The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

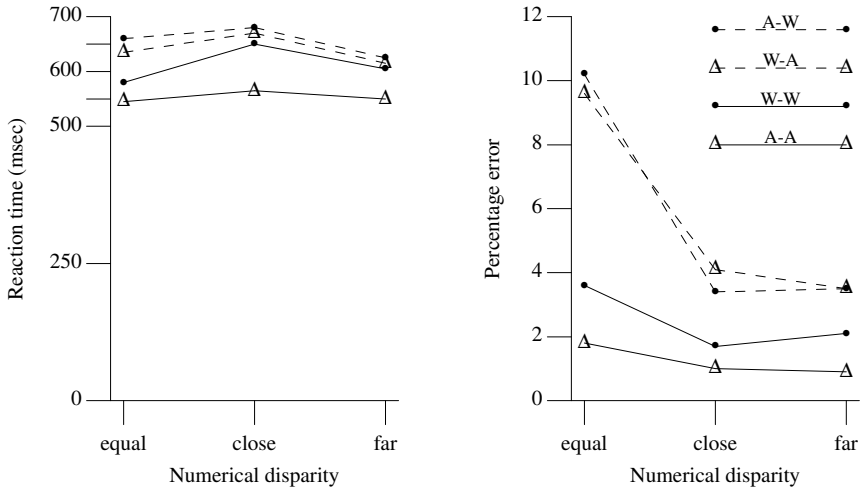

Figure 1212.1: Reaction time (in milliseconds) and error rates for same/different judgment for values between one and nine, expressed in *A*rabic or *W*ord form. Adapted from Dehaene and Akhavein.[1]

6.5.9 Equality operators

equality operators
syntax

1212

```
equality-expression:
                relational-expression
                equality-expression == relational-expression
                equality-expression != relational-expression
```

Commentary

The use of the term *equality* is based on the most commonly occurring of the two operators. The terms *equals* and *not equals* are commonly used by developers.

Other Languages

Those languages that use the character sequence **:=** to represent assignment usually use the **=** character to represent the equality operator token. Some languages use the character sequence **<>** to represent the not equal operator. Fortran uses the tokens **.EQ.** and **.NE.** to represent the quality operators.

Pascal supports the **IN** operator. Instead of comparing one value against another, this operator provides a mechanism for testing whether a value is **IN** a set of values. For instance, `i IN [4, 6]` tests whether `i` has the value 4 or 6; `i IN [2..5]` tests whether `i` has a value between 2 and 5, inclusive.

Some languages offer a number of different ways of comparing for equality. For instance, in Scheme `(eq? X Y)` tests whether `X` and `Y` refer to the same object, but `(equal? X Y)` tests, recursively, whether `X` and `Y` have the same structure (which for linked data structures involves walking them, comparing each pair of nodes).

Perl supports the **<=>** operator, which returns -1 if the left operand is less than the right operand, 0 if they are equal, and 1 if the left operand is greater than the right.

Coding Guidelines

A study by Dehaene and Akhavein[1] asked subjects to make same/difference judgments for pairs of numeric values. The values were either presented in Arabic form (e.g., 2 2), as written words (e.g., TWO TWO), or in mixed form (e.g., 2 TWO, or TWO 2). The two numeric values were either the same, close to each other (e.g., 1 2), or not close to each other (e.g., THREE NINE). The time taken for subjects to make a same/different judgment was measured, along with their error rate.

The results (see Figure 1212.1) for different the kinds of numeric value pairings followed the same pattern. This pattern is consistent with both the Arabic and written-word representation being converted to a common, internal, magnitude representation (in the subject's head). This *distance effect* is discussed elsewhere.

distance effect
numeric difference

Numeric values appear in source code in written form through the use of macro definitions. However, the common usage is to use a symbolic name to represent some numeric quantity, not the spoken words of the value; for instance, using the name `MAX_LINE_LENGTH` to denote the maximum number of characters that can appear on a line. Source code may contain a comparison of such a name against an integer constant (e.g., `#if MAX_LINE_LENGTH == 80`) or against another symbolic name (e.g., `#if MAX_LINE_LENGTH == XYZ_TERM_LINE_LENGTH`). Comparison of two decimal constants appearing in the visible source as Arabic numerals is rare. The affect of reader's prior knowledge of the current value of `MAX_LINE_LENGTH` on their processing of this equality operation is unknown. The Dehaene and Akhavein study showed that using symbolic names for numeric values does not prevent the distance effect from occurring.

An equality comparison may involve performing a mental calculation on one of the operands. Within source code, this may be a *what if?* kind of calculation or the values of the operands may be known at translation time and a reader may be attempting to verify if a condition is true or false (e.g., which arm of a conditional inclusion directive will be processed by a translator).

A study by Zbrodoff and Logan[3] measured subjects' performance in arithmetic verification tasks (e.g., $3\times5 = 17$, true/false?). The intent was to test the hypothesis that such verification tasks consisted of two stages: first performing the arithmetic operation and then comparing the computed value against the value given. In most cases the results were not consistent with this two-stage production/comparison process, but were consistent with verification involving a comparison of all the information presented (i.e., including the putative answer) against memory.

Table 1212.1: Common token pairs involving the equality operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files. Note: entries do not always sum to 100% because several token sequences that have a very low percentage are not listed.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier `!=`	0.6	69.2	`]` `!=`	1.4	5.1
identifier `==`	1.2	67.9	`==` **-v**	2.6	3.5
`)` `==`	1.6	25.1	`==` *integer-constant*	25.5	2.0
`)` `!=`	0.8	24.7	`==` identifier	62.1	1.1
`==` *character-constant*	7.1	22.8	`!=` *integer-constant*	22.7	0.9
`!=` *character-constant*	5.3	8.4	`!=` identifier	65.0	0.6
`]` `==`	3.1	5.6			

Constraints

1213 One of the following shall hold:

equality operators constraints

Commentary

This list is very similar to that given for simple assignment, except that there is no support for equality operations on structure or union types.

simple assignment constraints
assignment structure types

The C89 Committee considered, on more than one occasion, permitting comparison of structures for equality. Such proposals foundered on the problem of holes in structures. A byte-wise comparison of two structures would require that the holes assuredly be set to zero so that all holes would compare equal, a difficult task for automatic or dynamically allocated variables. The possibility of union-type elements in a structure raises insuperable problems with this approach. Without the assurance that all holes were set to zero, the implementation would have to be prepared to break a structure comparison into an arbitrary number of

Rationale

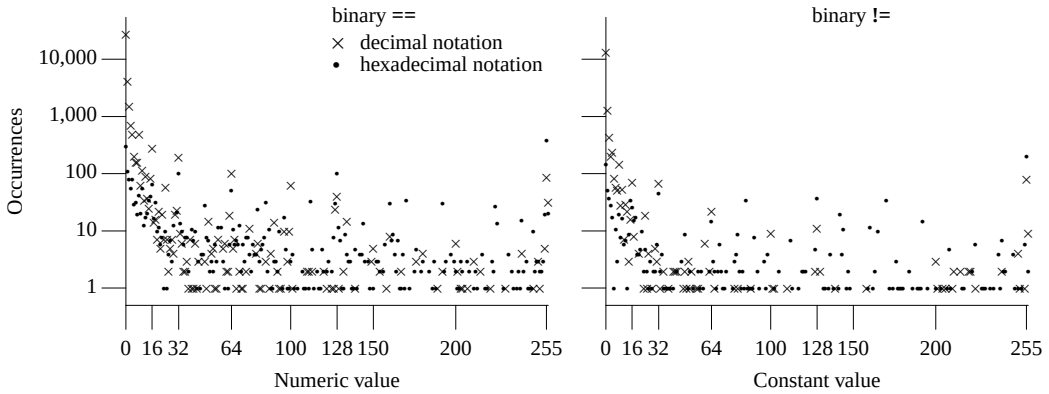

Figure 1212.2: Number of *integer-constant*s having a given value appearing as the right operand of equality operators. Based on the visible form of the `.c` files.

member comparisons; a seemingly simple expression could thus expand into a substantial stretch of code, which is contrary to the spirit of C.

C++

5.10p1 *The* `==` *(equal to) and the* `!=` *(not equal to) operators have the same semantic restrictions, conversions, and result type as the relational operators . . .*

relational operators constraints

See relational operators.

Other Languages

Other languages include support for equality operations on structures, strings, sets, and even linked lists.

---

equality operators arithmetic operands — both operands have arithmetic type; 1214

Commentary

These operators are defined for operands having a complex type.

Coding Guidelines

The equality operators are different from the relational operators in that an exact match is being tested for. In the case of the real and complex types, testing for an exact match does not always make sense. Rounding and other types of error in operations on floating-point values means that an exact value can never be expected, only a very close approximation.

The equality operator is sometimes used to check a property of floating-point numbers. Is one value so small, compared to a second value, that the effect of adding them together is to deliver the larger one as the result? In other words is the smaller value insignificant, compared to the larger value? This test can be made because we are interested in the smaller value, not the larger one. The natural logarithm of one plus a very small value is equal, within a reasonable error bound, to that very small value.[2]

```
1  #include <math.h>
2
3  double log_x_plus_1(double x)
4  {
5  if (1.0 + x == 1.0)
6     return x;
7  else
```

```
8       return log(1.0 + x) * x / ((1.0 + x) - 1.0);
9   }
```

The above algorithm relies on the value of the expression `1.0+x` calculated in the equality test being identical to the value passed as an argument to the log function. The value passed to the `log` function has type **`double`**. If the equality test is carried out using a precision that is different from **`double`** (for instance, using an extended precision), it is possible that the extra bits available in the result will cause the equality test to fail, invoking the `log` function which returns zero (the *log* of 1.0, since rounding to **`double`** will remove any extended precision bits)— a value that has a relative error of one. FLT_EVAL_METH

This example shows that using equality operators on values having real type requires more than a numerical analysis of the algorithm being used. (Goldberg[2] provides a readable analysis of error bounds.) What is also needed is knowledge of how an implementation actually performs the calculation. In this case use of extended precision can invalidate all of the error bounds deduced by careful numerical analysis.

Cg 1214.1

The operands of the equality operators shall not have floating-point or complex type.

Dev 1214.1

A numerical algorithm that performs equality tests on floating-point values may be used if an implementation always evaluates lexically identical expressions to the same result.

Comparing values having an enumerated type for equality (or inequality) need not imply any use of representation information (e.g., relative ordering of members).

Dev ??

Both operands of an equality operator may have an enumeration type or be an enumeration constant, provided it is the same enumerated type or a member of the same enumerated type.

Table 1214.1: Occurrence of equality operators having particular operand types (as a percentage of all occurrences of each operator). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
ptr-to	!=	ptr-to	28.5	**char**	!=	**_int**	3.9
int	==	**_int**	21.1	ptr-to	!=	**_int**	3.5
int	!=	**_int**	15.8	**unsigned long**	!=	**unsigned long**	2.5
ptr-to	==	ptr-to	15.3	**unsigned long**	!=	**_int**	2.2
other-types	==	other-types	12.7	**unsigned short**	!=	**_int**	2.0
other-types	!=	other-types	12.6	int:16 16	!=	**_int**	2.0
unsigned char	==	**_int**	9.5	**unsigned short**	!=	**unsigned short**	1.9
enum	==	**_int**	9.1	**unsigned int**	!=	**unsigned int**	1.9
int:16 16	==	**_int**	8.2	ptr-to	==	**_int**	1.8
int	!=	**int**	6.5	**unsigned short**	==	**_int**	1.7
int	==	**int**	6.5	**unsigned long**	==	**unsigned long**	1.7
char	==	**_int**	5.5	**unsigned long**	==	**_int**	1.6
unsigned char	!=	**_int**	4.8	**unsigned long**	!=	**_long**	1.3
enum	!=	**_int**	4.8	**unsigned char**	!=	**unsigned char**	1.3
unsigned int	!=	**_int**	4.4	**unsigned int**	==	**unsigned int**	1.1
unsigned int	==	**_int**	4.0				

1215 — both operands are pointers to qualified or unqualified versions of compatible types;

equality operators pointer to compatible types

Commentary

The rationale for supporting pointers to qualified or unqualified type is the same as for pointer subtraction and relational operators. Differences in the qualification of pointed-to types is guaranteed not to affect the equality status of two pointer values.

subtraction pointer operands relational operators pointer operands pointer converting qualified/unqualified

The operands may have a pointer to function type.

relational operators pointer operands

C++

The discussion on the relational operators is applicable here.

Other Languages

Languages containing a pointer data type allow values having such types to be compared for equality and inequality.

Common Implementations

pointer segmented architecture

Because the operands need not refer to the same objects, it is necessary to compare all the information represented in a pointer value (e.g., segment and offset in a segmented architecture). The constraint that both pointers point to compatible types ensures that, in those cases where different representations are used for pointers to different types, an implementation does not have to concern itself with implicitly converting one representation to another.

In implementations that use a flat address space an equality operator is usually implemented using one of the unsigned integer comparison machine instructions.

Table 1215.1: Occurrence of equality operators having particular operand pointer types (as a percentage of all occurrences of each operator with operands having a pointer type; an _ prefix indicates a literal operand, *_int* is probably the 0 representation of the null-pointer constant). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
struct *	==	_ **int**	59.9	**int** *	!=	_ **int**	3.0
struct *	!=	_ **int**	52.2	**void** *	==	_ **int**	2.2
union *	!=	_ **int**	18.3	**const char** *	==	_ **int**	1.8
union *	==	_ **int**	18.1	**int**	==	**void** *	1.4
other-types	==	other-types	8.1	**const char** *	!=	_ **int**	1.4
char *	!=	_ **int**	8.1	**int**	!=	**void** *	1.3
char *	==	_ **int**	7.3	**unsigned char** *	==	_ **int**	1.1
array-index	!=	**void** *	6.9	ptr-to *	!=	_ **int**	1.1
other-types	!=	other-types	6.4	**char** *	!=	array-index	1.1

equality operators pointer to incomplete type

— one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**; or 1216

Commentary

This combination of operands supports the idea that pointer to **void** represents a generic container for any pointer type. Relaxing the constraint in the previous C sentence, requiring the pointer types to be compatible, removes the need for an explicit cast of the operand having pointer to **void** type. A pointer to **void** is only guaranteed to compare equal to the original pointer value when it is converted back to that value's original pointer type.

converted via pointer to void compare equal

C++

This special case is not called out in the C++ Standard.

```
1  #include <stdlib.h>
2
3  struct node {
4              int mem;
5              };
6  void *glob;
7
8  void f(void)
9  {
10 /* The following is conforming */
11 // The following is ill-formed
```

```
12  struct node *p = malloc(sizeof(struct node));
13
14  /*
15   * There are no C/C++ differences when the object being assigned
16   * has a pointer to void type, 4.10p2.
17   */
18  glob = p;
19  }
```

relational operators constraints

See relational operators for additional issues.

Other Languages

Some languages include the concept of a generic pointer type, with some operators handling operands having this type.

Coding Guidelines

The use of pointer to **void** as a generic pointer type is common developer knowledge. There is no benefit in casting the operand having this type to the type of the other operand. An explicit cast creates a possible future maintenance cost, unless the pointer type is denoted by a typedef name. (In this case a single change to the definition of the typedef name changes all uses; otherwise, a change of pointer type requires all corresponding occurrences in the source to be changed.)

1217 — one operand is a pointer and the other is a null pointer constant.

equality operators null pointer constant

Commentary

The previous C sentence excluded the use of operands having pointer to function type. An equality test against the null pointer constant is a common loop termination condition when walking a dynamic data structure. This permission allows the loop termination test to include an operand having a pointer to function type. (It also covers the case of 0 being used to denote the null pointer constant.)

null pointer constant

Other Languages

Languages that support pointer data types invariably have some form of null pointer that can be compared for equality against all other pointer types.

Semantics

1218 The == (equal to) and != (not equal to) operators are analogous to the relational operators except for their lower precedence.[91)]

Commentary

They differ from the relational operators in their handling of NaN. The equality operators are defined not to raise an exception if one or more operators is NaNs, while the relational operators do raise an exception.

relational operators result value

Common Implementations

The processor instructions also perform in an analogous way to the relational operator instructions and translators generate analogous instruction sequences.

Coding Guidelines

Equality testing is one of the most fundamental operations people perform. In many contexts it is possible to assign a generally accepted meaning to an equality comparison, but not to a relational comparison. The C equality operators support a small subset of the possible equality tests (i.e., those between values having arithmetic and pointer types). Within this subset, it is analogous to the relational operators. (However, these C relational operations differ from the equality operators in that they often make up a significant percentage of the relational tests performed in a program; the other tests usually are between strings, often performed by calls to library functions.)

In most cases developers have to write code to perform equality tests when the operands do not have arithmetic types (e.g., to test whether two lists, or two arrays, are equal). However, there is one nonarithmetic type that is sometimes represented in an object having an integer type— a set. Representing a set type using an integer type creates the possibility for the equality operators to be inappropriately used. The following discussion looks at some of the difficulties of working out whether the use of an equality operator was intended when the operands appear to be sets.

enumeration
set of named constants

The enumeration constants defined in an enumerated type definition may be assigned values intended to fill a variety of roles. For instance, their representation may be numerically distinct because objects of that type are only intended to represent a single member, or the representation may be bitwise distinct, because objects of that type are intended to be able to represent more than one member at the same time. (A set type is created by ORing together different members.) In the latter case the operations performed by the C equality operators do not correspond to the most commonly seen tests, that of set membership (based on experience with Pascal, which supports a set type).

```
1  #define in_set(x, y)  (((x) & (y)) == (x))
2
3  enum T {mem_1 = 0x01, mem_2 = 0x02, mem_3 = 0x04, mem_4 = 0x10};
4
5  extern enum T glob;
6
7  void f(enum T p_1)
8  {
9  if (in_set(mem_2, p_1)) /* does p_1 include the member mem_2? */
10    ; /* ... */
11 /*
12  * The following test can be interpreted as either:
13  *    1) are the two sets equal?
14  *    2) does p_1 only include the member mem_2?
15  */
16 if (p_1 == mem_2)
17    ; /* ... */
18 }
```

How set types are represented as C types, and the interpretation given to C operators having operands of these types, is considered to be a level of abstraction that is outside the scope of these coding guideline subsections.

---

equality operators
true or false

Each of the operators yields 1 if the specified relation is true and 0 if it is false. 1219

Commentary

x != x yields 0, except when x is a NaN (when it yields 1). x == x yields 1, except when x is a NaN (when it yields 0). Possible limits on how strongly equality can be interpreted are discussed elsewhere.

footnote 43
subtraction
result of

C++

5.10p1 *The* **==** *(equal to) and the* **!=** *(not equal to) operators have the same . . . truth-value result as the relational operators.*

equality operators
result type 1220

This difference is only visible to the developer in one case. In all other situations the behavior is the same—false and true will be converted to 0 and 1 as needed.

Other Languages

Languages that support a boolean data type usually specify **true** and **false** return values for these operators.

Example

```
#include <math.h>

_Bool f(float x, float y)
{
if (isnan(x) && isnan(y))
   return 1;
return x==y;
}
```

1220 The result has type **int**.

equality operators result type

Commentary

The rationale for this choice is the same as for relational operators.

relational operators result type

C++

The == (equal to) and the != (not equal to) operators have the same . . . result type as the relational operators. 5.10p1

relational operators result type

The difference is also the same as relational operators.

Other Languages

Languages that support a boolean type usually specify a boolean result type for these operators.

Coding Guidelines

The coding guideline issues are the same as those for the relational operators.

relational operators result type

1221 For any pair of operands, exactly one of the relations is true.

equality operators exactly one relation is true

Commentary

This is a requirement on the implementation. No equivalent requirement is stated for relational operators (and for certain operand values would not hold). It is a moot point whether this requirement applies if both operands have indeterminate values since accessing either of them causes undefined behavior. It might be more accurate to say "for an evaluation of any pair of operands . . . ", since one or more of the operands may be volatile-qualified.

relational operators result value

type qualifier syntax

C90

This requirement was not explicitly specified in the C90 Standard. It was created, in part, by the response to DR #172.

C++

This requirement is not explicitly specified in the C++ Standard.

Example

```
#include <limits.h>

void f(void)
{
int i = INT_MIN;

if (i != INT_MIN)
   printf("This sentence will never appear\n");
if ((float)i != INT_MIN)
   printf("This sentence might appear\n");
```

```
11
12  while ((i != 0) && (i == -i)) /* When using two's compliment this is an infinite loop. */
13     { /* i not modified in loop. */ }
14  }
```

If both of the operands have arithmetic type, the usual arithmetic conversions are performed. 1222

Commentary

arithmetic conversions
integer promotions

This may also cause the integer promotions to be performed.

C90

Where the operands have types and values suitable for the relational operators, the semantics detailed in 6.3.8 apply.

Coding Guidelines

relational operators
usual arithmetic conversions

The unexpected results that can occur for the relational operators (when the operands have differently signed types), as a consequence of these conversions, are much less likely to occur for the equality operators. For instance, `signed_object == unsigned_object` is true when both objects have the same value and when the implicit conversion of a negative value to unsigned results in the same value (e.g., `UINT_MAX == -1`).

Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. 1223

Commentary

complex
component representation

Given the representation used for complex types, this form of equality testing is the obvious choice (had polar form been used, the obvious choice would have been to compare their modulus and angles).

C90

Support for complex types is new in C99.

Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal. 1224

Commentary

usual arithmetic conversions
real type
converted to complex

The usual arithmetic conversions specifies the type of the result. The details of real type to complex type conversions are specified elsewhere.

C90

Support for different type domains, and complex types, is new in C99.

C++

real type
converted to complex

The concept of type domain is new in C99 and is not specified in the C++ Standard, which defines constructors to handle this case. The conversions performed by these constructions have the same effect as those performed in C.

Coding Guidelines

equality operators 1214.1
not floating-point operands

The guideline recommendation dealing with the comparison of floating-point values for equality is applicable here.

---

footnote 91

91) The expression **a<b<c** is not interpreted as in ordinary mathematics. 1225

Commentary

In mathematical notation this would probably be interpreted as equivalent to the C expression `(a < b) && (b < c)`.

C++

The C++ Standard does not make this observation.

Other Languages

Cobol has a **`between`** operator that enables this mathematical test to be performed. BCPL allows any number of relational operators in a sequence; they're ANDed together; for instance, the BCPL expression `x > y >= z == q < r` is equivalent to the C `x > y && y >= z && z == q && q < r`.

Coding Guidelines

This issue is covered by the guideline recommendation dealing with the use of parenthesis.

?? expression shall be parenthesized

1226 As the syntax indicates, it means **`(a<b)<c`**;

Commentary

The **<** operator associates to the left.

1227 in other words, "if **a** is less than **b**, compare 1 to **c**; otherwise, compare 0 to **c**".

Commentary

As discussed elsewhere, the evaluation order of the operands is not guaranteed to be `a`, `b`, and `c`.

expression order of evaluation

1228 90) Because of the precedences, **`a<b == c<d`** is 1 whenever **`a<b`** and **`c<d`** have the same truth-value.

footnote 90

Commentary

The grouping here is `(a<b) == (c<d)`.

Coding Guidelines

This issue is covered by the guideline recommendation dealing with the use of parentheses.

?? expression shall be parenthesized

1229 Otherwise, at least one operand is a pointer.

Commentary

The case where only one operand is a pointer is when the other operand is the integer constant 0 (which is interpreted as a null pointer constant).

null pointer constant

C++

The C++ Standard does not break its discussion down into the nonpointer and pointer cases.

1230 If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer.

equality operators null pointer constant converted

Commentary

If different pointer types use different representations for the null pointer, this implicit conversion ensures that the equality test is performed against the appropriate representation.

null pointer conversion yields null pointer

C90

If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type.

In the case of the expression `(void *)0 == 0` both operands are null pointer constants. The C90 wording permits the left operand to be converted to the type of the right operand (type **`int`**). The C99 wording does not support this interpretation.

null pointer constant

equality 1217
operators
null pointer
constant

C++

The C++ Standard supports this combination of operands but does not explicitly specify any sequence of operations that take place prior to the comparison.

Other Languages

Other languages do not always go into this level of detail. They usually simply specify that two null pointers compare equal, irrespective of the pointer types involved.

equality 1216
operators
pointer to in-
complete type

Coding Guidelines

The issue of using explicit casts in this case is discussed elsewhere.

---

equality operators
pointer to void

If one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**, the former is converted to the type of the latter. 1231

Commentary

pointer
converted to
pointer to void
pointer
to void
same repre-
sentation and
alignment as

A pointer to **void** is capable of representing all the information represented in any pointer type (any pointer converted to pointer to **void** and back again will compare equal to the original pointer). There is no guarantee that any other pointer type will be capable of representing all the information in the pointer to **void** (although pointer to character types have the same representation and alignment requirements). For this reason the other operand has to be converted to pointer to **void**.

C++

This conversion is part of the general pointer conversion (4.10) rules in C++. This conversion occurs when two operands have pointer type.

equality 1216
operators
pointer to in-
complete type

Coding Guidelines

The issue of using explicit casts in this case is discussed elsewhere.

---

equality operators
pointer to object

For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type. 1232

Commentary

relational
operators
pointer to object
additive
operators
pointer to object

This wording was added by the response to DR #215 (the Committee response was to duplicate the wording from relational operators). The same sentence appears elsewhere in the standard and the issues are discussed there.

---

pointers
compare equal

Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[92)] 1233

Commentary

This is a requirement on the implementation. If the expression `px==py` returns 1 (with `px` and `py` both having pointer types), one of the conditions listed here must hold (the fact that two pointer values compare equal does not imply that they have the same value representation). In all other cases an implementation is required to return 0. All null pointers, whatever type they have been converted to, always compare equal. Here the phrase " . . . point to the same object . . . " means the same address.

pointer
segmented
architecture
null pointer
conversion yields
null pointer
object
lowest ad-
dressed byte
pointer
one past
end of object

The standard requires that pointers be able to point one past the last element of an array, but it does not place any requirements on the relative storage addresses of different objects. This can lead to the situation of a one past the last element pointer comparing equal to a pointer to a different object (which might not even be compatible with the pointed-to type). The standard does not require such behavior, it simply points out that it can occur in a strictly conforming program.

The relationship `p == q` does not imply `(p-q) == 0`, because the subtraction operator is only defined when its operands point at an object. If `p` and `q` have the null pointer value, the result is undefined behavior. The following equalities are true if `p != NULL`, but are not guaranteed to be true if `p == NULL` (although in practice they are always true on most implementations);

pointer subtraction
point at same object

```
1  p - p == 0
2  p + 0 == p
```

C90

If two pointers to object or incomplete types are both null pointers, they compare equal. If two pointers to object or incomplete types compare equal, they both are null pointers, or both point to the same object, or both point one past the last element of the same array object. If two pointers to function types are both null pointers or both point to the same function, they compare equal. If two pointers to function types compare equal, either both are null pointers, or both point to the same function.

The admission that a pointer one past the end of an object and a pointer to the start of a different object compare equal, if the implementation places the latter immediately following the former in the address space, is new in C99 (but it does describe the behavior of most C90 implementations).

C++

Two pointers of the same type compare equal if and only if they are both null, both point to the same object or function, or both point one past the end of the same array.

5.10p1

This specification does not include the cases:

- "(including a pointer to an object and a subobject at its beginning)", which might be deduced from wording given elsewhere,
- "or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space". The C++ Standard does not prevent an implementation from returning a result of **`true`** for the second case, but it does not require it. However, the response to C++ DR #073 deals with the possibility of a pointer pointing one past the end of an object comparing equal, in some implementations, to the address of another object. Wording changes are proposed that acknowledge this possibility.

object
lowest addressed byte

Other Languages

These requirements usually hold for implementations of other languages, but are rarely expressed explicitly.

Common Implementations

If two pointers point to the same object, and the lifetime of that object ends, most implementations will continue to compare those pointers as being equal, even though they no longer point at an object. If an integer value is cast to a pointer type and assigned to two pointers, most implementations will compare those pointers as equal, even though neither of them may be the null pointer constant or have ever pointed at an object.

References

1. S. Dehaene and R. Akhavein. Attention, automaticity, and levels of representation in number processing. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 21(2):314–326, 1995.
2. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.
3. N. J. Zbrodoff and G. D. Logan. On the relation between production and verification tasks in the psychology of simple arithmetic. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 16(1):83–97, 1990.