Doc No: SC22/WG21/N2431 = J16/07-0301

Date: 2007-10-02

Project: JTC1.22.32

References: Revision of SC22/WG21/N2214 = J16/07-0074

Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Email: hsutter@microsoft.com

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

A name for the null pointer: nullptr (revision 4)

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. For another example, consider the following oddity in Standard C++:

```
std::string s1( false );     // compiles, calls char* constructor with null
std::string s2( true );      // error
```

- *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:

1. The name for the null pointer should be a reserved word.
2. 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.
3. The null pointer can be converted to any pointer type, and cannot be converted to any other type including any integral type.
4. The integer 0 does not implicitly convert to any pointer type.

Obviously, (4) is infeasible because it would break spectacular amounts of code, so we don't propose that.

[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."

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 25:

```
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.
- Since nullptr doesn't implicitly convert to bool, 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 and incompatible variants are not uncommon.
- An elaborate class-based solution would cause problems in constant expressions as will become common with the adoption of generalized constant expressions (constexpr).

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). 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 <cstddef>. We do not expect to see much direct use of nullptr_t in real programs.

nullptr_t is a POD type that is convertible to 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 (it is a literal, just like 1 and true); 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% or over 5,000 use null in C++ code as an identifier or as a macro. Another favorite, nil, is worse still.
  - Any other name we have thought of is longer or clashes more often.
- The alternative spelling 0P or 0p, 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, 0P occurs as a part of the constant 0x0P2. Although using 0P or 0p 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 0P, 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 do not propose to define the standard library macro NULL to nullptr. We considered that and liked the idea, but the EWG opinion was that it would break too much 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.

```
char* ch = nullptr;  // ch has the null pointer value
char* ch2 = 0;       // ch2 has the null pointer value
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, no conversion to bool
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
typeid( nullptr );       // ok
throw nullptr;           // ok
```

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

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

```
// Deduction to nullptr_t, no deduction to pointer type
//
template<typename T> void g( T* t );

g( nullptr );                   // error
g( (float*) nullptr );          // deduces T = float

template<typename T> void h( T t );

h( 0 );                         // deduces T = int
h( nullptr );                   // deduces T = nullptr_t
h( (float*) nullptr );          // deduces T = float*
```

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, ~~strikethrough text~~ refers to existing text that is to be deleted, and <u>underscored text</u> 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.

In §2.13 add the alternative *pointer-literal* to *literal*.

Insert a new section after §2.13.5:

2.13.6 Pointer literal [lex.nullptr]

pointer-literal:
    `nullptr`

1 The pointer literal is the keyword `nullptr`. It is an rvalue of type `std::nullptr_t`.

Change §3.9(10) as indicated:

10 Arithmetic types (3.9.1), enumeration types, pointer types, ~~and~~ pointer to member types (3.9.2), and `std::nullptr_t`, and *cv-qualified* versions of these types (3.9.3) are collectively called *scalar types*. Scalar types, POD classes (clause 9), arrays of such types and *cv-qualified* versions of these types (3.9.3) are collectively called *POD types*. Scalar types, trivial class types (clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called *trivial types*. Scalar types, standard-layout class types (clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called *standard-layout types*.

Insert a new paragraph after §3.9.1(9):

9a A value of type `std::nullptr_t` is a null pointer constant (4.10). Such values participate in the pointer and pointer to member conversions (4.10, 4.11). `sizeof(nullptr_t)` shall be equal to `sizeof(void*)`.

Change §4.1(2) as indicated:

2 When an lvalue-to-rvalue conversion occurs in an unevaluated operand or a subexpression thereof (clause 5) the value contained in the referenced object is not accessed. Otherwise, if the lvalue has a class type, the conversion copy-initializes a temporary of type `T` from the lvalue and the result of the conversion is an rvalue for the temporary. Otherwise, if the lvalue has (possibly cv-qualified) type `std::nullptr_t`, the rvalue result is a null pointer constant (4.10). Otherwise, the value contained in the object indicated by the lvalue is the rvalue result.

Change §4.10(1) as indicated:

1 A *null pointer constant* is an integral constant expression (5.19) rvalue of integer type that evaluates to zero or an rvalue of type `std::nullptr_t`. 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 (4.4).

Change §5.2.10(9) as indicated:

9 The null pointer value (4.10) is converted to the null pointer value of the destination type. [*Note:* A null pointer constant~~, which has~~ of integral type~~,~~ is not necessarily converted to a null pointer value. (A null pointer constant of type `std::nullptr_t` cannot appear as the operand of `reinterpret_cast`, nor can any value be converted by `reinterpret_cast` to type `std::nullptr_t`.) —*end note*]

Change §5.9(1) as indicated:

1 The relational operators group left-to-right. [*Example:* `a<b<c` means `(a<b)<c` and *not* `(a<b)&&(b<c)`. —*end example*]

relational-expression:
  shift-expression
  relational-expression < shift-expression
  relational-expression > shift-expression
  relational-expression <= shift-expression
  relational-expression >= shift-expression

The operands shall have arithmetic, enumeration or pointer type or type `std::nullptr_t`. The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield false or true. The type of the result is `bool`.

Insert a new paragraph at the end of §5.9:

3 If two operands of type `std::nullptr_t` are compared, the result is `true` if the operator is <= or >=, and `false` otherwise.

Insert a new paragraph at the end of §5.10:

3 If two operands of type `std::nullptr_t` are compared, the result is `true` if the operator is ==, and `false` otherwise.

Change the beginning of §8.5(5) as indicated:

5 To *zero-initialize* an object of type `T` means:

— if `T` is `std::nullptr_t`, the object is set to the value of `nullptr`.

— otherwise, if `T` is a scalar type (3.9), the object is set to the value `0` (zero), taken as an integral constant expression, converted to `T`; [91)]

[etc. as before]

Change §14.3.2(5) as indicated:

5 The following conversions are performed on each expression used as a non-type *template-argument*. If a non-type *template-argument* cannot be converted to the type of the corresponding *template-parameter* then the program is ill-formed.

- for a non-type *template-parameter* of integral or enumeration type, integral promotions (4.5) and integral conversions (4.7) are applied.
- for a non-type *template-parameter* of type pointer to object, qualification conversions (4.4) and the array-to-pointer conversion (4.2) are applied; if the *template-argument* is of type `std::nullptr_t`, the null pointer conversion (4.10) is applied. [*Note:* In particular, neither the null pointer conversion for a zero-valued integral constant expression (4.10) nor the derived-to-base conversion (4.10) are applied. Although 0 is a valid *template-argument* for a non-type *template-parameter* of integral type, it is not a valid *template-argument* for a non-type *template-parameter* of pointer type. However, both `(int*)0` and `nullptr` are ~~is a~~ valid *template-arguments* for a non-type *template-parameter* of type "pointer to int." —*end note*]
- For a non-type *template-parameter* of type reference to object, no conversions apply. The type referred to by the reference may be more cv-qualified than the (otherwise identical) type of the *template-argument*. The *template-parameter* is bound directly to the *template-argument*, which must be an lvalue.
- For a non-type *template-parameter* of type pointer to function, ~~only~~ the function-to-pointer conversion (4.3) is applied; if the *template-argument* is of type `std::nullptr_t`, the null pointer conversion (4.10) is applied. If the *template-argument* represents a set of overloaded functions (or a pointer to such), the matching function is selected from the set (13.4).
- For a non-type *template-parameter* of type reference to function, no conversions apply. If the *template-argument* represents a set of overloaded functions, the matching function is selected from the set (13.4).
- For a non-type *template-parameter* of type pointer to member function, if the *template-argument* is of type `std::nullptr_t`, the null member pointer conversion (4.11) is applied; otherwise, no conversions apply. If the *template-argument* represents a set of overloaded member functions, the matching member function is selected from the set (13.4).
- For a non-type *template-parameter* of type pointer to data member, qualification conversions (4.4) are applied; if the *template-argument* is of type `std::nullptr_t`, the null member pointer conversion (4.11) is applied.

Change §15.3(3) as indicated:

1 A *handler* is a match for an exception object of type `E` if

- The *handler* is of type *cv* `T` or *cv* `T&` and `E` and `T` are the same type (ignoring the top-level *cv-qualifiers*), or
- the *handler* is of type *cv* `T` or *cv* `T&` and `T` is an unambiguous public base class of `E`, or
- the *handler* is of type *cv1* `T*` *cv2* and `E` is a pointer type that can be converted to the type of the *handler* by either or both of

    - a standard pointer conversion (4.10) not involving conversions to pointers to private or protected or ambiguous classes
    - a qualification conversion

- the *handler* is a pointer or pointer to member type and E is std::nullptr_t.

[*Note:* a *throw-expression* ~~which~~ whose operand is an integral constant expression of integer type that evaluates to zero does not match a handler of pointer or pointer to member type~~; that is, the null pointer constant conversions (4.10, 4.11) do not apply~~. —*end note*]

In §18.1, add nullptr_t to Table 15 as follows:

Table 15—Header <cstddef> synopsis

Kind	Name(s)		
Macros:	NULL	offsetof	
Types:	ptrdiff_t	size_t	nullptr_t

Also in §18.1, insert the following new paragraph:

6 nullptr_t is defined as follows:

```
namespace std {
  typedef decltype(nullptr) nullptr_t;
}
```

The type for which nullptr_t is a synonym has the characteristics described in 3.9 and 4.10. [*Note:* Although nullptr's address cannot be taken, the address of another nullptr_t object that is an lvalue could be taken. —*end note*]

Acknowledgments

Thanks to CWG members, and Mike Miller in particular, for multiple reviews of the wording in this paper.

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).