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.