

Doc No: SC22/WG21/N1601
J16/04-0041

Date: February 12, 2004

Project: JTC1.22.32

Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Fax: +1-928-438-4456
Email: hsutter@microsoft.com

Bjarne Stroustrup
Computer Science Dept.
Texas A&M University, TAMU 3112
College Station TX USA 77843-3112
Fax: +1-979-458-0718
Email: bs@cs.tamu.edu

A name for the null pointer: `nullptr` (revision 2)

This is a revision of our paper N1488 that incorporates the following EWG direction from the Kona meeting:

- Re `NULL` macro: No change is proposed, in order to preserve C compatibility.
- Re `nullptr`: It must be copyable, throwable, and passable through varargs.
- Re the type of `nullptr`: It should be named, and be a POD type with size equal to `sizeof(void*)`. Named variables of this type are allowed. `reinterpret_cast` to and from this type is allowed. It is both a pointer type and a pointer-to-member type; i.e., it matches both a `T*` and a `T::*` partial specialization. If it matches two partial specializations, you have an ambiguity.
- Re the name of the type of `nullptr`: `nullptr_t` is introduced as a typedef for `decltype(nullptr)`.

In addition, this paper includes proposed wording for the feature.

1. The Problem, and Current Workarounds

The current C++ standard provides the special rule that `0` is both an integer constant and a null pointer constant. From [C++03] clause 4.10:

A null pointer constant is an integral constant expression (expr.const) rvalue of integer type that evaluates to zero. A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type and is distinguishable from every other value of pointer to object or pointer to function type. Two null pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to cv-qualified type is a single conversion, and not the sequence of a pointer conversion followed by a qualification conversion (conv.qual).

This formulation is based on the original K&R C definition and differs from the definition in C89 and C99. The C standard [C99] says (clause 6.3.2.3):

*An integer constant expression with the value `0`, or such an expression cast to type `void *`, is called a null pointer constant.[55] If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.*

This use of the value `0` to mean different things (a pointer constant and an `int`) in C++ has caused problems since at least 1985 in teaching, learning, and using C++. In particular:

- *Distinguishing between null and zero.* The null pointer and an integer `0` cannot be distinguished well for overload resolution. For example, given two overloaded functions `f(int)` and `f(char*)`, the call `f(0)` unambiguously resolves to `f(int)`.¹ There is no way to write a call to `f(char*)` with a null pointer value without writing an explicit cast (i.e., `f((char*)0)`) or using a named variable.
- *Naming null.* Further, programmers have often requested that the null pointer constant have a name (rather than just `0`). This is one reason why the macro `NULL` exists, although that macro is insufficient. (If the null pointer constant had a type-safe name, this would also solve the previous problem as it could be distinguished from the integer `0` for overload resolution and some error detection.)

To avoid these problems, `0` must mean only one thing (an integer value), and we need to have a different name to express the other (a null pointer).

This problem falls into the following categories:

- Improve support for library building, by providing a way for users to write less ambiguous code, so that over time library writers will not need to worry about overloading on integral and pointer types.
- Improve support for generic programming, by making it easier to express both integer `0` and `nullptr` unambiguously.
- Make C++ easier to teach and learn.
- Remove embarrassments.

We propose that a desirable solution should be able to fulfill the following design goals:

- The name for the null pointer should be a reserved word.
- The null pointer is a value that has no nameable type, and its type cannot be deduced as a template argument.
- The null pointer cannot be used as the argument of a `sizeof`, `typeid`, or `throw` expression; a diagnostic is required.
- The null pointer cannot be used in an arithmetic expression, assigned to an integral value, or compared to an integral value; a diagnostic is required.

¹ An alternative description of this effect might be: “`0` is always both an integer constant and a null pointer constant, except when it’s not.”

- The null pointer can be converted to any pointer type, and cannot be converted to any other type including any integral type.

1.1 Alternative #1: A Library Implementation of `nullptr`

Perhaps the closest current workaround is to provide a library implementation of `nullptr`. This alternative is based on [Meyers96] Item 2:

```

const                                     // this is a const object...
class {
public:
    template<class T>                     // convertible to any type
        operator T*() const              // of null non-member
        { return 0; }                    // pointer...

    template<class C, class T>           // or any type of null
        operator T C::*() const          // member pointer...
        { return 0; }

private:
    void operator&() const;              // whose address can't be taken
} nullptr = {};                          // and whose name is nullptr

```

There is one real advantage to this workaround:

- It does not make `nullptr` a reserved word. This means that it would not break existing programs that use `nullptr` as an identifier, but on the other hand it also means that its name can be hidden by such an existing identifier. (Note: In practice, the name is intended to be pervasively used and so will still be effectively a reserved word for most purposes.)

There is one apparent advantage that we believe is less significant in practice:

- It provides `nullptr` as a library value, rather than a special value known to the compiler. We believe it is likely that compiler implementations will still treat it as a special value in order to produce quality diagnostics (see note below).

This alternative has drawbacks:

- It requires that the user include a header before using the value.
- It does not prevent `nullptr` from being used as the argument of a `sizeof`, `typeid`, or `throw` expression.
- It does not support usages like `if(nullptr)`, although these will probably not be common.
- Experiments with several popular existing compilers show that it generates poor and/or misleading compiler diagnostics for several of the common use cases described in section 2. (Examples include: “no conversion from ‘const’ to ‘int’”; “no suitable conversion function from ‘const class <unnamed>’ to ‘int’ exists”; “a template argument may not reference an unnamed type”; “no operator ‘==’ matches these operands, operand types are: int == const

class <unnamed>”). We believe that compilers will still need to add special knowledge of `nullptr` in order to provide quality diagnostics for common use cases.

- Although available for many years, it has not been widely adopted.

1.2 Alternative #2: `(void*)0`

A second alternative solution would be to accept `(void*)0` as a “magic” pointer value with roughly the semantics of the `nullptr` proposed in section 2.

However, this solution has serious problems:

- It would still be necessary for programmers to use the macro `NULL` to name the null pointer (the notation `(void*)0` is just too ugly).
- Furthermore, `(void*)0` would have to have a unique semantics; that is, its type would *not* be `void*`. We do not consider opening the C type hole by allowing any value of type `void*` to any `T*`.

The introduction of `nullptr` as proposed in section 2 is a far cleaner solution.

2. Our Proposal

We propose a new standard reserved word `nullptr`. The `nullptr` keyword designates a constant rvalue of type `decltype(nullptr)`, assuming that a `decltype` proposal passes. We also provide the typedef:

```
typedef decltype(nullptr) nullptr_t;
```

`nullptr_t` is not a reserved word. It is a typedef (as its `_t` typedef indicates) for `decltype(nullptr)` defined in `<cstdint>`. We do not expect to see much direct use of `nullptr_t` in real programs.

`nullptr_t` is a POD type, and `sizeof(nullptr_t) == sizeof(void*)`. `nullptr_t` is both a pointer type and a pointer-to-member type.

All objects of type `nullptr_t` are equivalent and behave identically, except that they may differ in cv-qualification and whether they are rvalues or lvalues. The address of `nullptr` itself cannot be taken; another `nullptr_t` object’s address could be taken, although this isn’t very useful. Objects of type `nullptr_t` can be copied and thrown.

An object of type `nullptr_t` can be converted to any pointer or pointer-to-member type by a standard conversion. It cannot be converted to any other type including any integral or `bool` type, cannot be used in an arithmetic expression, cannot be assigned to an integral value, and cannot be compared to an integral value; a diagnostic is required for these cases.

With this specification for `nullptr` and `nullptr_t`, the following points follow from the existing rules already in the standard:

- Performing a `reinterpret_cast` to and from a `nullptr_t` object is allowed (this is already covered by saying that `nullptr_t` is a pointer type, see [C++03] §5.2.10).

- `nullptr_t` matches both a `T*` and a `T::*` partial specialization. If it matches two partial specializations of the same template, the result is ambiguous because neither partial specialization is more specialized than the other (see [C++03] §14.5.4.2).

We recommend that the name of the reserved word be `nullptr` because:

- `nullptr` says what it is. For example, it is not a null reference.
- Programmers have often requested that the null pointer constant have a name, and `nullptr` appears to be the least likely of the alternative text spellings to conflict with identifiers in existing user programs. For example, a Google search for `nullptr cpp` returns a total of merely 150 hits, only one of which appears to use `nullptr` in a C++ program.
 - The alternative name `NULL` is not available. `NULL` is already the name of an implementation-defined macro in the C and C++ standards. If we defined `NULL` to be a keyword, it would still be replaced by macros lurking in older code. Also, there might be code “out there” that (unwisely) depended on `NULL` being `0`. Finally, identifiers in all caps are conventionally assumed to be macros, testable by `#ifdef`, etc.
 - The alternative name `null` is impractical. It is nearly as bad as `NULL` in that `null` is also a commonly used in existing programs as an identifier name and (worse) as a macro name. For example, a Google search for `null cpp` returns about 180,000 hits, of which an estimated 3%² or over 5,000 use `null` in C++ code as an identifier or as a macro.
 - Any other name we have thought of is longer or clashes more often.
- The alternative spelling `OP` or `Op`, adding the letter as a constant type suffix, is impractical. It overlaps with a C99 extension that already uses `P` or `p` in a constant to write the binary exponent part of a hexadecimal floating-point constant (see [C99] clause 6.4.4.2). For example, `OP` occurs as a part of the constant `0xOP2`. Although using `OP` or `Op` would not be ambiguous today (the C99 `P` or `p` must be preceded by `0x` and a hex number, and must be followed by a decimal number), it seems imprudent to reuse a constant type suffix already used for another type of constant in a sister standard. Also, using an obscure notation, such as `OP`, would encourage people to rely on a `NULL` macro.
- Our informal polling suggests that people seem to like `nullptr`. If nothing else, it is the spelling that has elicited the fewest strong objections to date in our experience.

We should not define the standard library macro `NULL` to `nullptr`. It is tempting to do so, but it would break code, even though in many cases that would be code that deserved to be broken. New code should use the cleaner and safer `nullptr`.

2.1 Basic Cases

The following example illustrates basic use cases: assignment, comparison, and arithmetic operations.

```
char* ch = nullptr; // ch has the null pointer value
char* ch2 = 0;     // ch2 has the null pointer value
```

² Based on inspection of the first 300 hits, in which there were nine code hits (most related to Qt’s `QString::null`).

A name for the null pointer: `nullptr` (revision 2)

```

int n = nullptr;    // error
int n2 = 0;        // n2 is zero

if( ch == 0 );     // evaluates to true
if( ch == nullptr ); // evaluates to true
if( ch );         // evaluates to false

if( n2 == 0 );    // evaluates to true
if( n2 == nullptr ); // error

if( nullptr );   // error
if( nullptr == 0 ); // error

// arithmetic
nullptr = 0;     // error, nullptr is not an lvalue
nullptr + 2;    // error

```

In particular, note that `0` can still be assigned to a pointer. This is essential for compatibility.

2.2 Advanced Cases

The following example illustrates additional use cases: the ternary operator, `sizeof`, `typeid`, `throw`, overload resolution, and template specialization.

```

// Ternary operator cases
//
char* ch3 = expr ? nullptr : nullptr;    // ch1 is the null pointer value
char* ch4 = expr ? 0 : nullptr;         // error, types are not compatible
int n3 = expr ? nullptr : nullptr;     // error, nullptr can't be converted to int
int n4 = expr ? 0 : nullptr;           // error, types are not compatible

```

```

// Sizeof, typeid, and throw
//
sizeof( nullptr );    // ok, must equal sizeof( void* )
typeid( nullptr );   // ok
throw nullptr;       // ok

```

```

// Overloading cases
//
void f( char* );
void f( int );

f( nullptr );        // calls f( char* )
f( 0 );              // calls f( int )

```

```

// Specialization cases
//
template<typename T> void g( T t );

g( 0 );           // specializes g, T = int
g( nullptr );    // specializes g, T = nullptr_t
g( (float*) nullptr ); // specializes g, T = float*

```

```

// Partial specialization cases
//
template<typename T> class X { };
template<typename T> class X<T*> X { }; // case X<T*>
template<typename T> class X<T::*> X { }; // case X<T::*>

X<nullptr_t> x; // error, ambiguous; nullptr_t is both a pointer
                // and pointer-to-member type, so it's undecidable which
                // partial specialization to use

```

3. Interactions and Implementability

3.1 Interactions

See §2.2.

Effects on legacy code: Existing code that uses `nullptr` as an identifier will have to change the name of that identifier because it will be a reserved word.

3.2 Implementability

There are no known or anticipated difficulties in implementing this feature.

4. Proposed Wording

In this section, where changes are either specified by presenting changes to existing wording, ~~struckthrough text~~ refers to existing text that is to be deleted, and underscored text refers to new text that is to be added. Existing footnotes are unchanged unless otherwise indicated. All clause references are to [C++03].

In §2.11, Table 3, add `nullptr` to the list of keywords.

Change §4.10 as indicated:

4.10 Pointer conversions

[conv.ptr]

- 1 A *null pointer constant* is an integral constant expression (5.19) rvalue of integer type that evaluates to zero. A null pointer constant or an object of type `nullptr_t` can be converted to a

pointer type; the result is the *null pointer value* of that type and is distinguishable from every other value of pointer to object or pointer to function type. Two null pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to cv-qualified type is a single conversion, and not the sequence of a pointer conversion followed by a qualification conversion (4.4).

- 2 An rvalue of type “pointer to cv **T**,” where **T** is an object type, can be converted to an rvalue of type “pointer to cv **void**.” The result of converting a “pointer to cv **T**” to a “pointer to cv **void**” points to the start of the storage location where the object of type **T** resides, as if the object is a most derived object (1.8) of type **T** (that is, not a base class subobject).
- 3 An rvalue of type “pointer to cv **D**,” where **D** is a class type, can be converted to an rvalue of type “pointer to cv **B**,” where **B** is a base class (clause 10) of **D**. If **B** is an inaccessible (clause 11) or ambiguous (10.2) base class of **D**, a program that necessitates this conversion is ill-formed. The result of the conversion is a pointer to the base class sub-object of the derived class object. The null pointer value is converted to the null pointer value of the destination type.
- 4 The `nullptr` keyword designates a constant rvalue of type `std::nullptr_t` (18.1). [*Note: The address of `nullptr` cannot be taken. —end note*]

Change §4.11 as indicated:

4.11 Pointer to member conversions

[conv.mem]

- 1 A null pointer constant (4.10) or an object of type `nullptr_t` (4.10) can be converted to a pointer to member type; the result is the *null member pointer value* of that type and is distinguishable from any pointer to member not created from a null pointer constant. Two null member pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to member of cv-qualified type is a single conversion, and not the sequence of a pointer to member conversion followed by a qualification conversion (4.4).
- 2 An rvalue of type “pointer to member of **B** of type cv **T**,” where **B** is a class type, can be converted to an rvalue of type “pointer to member of **D** of type cv **T**,” where **D** is a derived class (clause 10) of **B**. If **B** is an inaccessible (clause 11), ambiguous (10.2) or virtual (10.1) base class of **D**, a program that necessitates this conversion is ill-formed. The result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in **D**’s instance of **B**. Since the result has type “pointer to member of **D** of type cv **T**,” it can be dereferenced with a **D** object. The result is the same as if the pointer to member of **B** were dereferenced with the **B** sub-object of **D**. The null member pointer value is converted to the null member pointer value of the destination type.⁵²⁾

In §18.1, insert a new row in Table 15 as follows:

Table 15—Header `<cstdlib>` synopsis

Kind	Name(s)
Macros:	<code>NULL</code> <code>offsetof</code>
Types:	<code>ptrdiff_t</code> <code>size_t</code>
Typedefs:	<code>nullptr_t</code>

Also in §18.1, insert the following new paragraph:

6 `nullptr_t` is defined as follows:

```
namespace std {  
    typedef implementation defined nullptr_t;  
}
```

The type for which `nullptr_t` is a synonym shall have the following characteristics: It is a POD type, and `sizeof(nullptr_t)` shall be equal to `sizeof(void*)`. `nullptr_t` is both a pointer type and a pointer to member type. Objects of type `nullptr_t` can be copied and thrown. An object of type `nullptr_t` shall not be converted to an object of any type other than a pointer or pointer to member type, shall not be used in an arithmetic expression, shall not be assigned to an integral value, and shall not be compared to an integral value; a diagnostic is required. All objects of type `nullptr_t` are equivalent and behave identically (see also the definition of `nullptr` in 4.10), except that they may differ in cv-qualification and whether they are rvalues or lvalues. [*Note*: Although `nullptr`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue could be taken. —*end note*]

References

[C99] ISO/IEC 9899:1999(E), *Programming Language C*.

[C++03] ISO/IEC 14882:2003(E), *Programming Language C++*.

[Meyers96] S. Meyers. *More Effective C++, 2nd edition* (Addison-Wesley, 1996).