The [[`pure`]] attribute

Document number: P0078R0

Date: 2015-09-25

Revises: N3744

Project: JTC1.22.32 Programming Language C++, Evolution Working Group

Reply to: Karl-Étienne Perron <Karl-Etienne.Perron@USherbrooke.ca>

Contents

1 Introduction

The C++ programming language has long been associated with execution speed, optimization and flexibility. However, there is still place for improvement in C++ as well as any other programming language. This is the reason why many attempt to bring different kinds of optimization to the C++ core language.

Papers such as N1664 [1] and N3744 [2] both tried to do so by proposing function qualifiers such as **`nothrow`** and **`pure`**. Whereas **`nothrow`**'s purpose has ultimately been achieved with **`noexcept`**, this paper's goal is to clarify why the addition of a **`pure`** attribute to the C++ language could be beneficial. More precisely, we aim to provide a new clearer definition of the **`pure`** attribute for C++ based on both theoretical notions and the feedback received from the EWG regarding N3744's proposal of the same attribute. Ultimately, this will allow us to propose a formal wording to add to the Working draft [32].

2 Motivation

According to N1664 [1] in 2004, "…, it can sometimes be effectively impossible for a C++ compiler to determine whether any particular optimization can be safely applied in the context of a particular code fragment" [1]. This is the case with pure functions for which compilers do need indications in order to apply optimization. The opportunity for C++ to benefit from such optimization seemed a good idea ten years ago and still is today. As a matter of fact, during the review of N3744 [2], the EWG voted on having some form of the **`pure`** attribute and the result clearly showed the interest in having one:

- *Poll 3: In favor of some form of [[pure]]*

  SF 8 - 2 - 0 - 0 - 0 SA

[3]

Also, pure functions show great promise for the near future. During a recent telecom regarding contract programming for C++, participants quickly agreed that the predicate serving as a precondition should be pure. Therefore, we think it is in the best interest of everyone to better define the concept of **`pure`** as it pertains C++, in order to be able to add features such as a [[**`pure`**]] attribute.

3 In theory

The purity of a function is a fundamental aspect of functional programming. In its most theoretical (purest) form, this programming paradigm "treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data." [4]. In other words the purity of a function is defined by how much a function respects the mathematical principles of a function. Mathematical functions are, by definition, a relation between a set of inputs and an output [5] where "each input is related to exactly one output" [6]. This means that we should always end up with the same result for the same input.

This absolute relationship also means that nothing but the function itself may define how the input is related to the output. For example, in a purely functional program, for a given set of inputs, an additional input (say from the user or from reading a file) in the middle of the function's computation should not be required in order to produce the result since such intervention may change the outcome. This is the reason why pure functions should avoid using changing-state and mutable data since the modification of a variable's value may act as a form of extra input during computation. The possibility of having the result changed even if we give a function the same arguments would mean that mathematically, the same input is related to more than one output which is against the very definition of a function.

4 `pure` in current programming languages

4.1 C++

As of the writing of this paper, there are already ways to qualify C++ functions as **`pure`**. However, the definition and mechanics are different in each implementation and would certainly benefit from being standardized.

4.1.1 .Net

The .Net framework has allowed the use of a "PureAttribute" class in C++/CLI (and the framework's other languages as well) since version 4 [7]. The official documentation states that this attribute "indicates that a type or method is pure, that is, it does not make any visible state changes." [7]. Based on the above theoretical definition, the .Net framework seems to share the idea of avoiding the use of changing-state and mutable data. However, this definition only seems to consider state change as an outcome of the function and not the other way around, that is, a pure function should not change the state of anything as a result of its computation, but is free to use mutable data in order to produce a result. As stated earlier, a function which uses mutable data may produce different outputs given the same input. Consider this function:

```
int plus_five(int &x)
{
      return x + 5;
}
```

Here, the value of x at the time of input could be changed by another thread (therefore, a change not caused by the function itself) before the computation of the addition. Even worse, this definition does not restrict a pure function from producing multiple outputs given the same input which is theoretically wrong.

4.1.2 GCC and other compilers

With GCC, programmers have the possibility of using **`__attribute__((…))`** as a way of qualifying functions.

Since version 2.96, GCC has an **`__attribute__((pure))`** keyword which indicates that "... functions have no effects except the return value and their return value depends only on the parameters and/or global variables." [8]. This is an interesting definition since it explicitly applies the mathematical idea of inputs and output to the concrete use of a function's arguments and return value. It also states that function should not have any effect other than returning the return value, which means that it should not visibly change the state of anything outside the function. However, we presumably still have the same problem of being able to use mutable and state-changing data as inputs, especially here where it is clearly stated that global variables are a viable form of (read-only) input. There is no mention of whether or not global variables must be global **`const`** variables. This is where GCC gets confusing when talking about **`pure`** because there is also an **`__attribute__((const))`** keyword which is an even purer form of **`__attribute__((pure))`**. The official documentation declares that const 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 below, since function is not allowed to read global memory. Note that a function that has pointer arguments and examines the data pointed to must not be declared const. Likewise, a function that calls a nonconst function usually must not be const. It does not make sense for a const function to return void. [8]

It seems strange that the purest form of the **`pure`** attribute in GCC is not labeled **`pure`**, but **`const`**. As stated, the main difference between them is the fact that a const function may not read global memory which effectively solves our problem of having mutable data as inputs. Also, this definition makes an interesting point that functions must produce an output. Mathematical functions must, by definition, produce an output to be associated with the given inputs. Otherwise, the function would serve no purpose.

Other compilers such as LLVM/Clang [9] and ILE C/C++[10] also include attributes the same way GCC does. Both keywords are, therefore, applicable to those as well. In fact, "almost all major compilers (except MSVC, apparently) support extensions that allow one to mark a function as pure." [11].

The ARM compiler is a little bit more restrictive about **`__pure`** and is comparable to GCC's **`__attribute__((const))`**. Declaring a function with **`__pure`**:

> asserts that a function declaration is pure […] the result depends exclusively on the values of its arguments; the function has no side effects; [...] For example, pure functions: cannot call impure functions; cannot use global variables or dereference pointers, because the compiler assumes that the function does not access memory, except stack memory; must return the same value each time when called twice with the same parameters. [12]

This definition is interesting for a few reasons. First, it is stated that "the result depends exclusively on the values of its arguments". Whereas most definitions state that the result depends on the argument itself, here we let go of the container and use the content as the input instead. This solves our problem of a parameter being mutable. It is highly probable that this was implied in other definitions, but the fact that this is mentioned clearly in this one is appreciable. Second, stating that a pure function can only access the stack memory leaves less ambiguity regarding the type of variables which a pure function may or may not take in as parameters. Finally, the fact that the function must return the same value when called twice (or more) with the same parameters is also the first time it has been specified.

4.2 D

D already has a **`pure`** keyword. The documentation for the language states that:

> *Pure functions are functions which cannot access global or static, mutable state save through their arguments. […] 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 (i.e. the guarantee that the function will always return the same result for the same arguments). To that end, a pure function: does not read or write any global or static mutable state; cannot call functions that are not pure; can override an impure function, but an impure function cannot override a pure one; is covariant with an impure function; cannot perform I/O.* [13]

This definition is really close to the theoretical definition of **`pure`**. As a matter of fact, it explicitly says that the function will always return the same result for the same arguments as per the mathematical (pure) definition of a function while clearly preventing the use of any form of mutable data from outside or within the function's parameters (i.e. no side effects). It also lists a few other logical characteristics about pure functions.

4.3 Fortran

Fortran95 has "pure procedures": "we add the PURE keyword to the SUBROUTINE or FUNCTION statement—an assertion that the procedure (expressed simply): alters no global variable; performs no I/O; has no saved variables (variables with the SAVE attribute that retains values between invocations) and for functions, does not alter any of its arguments." [14]. This definition mostly revolves around the fact that a pure function must not have side effects. However, we still have the problem of being able to use global variables as inputs.

4.4 Haskell and other functional languages

Haskell is claimed to be a "pure functional language." [15]. While this is not entirely true, it does provide a mostly pure code base whilst retaining a small portion of practical, yet impure code (such as I/O) [15][16]. The official Haskell web site claims that the language is pure since "Every function in Haskell is a function in the mathematical sense (i.e., "pure"). Even side-effecting*[sic]* IO operations are but a description of what to do, produced by pure code. There are no statements or instructions, only expressions which cannot mutate variables (local or global) nor access state like time or random numbers." [17]. The official wiki also says this: "Languages that prohibit side effects are called pure. [...] Purely functional programs typically operate on immutable data. [...] Pure computations yield the same value each time they are invoked." [16]. Overall, functional purity in Haskell means that functions have no side effects, will produce the same output if given the same inputs and cannot use and/or change mutable data and states (even local variables). This is, in fact, very close to the theoretical definition of functional purity. As a matter of fact, it may even be a bit too restrictive since even creating a local variable (a temporary variable for instance) will not necessarily affect the purity of a function. For instance, when applied to C++:

```
int plus_temp_five(int x)
{
      const int five{5};
      return x + five;
}
```

The local variable “five” would never change the outcome of the function's computation and would not change the state or value of any data outside the function. Once we would be done with the function, the local variable would also be “deleted” from memory which means that no observable side effects would have occurred, that is, the program's environment will be the same after as it was before.

5 `pure` as a general computational concept

Functional purity does not have to be tied down to a particular set of languages. In general, **`pure`** is usually defined based on many characteristics. Whilst not all sources agree exactly on what gives a function its purity, here is what is generally said about pure functions.

5.1 No side effects / Referential transparency[1]

The fact that a pure function should not have any side effect is mainly accepted by everyone. Based on the idea that the purest functions possible are mathematical functions, a function is an absolute and strict relationship between inputs and exactly one output [5][6]. Some even speak of a property called “referential transparency” [4][16][18], that is, since a mathematical function puts a given set of inputs in equality with a particular output, it should be mathematically possible to simply replace a call to the function by the output.

However, not everyone agrees on what actually constitutes a side effect. For some, creating and modifying a local variable (local to the function only) is a form of side effect [17]. Others state that pure functions must not produce **observable** side effects, that is, side effects which alter the overall state of the system, but solely based on what's expected from it [18][19][20][21]. For example: “An observable side effect is one which is modelled in the semantics [What's monitored and expected]. In typical models of programming languages, memory consumption is not modelled, so a computation that requires 1TB of storage can be pure, even though if you try to run it on your PC it would observably fail.” [21]. Also, it should be noted that “observable” means observable from either the caller's perspective or any other threads which may be affected by the function’s computation. A pure function should be a black box to the caller. Within the box, a function should be

1 [4][7][8][9][10][11][12][13][15][16][18][19][20][22][25][27][29][30]

allowed to change and alter the system if needed as long as it replaces the system in its previous state when it returns [19][21].

5.2 Reproducible[2]

By extension to the previous point and once again taken from the idea of purely mathematical functions, one of the most fundamental aspect of a pure function seems to be the fact that regardless of any factor, the function will always return the same result. This makes the function memoizable. Should this requirement not be met, a pure function would actually disregard its mathematical nature: a set of inputs would be associated with more than one possible outcome [6].

5.3 Avoiding changing-state and mutable data / Having stable inputs[3]

Allowing a pure function to handle mutable data does not make the function impure by itself. However, there are many risks involved and the function is very likely to display an impure behaviour. Therefore, possibility of impurity should still be considered as impurity. As stated earlier in this paper, there can be a problem if mutable data is given to the function and then altered halfway through the function's computation by another mean (other than the function itself). For example: "If an argument is call by reference, any parameter mutation will alter the value of the argument outside the function, which will render the function impure." [19]. In order to overcome this problem, it seems necessary to specify that "arguments are fully determined before any output is generated." [20].

5.4 Returns[4]

It seems pretty counter intuitive to have a pure function without any value to return: "Pure functions cannot reasonably lack a return type." [22]. A function which receives inputs, but does not return an answer or produce any kind of side effects is quite simply a computational black hole and lacks any real purpose. Of course, this is incompatible with the theoretical definition of a pure function.

6 Redefining `pure`

Based on the theoretical principles and the other definitions and characteristics of a pure function we studied above, the following is what we consider a pure function should be. This is our recommendation to WG21 on how to approach the **`pure`** attribute.

A **`pure`** function is a function which:

- for the same set of arguments, will always return the same value;
  (because a mathematical function is the purest form of function possible)
- produces an output using solely the values given to it via its argument list;

[2] [4][9][10][12][13][15][18][19][25][26][27][29][30]

[3] [4][7][11][13][14][15][17][19][20][25][26][27][31]

[4] [8][9][10][11][12][20][22][27][29]

(because other forms of input may cause different outputs for the same arguments which would make the function impure)

- is only given arguments that cannot be externally altered during the function's computation;
  (because inputs which can be altered during computation may cause different outputs for the same arguments which would make the function impure)
- causes no observable side effects;
  (because a call to a mathematical function can be replaced directly by its result without altering the final result)
- does not call impure functions;
  (because calling impure functions may result in side effects with the pure function as its cause)
- outputs its result solely through its return value;
  (because any other mean of output would be an observable side effect)
- has a non-void return type;
  (because a mathematical function cannot return nothing)
- cannot fail to return and returns to its point of invocation.
  (because a mathematical function must return and only where it has been called)

7 Benefits and drawbacks

With this paper, we hope to bring even more optimization possibilities to the C++ programming language. As a matter of fact, adding a **`pure`** attribute would have many benefits, but also a few drawbacks.

7.1 Benefits

7.1.1 Subexpression elimination / Memoization[5]

Probably the most obvious use for a **`pure`** attribute is the exploitation of its referential transparency characteristic as a way to reduce computational time: “By definition, it is sufficient to evaluate any particular call to a pure function only once. Because the result of a call to the function is guaranteed to be the same for any identical call, each subsequent call to the function in code can be replaced with the result of the original call.” [23]. Ultimately we “get the same result; only the efficiency might change” [16]. The ARM Information Center [23] has a great example of how memoization may reduce the amount of computing necessary:

[5] [4][9][11][12][16][18][22][23][26][27][28]

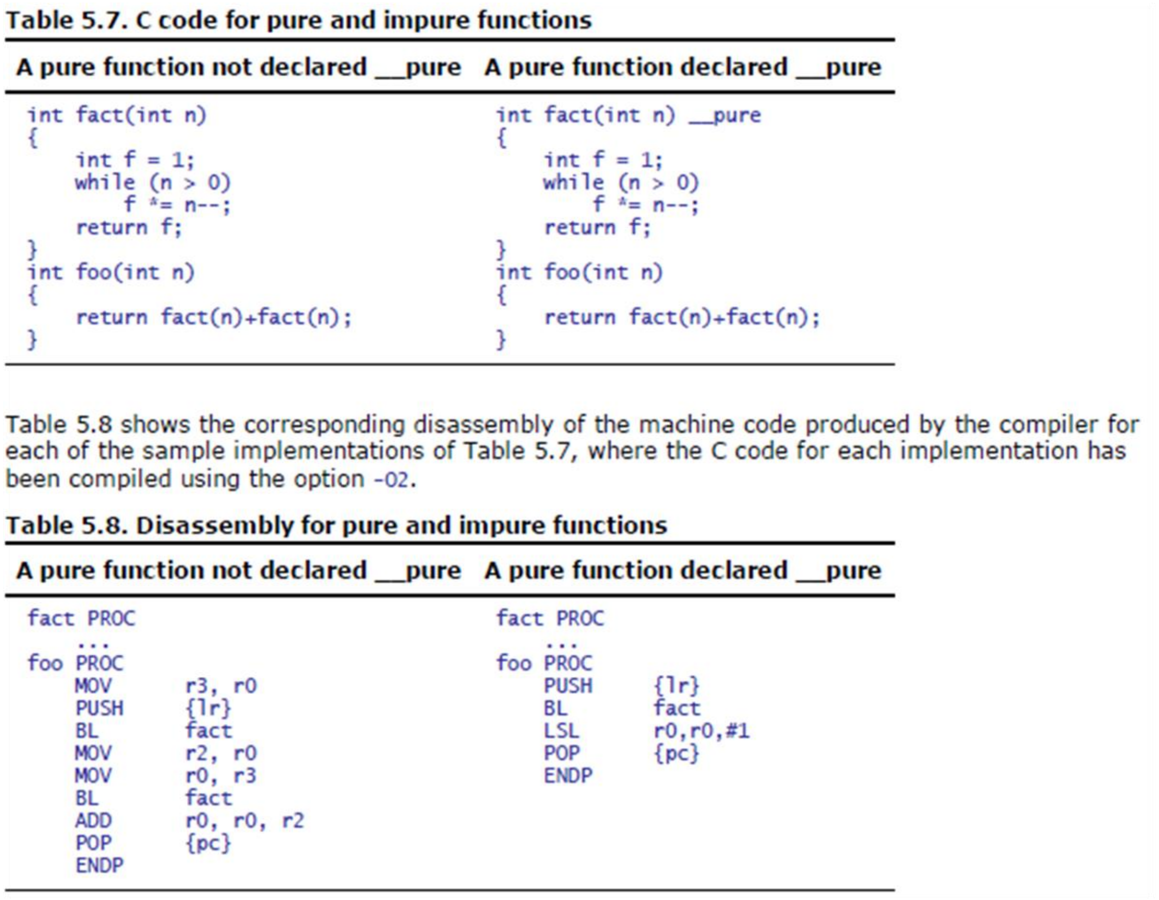
Table 5.7. C code for pure and impure functions

A pure function not declared __pure	A pure function declared __pure
`int fact(int n)`	`int fact(int n) __pure`
`{`	`{`
`int f = 1;`	`int f = 1;`
`while (n > 0)`	`while (n > 0)`
`f *= n--;`	`f *= n--;`
`return f;`	`return f;`
`}`	`}`
`int foo(int n)`	`int foo(int n)`
`{`	`{`
`return fact(n)+fact(n);`	`return fact(n)+fact(n);`
`}`	`}`

Table 5.8 shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of Table 5.7, where the C code for each implementation has been compiled using the option -O2.

Table 5.8. Disassembly for pure and impure functions

A pure function not declared __pure	A pure function declared __pure
`fact PROC`	`fact PROC`
`...`	`...`
`foo PROC`	`foo PROC`
`MOV r3, r0`	`PUSH {lr}`
`PUSH {lr}`	`BL fact`
`BL fact`	`LSL r0,r0,#1`
`MOV r2, r0`	`POP {pc}`
`MOV r0, r3`	`ENDP`
`BL fact`	
`ADD r0, r0, r2`	
`POP {pc}`	
`ENDP`	

As shown, the amount of assembly instructions produced is reduced by 45% when declaring the same function as **`pure`**.

7.1.2 Composition and abstraction[6]

Code composition and abstraction are probably two of the most useful and intuitive aspects of programming. Pure functions are inherently compatible with both.

It is very easy to see why pure functions are ideal to use for composition. A function that has no side effects can be used by absolutely any piece of code at any given moment since the function does not rely on anything external to itself for its computation. This means that the pure function will have the same behaviour regardless of the calling code's design.

They are also very useful when it comes to abstraction. Since a pure function communicates with the outside world solely via its arguments list and its return value, there is virtually no need for the calling code to know how exactly the function produces its result.

[6] [4][16][22][26][27]

7.1.3 Concurrency and thread safety / Code reorganization[7]

Expanding on the idea that pure functions are perfect for composition and abstraction, they are also thread safe and can be relocated within the code with ease.

Given that pure functions will always yield the same result if called with the same arguments, that they cannot have side effects and that their data is either local or immutable [24], there is no danger in calling the function with multiple thread in order to gain speed: "If we know that a function relies on nothing other than its parameters, then we (or the compiler) might be able to execute the function in a new thread, or even a different CPU" [25].

For the same reasons, pure functions may bring benefits to compilers' code reorganization optimizations. Since the function will always give the same output regardless of states and execution order, the compiler is free to call pure functions whenever it deems optimal without the risk of altering the end results: "If there is no data dependency between two pure expressions, then their order can be reversed…" [4].

7.1.4 Tests[8]

Pure functions are easier to test: "…it's very easy to unit test a pure function since there is no context to consider. Just focus on inputs / outputs." [26]. Tests are really useful when you want to ensure that your code behaves as expected. It is also a great way of documenting your code. However, some functions are tougher to test than others. When testing impure functions, you must ensure that each state and possibility is covered. This can be long and painful to plan and even when we think we covered every possibility, a new one appears. Since pure functions always produce a precise output for a certain input, it is easier to know whether they work or not: "When you are not sure that it is working correctly, you can test it by calling it directly from the console, which is simple because it does not depend on any context. It is easy to make these tests automatic — to write a program that tests a specific function. Nonpure functions might return different values based on all kinds of factors, and have side effects that might be hard to test and think about." [27].

7.1.5 Other benefits[9]

Apart from the big benefits listed above, pure functions also offers other advantages.

If a programmer really tries to use pure functions in his code, that will force him to "think locally". Given that pure functions should not use global variables or rely on mutable data, one should feel more inclined to using local variables which should make the overall code simpler and easier to understand:

> *A pure function can only access what you pass it, so it's easy to see its dependencies. We don't always write functions like this. When*

[7] [4][7][17][23][25][26]

[8] [23][25][26][27]

[9] [18][23][24]

> *a function accesses some other program state, such as an instance or global variable, it is no longer pure. Take global variables as an example. These are typically considered a bad idea, and for good reason. When parts of a program start interacting through globals, it makes their communication invisible. There are dependencies that, on the surface, are hard to spot. They cause maintenance nightmares. The programmer needs to mentally keep track of how things are related and orchestrate everything just right. Small changes in one place can cause seemingly unrelated code to fail.* [18]

Not only is it easier to keep track of our code with pure functions, but when we have a clear indication in order to tell whether a function is pure or not, it makes it easier to understand what is happening "under the hood". With a clear keyword qualifying a function as pure, it may help in better understanding how the execution of a program will behave after compilation and will reduce the overall cognitive load:

> *The first is self-documentation. A person trying to understand a code base, once they see that a function is pure, they know it only depends on its arguments, has no side effects, and there's no monkey business going on inside it. This greatly reduces the cognitive load of dealing with it. A big chunk of functions can be marked as pure, and just this benefit alone is enough to justify supporting it.* [24]

7.2 Drawbacks

7.2.1 I/O and side effects restrictions[10]

As useful as pure functions seem to be, they also bring us some trouble. The most problematic drawbacks are in fact what makes pure functions useful in the first place. Without the ability to cause side effects, pure functions cannot read a file or prompt the user for an input which is sometimes quite useful. Pure functions cannot benefit from impure functions since calling an impure function would mean that the pure function would cause side effects. Also, pure functions lack the possibility of using really efficient, but mutable data structures [16]. In certain cases, this could mean that using pure functions would be, in fact, slower and less practical than using their impure counterparts: "In many cases, nonpure functions are precisely what you need. In other cases, a problem can be solved with a pure function but the nonpure variant is much more convenient or efficient." [27].

7.2.2 Debugging[11]

GCC has the ability to warn you about which functions are viable candidates for either **`pure`** or **`const`** attribute qualifying. This is mainly done through code optimization analysis. However, such analysis frequently misses pure functions [11]. This is the reason

[10] [16][27]

[11] [11][28]

why it is up to the programmer to qualify the function or not. Imagine that we falsely qualify an impure function as **`pure`**. Based on what the compiler knows about pure functions, it might bring some "optimizations" to the code. In reality, this will probably result in an incoherent program execution which should be troublesome to debug: "… using this attribute incorrectly can lead to a nearly impossible to locate bug as actually seeing the redundant use of the function removed by the compiler requires looking at the assembly! Oh, and this type of bug will rarely show in a debug build since only highly optimized builds will have the bug." [28].

8 EWG feedback on N3744

The EWG already debated a **`pure`** attribute when they reviewed N3744 [2]. However, it seems that much confusion and issues arose from the review. Given our analysis and brand new definition, we will try and give a clearer explanation on what **`pure`** means.

8.1 Transaction safety

One of the first question to be asked, the EWG wondered if pure functions are transaction-safe by nature. Based on our analysis of functional purity, it must said that pure functions are transaction safe since they do not have side effects and must clean their environment before returning. This means that should anything happen during a pure function's computation which requires cancellation, its natural characteristics should result in few or no rollbacks.

8.2 Dynamic memory allocation

Given that what is considered to be an observable side effect is ambiguous, it is normal to question whether or not dynamically allocating memory in a pure function should be prohibited. As stated earlier, observable depends on what is monitored and by whom. In general for functions, we mean observable from the caller's point of view. This means that an observable side effect is one which can be seen outside of the function's lifetime. However, if a pure function requires dynamic storage while two or more threads are running concurrently, a pure function could cause other functions to experience **`bad_alloc`** exceptions. This is a form of side effect hence why dynamic memory allocation should be prohibited in pure functions.

8.3 Warnings

It has been discussed that trying to implement ways to warn programmers about functions being possibly pure or impure would somehow be difficult. Warnings indicating which functions should or should not be qualified as **`pure`** would be helpful to programmers who want to optimize their program's execution. Also, as stated earlier in this paper, enforcing a way to tell the programmer that an impure function has falsely been qualified as **`pure`** may help save much time and efforts during debugging.

8.4 **`pure`** vs **`const`**

During N3744's [2] review, many members of the EWG were confused about the difference between GCC's **`pure`** and **`const`** attributes which also led to questions about what should be considered as viable arguments for a pure function.

Of course any lvalues passed by-value are eligible since by-value arguments are local copies which will be destroyed after the function's computation and therefore can be altered without that being considered as a side effect.

However, we need to be careful when talking about pointers and arguments passed by-reference. These types of argument should not be used in pure functions since their mutable nature will most likely lead to impurity. It should be noted however that depending on the level of purity you require, arguments passed by const indirection can be used by pure functions. When we say level of purity, we are talking about the subtle difference between GCC'S **`pure`** and **`const`** attributes. **`const`** is the "purest" form of **`pure`**, that is, the only inputs allowed are the argument's values.

With **`const`**, we may use pointers and such, but since we cannot access global memory, we are left with the literal values of the arguments which are rarely useful on their own. An example of pure (**`const`**) functions that may use pointers is a function used for pointer arithmetic.

With **`pure`**, we may access global memory. This means that a pointer to const, for example, can be dereferenced in order to produce an output. Since the value pointed to by the pointer cannot change, the characteristics for functional purity will still apply. However, theoretically, this could make a function impure. For example, if we only change what a pointer to const points to between two calls to the same pure function, the arguments' value will technically still be the same, but the output will differ which theoretically means that the function is impure. It really depends on what you consider to be an "input". If we digress a bit from functions' mathematical roots and forget about pure functions requiring inputs to be passed through its arguments list, const global variables accessed through indirection could be considered as inputs therefore preserving a function's purity.

When N3744 [2] was reviewed, the EWG voted on whether we should support the equivalent of either **`const`** (the aggressive approach to **`pure`**) which can be memoized or **`pure`** (the conservative approach) which causes no side effects or even both. The results were:

- *Poll 4: Conservative approach.*

  SF 3 - 2 - 5 - 0 - 0 SA

- *Poll 5: Aggressive approach:*

  SF 6 - 1 - 3 - 0 - 0 SA

- *Poll 6: Support both, with different attributes.*

  SF 4 - 2 - 3 - 1 - 0 SA

[3]

As we can see, **`const`** received more support in general. Having both attributes with different meaning has been criticized for leading to too much confusion.

9 Proposed wording

Now that we have revisited the theory behind pure functions, offered a new theoretical definition and addressed what had caused confusion in previous proposition of such attribute, it is time to propose our own concrete wording of how a **`pure`** attribute could fit in the C++ programming standard language. Note that here, our definition of **`pure`** is inspired by its purest most theoretical form (similar to GCC's **`const`** or ARM's own **`pure`** attribute).

Incorporate the following new subclause into the WG21 Working Draft [32] under the Attributes section [dcl.attr]. (Note the proposed wording is greatly inspired by N3744's [2] own proposed wording and sometimes reuse similar to complete statements)

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

1. A function f is said to be *pure* if it embraces the mathematical nature of functions, that is, (a) with the same argument values, f will always return the same answer, (b) f will only communicate with client code via its argument list and return value (c) always returning to its point of invocation and (d) f will not cause side effects observable outside of its lifetime. A statement S in the body of a function g is said to be *pure* if S, when executed, exhibits behavior that is not inconsistent with a *pure* g. The opposite of a *pure* function or statement is said to be *impure*. Every function or statement is *impure* until specified as *pure*.
2. [Example: A function f is not *pure* if:
   - its arguments are not passed by value
   - it accesses global memory for reading or writing
   - it has a void return type
   - it calls an *impure* function
   - the calling code or other threads can "perceive" changes brought by f, that is, if:
     1. it relies on and/or alters variables and/or data structures that are not local to itself
     2. it dynamically allocates memory
     3. it throws an exception, but does not catch it

   - 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.

4. [Note: When applied to a theoretically *pure* 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 *impure*. — end note]
5. If an *impure* function f is called where f was previously declared with the **`pure`** attribute, it can be assumed that calls with the same arguments yield the same result.

10 Acknowledgements

Thanks to Walter E. Brown and Patrice Roy for their support and giving me the opportunity to write this paper.

11 References

1. Walter E. Brown, Marc F. Paterno: “*Toward Improved Optimization Opportunities in C++0X*”, 2004-07-16.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1664.pdf
2. Walter E. Brown: “*Proposing [[pure]]*”, 2013-08-30.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3744.pdf
3. WG21 Chicago, 2013
4. Wikipedia, “*Functional programming*”, retrieved on 2015-07-11.
https://en.wikipedia.org/wiki/Functional_programming
5. Maths Is Fun, “*Functions*”, retrieved on 2015-07-12
https://www.mathsisfun.com/sets/function.html
6. Wikipedia, “*Functions (mathematics)”*, retrieved on 2015-07-12.
https://en.wikipedia.org/wiki/Function_(mathematics)
7. Microsoft, “*Pure Attribute Class*”, retrieved on 2015-07-11.
https://msdn.microsoft.com/en-us/library/system.diagnostics.contracts.pureattribute(v=vs.110).aspx?cs-save-lang=1&cs-lang=cpp#code-snippet-1
8. Richard M. Stallman and the GCC Developer Community, “*Using the GNU Compiler Collection*“, 2015.
https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc.pdf
9. Mattt Thompson, “*__attribute__*”, 2013-01-14.
http://nshipster.com/__attribute__/
10. IBM, “*The pure function attribute*”, retrieved on 2015-07-11.

http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzarg/fn_attrib_pure.htm%23fn_attrib_pure?lang=fr
11. Pimiddy, “*Pure functions in C/C++*”, 2012-04-20.
https://pimiddy.wordpress.com/2012/04/20/pure-functions-in-cc/
12. ARM, “*__attribute((pure)) function attribute*”, 2011.
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/Cacigdac.html
13. D Programming language, “*Functions*”, retrieved on 2015-07-11.
http://dlang.org/function.html#pureㄳ_functions2/33
14. Wikipedia, “*Fortran 95 language feature*”, retrieved on 2015-07-11.
https://en.wikipedia.org/wiki/Fortran_95_language_features#Pure_Procedures
15. Joel Spolsky and others, “*Pure Functional Language: Haskell*”, 2010.
http://stackoverflow.com/questions/4382223/pure-functional-language-haskell/4#4
16. Haskell Wiki, “*Functional programming*”, retrieved on 2015-07-11.
https://wiki.haskell.org/Functional_programming#Purity
17. Haskell, 2015.
https://www.haskell.org/
18. Arne Brasseur, “*Functional programming: Pure functions*”, 2014-09-17.
https://learnable.com/topics/all?utm_source=sitepoint&utm_medium=link&utm_content=top-nav
19. Wikipedia, “*Pure functions*”, retrieved on 2015-07-10.
https://en.wikipedia.org/wiki/Pure_function
20. Shelby More III, “*terminology What is functional, declarative and imperative programming*”, 2011-12-02.
http://stackoverflow.com/questions/602444/what-is-functional-declarative-and-imperative-programming
21. Gilles, “*What exactly does semantically observable side effect mean*”, 2014-02-27.
http://cstheory.stackexchange.com/questions/21257/whatexactlydoessemanticallyobservablesideeffectmean
22. Diego Pettenò, ”*Implications of pure and constant functions*”, 2008-06-10.
https://lwn.net/Articles/285332/
23. ARM, “*RealView Compilation Tools Compiler User Guide: 5.3.3. __pure*”, 2010.
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0205j/CJACCJIJ.html
24. Walter Bright, “*Pure Functions*”, 2008-09-21.
http://www.drdobbs.com/architecture%ADand%ADdesign/pure%ADfunctions/228700129
25. Eric White, “*Pure Functions*”, 2006-10-03.
http://blogs.msdn.com/b/ericwhite/archive/2006/10/03/purefunctions.aspx
26. Nico Espeopn, “*Pure functions in JavaScript*”, 2015-01-25.
http://nicoespeon.com/en/2015/01/purefunctionsjavascript/
27. Marijn Haverbeke, ”*Eloquent JavaScript*”, 2014-12-14.
http://eloquentjavascript.net/1st_edition/chapter3.html
28. Nolan O’Brien, “*__attribute__ directives in Objective-C*”, 2014-03-10.

https://blog.twitter.com/2014/attribute-directives-in-objective-c

29. Geeks for geeks, "*Pure functions*", retrieved on 2015-07-11. http://www.geeksforgeeks.org/purefunctions/
30. Gleb Bahmutov, "Test if a function is pure", 2014-11-12. http://bahmutov.calepin.co/testifafunctionispure.html
31. Wikipedia, "*Variable (mathematics)*", retrieved on 2015-07-12. https://en.wikipedia.org/wiki/Variable_(mathematics)#Notation
32. Richard Smith, "*Working Draft, Standard for Programming Language C++*", 2015-05-22. http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4527.pdf