X3J16/91-0012
ISO/IEC JTC1/SC22/WG21
N0045

Proposed extension to C++: `~const`

J. Thomas Ngo
<ngo@tammy.harvard.edu>

August 28, 1991

Abstract

This document is presented to the ANSI committee for the standardization of C++ (X3J16), and is intended for consideration by the extensions group.

I propose the addition of a new *type-specifier*, `~const`. Only members of a class may be specified as `~const`. A data member that is specified as `~const` can be modified even if the object of which it is a part is specified as `const`. Additionally—and this suggestion is left open for debate—a member function that is specified as `~const` can modify any data member in the object for which it is called, even if that object is specified as `const`. The primary reason for introducing the `~const` specifier would be to provide a way to specify how the bitwise representation of a `const` object might change, entirely via specifiers in the class declaration.

In the first section I define the `~const` specifier in the context of the *Annotated C++ Reference Manual* (ARM) and explore its ramifications. In the second and third sections I describe additional suggestions that should be considered ancillary to the `~const` proposal. These potential enhancements, which emerged from discussions of `~const` on the Usenet group *comp.std.c++*, include generalizations of `~const`-like syntax, namely `~volatile`, `~inline`, and `~virtual`.

1 Core proposal

1.1 Motivation

One often needs to modify the bit representation of an object without changing what the object represents. This is usually done for efficiency, practicality, or improved encapsulation. The purpose of the language construct proposed here is to improve the ability of C++ to accommodate such situations in an elegant and expressive manner.

The distinction between *bitwise* and *abstract* constness is essential. In the following explanation, I draw from valuable personal communication with Brian Kennedy <bmk@csc.ti.com>. First, here are some important language-independent definitions: A `const` object is *bitwise const* if the program will not modify its bit representation. It is *abstractly const* if the program will not modify the object represented (as defined by the programmer), but might modify its bits. The compiler will, for the benefit of the programmer, enforce the abstract constness of such an object, as defined by the programmer through the constness of its member functions.

The current C++ language specification implies that a object declared `const` is bitwise const if it is a predefined type or is of a class that lacks any user-defined constructor. Otherwise it is considered to be abstractly const. The primary purpose of this proposal is to provide an alternative way for C++ to distinguish between bitwise and abstractly const objects.

The currently accepted way to alter the bit representation of an object that is declared `const` is to cast away the `const` attribute as required. It has been pointed out that such casts can be useful in that they do call attention to code that can change the bit representation of a `const` object. Furthermore, I am told that efforts are underway to define clearly the effects of casting away `const`. However, the value of permitting casts from `const` has been debated, and it is clear that the mechanism leaves much to be desired in the way of expressive capability.

This proposal outlines an alternative to casting away `const`, using a new *type-specifier* called `~const`. Since this new specifier can coexist with casts from `const`, it is not necessarily suggested that such casts be disallowed. Rather, it is hoped that the `~const` mechanism will provide an alternative, more elegant way to specify in what ways the bit representation of a `const` object can be altered.

1.2 Definition in terms of the ARM

1. Following ARM 7.1.6 [Type Specifiers], a *type-specifier* can be:

   ```
   simple-type-name
   ...
   const
   volatile
   ~const
   ```

   The `~const` specifier may appear in the *type-specifier* of a non-`static`, non-`const`, nonpointer data member (`~const`pointers are covered below).

2. Following the same section, each element of a `const` array is `const`. Each nonfunction, non-`static`, nonpointer member of a `const` class object is `const` unless it is specified as `~const`.

3. Following ARM 8 [Declarators], a *cv-qualifier* can be:

   ```
   const
   volatile
   ~const
   ```

   The `~const` specifier may appear in the *cv-qualifier* of a pointer that is a non-`static` member. And as usual, "The *cv-qualifiers* apply to the pointer and not to the object pointed to." (ARM 8.2.1)

4. Following ARM 9.3.1 [The `this` Pointer] (this item could be dropped from the proposal):

   The `~const` specifier may appear in the *type-specifier* of a non-`static`, non-`const` member function.

   A `const` member function may be called for `const` and non-`const` objects. The type of `this` in such a member function of a class `X` is `const X *const`. A non-`const` member function that is not specified as `~const` may be called only for a non-`const` object. The type of `this` in such a member function is `X *const`. A member function that is specified as `~const` may be called for both `const` and non-`const` objects. The type of `this` in either case is `X *const`.

5. The `~const` specifier does not create any new distinct types. It merely removes the `const` attribute from `this` under specific circumstances. Therefore the introduction of this specifier should require no new type matching rules, aside from the rules concerning `this` that have been outlined above.

1.3 Examples

In my own code I have found a few examples in which it was necessary to cast away `const`. In each case, the `~const` specifier would have been an appropriate alternative.

Internal buffer. I have a Trie class which represents a collection of strings. When I want to retrieve any one of those strings, I need to write the result to a character buffer. I want the user to be able to retrieve a string from a Trie that is declared `const`, but do not want him or her to have to worry about allocating the memory for the character buffer. Hence it is necessary for the buffer to be allocated and maintained by the Trie itself. Here a `~const` data member is appropriate to represent the character buffer. Thus:

```
class Trie {
  private:
    // [Data members for implementing the Trie itself]
    // ...
    ~const char *~const buf;     // Could have made buf, bufsize
    ~const bufsize;              // static, defeating example
  public:
    const char* operator () () const; // retrieves a string
};

const char* Trie::operator () () const
{
    // If necessary, reallocate buf and change bufsize.
    // Write the new string to buf.
    return buf;
}
```

Self-resizing array. The abstract data type that this class represents is an array of infinite extent. The elements of the array are held in contiguous storage. Enough memory is allocated to store only the initialized elements. When an attempt is made to access an element that would reside beyond the memory already allocated, the class calls a private method `grow()`, which allocates new memory and copies array contents as necessary. This application calls for `grow()` to be `~const`. That is, the old code

```
void DynArray::grow() const
{
    DynArray *const t = (DynArray *const) this;
    // refer to buffer explicitly through t
}
```

would be replaced by

```
void DynArray::grow() ~const
{
    // refer to buffer implicitly through this
}
```

Secondary representation. This is a trumped-up example, because my real example is too application-specific. Say you have an container class A. Based on your intended usage, you have decided that it is best to store the elements in unsorted form. But occasionally (very rarely) you will want to know the i'th, j'th and k'th elements of some A that has been declared `const`. So you have decided to do some kind of sort, but only when it is needed. It is appropriate to store the order information in `~const` data members.

1.4 Debate

Is it really worth adding this new feature to the language? Below, I have labeled each idea with `[+]` or `[-]`, depending on whether I think it supports or detracts from the proposal.

`[+]` Adding a `~const` specifier would introduce no new keywords, and would not break existing code.

`[+]` The semantics of `~const` are simple: `~const` breaks the propagation of the `const` attribute from an aggregate or object to its components.

`[-]` No program can be written using the `~const` specifier that cannot already be written using casts from `const` instead.

`[-]` In the hands of the wrong programmer, the `~const` specifier could lead to sloppiness. `[+]` On the other hand, all of C++ relies on the neatness of the programmer. Consider, for example, the decision as to whether the `const` specifier should have been included in the C++ language definition. A careless programmer could easily declare all members of a class public and never use `const`, except where constrained to do so by existing libraries. This would not have been a good reason not to have defined the `const` facility.

`[+]` The `~const` specifier decorates the class declaration with precise information about which data and function members might not preserve the bit representation of a `const` object. To the compiler this might suggest opportunities for optimization. To the human reader it can express concisely what is happening in the mind of the class programmer with regard to constructs such as internal caches and buffers.

`[-]` Against `~const` member functions specifically: if a class contains a `~const` member function, then *none* of its data members can be considered bitwise const. The notational advantages of being able to specify abstract constness in this sweeping manner are highly debatable, and the practical advantages are virtually nil.

1.5 Impact on casts from `const`

Casts which remove the `const` attribute have enjoyed considerable debate between purists and pragmatists; their existence impacts issues ranging from code readability to storage implementation. In the end it is generally agreed that they are here to stay in one form or another, if only because of the existing body of code that would break were they to be disfavored.

I choose not to make a specific recommendation about the fate of casts from `const` because decisions about their continued existence and the introduction of `~const` need not be tightly linked. Although the two constructs perform overlapping functions, they can easily coexist. Nevertheless, the two issues are related, and during discussions on *comp.lang.c++* a number of observations were made.

- As presently worded, the ARM (§5.4) describes the effects of casting away the `const` attribute in a confusing manner. However, according to Brian Kennedy <bmk@csc.ti.com>, efforts are underway to clarify that the effects of such casts are well-defined and implementation independent. Therefore casts from `const` should not be replaced on the grounds of appearing to be implementation dependent.

- Having said that, the knowledge that a program is free of casts from `const` could expand a compiler's latitude to perform optimizations in an implementation independent manner (see next subsection). In particular, the compiler could propagate constants and be more free to use flavors of storage such as read-only or write-once memory. (Both of these benefits accrue from the fact that the `~const` specifier is available in the class declaration, not just in the implementation module.) Obviously it is not possible for a compiler to figure out whether a program is free of casts from `const`, but the `~const` specifier empowers the compiler writer to provide optional flags which permit meaningful optimizations at the risk of rendering casts from `const` implementation dependent.

- It is not clear whether there exist applications in which the cast from `const` mechanism cannot easily be replaced by the `~const` mechanism. Whether or not such applications exist obviously affects the relationship between `~const` and casts from `const`.

- Jim Adcock <jimad@microsoft.UUCP> made a distinction between "enabling" and "supporting" a feature. In my mind, the distinction is mostly aesthetic but it also has some practical aspects. A language construct *supports* a feature if one can use that feature in a way that is elegant, easily maintained, and simple to compile and optimize. Otherwise, it merely *enables* the feature. (Here is an extreme example: a colleague claims that C enables object-oriented programming; after all, with an appropriate set of coding conventions and good discipline one can manually implement virtual function tables and other object-oriented constructs.)

  I agree with Jim in his statement that cast-from-`const` merely enables the ability to alter the bit representation of a `const` object, whereas the `~const` mechanism would support it.

1.6 Impact on compiler implementation

In this subsection I explore in greater detail compiler implementation issues related to `~const`, namely optimization and choice of storage. As I am unfamiliar with compiler design, I have kept my remarks informal.

An object that is `const` might be fully, partially, or not at all bitwise const. The bitwise const parts of an object may be subject to constant propagation, even across calls to `const` methods. If their bit representations can be determined at compile time, it may be possible to place them in read-only memory. Otherwise, it may still be possible to place them in write-once or *discardable* memory—*i.e.*, virtual memory that is paged to disk only the first time after initialization. To my knowledge, calls to functions— even ones whose arguments are all bitwise const—cannot be subject to common subexpression elimination because C++ has no facility for declaring that a function has no side effects.[1]

[1]It is noted that GNU C++ contains or has contained such a facility, but that it was incompatible with the language defined by the ARM.

In a program that is known to be free of casts from `const`,[2] the extent to which a `const` object is bitwise const can be determined by inspection of the class declaration. A `const` object of a class that contains one or more `~const` member functions is not at all bitwise const, since any of its `const` member functions could contain a call to a `~const` method. If a class contains no `~const` member function, each data member that is not `~const` is bitwise const, unless that data member itself cannot be bitwise const.

1.7 Side note: constness can be subverted during a constructor

Reid Ellis <rae@utcs.toronto.edu> brought up an issue that might need to be addressed more explicitly in the ARM. Consider:

```
class vulnerable {
  public:
    vulnerable();
    int nonconst();
    int subverter() const;
  private:
    vulnerable& vref;
};
vulnerable::vulnerable() : vref(*this) {}

const vulnerable v;
v.subverter();
```

If `vulnerable::subverter()` needs to alter a data member of `v`, all it needs to do is refer to `v` through vref, *e.g.*, `vref.nonconst()`. I feel this is a violation (of the spirit, not the letter) of ARM 8.4.3 [References], which states: "A reference to a plain `T` can be initialized only with a plain `T`."

The problem is that while a `const` object is being constructed, it is considered non-`const` so that its data members can be initialized and manipulated. This provides a window of opportunity to initialize a non-`const` pointer or reference that is durable, *i.e.*, will still be available after the constructor is finished.

Perhaps the ARM ought to specify explicit rules regarding the initialization of non-`const` pointers and references within a constructor. It is not enough to say that `this` cannot be used to initialize such entities; at least one must also prohibit such initializations from components of `this`.

2 Other possible uses of `~const` syntax (ancillary)

2.1 Partial cast

The following item was suggested to me, but it causes problems. Unless these problems can be resolved, I would not suggest that it be included in the language.

I am told that a possible extension to C++ is to have run-time type information, through a facility similar to `typeof` in gcc/g++. The next few paragraphs use the syntax that appears in the

[2]As mentioned above, a compiler cannot figure out whether a program is free of casts from `const` without seeing the entire source code. However, the programmer could supply the compiler with this information through appropriate flags. Admittedly, this is an implementation dependent issue and is therefore not a direct concern of X3J16! I include §1.6, anticipating that the committee will ask what additional latitude for optimization a compiler implementor would have were the `~const` attribute to be admitted into the language specification.

gcc documentation. Briefly, the `typeof` facility permits referring to the type of an expression, and can be used any place that a type name could normally be used. Here is a simple example:

```
int x;
const typeof(x) y;    // y is a const int
```

The suggested extension is to permit the use of `~const` to remove the `const` attribute from an unknown type, as when `x` happens to be a macro parameter:

```
#define nonconst(a) ~const typeof(a)
nonconst(y) z;        // z is a non-const int
```

The semantics of `~const` in this context are very different from the semantics in the core part of this proposal. Here, `~const` removes the `const` attribute from the type that it modifies. In the main part of this proposal, `~const` breaks the propagation of the `const` attribute from an enclosing structure. This difference in semantics leads to ambiguity if (referring to the example above) `z` happens to be a member of some class, *e.g.*

```
class Bar {
    ~const typeof(y) z;
};
```

Do we mean for `z` to have the type of `x`, except with the `const` attribute removed? Or do we mean for `z` to be modifiable even if the Bar object of which it is a part happens to be `const`?

3 Similar specifiers: `~volatile`, `~virtual` (ancillary)

During Usenet discussions of `~const`, four additional specifiers were suggested:

`~volatile` Do allow optimization on this object, even if the enclosing object is declared `volatile`.

`~virtual` This method should not override any base class method, even if a virtual method with the same name and type signature exists.

`~register` Do not enregister this variable, even when optimizing.

`~inline` Do make this method a real function call.

Of these four specifiers, all except `~register` were deemed reasonable to propose. Good arguments in favor of `~register` were not found.

3.1 `~volatile`

It was agreed that `~volatile` would provide little in the way of optimizability and nothing in expressiveness. However, it was pointed out that since the `const` and `volatile` specifiers are generally handled together in a compiler, it would be easier than not to define `~volatile` in a manner symmetrical to `~const`. Hence, following ARM 7.1.6 [Type Specifiers], a *type-specifier* can be:

```
simple-type-name
...
const
volatile
~const
~volatile
```

.... Each element of a `volatile` array is `volatile`. Each nonfunction, non-`static` member of a `volatile` class object is `volatile` unless it is declared `~volatile`. And following ARM 8 [Declarators], a *cv-qualifier* can be:

```
const
volatile
~const
~volatile
```

3.2 `~inline`

Inline functions are inconvenient to debug using a source-level debugger. An `inline` function must be defined in the header file of a class so that its body will be available to the compiler in every module in which it is invoked. To use a debugger with the function, one must remove the `inline` specifier. Since this causes the function to have external linkage, one must move the function definition into the implementation module of the class so that it will be linked only once.

This procedure would be greatly simplified were it possible to leave the function definition in the header file. For nonmember functions it is sufficient to change the `inline` specifier to `static`, thus giving the function internal linkage. (According to ARM 7.1.1, "For a nonmember function an `inline` specifier is equivalent to a `static` specifier for linkage purposes.) However, for member functions, the `static` keyword additionally removes the `this` pointer from the parameter list, radically changing its meaning.

Thus, it is proposed that ARM 7.1.2 [Function Specifiers] be modified so that a *fct-specifier* can be:

```
inline
~inline
virtual
~virtual      (see below)
```

A member or nonmember function with the `~inline` specifier has default internal linkage (§3.3). It is guaranteed that the compiler will not inline calls to such a function.

It is noted that some compilers provide a flag that serves precisely this purpose.

3.3 `~virtual`

The purpose of `~virtual` would be to permit the compiler to diagnose a common *faux pas*. The programmer of a base class and the programmer of a derived class can unwittingly write methods with identical names and types. If the base class method is virtual, then the derived class method is placed, without warning, in the virtual function table.

To make matters worse, if the base and derived class methods are declared `public` and `private` respectively, then the derived class method is unexpectedly `public` when invoked through a call to the base class method. This follows from ARM 11.6 [Access to Virtual Functions].

The `~virtual` specifier would have the following definition. Adding to ARM 7.1.6 [Function Specifiers]:

> Some specifiers can be used only in a function declarations. A *fct-specifier* can be:
>
> ```
> inline
> ~inline
> virtual
> ~virtual
> ```
>
> The `virtual` and `~virtual` specifiers may be used only in declarations of non-`static` member functions within a class declaration; see §10.2.

And rewording the first paragraph of ARM 10.2 [Virtual Functions],

> A function `vf` that is a member of a class `base` is said to be *overridden* by a function `vf` that is a member of a class `derived` that is derived from `base`, if the following conditions are met:
>
> 1. The types (§8.2.5) of `base::vf` and `derived::vf` are identical.
> 2. The function `base::vf` is declared `virtual`.
> 3. The function `derived::vf` is not declared `~virtual`.
>
> If all of these conditions are met, then a call of `vf` for an object of class `derived` invokes `derived::vf` (even if the access is through a pointer or reference to `base`).
>
> It is conceivable that a method could be declared `~virtual virtual`. The two declarations do not clash; the `~virtual` specifier states that the method is not to be placed in the vtable of any base class, while the `virtual` specifier causes a new vtable entry to be created.
>
> In cases of multiple inheritance, ambiguities can arise. For example:
>
> ```
> class A {
>   public:
>     virtual void foo();
> };
> class B : public A {
>   public:
>     ~virtual void foo();
> };
> class C : public A, public B {
>   public:
>     void foo();
> };
> ```
>
> Does `C::foo` override `A::foo` or not? Applying the ambiguity rules described in ARM 10.1.1, `C::foo` is not virtual since the nonvirtual function `B::foo` dominates `A::foo`.
>
> In general, for a given function `derived::foo` the directed acyclic graph of base classes is searched for methods named `foo` that have the same type as `derived::foo`. If none

is found, then `derived::foo` is not virtual. If many are found and none dominates (§10.1.1) all of the others, there is an ambiguity and the compiler should report an error. Otherwise, it must be that exactly one is found, or many are found but one dominates all of the others. If this function is `virtual` then `derived::foo` overrides it; otherwise `derived::foo` is not virtual.

It might be useful also to provide a way to make an entire class `~virtual`. This would be tantamount to specifying that all of its member functions are `~virtual`, *i.e.*, that no base class virtual function should be overridden. By contrast with ARM 10.5c [Virtual Base Classes], this would be a property of the class, not of the derivation. I am not in favor of permitting base classes to be `~virtual` because of the potential for confusion with the semantics of virtual base classes.

Finally, it was pointed out that a declaration with complementary semantics would permit compilers to catch the opposite error, in which a derived class method is declared `virtual` with the intention of having it override a base class method with the same name and type signature, but a small type mismatch causes the derived class method to be given its own vtable entry—silently. For example:

```
class B {
  public:
    virtual void foo();
};
class D : public B {
  public:
    virtual void foo() const;
};
```

In this case, calls to foo() through a pointer of type B* that points to an object of type D will invoke B::foo(), not D::foo(), as the programmer may have intended. Errors of this kind are generally caught only after laborious debugging. In an independent discussion thread, I have seen John Chapin <jchapin@neon.stanford.edu> suggest that the `catch` keyword be given the following usage:

```
class B {
  public:
    virtual void foo();
};
class D : public B {
  public:
    catch virtual void foo() const;     // compile-time error
};
```

It would be a compiler error for a method specified with the `catch` keyword not to have a base class method which it properly overrides.

4 Flexibility of syntax

As an alternative to `~const`, the syntax `!const` was suggested. Here are two reasons why I originally chose `~const` instead of `!const`:

- `!const` seems to connote that the specified member is merely 'not const'. The semantics of my core proposal call for a more active *overriding* of constness that would otherwise be taken on by virtue of membership in an enclosing structure. I feel that this meaning is more closely suggested by `~const` ("destroy constness"?).
- People are used to seeing `~` in declarations (*i.e.*, in destructors), whereas the idea of seeing `!` is more foreign.

5 Acknowledgments

Many thanks to all those who have criticized this proposal so thoughtfully and constructively. In particular, let me thank some major contributors:

Jim Adcock <jimad@microsoft.UUCP> gave detailed opinions on a number of important issues, including the relationship with casts from **const**, and the idea of discardable memory. Also, he is to blame for **~inline** and **~virtual**.

Dag Bruck <dag@control.lth.se> looked at several early versions, convinced me not to leave out **~const** member functions, and brainstormed on some possible alternative uses of the **~const** syntax.

Brian Kennedy <bmk@csc.ti.com> explained to me the all-important distinction between bitwise and abstract constness, and shared parts of his X3J16 proposal to define more sharply the effects of casting away the **const** attribute in terms of bitwise and abstract constness.