Your Help Wanted:

Language Proposals in Flight

WALTER E. BROWN, PH.D.

<webrown.cpp @ gmail.com>

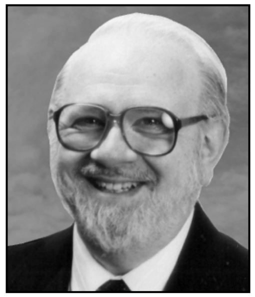
Abstract

- “Want to collaborate in designing and implementing a new feature for C++17? Then this session is for YOU!
- “After reviewing the process by which a new C++ feature enters the language, we will look at one or two of the speaker's proposals that have received early favorable review from the standards committee, and that are awaiting sample implementation and/or formal wording.
- “Attendee feedback will be solicited, and collaborators will be sought to help bring the proposal(s) to fruition.”

A little about me

- B.A. (math’s); M.S., Ph.D. (computer science).
- Professional programmer for almost 50 years, programming in C++ since 1982.
- Experienced in industry, academia, consulting, and research:
  - Founded a Computer Science Dept.; served as Professor and Dept. Head; taught and mentored at all levels.
  - Managed and mentored the programming staff for a reseller.
  - Self-employed as a software consultant and commercial trainer; international lecturer.
  - Retired from the Scientific Computing Division at Fermilab, specializing in C++ programming and in-house consulting.
- Not dead — still available for free-lance work. (Email me!)

Emeritus participant in C++ standardization



- Written 85+ papers for WG21, introducing such now-standard C++ library features as cbegin/cend and common_type, as well as the entirety of headers <random> and <ratio>.
- Heavily influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*.
- Conceived and served as Project Editor for ISO/IEC 29124 (Int'l Standard on Mathematical Special Functions in C++); now serving as an Associate Project Editor for C++17.
- Be forewarned: Based on my training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

Introducing ISO JTC 1 SC22 / WG21

- ISO: International Standards Organization
  - JTC1: Joint Technical Committee for Information Technology
  - SC22: Subcommittee for Programming Languages, their Environments, and System Software Interfaces
  - WG21: Working Group for C++
- ISO membership is open only to national standards bodies:
  - "Each member represents ISO in its country. **Individuals or companies cannot become ISO members**" [www.iso.org].
  - *E.g.*, BSI (British Standards Institution)
  - *E.g.*, AFNOR (Association Française de Normalisation)
  - *E.g.*, DIN (Deutsches Institut für Normung)
  - *E.g.*, Standards Council of Canada

ANSI INCITS / PL22.16

- ANSI: American National Standards Institute
  - INCITS: InterNational Committee for Information Technology Standards
    - "An ANSI-accredited forum of IT developers"
    - Formerly known as NCITS: National Committee for Information Technology Standards
    - Formerly known as Accredited Standards Committee X3
  - PL22: Technical Committee for Programming Languages
  - .16: Task Group for Programming Language C++
    - Formerly known as J16

Meeting attendees, February 2014, Issaquah, WA

(We had just finished work on C++14!)

Working arrangements

- Meetings of WG21 and .16 are co-located:
  - Held 2x-3x/year, as arranged by the WG21 Convener, all over the world; roughly half the meetings are in North America.
  - Each organization publishes its own minutes, and all formal votes are taken twice, if needed:
    - .16 first, with only its (U.S.) members voting.
    - WG21 second, with only national bodies voting.
- Informal consensus is reached before formal motions are brought to a vote:
  - Hence formal motions generally pass with no significant opposition (but there are exceptions!).
  - All participants share a strong commitment to cooperation.

Internal organization

- All meeting attendees (“technical experts”) work closely together for the common goal:
  - .16 and WG21; voting “members” and non-voting “observers”; famous (notorious?) and unknowns.
  - Nobody pays us to do this work — in fact, we pay! (Anyone want to underwrite a retiree?)
- Formally organized into:
  - Working Groups: CWG (core language), LWG (library), EWG (evolution), and LEWG (library evolution).
  - Study Groups appointed by the Convener to consider and report on specific topics; SG1 (concurrency) is the oldest.
  - Each Study Group reports to a Working Group per the topic.
- Agenda is always based on submitted papers and issues.

Plenary session (only half the room shown)

WG21's approach to C++ evolution

- General principles:
  - Preserve source compatibility.
  - Support novices ($n_{novices} \gg n_{experts}$).
  - Help programmers to write better programs:
    - Maintain or increase type safety.
    - Keep to the zero-overhead principle.
- Removing a feature is only rarely feasible:
  - Stability and compatibility are major concerns.
  - It's very hard to remove a feature without breaking somebody's code, but (in C++11):
    - Keyword auto was so rarely used that we gave it new semantics.
    - Library's auto_ptr was heavily used, but has inherent issues;
      we deprecated it and provide/promote unique_ptr instead.

Goals

- For the core language:
  - Make C++ easier to teach and learn.
  - Make the rules more general and more uniform.
  - Make C++ better for building libraries;
    prefer libraries over language extensions.
- For the standard library:
  - Improve support for generic programming
    and other programming paradigms.
  - Extend the library into new domains.
  - Apply new core language technologies.

A successful proposal needs …

- An expository paper:
  - Statement of issue to be addressed.
  - Motivation/rationale (*i.e.*, why is it worthwhile to address?).
  - Pertinent background information.
  - Description of proposed feature; discussion of alternatives.
  - Interactions with the rest of the language and library.
  - Implementation experience (“prior art”).
  - Wording to “specify” the proposed feature.
- Someone to present, defend, and generally shepherd the proposal, in person, through the various stages of review.
- Patience; it’s not uncommon to have favorable reviews, yet need to adjust a paper’s details several times.

A model paper: N3887

- Michael Park: *Consistent Metafunction Aliases*, 2013-12-26.
- A first-time contribution, which I was pleased to help edit and then to shepherd at the author’s request.
- Uniformly praised, during LEWG and LWG formal reviews, for its approach, its clarity, and its thoroughness.
- Adopted for C++14 by unanimous consent.
- A home run!

Papers "In Flight" (in various stages of consideration)

Doc #	Title	Status
N3339	"A … Deep-Copying Smart Pointer"	LEWG
N3840	"… World's Dumbest Smart Pointer"	LEWG
N3741	"… Opaque Typedefs …"	EWG
N3744	"… [[pure]]"	EWG
N3925	"A sample Proposal"	✓ TS
N3843	"A SFINAE-Friendly std::common_type"	TS→C++17
N3923	"A SFINAE-Friendly std::iterator_traits"	TS→C++17
N4061	"Greatest Common Divisor …"	→TS
N3928	"Extending static_assert"	→C++17
N3911	"… void_t"	LEWG

I’m seeking your help

- Implementations of new proposals are always useful:
  - Proof of concept to ensure it’s doable.
  - Experimental platform to fine-tune the new rules .
  - Let the world try it out before formally proposing it to WG21.
- How is your gcc- or clang-fu?
  - I haven’t touched a compiler’s internals since the 1980’s …
  - In any case, I’d rather play with my grandchildren …
  - But I want to develop my language designs in time for C++17.
- So please listen to my proposals, already in the pipeline:
  - Your feedback, today or later, will be welcome …
  - But your active assistance (as a future co-author) would be appreciated even more!

Your Help Wanted: Language Proposals in Flight

FIN

Walter E. Brown, Ph.D.

<webrown.cpp @ gmail.com>

Proposing `[[pure]]`

Document #: WG21 N3744
Date: 2013-08-30
Revises: N1664
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <`webrown.cpp@gmail.com`>

Contents

Abstract

Following significant prior art, this paper proposes a **`pure`** attribute to specify that a function or statement is free of observable side effects.

1 Background

Our nearly decade-old paper [Bro04] aimed to provide "...Improved Optimization Opportunities in C++0X" by proposing new function qualifiers **`nothrow`** and **`pure`**. Alas, that paper did not find favor with EWG at the time. However, hindsight strongly suggests that we were on the right track after all, because our proposed **`nothrow`** qualifier has turned out to be a precursor of the **`noexcept`** qualifier proposed five years later [GA09] and adopted for C++11.[1]

We believe it is time to revisit our other proposed annotation, **`pure`**. In doing so, we selectively borrow from relevant parts of our earlier paper because the rationale, expectations, benefits, and prior art seem as applicable today as they were in 2004. However, we adapt our proposal to take advantage of C++11 *attribute* syntax.

2 Well- and ill-behaved functions

A free function or a member function is described as *well-behaved* or, equivalently, as *pure*[2] if it:

1. communicates with client code solely via the function's argument list[3] and return value, and
2. is incapable of observable side effects.

[1] We had even proposed functionality, then in the form of a trait named **`isnothrow`**, that C++11 has since incarnated in the form of a *noexcept-expression* [expr.unary.noexcept].

[2] See http://en.wikipedia.org/wiki/Pure_function as of 2013-07-10. This *pure* nomenclature is of long standing, so we have adopted it even though it overlaps and potentially could be confused with the established C++ terms *pure virtual* and *pure-specifier* [class.abstract]/2.

[3] Where applicable, we treat **`this`** as an (implicit) argument.

Eric White puts it less formally, saying that “A pure function is one that doesn’t affect the state of anything outside of it, nor depends on anything other than the arguments passed to it” [Whi06]. Equivalently, Keith Sparkjoy says that “A pure function doesn’t rely on any state beyond what’s passed to it via its argument list, and the only output from a pure function is its return value” [Spa13].

Among its other characteristics, a well-behaved function exhibits results that are *reproducible*: no matter how often such a function is called, its results will be identical so long as all values provided via the argument list remain unchanged. In the standard library, **`forward<>`**, **`string::length`**, and **`hash<>::operator()`** for the specializations specified in [unord.hash] exemplify well-behaved functions.

In contrast, a function that is not well-behaved is said to be *ill-behaved* or, equivalently, *impure*. An ill-behaved function may violate the above strictures via such behaviors as:

- relying on the value of a non-local object outside its argument list,
- modifying the value of a non-local object outside its argument list,
- throwing an exception without catching it,
- modifying and relying on the value of non-const private state such as a local **`static`** variable,
- failing to return,
- allocating dynamic storage without freeing it, or
- calling any ill-behaved function.

Standard library examples of ill-behaved functions include

- **`printf`**, because I/O is an observable side effect;
- **`tan`**, because it may update the global **`errno`** variable;
- **`longjmp`**, because it fails to return; and
- **`mersenne_twister_engine<>::operator()`**, because it updates and relies on the private state of **`*this`**.

3 Benefits

The ability to discriminate between well- and ill-behaved functions can be significant for the generation of high-performance code at and near a call site: If it can be determined at a point of call that the callee is well-behaved (and thus that its results are reproducible), additional caller optimizations may be applicable.

Walter Bright provides the following example in [Bri08]:

> Common subexpression elimination is an important compiler optimization, and with pure functions this can be extended to cover them, so: **`int x = foo(3) + bar[foo(3)];`** need only execute **`foo(3)`** once.

An article by embedded development tools producer KEIL [KEIL] confirms this analysis: A side-by-side comparison of the object code produced by the ARM C/C++ Compiler from identical calls to unannotated and pure-annotated versions of the same function shows that approximately 33% less code is generated when the called function is declared as pure. The reduction is attributed to a common subexpression elimination.

Selected optimizations may be applicable even if a callee is ill-behaved, but in such a case the safe application of optimizing code transformations generally requires more detailed knowledge regarding callee behavior. Such information is traditionally available only by inspecting the body of the callee. Because function inlining, by definition, makes function bodies visible at the point of call, compilers can make better decisions regarding both local and global code improvement opportunities relative to such a call site; such additional knowledge therefore contributes significantly toward the improved code very often attributed to inlining technology.[4]

Conversely, in the absence of inlining, a called function's body is traditionally opaque to its callers. Caller code improvement may therefore be inhibited by such lack of knowledge regarding callee behavior, especially with respect to side effects.[5]

In [Raa12], Frerich Raabe points out that "In addition to possible run-time benefits, a pure function is much easier to reason about when reading code." Walter Bright refers to this characteristic as *self-documentation* and argues [*op. cit.*] that "This greatly reduces the cognitive load of dealing with [a function]. A big chunk of functions can be marked as pure, and just this benefit alone is enough to justify supporting it."

Raabe continues, "Furthermore, it's much easier to test a pure function since you know that the return value only depends on the values of the parameters." Keith Sparkjoy [*op. cit.*] makes and expands on the same point: "Pure functions are straighforward to test. They only depend on their arguments; there's no obscure environmental setup that's required—just input and output. And testing is only one example of reuse—if something is easy to test, it's usually easy to reuse in other parts of your code."

Finally, Bright also promotes [*op. cit.*] the following additional benefits:

- "Pure functions do not require synchronization for use by multiple threads, because their data is all either thread local or immutable."
- "User level code can take this further by noting that the result of a pure function depends only on the bits passed to it on the stack (as the transitivity of invariants guarantees the constancy of anything they may refer to). Those bits can therefore be used as a key to access memoized results of the function call, rather than calling the function again."
- "Pure functions can be executed asynchronously. This means that not only can the function be executed lazily, it can also be farmed out to another core. . . ."
- "[Pure functions] can be hot swapped (meaning replaced at runtime), because they do not rely on any global initialization or termination state."

As John Cook summarizes: "You can't avoid state, but you can partition the stateful and stateless parts of your code. 100% functional purity is impossible, but 85% functional purity may be very productive" [Coo10].

4 Characteristics of pure functions[6]

A well-behaved free function will exhibit the following characteristics and behaviors of interest, as will a well-behaved **`static`** member function:

1. It takes arguments passed:
   a) by value, or
   b) by *const indirection* (reference-to-**`const`**, pointer-to-**`const`**, **`const_iterator`**, etc.).
2. It uses a modifiable lvalue to refer to an object, in whole or in part, only if that object:
   a) is local to the function, and has automatic lifetime (storage class), or
   b) is dynamically allocated by the function and is freed before returning.

[4]Whole-program optimization promises similar improvements, but tends to be far more demanding on resources during compilation and linking.

[5] A future proposal for modules might ameliorate such difficulties, but for now we can only speculate.

[6] Note that the core language already constrains **`constexpr`** functions to meet these expectations.

3. It uses either a non-modifiable lvalue or an rvalue to refer to (any part of) an object only if that object is non-**`volatile`** and:
   a) is an argument to the function, or
   b) is local to the function and has automatic lifetime, or
   c) is dynamically allocated by the function and is freed before returning, or
   d) is local to the function and has **`static`** lifetime and is declared **`const`**.
4. Its return type is non-**`void`**.
5. It respects all **`const`** qualifications. (That is, it does not circumvent any const-qualification via **`const_cast`** or the like.)
6. It permits no exceptions to escape.
7. It invariably returns control to the point of invocation.
8. It calls other functions (or member functions; see below) only if such functions are likewise well-behaved.

A well-behaved non-**`static`** member function shares the same characteristics and behaviors exhibited by a well-behaved free function. In addition:

9. It is always declared **`const`** so that its invoking object is always passed by pointer-to-**`const`** (i.e., **`this`** will always have a pointer-to-**`const`** type).
10. It respects all **`const`** qualifications, even in the presence of a **`mutable`** declaration.
11. If declared **`virtual`**, it is never overridden by an ill-behaved function.

5 Prior art

Several C++ compiler vendors provide implementations that encompass language extensions substantively corresponding to a **`pure`** annotion as proposed herein. Diego Pattenò argues in [Pet08] that these can successfully lead to improvements in generated code when consistently applied.

- GCC has long supported an **`__attribute__((pure))`**, as described in [Sta13, §6.30], "Many functions have no effects except the return value and their return value depends only on the parameters and/or global variables. . . . These functions should be declared with the attribute **`pure`**." The same section describes a similar **`__attribute__((const))`**: "Many functions do not examine any values except their arguments, and have no effects except the return value. Basically this is just slightly more strict class than the **`pure`** attribute. . . . "
- Further, "GCC does have the warning options **`-Wsuggest-attribute=pure`** and **`-Wsuggest-attribute=const`**, which suggest functions that might be candidates for the **`pure`** and **`const`** attributes" [ehi12].
- The .NET Framework 4.5 documents[7] a **`PureAttribute`** class that "Indicate[s] that a type or method is pure, that is, it does not make any visible state changes."
- The D programming language permits functions to be declared **`pure`**: "Pure functions are functions which cannot access global or static, mutable state save through their arguments. This can enable optimizations based on the fact that a pure function is guaranteed to mutate nothing which isn't passed to it, and in cases where the compiler can guarantee that a pure function cannot alter its arguments, it can enable full, functional purity. . . ."[8]
- The ARM C/C++ Compiler supports a **`__pure`** keyword with semantics equivalent to GCC's **`__attribute__((const))`**.

[7] See http://msdn.microsoft.com/en-us/library/system.diagnostics.contracts.pureattribute.aspx as of 2013-07-13.
[8] See http://dlang.org/function.html as of 2013-07-13.

- Other compilers (e.g., ICC and clang) are also reported[9] to mimic or directly support the above-described GCC attributes.
- While Mathematica supports the nomenclature of a pure function, it uses (abuses?) the term to denote a lambda-like construct instead.[10] Mathematica is therefore an inappropriate precedent for the present purpose.

It has been argued that these attribute declarations are redundant in the sense that whole-program interprocedural analysis can determine these properties. However, we believe that such analysis is often not feasible and sometimes not possible. For example, (a) pre-compiled code, (b) dynamically-linked libraries, and (c) time-sensitive compilation issues each provide challenges to the methods underlying interprocedural analysis.

6 Proposed wording

Incorporate the following new subclause within [dcl.attr] in WG21 Working Draft [DuT13]. (Note that paragraphs 3 through 5 are substantively based on wording in current use to specify existing attributes.)

7.6.x Pure attribute [dcl.attr.pure]

1 A function **`f`** is said to be *well-behaved* if, when called, (a) **`f`** commits no observable side effects ([intro.execution]) and (b) **`f`** communicates with client code solely via the function's return value and argument list (including **`this`**, where applicable). A statement *S* in the body of a function **`g`** is said to be *well-behaved* if *S*, when executed, exhibits behavior that is not inconsistent with a well-behaved **`g`**. If a function or a statement is not well-behaved, it is said to be *ill-behaved*.

2 [*Example:* A function **`f`** is ill-behaved if:

- it can ever fail to **`return`** to its caller, or
- its body contains an ill-behaved statement **`S`** that
  - reads (loads) the value of any **`volatile`**, **`mutable`**, or non-const variable whose lifetime began before **`f`** is called or whose lifetime will end after **`f`** returns,
  - updates (stores) the value of any variable whose lifetime began before **`f`** is called or whose lifetime will end **`f`** after returns,
  - performs I/O or commits any other observable side effect, or
  - calls an ill-behaved function.

— *end example*]

3 The *attribute-token* **`pure`** specifies that a function or statement is well-behaved. [*Footnote:* The **`pure`** attribute is unrelated to the C++ terms *pure virtual* and *pure-specifier* ([class.abstract]). — *end footnote*] The attribute shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute may be applied to the *declarator-id* in a function declaration. The first declaration of a function shall specify the **`pure`** attribute if any declaration of that function specifies the **`pure`** attribute. If a function is declared with the **`pure`** attribute in one translation unit and the same function is declared without the **`pure`** attribute in another translation unit, the program is ill-formed; no diagnostic required.

[9] See http://stackoverflow.com/questions/2798188/pure-const-function-attributes-in-different-compilers as of 2013-07-13.

[10] See http://reference.wolfram.com/mathematica/tutorial/PureFunctions.html as of 2013-07-22.

4 [*Note:* When applied to a well-behaved function, the **`pure`** attribute does not change the meaning of the program, but may result in generation of more efficient code. When applied to an arbitrary statement *S*, the **`pure`** attribute does not change the meaning of the program, but specifies that the implementation may assume (without further analysis) that, when executed, *S* will not cause the containing function to be ill-behaved. — *end note*]

5 If an ill-behaved function **`f`** is called where **`f`** was previously declared with the **`pure`** attribute, the behavior is undefined.

6 [*Note:* Implementations are encouraged to issue a warning if a function **`f`** marked **`[[pure]]`** is ill-behaved or exhibits any characteristic that is inconsistent with a well-behaved function. However, implementations should issue no such warning on the basis of any statement (even if ill-behaved) that is marked **`[[pure]]`**. [*Example:* A warning would be in order if **`f`**:

- has a **`void`** return type,
- is declared with the **`noreturn`** attribute,
- is declared **`noexcept(false)`**,
- is a non-const non-static member function,
- is overridden by a function that is neither declared **`constexpr`** nor marked **`[[pure]]`**, or
- calls a function that is neither declared **`constexpr`** nor marked **`[[pure]]`**, unless the call is part of a statement that is marked **`[[pure]]`**.

— *end example*] — *end note*]

7 Acknowledgments

Many thanks, for their insightful comments, to the readers of early drafts of this paper.

8 Bibliography

[Bri08] Walter Bright: “Pure Functions.” 2008-09-21.
http://www.drdobbs.com/architecture-and-design/pure-functions/228700129.

[Bro04] Walter E. Brown and Marc F. Paterno: “Toward Improved Optimization Opportunities in C++0X.” ISO/IEC JTC1/SC22/WG21 document N1664 (mid-Sydney/Redmond mailing). 2004-07-16.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1664.pdf.

[CO79] Robert Cartwright and Derek Oppen: “The Logic of Aliasing.” Computer Science Department Report No. STAN-CS-79-740. September 1979.
ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/79/740/CS-TR-79-740.pdf.

[Coo10] John D. Cook: “Pure functions have side-effects.” 2010-05-18.
http://www.johndcook.com/blog/2010/05/18/pure-functions-have-side-effects/.

[DuT13] Stefanus Du Toit: “Working Draft, Standard for Programming Language C++.” ISO/IEC JTC1/SC22/WG21 document N3691 (mid-Bristol/Chicago mailing), 2013-05-06.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3691.pdf.

[ehi12] ehird [*sic*]: Response to “Can a compiler automatically detect pure functions without the type information about purity?” 2012-01-12.
http://stackoverflow.com/questions/8760956/can-a-compiler-automatically-detect-pure-functions-without-the-type-information.

[GA09] Douglas Gregor and David Abrahams: “Rvalue References and Exception Safety.” ISO/IEC JTC1/SC22/WG21 document N2855 (post-Summit mailing). 2009-03-23.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2855.html.

[KEIL] KEIL: “Comparison of pure and impure functions.” undated.
http://www.keil.com/support/man/docs/ARMCC/armcc_BABBCHHF.htm.

[Pet08] Diego Pattenò: "Implications of pure and constant functions." 2008-06-10. http://lwn.net/Articles/285332/.

[Raa12] Frerich Raabe: "Benefits of pure function" (response). 2012-06-22. http://stackoverflow.com/questions/11153796/benefits-of-pure-function.

[Spa13] Keith Sparkjoy: "My Clojure journey: pure functions." 2013-03-13. http://blog.pluralsight.com/2013/03/13/my-clojure-journey-pure-functions/.

[Sta13] Richard M. Stallman and the GCC Developer Community: "Using the GNU Compiler Collection for GCC version 4.8.1." 2013. http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc.pdf.

[Whi06] Eric White: "Pure Functions." 2006-10-03. http://blogs.msdn.com/b/ericwhite/archive/2006/10/03/pure-functions.aspx.

9 Revision history

Version	**Date**	**Changes**
1	2013-08-30	• Published as N3744.

Toward Opaque Typedefs for C++1Y, v2

Document #: WG21/N3741
Date: 2013-08-30
Revises: N3515
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <`webrown.cpp@gmail.com`>

Contents

1 Introduction

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*. 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.[1] Alas, the press of other obligations has until recently not permitted us to resume our explorations.

Now with C++11 as a basis, we return to the topic. Where our earlier thinking and nomenclature seem still valid, we will repeat and amplify our exposition; where we have new insights, we will follow our revised thinking and present for EWG discussion a high-level proposal for a C++1Y language feature to be known as an *opaque alias*.[2]

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.

[1] 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.”

[2] Citations that look [like.this] refer to subclauses of C++ draft N3691. We will generally omit cross-references from quoted text.

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.[3] 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:

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

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

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 recent WG21 reflector message, "The **typedef** problem is one that we know badly bites us ever so often . . . vis-a-vis overloading."[4] 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,[5] 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.

[3] "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).

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

[5] 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.

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.

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

class PhysicsVector {
public:
  PhysicsVector(X, Y, Z);
  PhysicsVector(Rho, Theta, Phi);
  ...
};  // 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 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[6] 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[7] with its underlying type. The intent is to allow programmer control (1) over substitutability between an opaque alias and its underlying type, and (2) over overloading (including operator overloading) based on any parameter whose type is or involves an opaque alias. Unlike the similar relationship of a derived class to its underlying base class, it is desired 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[8] **`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.

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

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

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

- 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.[9]
- 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*[10] 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.

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.[11] 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

A macro implementing a kind of opaque alias has been distributed with Boost's serialization library for over a decade. Internal documentation[12] summarizes its behavior as "**`BOOST_STRONG_TYPEDEF(T,D)`** creates a new type named **`D`** that operates as a type **`T`**." Using this paper's nomenclature, we would say that **`D`** denotes an opaque type whose underlying type is denoted by **`T`**.

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

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

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

[12] Found in header **`boost/strong_typedef.hpp`**, © 2002 by Robert Ramey.

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 hindsight, the macro's author has posted[13] an evaluation of his experience in creating and using the macro. He writes in significant part (lightly reformatted and with some typos corrected):

> 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 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.

6 A start on *opaque alias* syntax

We propose that C++1Y complement traditional type aliases by providing an *opaque alias* facility, and nominate the following variation of *alias-declaration* syntax:

```
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*:

[13] Robert Ramey: "[std-proposals] Re: Any plans for strong typedef." 2013-01-11.

```
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,[14] 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:

- 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.[15]

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:[16]

```
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:

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

[15] 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.

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

- 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,[17] 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.[18]

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.

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
```

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

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

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.[19] 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.[20]

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 (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 (1) a brace-enclosed body[21] or (2) 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.[22] Here is our example from an earlier section, rewritten in template form:

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

[20] 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.

[21] 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`**.

[22] Specified in [temp.alias].

```
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
```

10 Summary and conclusion

This paper has outlined a new solution, known as *opaque aliases*, 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 opaque aliases to be a viable new tool, complementing traditional type aliases, that allows programmers to address practical problems. We invite feedback from WG21 participants and other knowledgeable parties, and especially invite implementors to collaborate with us in order to experiment and gain experience with this proposed new language feature.

11 Acknowledgments

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

12 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.

[N3691] Du Toit, Stefanus: “Working Draft, Standard for Programming Language C++.”
ISO/IEC JTC1/SC22/WG21 document N3691 (mid-Bristol/Chicago mailing), 2013-05-16.
Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3691.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.

13 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 to §3. • Tweaked formatting, punctuation, and grammar. • Published as N3741.