Introducing Object Aliases

Document #: WG21 N3552
Date: 2013-03-12
Revises: None
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

1 Introduction

A number of years ago, [Bro05] explored a prospective language feature then known as *Object Templates*. That feature died on the vine due to healthy skepticism voiced by EWG, as well as the usual lack of cycles to develop the idea much further. However, the original underlying issue is still present in C++11, after even many additional years of experimental effort to achieve the desired effect via C++03 or (more recently) C++11 syntax.

Fortunately, C++11 also points the way toward a more general solution, well beyond that envisioned over a decade ago. Accordingly, this paper incorporates and updates a significant fraction of the earlier [Bro05] in introducing a different proposed C++ feature. Inspired by the C++11 template alias for types, we will propose a template alias for objects, known in brief as an *object alias*.

2 Motivating examples

2.1 Manifest constants

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.

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 (and more plentiful) experimental data and from improved statistical techniques. 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:

```
auto  area_of_circle( double radius ) -> decltype(radius)  {
  return pi * radius * radius;
}
```

How could the non-local name **pi** have been defined? Several straightforward possibilities are readily apparent in C++03/11 (formatting selected to emphasize similarities):

```
#define                 pi   3.141592653589793
double const            pi = 3.141592653589793;
static double const     pi = 3.141592653589793;
static constexpr double pi = 3.141592653589793;
```

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:

```
auto  area_of_circle( float radius ) -> decltype(radius)  {
  return pi * radius * radius;
}

auto  area_of_circle( long double radius ) -> decltype(radius)  {
  return pi * radius * radius;
}
```

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 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:

[1] For example, the Committee on Data for Science and Technology (CODATA) most recently introduced the "2010 set" of self-consistent values of basic constants and conversion factors. These values replaced the "2006 set," which replaced the "2002 set," which replaced the "1998 set," etc.; see [CODATA] for details.

[2] BCPL, anyone?

```
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;
}
```

2.2 Constraints

More recently, [SS12] and its successor drafts [SS13a, SS13b] have proposed (as part of a broader effort) a new **requires** keyword for C++. According to [SS13a], "The requires clause is followed by a Boolean expression that evaluates predicates." These predicates are termed *constraints* in the proposal:

> A constraint is simply an unconstrained **constexpr** function template that takes no function arguments and returns **bool**. It is — in the most literal sense — a predicate on template arguments. This also means that the evaluation of constraints in a **requires** clause is the same as **constexpr** evaluation.

Among others, the paper gives the following (lightly reformatted) example of a simple constraint declaration:

```
template< typename T >
  constexpr bool  Equality_comparable();
```

The constraint can be called as any ordinary function template might. In the context of a **requires** clause, such a call is termed the *explicit form*. There is also a *shorthand form*, in which only the name of the constraint is used; the call is implied and is expanded by the compiler into the corresponding explicit form.

```
template< typename T >
  requires Equality_comparable<T>()  // explicit form
struct equal_to;

template< Equality_comparable T >  // shorthand form
struct equal_to;
```

Given an *object alias* facility as proposed herein, we could avoid the need for any call in such contexts. Since any call to a niladic function (or function template) can be obviated via a suitable *object alias*, we could have the following instead:

```
template< typename T >
  using auto Equality_comparable = ··· ;  // see next section

template< typename T >
  requires Equality_comparable<T>  // explicit form
struct equal_to;

template< Equality_comparable T >  // shorthand form (unchanged from above)
struct equal_to;
```

3 Working syntax and notional semantics

Given a definition such as the following:

```
template< class T = double >
struct pi_constant  {
  static constexpr T value = static_cast<T>(3.141592653589793L);
};
```

we envision that an *object alias* named **pi** may thereafter be defined as follows:

```
template< class T = double >
  using auto pi = pi_constant<T>::value;
```

such that in subsequent usage:

- each appearance of **pi** or **pi<>** would be the semantic equivalent of **pi_constant<>::value** (each implying the default template argument), and
- each appearance of **pi<***type***>** would be the semantic equivalent of **pi_constant<***type***>::value**.

If the requirements on **constexpr** functions are somewhat relaxed in the future, we would also expect the following alternative definitions to behave equivalently to those above:

```
template< class T = double >
constexpr T&  pi_constant( )  {
  static constexpr T value = static_cast<T>(3.141592653589793L);
  return value;
};
···
template< class T = double >
  using auto pi = pi_constant<T>();
```

The advantage of this latter approach would be that it is free of order-of-initialization issues in case one such value depends on another. In use, there is no discernable difference to the programmer.

As a final (preferred) alternative, if the feature were to allow the direct aliasing of constant expressions (á la manifest constants), our intent could be specified far more succinctly,[3] with no need for any function template:

```
template< class T = double >
  using auto pi = static_cast<T>(3.141592653589793L);
```

[3] It is unclear as of this writing whether **constexpr** may be a desirable or even a necessary part of the declaration.

In brief, it seems that **any call to a niladic function (or function template) can be obviated via a suitable *object alias***.

4 What's wrong with function templates?

There's nothing wrong with function templates. Niladic/nullary function templates certainly can mimic most *object alias* functionality:

```
template< class T = double >
T  pi( )  {  // read-only
  static T constexpr 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 would instead return a reference to a 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, oft-repeated user surveys of a representative programmer community clearly, consistently, and convincingly 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, many/most programmers' mindsets evidently do not permit easy application of such a model to their coding practices.

But it's just syntactic sugar, right?

Certainly. However, based on the above-described surveys, it appears to be syntactic sugar that is *extremely* 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.

However, despite some fifteen years of experimentation, no one (up to and including our most respected WG21 colleagues) has been able to make such `pi<T>` syntax work in C++. We believe this problem is well worth solving, and so we believe a new core language feature is called for.

5 Conclusion

Our primary goal in writing this paper was to present use cases, both old and new, for a new language feature that we have termed an *object alias*. We view this feature as a logical extension of concepts and features already supported by contemporary C++, and believe *object alias*es represents an important direction along which C++11 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 notion. We also presented, as a working syntax for *object alias*es, the precise manner in which these same colleagues have expected the feature to be used in their code. We did so because we concur with their judgment that this notation provides "convenience, economy, and clarity of expression."

This paper was generally intended as an exploratory document. Our purpose is solely to inaugurate and stimulate discussion exploring interest in and feasibility of *object alias*es. We therefore respectfully request that our readers provide us their useful feedback in order that we may determine how next to proceed.

6 Acknowledgments

We are grateful to the many individuals who have helped us, over many years, to refine the ideas underlying what has emerged as the *object alias*. Many thanks also to the readers of early drafts of this paper for their helpful feedback.

7 Bibliography

[Bro05] Walter E. Brown: "Toward a Proposal for *Object Templates* in C++0x." ISO/IEC JTC1/SC22/WG21 document N1785 (post-Lillehammer mailing), 2005-04-11. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1785.pdf.

[CODATA] Peter J. Mohr, Barry N. Taylor, and David B. Newell: "The 2010 CODATA Recommended Values of the Fundamental Physical Constants" (Web Version 6.0). Database developed by J. Baker, M. Douma, and S. Kotochigova. National Institute of Standards and Technology, Gaithersburg, MD 20899, 2011-07-22. http://physics.nist.gov/constants.

[DuT12] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3485 (post-Portland mailing), 2012-11-02. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf.

[SS12] Andrew Sutton and Bjarne Stroustrup: "Template Constraints." Undated pre-publication draft made available 2012-11. http://wiki.edg.com/twiki/pub/Wg21portland2012/EvolutionWorkingGroup/template-constraints.pdf (login required).

[SS13a] Andrew Sutton and Bjarne Stroustrup: "Concepts Lite: Constraining Templates with Predicates." Undated pre-publication draft posted 2013-02-08 to concepts study group reflector. https://groups.google.com/a/isocpp.org/forum/?hl=en-US&fromgroups=#!topic/concepts/qflNpHvvE90.

[SS13b] Andrew Sutton and Bjarne Stroustrup: "Concepts Lite: Constraining Templates with Predicates." Undated pre-publication draft posted 2013-02-21 to concepts study group reflector. https://groups.google.com/a/isocpp.org/forum/?hl=en-US&fromgroups=#!topic/concepts/J7YhOhD8-Bs.

8 Revision history

Revision	Date	Changes
1.0	2013-03-12	• Published as N3552.