The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.5.8 Relational operators

relational operators syntax

1197

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

Commentary

The term *comparison operators* is commonly used by developers to refer to the relational operators. These operators are often combined to specify intervals. (C does not support SQL's **between** operator, or the Cobol form `(x > 0 and < 10)`, which is equivalent to `(x > 0 and x < 10)`.)

Other Languages

Nearly every other computer language uses these tokens for these operators. Fortran uses the tokens **.LT.**, **.GT.**, **.LE.**, and **.GE.** to represent the above operators. Cobol supports these operators as well as the equivalent keywords **LESS THAN**, **LESS THAN OR EQUAL**, **GREATER THAN**, and **GREATER THAN OR EQUAL**.

Some languages (e.g., Ada and Fortran) specify that the relational and equality operators have the same precedence level.

Common Implementations

relational operators unordered

Some implementations support the unordered relational operators `!<`, `!<=`, `!>=`, and `!>`. The NCEG also included `!<>` for unordered or equal; `!<>=` for unordered; `<>=` for less, equal, or greater; and defines `!=` to mean unordered, less or greater; which enables the 26 distinct comparison predicates defined by IEC 60559 to be used. These operators were included in the Technical Report produced by the NCEG FP/IEEE Subcommittee. The expressions `(a !op b)` and `!(a op b)` have the same logical value. Without any language or library support, `a !> b` could be implemented as: `(a != a) || (b != b) || (a <= b)`.

NCEG

Coding Guidelines

distance effect numeric difference

A study by Moyer and Landauer[8] found that the time taken for subjects to decide which of two single-digit values was the largest was inversely proportional to the numeric difference in their values (known as a *distance effect*).

How do people compare multi-digit integer constants? For instance, do they compare them digit by digit (i.e., a serial comparison), or do they form two complete values before comparing their magnitudes (the so-called *holistic* model)? The following two studies show that the answer depends on how the comparisons are made:

- A study by Dehaene, Dupoux, and Mehler[2] told subjects that numbers distributed around the value 55 would appear on a screen and asked them to indicate whether the number that appeared was larger or smaller than 55 (other numbers— e.g., 65— were also used as the center point). The time taken for subjects to respond was measured. The results were generally consistent with subjects using the holistic model (exceptions occurred for values ending in zero, where subjects may have noticed that a single-digit comparison was sufficient, and when the value being compared against contained two identical digits). The response times also showed a distance effect.
- A study by Zhang and Wang[9] told subjects that they would see two numbers, both containing two digits, on a screen and asked them to indicate whether the number that appeared on the left or the right was the largest. The time taken for subjects to respond was measured. When the largest value was less than 65, the results were generally consistent with subjects using a modified serial comparison of the digits (modified to take account of Stroop-like interference between the unit and ten's digit). When the largest value was greater than 65, a serial comparison gave a slightly better fit to the results than the modified serial comparison model.

stroop effect

symbolic distance effect

Other studies have found that people do not treat all relational comparisons in the same way. A so-called *symbolic distance effect* exists. This is a general effect that occurs when people compare numbers or other symbols having some measure that varies along some continuum.

For instance, a study by Moyer and Bayer[7] gave four made-up names to four circles of different diameters. One set of four circles (the small range set) had diameters 11, 13, 15, and 17 mm, while a second set (the large range set) had diameters 11, 15, 19, and 23 mm. On the first day one group of subjects learned the association between the four made-up names and their associated circle in the small range set, while another group of subjects learned the word associations for the large range set. On the second day subjects were tested. They were shown pairs of the made-up names and had to indicate which name represented the larger circle. Their response times and error rates were measured. In both cases, response rate was faster, and error rate lower, when comparing circles whose diameters differed by larger amounts. However, performance was better at all diameter differences for subjects who had memorized an association to the circles in the large range set; they responded more quickly and accurately than subjects using the small range set. The results showed a distance effect that was inversely proportional to the difference of the area of the circles being compared.

A symbolic distance effect (which is inversely proportional to the *distance* between the quantities being compared) has also been found for comparisons involving a variety of objects that differ in some way.

Freksa[3] adapted Allen's temporal algebra[1] to take account of physical constraints on perception, enabling *cognitively plausible* inferences on intervals to be performed.[4]

Table 1197.1: Common token pairs involving relational operators (as a percentage of all occurrences of each token). Based on the visible form of the `.c` files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
identifier <	0.7	87.9	>= *character-constant*	3.6	1.5
identifier >=	0.2	85.9	< *integer-constant*	40.0	1.3
identifier >	0.3	85.0	> *integer-constant*	53.2	0.9
identifier <=	0.1	84.8	>= *integer-constant*	41.2	0.4
) <=	0.1	10.4	< identifier	53.9	0.4
) >=	0.1	10.1	<= *integer-constant*	41.0	0.2
) <	0.3	9.9	> identifier	40.1	0.2
) >	0.1	9.6	>= identifier	50.0	0.1
<= *character-constant*	7.1	1.7	<= identifier	45.7	0.1

Table 1197.2: Occurrences (per million words) of English words used in natural language sentences expressing some relative state of affairs. Based on data from the British National Corpus.[5]

Word	Occurrences per Million Words	Word	Occurrences per Million Words
great	464	less	344
greater	154	lesser	18
greatest	51	least	45
greatly	33	–	–
–	–	less than	40

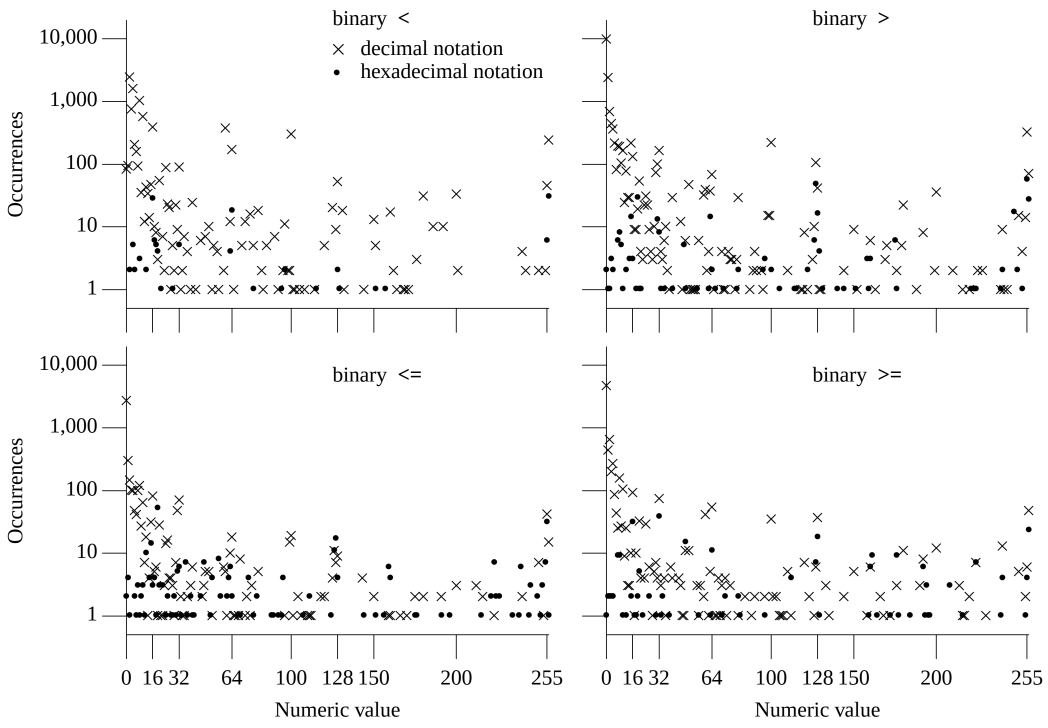

Figure 1197.1: Number of *integer-constant*s having a given value appearing as the right operand of relational operators. Based on the visible form of the `.c` files.

Table 1197.3: Occurrence of relational operators (as a percentage of all occurrences of the given operator; the parenthesized value is the percentage of all occurrences of the context that contains the operator). Based on the visible form of the `.c` files.

Context	% of <	% of <=	% of >	% of >=
if control-expression	76.7 (3.4)	45.5 (6.7)	68.5 (1.8)	80.5 (6.0)
other contexts	11.5 (—)	4.8 (—)	9.5 (—)	8.4 (—)
while control-expression	4.8 (3.9)	4.6 (12.0)	4.8 (2.2)	7.6 (10.4)
for control-expression	7.1 (3.1)	45.2 (65.9)	17.2 (4.5)	3.5 (2.6)
switch control-expression	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)

Constraints

relational operators constraints

One of the following shall hold: 1198

Commentary

This list does not include pointer to function types.

Other Languages

Some languages support string values as operands to the relational operators. The comparison is usually made by comparing the corresponding characters in each string, using the designated operator.

relational operators real operands

— both operands have real type; 1199

equality operators arithmetic operands

Commentary

Relational operators applied to complex types have no commonly accepted definition (unlike equality) and the standard does not support this usage.

Some mathematical practice would be supported by defining the relational operators for complex operands so that `z1` *op* `z2` would be true if and only if both `creal(z1)` *op* `creal(z2)` and `cimag(z1) == cimag(z2)`. Believing such use to be uncommon, the C99 Committee voted against including this specification.

C90

both operands have arithmetic type;

The change in terminology in C99 was necessitated by the introduction of complex types.

Coding Guidelines

As discussed elsewhere, it is sometimes necessary to step through the members of an enumeration type. This suggests the use of looping constructs, which in turn implies using a member of the enumerated type in the loop termination condition.

postfix operator
operand

Dev **??**

Both operands of a relational 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.

Example

```
enum color {first_color, red=first_color, orange, yellow, green, blue,
            indigo, violet, last_color};

void f(void)
{
for (enum color index=first_color; index < last_color; index++)
   ;
}
```

Table 1199.1: Occurrence of relational operators having particular operand types (as a percentage of all occurrences of each operator; an _ prefix indicates a literal operand). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
int	>=	_int	35.3	unsigned char	>	_int	2.3
int	>	_int	35.2	unsigned char	>=	_int	2.3
int	<	_int	34.8	ptr-to	<=	ptr-to	2.3
int	<=	_int	28.2	unsigned int	>=	unsigned int	2.1
int	<	int	25.5	long	<=	long	2.1
int	<=	int	17.5	long	>=	_int	2.0
other-types	>	other-types	15.8	float	>	_int	2.0
other-types	<	other-types	15.4	unsigned long	>	unsigned long	1.9
int	>	int	15.0	unsigned short	>	unsigned short	1.8
other-types	<=	other-types	14.5	unsigned short	>	_int	1.8
other-types	>=	other-types	13.2	unsigned int	<=	unsigned int	1.7
enum	<=	_int	12.6	ptr-to	>=	ptr-to	1.7
int	>=	int	10.8	int	<=	unsigned long	1.7
enum	>=	enum	7.5	float	>	float	1.7
unsigned int	>=	int	7.3	char	>=	_int	1.7
unsigned int	>	_int	6.0	unsigned long	>=	unsigned long	1.6
long	<	_int	5.3	unsigned long	>	_int	1.5
ptr-to	>	ptr-to	4.1	double	<=	_double	1.5
unsigned int	<=	_int	4.0	unsigned long	<=	unsigned long	1.4
unsigned int	<	unsigned int	3.7	long	>=	long	1.4
unsigned int	>=	_int	3.5	int	<	unsigned long	1.4
char	<=	_int	3.5	unsigned long	<	unsigned long	1.3
unsigned int	>	unsigned int	3.3	long	<	long	1.3
unsigned char	<=	_int	3.1	_long	>=	_long	1.3
long	>	long	2.9	unsigned short	<=	unsigned short	1.2
ptr-to	<	ptr-to	2.8	unsigned int	>	int	1.2
int	<	unsigned int	2.7	float	<	_int	1.2
unsigned long	<=	_int	2.6	unsigned short	<=	_int	1.1
unsigned int	<	_int	2.5	unsigned char	<	_int	1.1
_long	>=	long	2.5	float	<	float	1.1
long	>	_int	2.5	unsigned long	>	int	1.0
enum	>=	_int	2.5	long	>=	int	1.0
unsigned long	>=	int	2.4	float	<=	_int	1.0

relational operators pointer operands

— both operands are pointers to qualified or unqualified versions of compatible object types; or 1200

Commentary

subtraction pointer operands

pointer converting qualified/unqualified

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

C++

5.9p2 *Pointers to objects or functions of the same type (after pointer conversions) can be compared, with a result defined as follows:*

subtraction pointer operands

The pointer conversions (4.4) handles differences in type qualification. But the underlying basic types have to be the same in C++. C only requires that the types be compatible. When one of the pointed-to types is an enumerated type and the other pointed-to type is the compatible integer type, C permits such operands to occur in the same relational-expression; C++ does not (see pointer subtraction for an example).

Other Languages

Few languages support relational comparisons on objects having pointer types.

Table 1200.1: Occurrence of relational operators having particular operand pointer types (as a percentage of all occurrences of each operator with operands having a pointer type). Based on the translated form of this book's benchmark programs.

Left Operand	Op	Right Operand	%	Left Operand	Op	Right Operand	%
char *	>	char *	67.5	const char *	>	const char *	4.0
char *	<=	char *	39.6	other-types	>	other-types	3.8
char *	>=	char *	26.9	int *	>=	int *	3.6
char *	<	char *	25.8	const char *	>=	const char *	3.6
struct *	<=	struct *	23.2	struct *	>	struct *	3.1
unsigned char *	>=	unsigned char *	22.8	short *	<=	short *	3.0
unsigned char *	<	unsigned char *	21.0	other-types	<	other-types	2.8
short *	>=	short *	16.1	unsigned int *	>=	unsigned int *	2.6
struct *	<	struct *	14.9	const char *	<	const char *	2.6
unsigned char *	<=	unsigned char *	13.4	const unsigned char *	<	const unsigned char *	2.0
signed int *	<	signed int *	13.1	unsigned int *	>	unsigned int *	1.9
struct *	>=	struct *	13.0	unsigned long *	<=	unsigned long *	1.8
void *	>	void *	11.0	other-types	<=	other-types	1.8
void *	<	void *	9.4	const char *	<=	const char *	1.8
unsigned char *	>	unsigned char *	8.7	void *	>=	void *	1.6
unsigned short *	<=	unsigned short *	7.9	unsigned short *	<	unsigned short *	1.2
const unsigned char *	<=	const unsigned char *	4.9	unsigned int *	<	unsigned int *	1.2
ptr-to *	<	ptr-to *	4.8	union *	<=	union *	1.2
unsigned short *	>=	unsigned short *	4.7	int *	<	int *	1.2
const unsigned char *	>=	const unsigned char *	4.7	int *	<=	int *	1.2

1201 — both operands are pointers to qualified or unqualified versions of compatible incomplete types. relational operators pointer to incomplete type

Commentary

Because the operands are pointers to compatible types, a relational operator only needs to compare the pointer values (i.e., information on the pointed-to type is not needed to perform the comparison).

C++

C++ classifies incomplete object types that can be completed as object types, so the discussion in the previous C sentence is also applicable here. object types

Other Languages

Those languages that do not support pointer arithmetic invariably do not support the operands of the relational operators having pointer type.

Example

```
extern int a[];

void f(void *p)
{
if (a+1 > p)   /* Constraint violation. */
   ;
if (a+1 > a+2) /* Does not affect the conformance status of the program. */
   ;
}
```

Semantics

relational operators usual arithmetic conversions
arithmetic conversions integer promotions

If both of the operands have arithmetic type, the usual arithmetic conversions are performed. 1202

Commentary

This may also cause the integer promotions to be performed.

Common Implementations

The *comparison* (the term *relational* is not usually used by developers) instructions on some processors can operate on operands of various widths, provided both widths are the same (e.g., those on the Intel x86 processor family can operate on 8-, 16-, or 32-bit operands). If both operands have the same type before the usual arithmetic conversions, an implementation may choose to make use of such instructions.

Coding Guidelines

object ?? int type only

The relational operators produce some of the most unexpected results from developers' point of view. The root cause of the unexpected behavior is invariably a difference in the sign of the types after the integer promotions of the operands. Following the guideline recommendation specifying the use of a single integer type reduces the possibility of this unexpected behavior occurring. However, the use of typedef names from the standard header, or third-party libraries (or even the use of some operators), can cause of a mixture of signed types to appear as operands.

Cg 1202.1 The types of the two operands in a *relational-expression*, after the integer promotions are performed on each of them, shall both be either signed or both unsigned.

Example

```
#include <stdio.h>

extern int glob;

void f(void)
{
if (glob > sizeof(glob))
   {
   if (glob < (int)sizeof(glob))
      print("glob has a negative value\n");
   else
      print("glob has a positive value\n");
   }
else if (glob != 0)
   print("glob has at most 66.66....% chance of being negative\n");
}
```

relational 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. 1203

Commentary

additive operators pointer to object

The same sentence appears elsewhere in the standard and the issues are discussed there.

C++

This wording appears in 5.7p4, Additive operators, but does not appear in 5.9, Relational operators. This would seem to be an oversight on the part of the C++ committee, as existing implementations act as if the requirement was present in the C++ Standard.

1204 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to.

Commentary

Here the term *address space* refers to the object within which the two pointers point. It is not being used in the common usage sense of the address space of the program, which refers to all the storage locations available to an executing program. The standard does not define the absolute location of any object or subobject. However, in some cases it does define their locations relative to other subobjects (and the relational operators are about relative positions).

1206 structure members later compare later

C++

The C++ Standard does not make this observation.

Common Implementations

In most implementations all objects have a location relative to all other objects in the storage used by a program. Processors rarely perform any checks on the values of pointers. Pointers to different storage locations (which may or may not currently be used to hold an object) return a result based on these relative positions in that storage.

Coding Guidelines

Having provided a mechanism to index subcomponents of an object, relational operations on the values of indexing expressions is the same whether the types involved are integers or pointers.

1205 If two pointers to object or incomplete types both point to the same object, or both point one past the last element of the same array object, they compare equal.

Commentary

Relational comparison of pointer values has a meaningful interpretation in C because of the language's support for arithmetic on pointer values. A more detailed specification of pointer equality is given elsewhere in the standard.

pointers compare equal

Some expressions, having pointer type, can be paired as operands of a relational operator but not as operands of the subtraction operator; for instance, given the declarations:

```
struct S {
          int    mem_1;
          double mem_2[5];
          int    mem_3;
          double mem_4[5];
         } x,
           y[10];
```

then the following pairs of expressions may appear together as operands of the relational operators, but not as operands of the subtraction operator:

```
       x   .mem_1    op   x   .mem_3
       y[1].mem_1    op   y[3].mem_3
       y[1].mem_1    op   y[3].mem_3
       x   .mem_2[1] op   x   .mem_4[1]
```

C++

This requirement can be deduced from:

5.9p2

*— If two pointers `p` and `q` of the same type point to the same object or function, or both point one past the end of the same array, or are both null, then `p<=q` and `p>=q` both yield **`true`** and `p<q` and `p>q` both yield **`false`**.*

structure members later compare later
array elements later compare later

If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. 1206

Commentary

member address increasing

The standard does not specify a representation for pointer types, but rather some of the properties they must have. The relative addresses of members of the same structure type are specified elsewhere. This wording specifies that increasing the value of an address causes it to compare greater than the original address, provided both addresses refer to the same structure object. This requirement is not sufficient to calculate the address of subsequent members of a structure type. For instance, the expression `&x.m1+sizeof(x.m1)` does not take into account any padding that may occur after the member `m1`. There is existing source code that uses pointer arithmetic to access the members of a structure object. However, the more widespread usage is interpreting the same object using different types. This requirement and the creation of the `offsetof` macro codifies existing practice.

There are two possible ordering of array elements in storage. Established practice, prior to the design of C, was for increasing index values to refer to elements ever further away from the first. This C sentence specifies this implementation behavior.

structure unnamed padding

Nothing is said about the result of comparing pointers to any padding bytes. Neither is anything said about the behavior of relational operators when their operands point to different objects.

C++

This requirement also applies in C++ (5.9p2). If the declaration of two pointed-to members are separated by an access-specifier label (a construct not available in C), the result of the comparison is unspecified.

Other Languages

The implementation details of array types in Java is sufficiently opaque that storage for each element could be allocated on a different processor, or in contiguous storage locations on one processor.

Coding Guidelines

pointer arithmetic addition result

Common existing practice, for C developers, is to use the less than rather than the not equal operator as a loop termination condition (see Table **??**). Pointer arithmetic is based on accessing the elements of an array, not its bytes. Looping through an array object using pointer arithmetic and relational operators has a fully defined behavior.

Example

```
struct T {
          int first;
          /* Some member declarations. */
          int middle;
          /* More member declarations. */
         } glob;

void f(void)
{
int *p_loc = &glob.first;

/*
```

```
    * Set all members before middle to zero.
    */
   while (p_loc < &glob.middle)
      {
      *p_loc=0;
      p_loc++;
      }
   }
```

1207 All pointers to members of the same union object compare equal.

pointer to union members compare equal

Commentary

This behavior can also be deduced from pointers to the members of the same union type also pointing at the union object and pointers to the same object comparing equal.

union members start same address
pointers compare equal

1208 If the expression **P** points to an element of an array object and the expression **Q** points to the last element of the same array object, the pointer expression **Q+1** compares greater than **P**.

Commentary

This special case deals with pointers that do not point at the same object (but have an association with the same object).

C90

The C90 Standard contains the additional words, after those above:

even though Q+1 does not point to an element of the array object.

Common Implementations

pointer segmented architecture

On segmented architectures incrementing a pointer past the end of a segment causes the address to wrap around to the beginning of that segment (usually address zero). If an array is allocated within such a segment, either the implementation must ensure that there is room after the array for there to be a one past the end address, or it uses some other implementation technique to handle this case (e.g., if the segment used is part of a pointer's representation, a special *one past the end* segment value might be assigned).

1209 In all other cases, the behavior is undefined.

relational pointer comparison undefined if not same object

Commentary

The C relational operator model enables pointers to objects to be treated in the same way as indexes into array objects. Relational comparisons between indexes into two different array objects (that are not both subobjects of a larger object) rarely have any meaning and the standard does not define such support for pointers. Some applications do need to make use of information on the relative locations of different objects in storage. However, this usage was not considered to be of sufficient general utility for the Committee to specify a model defining the behavior.

C90

If the objects pointed to are not members of the same aggregate or union object, the result is undefined with the following exception.

C++

5.9p2

— *Other pointer comparisons are unspecified.*

Source developed using a C++ translator may contain pointer comparisons that would cause undefined behavior if processed by a C translator.

Common Implementations

pointer segmented architecture

Most implementations perform no checks prior to any operation on values having pointer type. Most processors use the same instructions for performing relational comparisons involving pointer types as they use for arithmetic types. For processors that use a segmented memory architecture, a pointer value is often represented using two components, a segment number and an offset within that segment. A consequence of this representation is that there are many benefits in allocating storage for objects such that it fits within a single segment (i.e., storage for an object does not span a segment boundary). One benefit is an optimization involving the generated machine code for some of the relational operators, which only needs to check the segment offset component. This can lead to the situation where `p >= q` is false but `p > q` is true, when `p` and `q` point to different objects.

Coding Guidelines

storage layout

Developers sometimes do intend to perform relational operations on pointers to different objects (e.g., to perform garbage collection). Such usage makes use of information on the layout of objects in storage, this issue is discussed elsewhere.

---

relational operators result value

Each of the operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.90) 1210

Commentary

relational operators unordered 1197

The four comparisons (<, <=, >=, >) raise the invalid exception if either operand is a NaN (and returns 0). Some implementations support what are known as *unordered* relational operators. The floating-point comparison macros (`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, and `isunordered`), new in C99, can be used in those cases where NaNs may occur. They are similar to the existing relational operators, but do not raise invalid for NaN operands.

C++

5.9p1 *The operators* **<** *(less than),* **>** *(greater than),* **<=** *(less than or equal to), and* **>=** *(greater than or equal to) all yield* ***false*** *or* ***true****.*

relational operators result type 1211

This difference is only visible to the developer in one case, the result type. 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.

Common Implementations

logical negation result is

The constant 0 is commonly seen as an operand to these operators. The principles used for this case generally apply to all other combinations of operand (see the logical negation operator for details). When one of the operands is not the constant 0, a comparison has to be performed. Most processors require the two operands to be in registers. (A few processors[6] support instructions that compare the contents of storage, but the available addressing modes are usually severely limited.) The comparison of the contents of the two specified registers is reflected in the settings of the bits in the condition flags register. RISC processors do not contain instructions for comparing integer values, the subtract instruction is used to set condition flags (e.g., `x > y` becoming `x-y > 0`). Most processors contain compare instructions that include a small constant as part of their encoding. This removes the need to load a value into a register.

iteration statement syntax

Relational operators are often used to control the number of iterations performed by a loop. The implementation issues involved in this usage are discussed elsewhere.

Coding Guidelines

While the result is specified in numeric terms, most occurrences of this operator are as the top-level operator in a controlling expression (see Table 1197.3). These contexts are usually treated as involving a boolean role, rather than a numeric value. boolean role

1211 The result has type **`int`**. relational operators result type

Commentary

The first C Standard did not include a boolean data type. C99 maintains compatibility with this existing definition.

C++

*The type of the result is **`bool`**.* 5.9p1

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **`sizeof`** operator. Such usage is rare.

Other Languages

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

Coding Guidelines

Most occurrences of these operators are in controlling expressions (see Table 1197.3), and developers are likely to use a thought process based on this common usage. Given this common usage developers often don't need to consider the result in terms of delivering a value having a type, but as a value that determines the flow of control. In such a context the type of the result is irrelevant. Some of the issues that occur when the result of a relational operation is an operand of another operator (e.g., `x=(a<b)`) are discussed elsewhere. boolean role

References

1. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
2. S. Dehaene, E. Dupoux, and J. Mehler. Is numerical comparison digits? Analogical and symbolic effects in two-digit number comparisons. *Journal of Experimental Psychology: Human Perception and Performance*, 16(3):626–641, 1990.
3. C. Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54(1):199–227, 1992.
4. M. Knauff, R. Rauh, and C. Schlieder. Prefered mental models in qualitative spatial reasoning: A cognitive assessment of Allen's calculus. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 200–205. Lawrence Erlbaum Associates, 1995.
5. G. Leech, P. Rayson, and A. Wilson. *Word Frequencies in Written and Spoken English*. Pearson Education, 2001.
6. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.
7. R. S. Moyer and R. H. Bayer. Mental comparison and the symbolic distance effect. *Cognitive Psychology*, 8:228–246, 1976.
8. R. S. Moyer and T. K. Landauer. Time required for judgements of numerical inequality. *Nature*, 215:1519–1520, 1967.
9. J. Zhang and H. Wang. The effect of external representations on numeric tasks. *Quarterly Journal of Experimental Psychology*, 58(5):817–838, Oct. 2005.