Doc No: SC22/WG21/N1488
J16/03-0071

Date: September 10, 2003

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

Abstract

We propose a new constant called nullptr of the distinct type decltype(nullptr). nullptr can be assigned to any pointer type (incl. pointer to member and function pointer types) but not to integral types. We further propose that the standard library macro NULL be defined to be nullptr. The result will be more readable code, better error detection, and better overload resolution.

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). 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. Note that this implies that today's null pointer, 0, has no utterable type.

- *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, we need to have a different name to express the null pointer.

There is widely used prior art where compilers have introduced implementation-specific extensions to provide a distinguishable name for the null pointer. For example, since January 1998, gcc 2.8.0 and later have provided __null as a magic keyword that can convert to any pointer type without a warning, or also to an integer 0 with a warning; the gcc implementation also has #define NULL __null so that users of NULL would by default use the new feature. This is known to catch genuine errors that would otherwise have compiled silently (including a report from Matt Austern while discussing a draft of this paper before the pre-meeting mailing), and is not known to have generated any user complaints about the warning.

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 an utterable type, and its type can be deduced as a template argument and templates can be specialized for the null pointer.
- 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.
- The null pointer can be converted to any pointer type (including pointer to function) and pointer to member type (including pointer to data member and pointer to member function), and cannot be converted to any other type including any integral or bool type.

We do not propose to eliminate the conversion of 0 to a pointer type. In a new language that would have been preferable. However, there are millions of lines of code that depends on 0 being used as an initializer for a pointer, as an argument to a function, etc.

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.
- 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, 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.

This alternative has one significant advantage:

- It increases the level of C compatibility beyond what would be achieved by #defining NULL to nullptr.

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. nullptr designates a constant rvalue whose address cannot be taken. It has adistinct and utterable type. It can be converted to any pointer type (including pointer to function) and pointer to member type (including pointer to data member and pointer to member function), and cannot be converted to any other type including any integral or bool type. It can be used only in a context where it will be converted to a uniquely determined pointer type.

nullptr may not be used in an arithmetic expression, assigned to an integral variable, or compared to an integral value; a diagnostic is required for these cases.

nullpr cannot be stored; when mentioned, it is always immediately converted into another type. Therefore, nullptr will not have a representation in memory.

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%[1] or over 5,000 use null in C++ code as an identifier or as a macro.

[1] Based on inspection of the first 300 hits, in which there were nine code hits (most related to Qt's QString::null).

- 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.

2.1 Basic Cases

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

```
struct C { };

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

char C::* pmem = nullptr;    // pmem has the null pointer value
char C::* pmem2 = 0;         // pmem2 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
if( nullptr == 0 );          // error

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

char** p = &nullptr;         // error, nullptr is not addressable

char** q = nullptr;          // ok

int (*pf)(int) = nullptr;    // ok
```

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

The reason that nullptr == 0 is an error is that nullptr cannot be converted to 0's type (int) and that 0 cannot be converted to nullptr's type because the type of a use of nullptr is always deduced.

2.2 Advanced Cases

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

```
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


void f( char* );
void f( int );

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


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

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


void f(int ...);

f(2,p,nullptr);                        // error: nullptr cannot be represented in memory

f(2,p,static_cast<int*>(nullptr));  // ok
```

2.3 Backward compatibility and NULL

The macro NULL shall be defined to be nullptr and nullptr only (not 0 or 0L).

This will break existing code that relies on a conversion from NULL to an integral type, but such code is likely far more often code that contains a conceptual or actual error (where "breaking" gives the user an opportunity to correct it) than code that is actually correct and valid. New code will use the cleaner and safer nullptr. Code using the standard library NULL macro would behave identically in C and C++.

In prior art, the gcc implementation has for many years had #define NULL __null, where __null is a magic keyword similar to nullptr.

We must investigate whether NULL is used so frequently as a varadic argument that we should consider defining a representation for nullptr to allow such use.

2.4 Type of nullptr

It is desirable to allow nullptr's type to be deducible as a template argument and to be overloadable, but it is not desirable to allow additional objects of its type to be created.

Therefore, nullptr shall have an utterable type of which no other objects can be declared. This choice has several direct consequences:

- Additional objects of nullptr's type cannot be created, so we do not have to answer questions as to their meaning, copyability, or assignability. for example

```
decltype(nullptr) x = nullptr; // error
```

- nullptr's type may be deduced and functions may be overloaded on it. In particular, tr1::function has an overload taking an integer where the Boost implementation has deliberately enabled tr1::function<void(void)>(0) but disabled usages like tr1::function<void(void)>(42). This would allow tr1::function<void(void)>(nullptr) instead, distinctly, instead of tr1::function<void(void)>(0). Note that in such deduced contexts, null pointer objects appearing as parameter must be taken by reference, not by value, because their copy constructor is private.
- sizeof(nullptr), typeid(nullptr), and throw nullptr are permitted, where otherwise they would not have been.

The key remaining question is: Is it sufficient that nullptr's type be utterable without actually giving it a name?

- If yes, then for example we could spell it decltype(nullptr) (making use of EWG type identification extensions proposal). Doing this avoids the following problems associated with choosing its name, below. (It is also possible to later give it a name, for example by having typedef decltype(nullptr) nullptr_t; in a standard header.)
- If no, then:
  - What is that name?
    - Nullptr would be reasonable, but we don't capitalize names of built-in types.
    - nullptr_t could be viewed as inappropriate because _t is ugly and should be reserved for typedefs (wchar_t arguably should have been wchar).
    - _nullptr (and variations like __nullptr and _Nullptr) we view as inappropriate because we should not use "nonstandard underscore-prefix names."
  - Is that name a reserved word?
    - If no, it would be inconsistent with making nullptr a reserved word.
    - If yes, eating a keyword for this seems like a wastefully high-cost solution.

We therefore propose that nullptr's type be unnamed but utterable via the parallel EWG type manipulation proposal (e.g., decltype(nullptr)).

3. Interactions and Implementability

3.1 Interactions

See section 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. Because of #define NULL nullptr, code that assigns NULL to an arithmetic type will no longer compile on a C++0x-compliant compiler.

3.2 Implementability

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

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).