Const functions N2477

ISO/IEC JTC 1/SC 22/WG14 February 3rd, 2020

Proposal for C2x

Étienne Alepins

Thales Canada, Avionics

Proposal category: New feature

Purpose

Multiple compilers on the market support the concept of const and pure functions, also sometimes no-side-effects functions. 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. This concept is absent from the C standard.

Use cases

Const 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 a const function. 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 const 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 const 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 const, the compiler would 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 could be declared const, thus allowing portable optimization improvements in all systems.

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 sum
#pragma no_side_effects descriptor,...
```

These pragma exist in Diab compiler since at least version 4.4. Diab uses "pure_function" for const functions and "no_side_effects" for pure 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)."

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.

Const and pure

It is proposed to add support to both const and pure functions in C2x. Otherwise, functions needed to read global variables could not benefit at all from the const optimisation.

Standardization

One could argue that const/pure could be left as compiler-specific features rather than being added to C2x standard. However, this has two drawbacks:

- That optimization would not be available on some systems due to lack of compiler support. Standardization will encourage wider support for that optimization feature.
- 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

Const/pure could be added as pragmas. However, there are currently only a few standard pragmas and they are more floating-point focused.

A new keyword could be added. Indeed, const/pure is close to _Noreturn which was implemented as a new keyword in C11. However, C2x is proposing to move _Noreturn to an attribute. Adding const/pure as keyword would complexify the list of basic keywords.

This paper proposes to use the new attribute system to implement const/pure. "const" and "pure" attribute names are proposed rather than the longer "const_function" and "pure_function" although the "const" attribute will have the exact same name as the "const" keyword. However, no entity can be applied both the "const" keyword (on types) and the "const" attribute (on functions).

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, it is proposed to have a recommended practice diagnostic warning when a const function reads or modifies the global context, or when a pure function modifies the global context. Indeed, wrongly declaring as pure/const a function might produce functionally incorrect executable code.

Modifying global context includes calling non-pure/const functions. Reading global context includes calling non-const 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 complexity.

Rather, for the cases described by GNAT documentation, warnings could simply be disabled, either globally or locally to the function. However, for that reason, warnings are proposed, not errors.

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, const/pure function does not conflict with const, restrict, volatile or _Atomic keywords.

inline

Although there is no conflict here, it makes little sense to add const/pure to an inline function since the implementation already sees the function's body and can already assess whether it has side effects or depends on the global context. It is proposed to allow that combination.

_Noreturn

Although none of the two above mentioned const/pure use cases apply for a _Noreturn function, it might still be useful to declare const/pure a _Noreturn function: 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 pure than for const though.

nodiscard

nodiscard attribute requires function calls to use a function's return value. This is compatible with const/pure.

Proposed Wording

Proposed changes based on ISO/IEC 9899:202x working draft — November 18, 2019. 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` `fallthrough` `maybe_unused` `nodiscard` `const` `pure`

[...]

6.7.11.6 The const attribute

Constraint
1 The **const** attribute shall be applied to the identifier in a function declarator. It shall appear at most once in each attribute list and no attribute argument clause shall be present.

Semantics
2 A name or entity declared without the **const** attribute can later be redeclared with the attribute and vice versa. An entity is considered marked after the first declaration that marks it.

Recommended Practice
3 The **const** attribute indicates that a function has no side effects and that it does not depend on the state of the execution environment. The only objects on which the return value (if any) and value of objects pointed by parameters of such a function (if any) may depend upon are its parameters (if any).

4 NOTE 1 This allows implementations to perform optimizations.

5 A function marked with the **const** attribute is a *const function*. Implementations are encouraged to issue a diagnostic when a const function depends on or modifies the state of the execution environment. This includes const functions calling non-const functions.

6 EXAMPLE 1

```
int g, h;
[[const]] int s(int a);
void f()
{
   g = s(4) + s(5);
   h = s(5) + s(6);   // s(5) call can be optimized-out
}
```

The call to s(5) in the assignment to h can be optimized-out by an implementation, re-using the return value of the first s(5) call.

7 EXAMPLE 2

```
int g, h, i, j;
[[const]] int s(int a, int b);
void f()
{
   g = s(4, i);
   j = 10;         // j assignment can be moved across calls
   h = s(5, i);    // i reload can be avoided
}
```

The value of i from the first call to s() can be used for the argument of the second call to s() since s() is known not to modify i. The assignment of j can be moved before the first s() call or after the second s() call if the implementation prefers this for optimization; indeed, s() is known not to depend on j's value.

8 EXAMPLE 3

```
int g;
```

```
int f(int a);
[[pure]] int r(int a);
[[const]] int s(int a)
{
   static int p; // Diagnose declaration of static variable
   p += g;       // Diagnose use of g global variable
   p += f(a);    // Diagnose use of non-const function f()
   p += r(a);    // Diagnose use of non-const function r()
   g = p;        // Diagnose modification of g global variable
   return p;
}
```

Diagnostic messages can be emitted when global or static variables are used or updated and when non-const function are called.

6.7.11.7 The pure attribute

Constraint

1 The **pure** attribute shall be applied to the identifier in a function declarator. It shall appear at most once in each attribute list and no attribute argument clause shall be present.

Semantics

2 A name or entity marked both as **pure** and **const** is considered **const**. A name or entity declared without the **pure** attribute can later be redeclared with the attribute and vice versa. An entity is considered marked after the first declaration that marks it.

Recommended Practice

3 The **pure** attribute indicates that a function has no side effects. It may however depend on the state of the execution environment.

4 NOTE 1 This allows implementations to perform optimizations.

5 A function marked with the **pure** attribute is a *pure function*. Implementations are encouraged to issue a diagnostic when a pure function modifies the state of the execution environment. This includes pure functions calling functions which are not pure or const.

6 EXAMPLE 1

```
int g, h;
[[pure]] int s(int a);
void f()
{
   int p;
   g = s(4) + s(5);
   p = s(5) + s(6); // s(5) call cannot be optimized-out
   h = s(6) + p;    // s(6) call can be optimized-out
}
```

The call to s(6) in the assignment to h can be optimized-out by an implementation, re-using the return value of the first s(6) call. However, calls to s(5) cannot be merged because the value of g has changed in between the calls and s() may use g.

7 EXAMPLE 2

```
int g, h, i;
[[pure]] int s(int a, int b);
void f()
{
   g = s(4, i);
   h = s(5, i);   // i reload can be avoided
```

```
}
```

The value of i from the first call to s() can be used for the argument of the second call to s() since s() is known not to modify i.

8 EXAMPLE 3

```
int g;
int r(int a);
[[const]] int s(int a)
{
   static int p; // Diagnose declaration of static variable
   p += g;
   p += r(a);    // Diagnose use of non-pure and non-const function r()
   g = p;        // Diagnose modification of g global variable
   return p;
}
```

Diagnostic messages can be emitted when global or static variables are used or updated or when functions are called.

Acknowledgements

Thanks to Rajan Bhakta for his support and corrections.