Unsequenced functions N2539

ISO/IEC JTC 1/SC 22/WG14

June 25th, 2020

Proposal for C2x

Proposal category: New feature

Étienne Alepins

Jens Gustedt

Thales Canada, Avionics

INRIA, France

Revision history

Paper number	Changes
N2477 Const functions	Initial version
N2539 Unsequenced functions	Supersedes N2477 • Brand new language wording (extracted from N2522) • Withdrawal of applicability to standard library functions (moved to future papers)

Purpose

Multiple compilers on the market support the concept of ``const'' and ``pure'' functions (so coined by GCC), also sometimes called no-side-effects functions. In that terminology, a ``const'' function is a function that does not depend on or modify the global context of the program: it only uses its input parameters to modify its output parameters and return value. To avoid ambiguities with the properties of the const qualifier, we call such a function *unsequenced* because a call to such a function can be executed unsequenced with respect to unrelated evaluations, namely as soon as its input is available and as late as its result is used by another statement. Although widely supported, this concept is currently absent from the C standard. Unsequenced property can be gradually built from sub-properties:

- Noleak: function which does not leak dynamically allocated memory
- Stateless: function that does not define mutable static objects
- Idempotent: noleak, stateless and the function may read globals, but not write to them – **equivalent to GCC pure**
- Independent: noleak, stateless and does not depend on other state than the arguments (but may write to globals)
- Unsequenced: idempotent + independent – **equivalent of GCC const**

It is proposed to add support to all those functions attributes in C2x such that functions that aren't unsequenced can still benefit some of optimisations.

Use cases

Unsequenced functions can be leveraged in multiple ways to enhance performance and reduce code size. Optimization is key in a compiler: it allows adding more features to a given system or conversely selecting lower-power CPUs to perform the same tasks.

Global variables reload

Model-based systems define software using a series of graphical blocks such as confirmators, delays, digital filters, etc. Code generators are often used to produce source code. Some of these generators use global variables to implement the data flow between models. Each block is implemented by a call to a library function. Let's take a model with two confirmator blocks:

```
bOutput1 = CONF (bInput1, uConfTime1, bInit, bInitState1, &uCounter1);

bOutput2 = CONF (bInput2, uConfTime2, bInit, bInitState2, &uCounter2);
```

where all these are global variables. `bInit` represents the initialization state of the whole system (for `CONF`, it resets the counter output parameter to zero). Since the compiler must assume `CONF` may be modifying `bInit` (through a direct access to this global variable), it is forced to load it's value once before the first `CONF` call and a second time before the second `CONF` call. The second load can be avoided if the compiler is told `CONF` is an unsequencedfunction. It will then re-use `bInit` value from a register or stack, thus reducing the number of assembly instructions, i.e. potentially saving code size and performance. Code generators are not always flexible enough to put `bInit` in a local variable before calling blocks in order to work around compilers not supporting unsequenced functions.

In large model-based systems, there can be hundreds of such optimization opportunities, thus having a major overall positive impact.

Function calls merge

Multiple use cases of potentially unsequenced function exist in the standard library, taking for example the `cos` <math.h> function:

```
if ((cos (angle1) > cos (angle2)) || (cos (angle1) > cos (angle3)))
{
        …
}
```

In this case, the compiler will invoke four times the `cos` function, with three different parameters. If `cos` would be declared unsequenced, the compiler could have merged the first and the third calls, reusing the result of the first call for the third, thus reducing CPU throughput and code size.

A large portion of the <math.h> functions fit the unsequenced objective for most of their input domain. However, in case of errors, these functions may also modify errno or the floating-point environment, which contradicts the strict unsequenced property as formulated above. Special care has to be taken to allow the functions to be declared unsequenced and in a manner where additional constraints can be spelled out. More work needs to be done on that topic and this paper does not propose any attribute addition to standard library functions, yet.

Prior art

GCC

```
int square (int) __attribute__ ((const));
int hash (char *) __attribute__ ((pure));
```

Const attribute is implemented since GCC 2.5 and pure since GCC 2.96.

GCC distinguishes const from pure:

- Const: "functions [that] do not examine any values except their arguments, and have no effects except the return value."
- Pure: "functions [that] have no effects except the return value and their return value depends only on the parameters and/or global variables."

LLVM Clang

Clang supports all GCC attributes.

WindRiver Diab 5.x

```
#pragma pure_function function({global | n},...), ...
#pragma no_side_effects function({global | n},...), ...
```

These pragma exist in Diab compiler since at least version 4.4. Diab uses "pure_function" for unsequenced functions and "no_side_effects" for idempotent functions.

- Const: "function does not modify or use any global or static data."
- Pure: "function does not modify any global variables (it may use global variables)."

In addition to the basic idempotent/unsequenced, Diab allows functions to still access some specially marked global variable. This is achieved by passing parameters to the pragma. It is the only compiler to our knowledge that goes beyond the GCC const/pure concept. As proposed in N2522, this could be used to solve the errno and floating-point environment challenge of standard library functions. Nevertheless, we think that the general idempotent and unsequenced properties are currently more widely supported and so we concentrate this paper on these.

Diab 5.x compiler is a WindRiver proprietary compiler, not based on GCC or LLVM. It is in particular used in the highly popular WindRiver VxWorks RTOS for aerospace, automotive and industrial. It supports C99 (C11 experimental) and C++14 (except thread_local, some Unicode types and some library functions related to atomic, chron/signal, thread, filesystem and localization).

Not to be confused with Diab 7.x which is a branch based on LLVM.

GreenHills MULTI

```
__attribute__((const))
__attribute__((pure))
```

MULTI supports GCC const and pure attributes since at least version 4.0.

Ada language – GNAT

```
pragma Pure_Function ([Entity =>] function_LOCAL_NAME);
```

AdaCore GNAT Ada uses "Pure_Function" for const functions, i.e. functions where "the compiler can assume that there are no side effects, and in particular that two calls with identical arguments produce the same result."

SPARK language also has the "Global" aspects which is similar. Efforts are in progress to incorporate const/pure support in the next revision of Ada language.

Fortran

Fortran allows a function to be declared PURE, meaning that the function has no side effect. That is the equivalent of GCC const.

Standardization

Const/pure compiler-specific features should be promoted to language features in order to:

- Increase performance in systems written in C, whatever the compiler used. Standardization will encourage wider support for that optimization feature.

- Allow portability. Although some macro *magic* might be possible in order write portable code using const/pure – macros could map to compiler-specific feature – this would be more complex and might not even be possible (e.g. reconciliation of pragmas and attributes implementations, positioning constraints of const/pure within function declaration/definition, etc.)

Implementation in C2x

Unsequenced and its sub-properties fit well into realm of the new attribute feature of C2x, because conforming code that has these attributes would remain conforming if the attribute is removed or ignored. The use the attribute feature is clearly advantageous over pragmas (attributes have well-defined applicability) and the introduction of new keywords which suggest a semantic change.

Therefore this paper proposes to use the new attribute system to implement the properties. The names "const" and "pure" are not used because their definition is split into 5 sub-properties and because "pure" has been used in the industry for different semantics (Fortran, Ada, Diab compiler).

Diagnostics

GCC 8.3.0 does not seem to raise warnings when const functions modify global variables.

AdaCore GNAT Ada allows pure/const functions to modify global variables. This is to support cases like instrumentation (e.g. call counter), table-caching implementations, debug traces, etc.

However, in order to avoid source code bugs, when violation of the properties' objective is encountered, it is proposed to have compiler errors when possible, recommended practice diagnostic warning otherwise. Indeed, wrongly declaring as unsequened a function might produce functionally incorrect executable code.

Modifying global context includes calling non-idempotent functions. Reading global context includes calling non-unsequenced functions.

WindRiver Diab const/pure attribute allows to specify exceptions, i.e. global variables that a const/pure function might still read/write. That may be used to work around above mentioned GNAT use case. However, it is proposed not to introduce this for now.

We propose a more constraining approach than most current implementations and aim to enforce consistency of the sought properties where this is possible. Thus, such properties are generally formulated as constraints when visible in the same translation unit (and require a diagnostic) and they make any program execution undefined when inconsistent translation units are linked into one program.

Relationship with other language features

Function pointers

ISO/IEC 9899:202x "6.7.3 Type qualifiers" mentions that *"If the specification of a function type includes any type qualifiers, the behavior is Undefined."* Hence, these properties do not conflict with const, restrict, volatile or _Atomic keywords.

However, this paper proposes these properties not to be applicable to function pointers. To take advantage of optimizations brought by these properties when using function pointers, the attributes should be part of the function prototype, which is not currently compatible with C. Applicability to function pointers may be discussed in a future paper.

inline

Although there is no conflict here, it may seem superficial to add any of these properties to an inline function since the translator sees the function's body and can assess whether it has side effects or depends on the global context. However, it is proposed to allow that combination to allow the programmer to profit from the corresponding diagnosis and to explicitly enforce constraints where these properties can't be automatically deduced.

_Noreturn

Although none of the two above mentioned use cases apply for a _Noreturn function, it might still be useful to declare a _Noreturn function with one of the properties: it allows the _Noreturn function programmer to add static check that it's function does not rely on or modifies global variables. This combination is more likely to be useful for idempotent than for unsequenced, though.

nodiscard

nodiscard attribute requires function calls to use a function's return value. This is compatible with the properties.

Rationale for some of the choices

The current implementations of these features leave a lot of the properties to be defined by the user, in particular which accesses would constitute an access to global state, be it for reading or writing. Here, this paper proposes to make three dedicated choices:

1. An access to a const-qualified object is only considered invariant if the object is not additionally volatile-qualified. This is because any access to a volatile-qualified object may have a different result according to the context where it is placed, and in particular repeated such accesses can lead to different results.

2. A call to an allocation function changes the global state of the current thread and thus at a first glance any such calls should be forbidden for idempotent or independent attributes. Nevertheless, we consider temporary allocations that are freed before the end of the function as not modifying the state. Because such temporary allocations may constitute an important use case, we think that these should be allowed and asserted by the noleak attribute. Because such a "noleak" property may not be automatically verifiable, its assertion will usually be the duty of the programmer.

3. Calls to the special library function call_once have a special status, because they basically enforce that a certain state is reached before any execution may progress beyond such a call. The callbacks that are used for such calls usually change global state (otherwise there is not much point to them) and the change in the underlying once_flag by itself constitutes a state change. But since the guarantee is that the call is only effected exactly once, it can be thought to take place before any execution of the function that we want to annotate.

WG21 C++ liaison

These properties seem compatible with C++. It would be beneficial for the whole C and C++ communities that they be implemented the same way in both languages. Discussions should take place for this harmonization.

Link-Time Optimization (LTO)

Link-time optimization can achieve some of the optimization benefits these properties bring. However, in addition to lacking manually-enforced constraints addition capabilities, it is not always possible to enable LTO. For example, multiple products in safety-critical markets such as avionics, nuclear, automotive and railway explicitly disable LTO in order to keep testing credit obtained by testing individual object files. Moreover, not all linkers offer LTO.

Proposed Wording

Proposed changes are extracted from N2522. Based on ISO/IEC 9899:202x working draft N2478. Original text is in black or purple, modified/new text is in blue. *[…]* indicates skipped text.

6.7.11 Attributes

[…]

6.7.11.1 General

[…]

Constraints
2 The identifier in a standard attribute shall be one of:

deprecated	**idempotent**	**maybe_unused**	**noleak**	**unsequenced**
fallthrough	**independent**	**nodiscard**	**stateless**	

[…]

6.7.11.6 Function attributes

Constraints
1 The identifier in a function attribute shall be one of:

idempotent	independent	noleak	stateless	unsequenced

2 Unless specified otherwise, the attribute tokens of function attributes shall only appear in the attribute specifier sequence that belongs to a function declarator. *FOOTNOTE)* If they appear in a function declarator, the function declarator shall be used to declare a function, and the corresponding attribute is a property of the function itself and not of its type. The attribute tokens shall appear at most once in each attribute list. Unless stated otherwise, the attribute argument clause shall be omitted.

3 For a given attribute token and a declaration of a function with that attribute the following constraints hold.

- If the translation unit forms the definition of the function, that definition shall have the attribute and the attribute argument clause, if any, shall be identical.
- Otherwise, if the function has external linkage and refers to a definition in another translation unit that does not have the attribute or with a different attribute argument clause, the behavior is undefined.

Description
4 The attributes described are not part of the prototype of a such annotated function, and the knowledge about the attribute might get lost when forming a function pointer, and, in particular, when passing such a function pointer between different translation units. Thus, the sought properties by the annotation with such attributes are only effectively diagnosable if the designator in a function call is the name of a function.

5 The attributes defined in this clause provide optimization opportunities for functions. Their main goal is to provide the translator with information about the access of objects such that it may deduce certified properties. This

certification is ensured by forcing the attributes to be consistently present for a function definition if any declaration has them.

FOOTNOTE) That is, they appear right after the parameter list, if any, and before the function body or semicolon.

6.7.11.6.1 The noleak attribute

Constraints
1 For a function definition that fullfils the following conditions the noleak attribute is implied:

- — Any function that is called has the noleak attribute or is the call_once library function. For the latter, the first argument to such a call is a pointer to an object of type once_flag that has static storage duration.
- — It calls none of the allocation functions (??).

2 If the noleak attribute is applied to the definition of a function, any function specifier that is evaluated within the function body, shall have the noleak attribute or shall be the call_once library function. For the latter, the first argument to such a call shall be a pointer to an object of type once_flag that has static storage duration.

Description
3 A storage leak is an allocation and a possible control flow such that no corresponding deallocation can be deduced by the translator. The noleak attribute asserts that the annotated function does not leak any allocations.

4 Any storage that is allocated during any call to a function annotated with the noleak attribute shall be deallocated before the end of the call, with the exception of an allocated return value.

Recommended Practice
5 It is recommended that for functions with a noleak attribute implementations diagnose if a pointer to newly allocated storage is not used as an argument to free or realloc, unless it is the return value of the function.

6 NOTE For this definitions, allocations that are effected by an initializer function that is guarded by a once_flag with static storage duration are not considered leaks. In contrast to that, because the number of threads that an application starts can in general not be bounded, such allocations that are only guarded by a once_flag of thread storage duration are leaks. In both cases, applications are encouraged to deallocate storage that is such allocated by the appropriate means, that is by atexit handlers or tss_t destructors.

6.7.11.6.2 The stateless attribute

Constraints
1 If the attribute is applied to the definition of a function, any objects of static or thread storage duration that are defined in the function body shall be const-qualified and not volatile-qualified or have the type once_flag; if such an object has the type once_flag, outside of its definition it shall only be used by passing its address as the first argument to the call_once library function.

2 If the stateless attribute is applied to the definition of a function, any function specifier that is evaluated within the function body, shall have the stateless attribute.

3 For any function definition that fulfills the above constraints the stateless attribute is implied.

Description
4 A function that could be declared with the stateless attribute is called stateless.

6.7.11.6.3 The idempotent attribute

Constraints
1 The stateless and noleak attributes are implied and the corresponding constraints apply. No file scope identifier shall be accessed to modify the corresponding object.

Description
2 An evaluation E is idempotent if it can be replaced by the evaluation (E,E) without changing the observable state of any execution. A function designator identified by an identifier f is idempotent, if the evaluation r = f(a1, ..., an) is idempotent, where the list a1, ..., an are const qualified variables that may range over the whole admissible set of function arguments (which may be empty), and where r is a variable with the non-void return type of the function. Analogously, f with a return type of void is idempotent if f(a1, ..., an) is idempotent with a1, ..., an as above.

3 No file scope identifier shall be accessed, even recursively, to modify the corresponding object.

4 A function definition that has the idempotent attribute shall be such that the function designator is idempotent.

6.7.11.6.4 The independent attribute

Constraints
1 The stateless and noleak attributes are implied and the corresponding constraints apply. No file scope identifier shall be used to access the corresponding object unless the type is const-qualified but not volatile-qualified.

Description
2 A function call is independent, if all lvalue conversions or accesses to objects that are effected during the call, inclusive calls into other functions, refer to objects that are

- — function parameters of the called function,
- — objects with static or thread-local storage duration that are const but not volatile qualified or that have type once_flag,
- — objects for which the definition is met during the call, or
- — objects that are allocated during the call.

This not withstanding, it may read or modify objects or other execution state

- — indirectly through a call to the library function call_once with a first argument of static or thread storage duration,
- — or by using memory management functions provided that the use is consistent with the noleak attribute.

3 No file scope identifier shall be used, even recursively, to access the corresponding object unless the type is const-qualified but not volatile-qualified.

4 A function definition is independent, if all function calls with the function designator and valid parameters are independent.

5 A function definition that has the independent attribute shall be such that the function designator is independent.

6.7.11.6.5 The unsequenced attribute

Constraints

1 The independent and idempotent attributes are implied and the corresponding constraints apply.

2 For any function declaration that has the independent and idempotent attributes the unsequenced attribute is implied.

Description

3 The unsequenced attribute indicates that a call to a function can be effected as soon as the values of its parameters have been determined, and that it can be effected as late as any of its return value or modified pointed-to parameters are accessed.

4 **NOTE** The unsequenced attribute asserts strong properties for the annotated function. Thereby, calls to such functions are natural candidates for optimization techniques such as common subexpression elimination, local memoization or lazy evaluation.

Acknowledgements

Thanks to Rajan Bhakta for his support and corrections.