referring to pi as `3.141592653589793` at every appearance, the variable `Pi` can be given that value with a `DATA` statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of pi change.

While humorously expressed — after all, how can π, a constant of nature, change? — programmers have come to understand that different environments can easily require different numeric approximations to π. Porting a program to a new architecture, for example, may provide an opportunity for increased computational precision, requiring additional significant digits to achieve.

In addition, there are constants of nature whose values are subject to reconsideration from time to time. For example, some constants' values are determined only by analysis of experimental data. Over time, as experiments improve their measurements of physical phenomena, these constants' generally-accepted values are adjusted in the literature[1] so as to reflect the improved understanding that results from better experimental data. Maintenance of software that employs such constants therefore often includes tracking these constants' values as their accuracy improves over time.

Thus, it has long been considered good coding practice and style to employ named constants in lieu of literals. As above, typical rationale for this recommended practice (denoted, in some contexts[2], as *manifest constants*) cites such benefits as:

- **Clarity of exposition**: making the code's intent more obvious to a reader;
- **Consistency of use**: ensuring a common value is used throughout the code; and
- **Ease of maintenance**: requiring adjustment, when needed, to but a single specification of the desired value.

Even a simple C++ function to calculate the area of a circle can profit from the application of this technique:

```
1 double  area_of_circle( double radius )  {
2   return pi * radius * radius;
3 }
```

How could the non-local name `pi` have been made available? Several straightforward possibilities are readily apparent in C++ (formatting selected to emphasize similarities):

```
1 #define               pi    3.1415926
2 double const          pi  = 3.1415926;
3 static  double const  pi  = 3.1415926;
```

However, each of these is problematic, in the same way, in the context described below.

To set the scene, let us overload our area-computing function for additional types of its `radius` parameter:

```
1 float  area_of_circle( float radius )  {
2   return pi * radius * radius;
3 }
4
5 long double  area_of_circle( long double radius )  {
6   return pi * radius * radius;
7 }
```

If, as shown, all overloads share a single instance of `pi`, then two of the three overloads may well incur the cost of a cast or two, no matter which technique was used to declare and define `pi`. Further, depending on the type of that single instance of `pi`, one or two of the overloads may yield a result with fewer bits of accuracy than otherwise possible.

If each overload were instead provided a `pi` object whose type matched the type of the function's parameter, then no casting would be needed. This approach represents one possible trade-off between performance and computational accuracy. However, we now require additional names in order to refer to the `pi`'s of the various desired types. One possible approach to

[1] For example, the Committee on Data for Science and Technology (CODATA) has recently introduced the "2002 set" of self-consistent values of basic constants and conversion factors. These values replace the "1998 set"; see [CODATA] for details.

[2] BCPL, anyone?

selecting such names follows the naming convention of many of the functions in the C portion of the C++ standard library: use a canonical name (here, `pi`) for the `double` version, and attach distinct suffixes to denote the `float` and `long double` versions:

```
static  float       const  pif  = 3.14159F;
static  double      const  pi   = 3.1415926;
static  long double const  pil  = 3.141592653589793L;
```

But now suppose we wish to provide a single generic computation, rather than a family of overloaded functions. While it seems straightforward to express most of this in the form of a function template, the desire to employ a `pi` whose type matches the deduced function template parameter first suggests we write:

```
template< class T >
T  area_of_circle( T radius )  {
  return static_cast<T>(pi) * radius * radius;
}
```

Because this approach uses a single value of `pi` in all its instantiations, it encounters the performance and accuracy issues described above. If, however, we could provide specializations of `pi` (*e.g.*, `pi<float>`, `pi<double>`, *etc.*) to accommodate each intended template parameter `T`, we could write:

```
template< class T >
T  area_of_circle( T radius )  {
  return pi<T> * radius * radius;
}
```

In such a context, `pi<T>` exemplifies use of an *Object Template*. The next section describes this feature's intended semantics.

4 Working syntax and notional semantics

We envision that (1) an *Object Template* named `pi` may be defined (for example) via such syntax as:

```
template< class T = double >
T const  pi  = 3.1415926L;
```

and that (2) this would correspond in meaning to:

```
template< class T = double >
struct pi_constant  {
  static  T const  value;
};

template< class T >
T const  pi_constant::value  = 3.1415926L;
```

such that (3) in subsequent usage,

1. any mention of the *Object Template* `pi` *in that form* would be the semantic equivalent of mentioning `pi_constant<>::value` (both cases implying the default template argument(s), else error), and
2. any mention of the *Object Template* in the form `pi<`*argument(s)*`>` would be the semantic equivalent of mentioning `pi_constant<`*argument(s)*`>::value`.

We would hope to permit *Object Template* specialization and explicit instantiation in what seem to be the obvious ways:

```
template<>  float const  pi<float> = 3.14159F;  // specialization

template  pi<long double>;  // explicit instantiation
```

5 Are *Object Templates* worthwhile?

A Function templates instead of *Object Templates*

It may have occurred to the reader that function templates can mimic most *Object Template* functionality:

```
template< class T = double >
T  pi( )  {  // read-only
  static T const  pi(3.141592653589793L);
  return pi;
}

cout << pi<float>();  // use
```

The above code demonstrates one implementation for a read-only version. A read-write variant, if needed, would follow identical logic but (a) return a reference to a (b) local non-`const static` object:

```
template< class T = double >
T &  adjustable_pi( )  {  // read-write
  static T  adjustable_pi(3.141592653589793L);
  return adjustable_pi;
}

cin >> adjustable_pi<float>();  // use
```

It is important, however, to explore how to use the result of such an approach. Because instantiation of a function template produces a function, obtaining access to the function's embedded value would require the syntax of a function call. Thus, under current language syntax rules and as illustrated above, parentheses are required to designate the function-call operator.

However, repeated (informal) user surveys of a representative programmer community clearly demonstrate that, in our context for our intended use, this requirement for parentheses to perform a straightforward access to what is perceived as an ordinary constant (read-only case) or a straightforward variable (read-write case), is at best deemed "unnatural" and is at worst considered to be "odious." Even though a constant can certainly be mathematically modeled via a niladic function, programmers' mindsets evidently do not permit easy application of such a model to their coding practices.

B But it's just syntactic sugar, right?

Certainly. However, based on the above-described surveys, it appears to be syntactic sugar that is *e*xtremely important to users: Not only does it meet their expectations, the notation provides considerable convenience, economy, and clarity in expressing a programming idiom (manifest

constants) that is both in common (near-ubiquitous) use and is highly recommended for its well-known benefits.

C Do we need *Object Templates* if we have ICFs?

In a companion paper [N1611], we proposed a feature we had tentatively named *Implicitly-Callable Functions* (ICFs, for short). In brief, an ICF is a niladic function that is called without the traditional parentheses denoting function call syntax. Such a function call would be indistinguishable in appearance from use of an object.

While we believe that ICFs do represent a viable and useful approach, we also believe they do not preclude the incorporation of *Object Templates*.

6 Representative issues regarding the *template* part of *Object Templates*

A Can an *Object Template* have more than one template parameter?

Yes. However, it's non-obvious whether such a capability is particularly useful. By its nature, an object has a single type. It seems most natural that this type serve in the role of the template's parameter. It is unclear that additional template parameters have any significant role to play. However, there also seems to be no reason to forbid multiple *Object Template* template parameters.

B Can an *Object Template* be separately declared and defined?

Yes; the usual cppcodeextern keyword is applicable in doing so:

```
1 template< class T = double >  extern  T const  pi;
2 // elsewhere:
3 template< class T = double >  T const  pi  = 3.1415926L;
```

Separating an *Object Template*'s declaration from its definition may be useful in connection with `export`.

C Will *Object Templates* be exportable?

Yes.

D Can an *Object Template* be specialized?

Yes. This notion was introduced in §4 above. Here are some additional examples:

```
1 template<> float       const  pi<float      >  = 3.14159;
2 template<> double      const  pi<double     >  = 3.1415926;
3 template<> long double const  pi<long double>  = 3.1415926L;
```

E Can we have partial specialization of *Object Templates*?

There's no obvious reason to forbid such a feature. However, see §6.A regarding multiple template parameters.

F Are *Object Template* parameters limited to type parameters?

We see no inherent reason for such a limitation. However, we have explored the utility of neither non-type nor template template parameters in conjunction with the *Object Template* feature.

G Do *Object Templates* participate in template argument deduction?

No. While the *type* of an instantiated *Object Template* can, of course, participate in template argument deduction, the object itself may not: the use context is non-deducible.

7 Representative issues regarding the *object* part of *Object Templates*

A What is the lifetime of an *Object Template* instantiation?

Since an *Object Template* instantiation is an object, it must have a lifetime. Consistent with the model of a class' static data member, the intended lifetime of an instantiated *Object Template* is static. That is, if defined at namespace scope, the object comes into existence and is initialized before `main()` is called, and is terminated after `main()` exits. If defined at block scope, its lifetime begins if and when it is first encountered during execution, and terminates after `main()` exits.

B Can we have *Object Template* instantiations that are mutable?

Yes. While it's not obvious to what extent mutable instantiations are useful, it's also non-obvious that such a feature ought to be prohibited.

C How is linkage determined?

The following table summarizes one possible set of rules for deciding the linkage of an object template. Note that the type of an object that results from an *Object Template* instantiation does not affect the object's linkage, even if such type is `const`-qualified.

Object Template's declaration scope	declared `static`?	declared `extern`?	resulting linkage
block	—	yes	external
	—	no	none
global or named namespace	yes	—	internal
	no	—	external
unnamed namespace	—	—	internal

It seems important that all instantiations of an *Object Template* with external linkage are to be deemed identical whenever their corresponding template arguments are identical. Linkers are

expected to strip out such duplicates in much the same way that they today strip out duplicate function template instantiations.

D What about initialization semantics?

Since the intended *Object Template* initialization semantics match the initialization semantics of class templates' static data members, there appear to be no new issues with respect to initialization.

8 Conclusion

Our primary goal in writing this paper was to present a use case for a new language feature that we have termed *Object Templates*. We view this feature as a logical extension of concepts and features already supported by contemporary C++, and believe *Object Templates* represents an important direction along which C++ 0x might be usefully enhanced.

In so doing, we have given significant weight to the consistent input we have received over many years as we surveyed respected professional colleagues regarding the utility and significance of the underlying concept. We also presented, as a working syntax for *Object Templates*, the precise manner in which these same colleagues have expected the feature to be used in their generic code. We did so because we concur with their judgment that this notation provides "convenience, economy, and clarity of expression."

While we additionally explored (as a "notional semantics") one particular avenue by which *Object Templates* might be provided, we did so in the interest of stimulating discussion toward such a new language feature. We would be pleased to obtain consider alternative means of obtaining C++ support for *Object Templates*, and look forward to useful feedback on this and related matters.

9 Acknowledgments

We are grateful to the many individuals who (a) have helped us, over many years, to refine the ideas underlying *Object Templates*, and (b) have reviewed this document's expression of those ideas. While there are too many to name individually, we do wish to acknowledge, with special thanks, the contributions of Marc Paterno, Bjarne Stroustrup, and David Vandevoorde.

We also appreciate the support of the Fermi National Accelerator Laboratory's Computing Division, sponsors of our participation in the C++ standards effort.

Bibliography

[N1611] Brown, Walter E.: *Implicitly-Callable Functions in C++0x*. WG21/N1611 (same as J16/04-0051). 17 February 2004. Available at http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2004/n1611.pdf.

[CODATA] Mohr, P. J. and B. N. Taylor: *The 2002 CODATA Recommended Values of the Fundamental Physical Constants, Web Version 4.0*. National Institute of Standards and Technology, Gaithersburg, MD 20899, 9 December 2003. Available at http://physics.nist.gov/constants.