

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.5.6 Additive operators

additive-expression
syntax
additive operators *additive-expression*:

multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression - *multiplicative-expression*

Commentary

The use of the term *additive* is based on the most commonly occurring of the two operators.

The probability that the sum of two floating-point numbers, randomly chosen from a logarithmic distribution, overflows is given by^[5] (the probability of the subtraction underflowing has a slightly more complicated form, however it differs by at most 1 part in 10^{-9} of the probability given by this formula):

$$\frac{\pi^2}{6(\ln(\Omega/\omega))^2} \quad (1153.1)$$

where Ω and ω are the largest and smallest representable values respectively.

EXAMPLE
IEC 60559
floating-point

In the case of a representation meeting the minimum requirements of IEC 60559 the overflow probabilities are 7×10^{-7} for single-precision and 4×10^{-11} for double-precision. In practice application constraints may significantly limit the likely range of operand values (see Table ??), resulting in a much lower probability of overflow.

Table ?? lists various results from operating on infinities and NaNs; annex F (of the C Standard) discusses some of the expression optimizing transformations these results may prohibit.

Other Languages

These operators have the same precedence in nearly all languages. Their precedence relative to the multiplicative operators is also common to nearly all languages and with common mathematical usage.

Coding Guidelines

A study by Parkman and Groen^[14] showed subjects a series of simple addition problems of the form $p + q = n$ and asked them to respond as quickly as possible as to whether the value n was correct (both p and q were single-digit values). The time taken for subjects to respond was measured. It was possible to fit the results to a number of straight lines (i.e., $RT = ax + b$). In one case x was determined by the value $\min(p, q)$ in the other by $p + q$. That is, the response time increased as the minimum of the two operands increased, and as the sum of the operands increased. In both cases the difference in response time between when n was correct and when it was not correct (i.e., the value of b) was approximately 75 ms greater for incorrect. The difference in response time over the range of operand values was approximately 175 ms.

rule-base
mistakes

binary *
result

VanLehn studied the subtraction mistakes made by school-children. The results showed a large number of different kinds of mistakes, with no one mistake being significantly more common than the rest (unlike the situation for multiplication).

The so-called *problem size* effect describes the commonly found subject performance difference between solving arithmetic problems involving large numbers and small numbers (e.g., subjects are slower to solve $9 + 7$ than $2 + 3$). A study by Zbrodoff^[20] suggested that this effect may be partly a consequence of the fact that people have to solve arithmetic problems involving small numbers more frequently than larger numbers (i.e., it is a learning effect); interference from answers to other problems was also a factor.

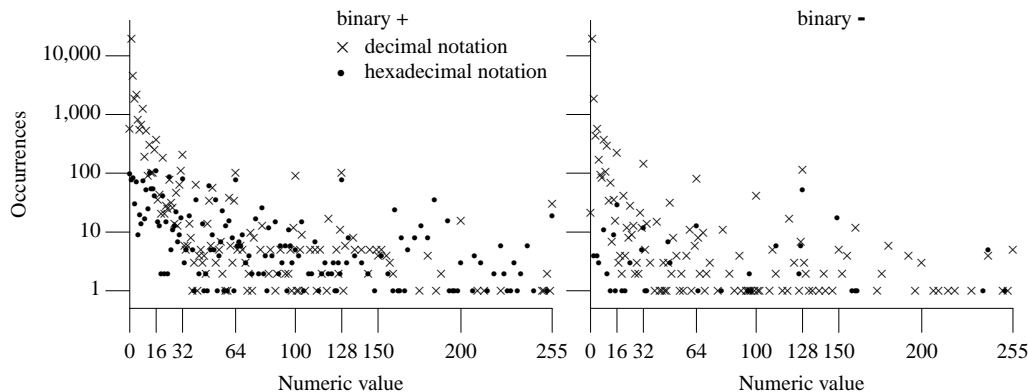


Figure 1153.1: Number of *integer-constants* having a given value appearing as the right operand of additive operators. Based on the visible form of the `.c` files.

Table 1153.1: Common token pairs involving additive 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 +	1.0	77.5	+ sizeof	1.5	3.8
identifier -	0.5	75.7	+ <i>integer-constant</i>	33.7	1.9
) -	0.3	14.7	- <i>integer-constant</i>	44.0	1.3
) +	0.6	12.9	+ identifier	55.4	0.7
+ <i>floating-constant</i>	0.4	7.7	+ (8.3	0.4
<i>integer-constant</i> +	0.4	6.3	- identifier	46.1	0.3
<i>integer-constant</i> -	0.2	5.8	- (6.2	0.1

Constraints

1154 For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to an object type and the other shall have integer type.

addition
operand types

Commentary

Both operands in the expression `0+0` have to be interpreted as having an arithmetic type. Although `0` can be interpreted as the null pointer constant, treating it as such would violate this constraint (because it is not a pointer to object type).

null pointer
constant

C++

The C++ Standard specifies that the null pointer constant has an integer type that evaluates to zero (4.10p1). In C the `NULL` macro might expand to an expression having a pointer type. The expression `NULL+0` is always a null pointer constant in C++, but it may violate this constraint in C. This difference will only affect C source developed using a C++ translator and subsequently translated with a C translator that defines the `NULL` macro to have a pointer type (occurrences of such an expression are also likely to be very rare).

Other Languages

Many languages do not support pointer arithmetic (i.e., adding integer values to pointer values). Some languages use the `+` operator to indicate string concatenation.

Coding Guidelines

When some coding guideline documents prohibit the use of pointer arithmetic, further investigation often reveals that the authors only intended to prohibit the operands of an arithmetic operation having a pointer type. As discussed elsewhere, it is not possible to prevent the use of pointer arithmetic by such a simple-minded

array sub-
script
identical to

interpretation of C semantics.

Enumerated types are often thought about in symbolic, not arithmetic terms. The use of operands having an enumerated type is discussed elsewhere. The rationale for the deviation given for some operators does not apply to the addition operator.

The possibility of arithmetic operations on operands having a boolean role is even less likely than for the multiplication operators.

Table 1154.1: Occurrence of additive 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	37.5	unsigned long	+	_int	2.6
int	-	_int	19.5	unsigned long	-	unsigned long	2.4
other-types	+	other-types	16.2	unsigned int	-	unsigned int	2.2
other-types	-	other-types	16.0	long	-	_int	2.2
_int	+	_int	11.8	_int	-	int	2.1
int	-	int	10.8	ptr-to	-	_int	2.0
_int	-	_int	8.8	long	-	long	2.0
ptr-to	-	ptr-to	6.4	unsigned int	+	_int	1.7
ptr-to	+	unsigned long	6.2	float	+	float	1.7
ptr-to	+	long	5.8	unsigned short	-	int	1.5
float	-	float	5.0	unsigned long	+	unsigned long	1.4
unsigned long	-	_int	4.9	int	-	unsigned short	1.4
int	+	int	4.7	_int	+	int	1.4
unsigned int	-	_int	4.2	unsigned short	+	_int	1.2
ptr-to	+	int	3.7	unsigned short	-	_int	1.1
_unsigned long	-	_int	3.1	unsigned char	-	_int	1.1
ptr-to	-	unsigned long	3.1	unsigned int	+	unsigned int	1.0
ptr-to	+	_int	3.0				

(Incrementing is equivalent to adding 1.)

1155

Commentary

This specifies how the term *incrementing* is to be interpreted in the context of the additive operators.

Coding Guidelines

A common beginner programming mistake is to believe that incrementing an object used for input will cause the next value to be input.

For subtraction, one of the following shall hold:

1156

Commentary

Unlike addition, subtraction is not a symmetrical operator, and it is simpler to specify the different cases via bullet points.

88) This is often called "truncation toward zero".

1157

Commentary

Either the term *truncation toward zero* or *rounded toward zero* is commonly used by developers to describe this behavior.

C90

This term was not defined in the C90 Standard because it was not necessarily the behavior, for this operator, performed by an implementation.

symbolic
name
enumeration
set of named
constants
postfix
operator
operand
multiplicative
operand type

C++

Footnote 74 uses the term *rounded toward zero*.

Other Languages

This term is widely used in many languages.

1158— both operands have arithmetic type;

Commentary

This includes the complex types.

Other Languages

Support for operands of this type is universal to programming languages.

Coding Guidelines

These are the operand types that developers use daily in their nonprogramming lives. The discussion on operands having enumerated types or being treated as boolean, is applicable here.

arithmetic
type
1154 addition
operand types

1159— both operands are pointers to qualified or unqualified versions of compatible object types; or

Commentary

The additive operators are not defined for operands having pointer to function types. The expression 0-0 is covered by the discussion on operand types for addition.

Although the behavior is undefined unless the two pointers point into the same object, one of the operands may be a parameter having a qualified type. Hence the permission for the two operands to differ in the qualification of the pointed-to type. Differences in the qualification of pointed-to types is guaranteed not to affect the equality status of two pointer values.

subtraction
pointer operands

1154 addition
operand types
1173 pointer subtraction
point at same
object

pointer
converting quali-
fied/unqualified

C++

— both operands are pointers to cv-qualified or cv-unqualified versions of the same completely defined object type; or

5.7p2

Requiring the same type means that a C++ translator is likely to issue a diagnostic if an attempt is made to subtract a pointer to an enumerated type from its compatible integer type (or vice versa). The behavior is undefined in C if the pointers don't point at the same object.

```

1  #include <stddef.h>
2
3  enum e_tag {E1, E2, E3}; /* Assume compatible type is int. */
4  union {
5      enum e_tag m_1[5];
6      int m_2[10];
7      } glob;
8
9  extern enum e_tag *p_e;
10 extern int *p_i;
11
12 void f(void)
13 {
14     ptrdiff_t loc = p_i-p_e; /* does not affect the conformance status of the program */
15                             // ill-formed
16 }
```

The expression NULL-0 is covered by the discussion on operand types for addition.

1154 addition
operand types

Other Languages

Support for subtracting two values having pointer types is unique to C (and C++).

Table 1159.1: Occurrence of operands of the subtraction operator having a pointer type (as a percentage of all occurrences of this operator with operands having a pointer type). Based on the translated form of this book's benchmark programs.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
char *	-	char *	48.9	void *	-	void *	1.4
unsigned char *	-	unsigned char *	26.2	int *	-	int *	1.4
struct *	-	struct *	13.7	unsigned short *	-	unsigned short *	1.2
const char *	-	const char *	4.6	other-types	-	other-types	0.0

— the left operand is a pointer to an object type and the right operand has integer type.

1160

Commentary

addition 1154
operand types

The issues are the same as those discussed for addition.

Table 1160.1: Occurrence of additive operators one of whose operands has a pointer type (as a percentage of all occurrences of each operator with one operand having a pointer type). Based on the translated form of this book's benchmark programs. Note: in the translator used the result of the `sizeof` operator had type **unsigned long**.

Left Operand	Operator	Right Operand	%	Left Operand	Operator	Right Operand	%
char *	-	unsigned long	46.0	unsigned char *	-	int	1.7
char *	+	unsigned long	27.3	const char *	-	_int	1.7
char *	+	long	26.8	short *	-	_int	1.6
other-types	+	other-types	10.6	char *	+	unsigned char	1.6
char *	-	_int	9.5	char *	-	int	1.6
struct *	-	array-index	9.1	char *	-	array-index	1.4
unsigned char *	-	_int	8.8	unsigned char *	+	unsigned int	1.3
unsigned char *	+	_int	7.4	unsigned char *	-	array-index	1.3
char *	+	int	6.6	void *	-	_int	1.2
unsigned char *	+	int	5.7	char *	+	signed int	1.2
struct *	-	_int	4.7	unsigned long *	+	int	1.1
char *	+	_int	3.6	struct *	+	_int	1.1
unsigned char *	-	_unsigned long	2.1	unsigned char *	+	unsigned short	1.0
char *	+	unsigned int	1.9	char *	+	unsigned short	1.0
struct *	+	int	1.8	other-types	-	other-types	0.0

(Decrementing is equivalent to subtracting 1.)

1161

Commentary

This specifies how the term *decrementing* is to be interpreted in the context of the additive operators.

Semantics

additive operators
semantics

If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

1162

Commentary

There is no requirement to perform the usual arithmetic conversions if only one of the operands has arithmetic type. The only possible other operand type is a pointer and the result has its type. Performing the usual arithmetic conversions may result in the integer promotions being performed.

Common Implementations

Most processors use the same instructions for performing arithmetic involving pointer types as they use for arithmetic types. An operand having an arithmetic type is likely to undergo the same conversions, irrespective of the type of the other operand (and the same optimizations are likely to be applicable).

usual arith-
metic con-
versions
arithmetic
conversions
integer promotions

Coding Guidelines

There is a common incorrect assumption that if the operands of the additive operators have the same type, the operation will be performed using that type. This issue is discussed elsewhere. If the guideline recommendation specifying the use of a single integer type is followed these conversions will have no effect.

operand
convert automati-
cally
?? object
int type only

1163 The result of the binary + operator is the sum of the operands.

Commentary

It is quite common for a series of values to be added together to create a total. When the operands have a floating type, each addition operation introduces an error. Are some algorithms better than others at minimizing the error in the final result? The following analysis is based on Robertazzi and Schwartz.^[15] This analysis assumes that the relative error for each addition operation will be independent of the others (i.e., have a zero mean) with a variance (mean square error) of σ^2 . Let N represent the number of values being added, which are assumed to be positive. These values can have a number of distributions. In a uniform distribution they are approximately evenly distributed between the minimum and maximum values; while in an exponential distribution they are distributed closer to the maximum value.

A series of values can be added to form a sum in a number of different ways. It is known^[19] that the minimum error in the result occurs if the values are added in order of increasing magnitude. In practice most programs loop over the elements of an array, summing them (i.e., the ordering of the magnitude of the values is random). The mean square error for various simple methods of summing a list of values is shown in columns 2 to 6 of Table 1163.1.

Table 1163.1: Mean square error in the result of summing, using five different algorithms, N values having a uniform or exponential distribution; where μ is the mean of the N values and σ^2 is the mean square error that occurs when two numbers are added.

Distribution	Increasing Order	Random Order	Decreasing Order	Insertion	Adjacency
Uniform ($0, 2\mu$)	$0.2\mu^2 N^3 \sigma^2$	$0.33\mu^2 N^3 \sigma^2$	$0.53\mu^2 N^3 \sigma^2$	$2.6\mu^2 N^2 \sigma^2$	$2.7\mu^2 N^2 \sigma^2$
Exponential (μ)	$0.13\mu^2 N^3 \sigma^2$	$0.33\mu^2 N^3 \sigma^2$	$0.63\mu^2 N^3 \sigma^2$	$2.63\mu^2 N^2 \sigma^2$	$4.0\mu^2 N^2 \sigma^2$

The simple algorithms maintain a single intermediate sum, to which all values are added in turn. Using the observation that the error in addition is minimized if values of similar magnitude are used,^[11] more accurate algorithms are possible. An insertion adder takes the result of the first addition and inserts it into the remaining list of values to be added. The next two lowest values are added and their result inserted into the remaining list, and so on until a single value, the final result, remains. The adjacency algorithm orders the values by magnitude and then pairs them off. The smallest value paired with the next smallest and so on, up to the largest value. Each pair is added to form a new list of values and the process repeated. Eventually a single value, the final result, remains. The error analysis for these two algorithms shows (last two columns of Table 1163.1) a marked improvement. The difference between these sets of results is that the latter varies as the square of the number of values being added, while the former varies as the cube of the number of values.

These formula can be expressed in terms of information content by defining the signal-to-noise ratio as the square of the final sum divided by the mean square error of this sum, then for the simple ordering algorithm the ratio decreases linearly with N . For the insertion and adjacency addition algorithms, it is independent of N . Another way to decrease the error in the simple ordering algorithms is to increase the number of bits in the significand of the result. Performance can be improved to that of the insertion or adjacency algorithm by using $\log_2(\text{sqrt}(N))$ additional bits.

As an example, after adding 4,096 values, the mean square error in the result is approximately 1×10^{-2} using the simple ordering algorithms, and approximately 3×10^{-5} for the insertion and adjacency algorithms.

When negative, as well as positive, values are being summed, there are advantages to working from the largest to the smallest. For an analysis of the performance of seven different algorithms, see Mizukami.^[12]

The preceding analysis provides a formula for the average error, what about the maximum error? Demmel Demmel and Hida

ULP
precision
floating-point

and Hida^[4] analyzed the maximum error in summing a sorted list of values. They proved an upper bound on the error (slightly greater than 1.5 ULPs) provided the condition $N \leq M$ was met, where N is the number of values in the list and $M = 1 + \text{floor}(2^{F-f}/(1 - 2^{-f}))$, f is the precision of the values being added, and F is the precision of the temporary used to hold the running total. They also showed that if $N \geq M + 2$, the relative error could be greater than one (i.e., no relative accuracy).

Using IEC 60559 representation for single- and double-precision values: if 65,537, single-precision values are summed, storing the running total in a temporary having double-precision; the maximum error will only be slightly greater than 1.5 ULPs. If the temporary has single-precision, then three additions are sufficient for the error to be significantly greater than this.

Other Languages

Some languages also use the binary + operator to indicate string concatenation (not just literals, but values of objects).

Common Implementations

Error analysis on the Cray

error analysis

Some Cray processors do not implement IEC 60559 arithmetic. On the Cray the error analysis formula for addition and subtraction differs from that discussed elsewhere. Provided $fl(a \pm b)$ does not overflow or underflow, then:

$$fl(a \pm b) = ((1 + \epsilon_1) \times a) \pm ((1 + \epsilon_2) \times b) \tag{1163.1}$$

In particular, this means that if a and b are nearly equal, so $a - b$ is much smaller than either a or b (known as *extreme cancellation*), then the relative error may be quite large. For instance, when subtracting $1 - x$, where x is the next smaller floating-point number than 1, the answer is twice as large on a Cray C90, and 0 when it should be nonzero on a Cray 2. This happens because when x 's significand is shifted to line its binary point up with 1's binary point, prior to actually subtracting the significands, any bits shifted past the least significant bit of 1 are simply thrown away (differently on a Cray C90 and Cray 2) rather than participating in the subtraction. On IEC 60559 machines, there is a so-called guard digit (actually 3 of them) to store any bits shifted this way, so they can participate in the subtraction.

Usage

A study by Sweeney^[17] dynamically analyzed the floating-point operands of the addition operator. In 26% of cases the values of the two operands were within a factor of 2 of each other, in 13% of cases within a factor of 4, and in 84% of cases within a factor of 1,024.

The result of the binary - operator is the difference resulting from the subtraction of the second operand from the first.

Commentary

If the floating-point operands of a subtraction operator differ from each other by less than a factor of 2, and the result is correctly rounded, then the result is exact^[6] (mathematically, $x/2 < y < 2x \Rightarrow x \ominus y = x - y$).

Common Implementations

Some implementations of this operator complement the value of the right operand and perform an addition. For IEC 60559 arithmetic the expression 0-0 yields -0 when the rounding direction is toward $-\infty$. Cuyt and Verdonk^[3] describe the very different results obtained, using two different processors with various combinations of floating-point types, when evaluating an expression that involves subtracting operands having very similar values.

Example

subtraction
result of

correctly
rounded
result

```

1  #include <stdio.h>
2
3  extern float ef1, ef2;
4
5  void f(void)
6  {
7  if (((ef1 - ef2) == 0.0) &&
8      (ef1 != ef2))
9      printf("ef1 and ef2 have very small values\n");
10 }
```

-
- 1165 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.

additive operators
pointer to object

Commentary

Pointer types specify the type they point at; nothing is assumed about how many objects of that type may be contiguous in storage at the pointed-to location. Common developer terminology is to refer to each of these *elements* as an *object*. The same sentence appears elsewhere in the standard.

relational
operators
pointer to object
equality
operators
pointer to object

Other Languages

More strongly typed languages require that pointer declarations fully specify the type of object pointed at. A pointer to **int** is assumed to point at a single instance of that type. A pointer to an object having an array type requires a pointer to that array type. Needless to say these strongly typed languages do not support the use of pointer arithmetic.

Common Implementations

Implementations that do not treat storage as a linear array of some type are very rare. The Java virtual machine is one such. Here the intent is driven by security issues; programs should not be able to access protected data or to gain complete control of the processor. All data is typed and references (Java does not have pointers as such) to allocated storage must access it using the type with which it was originally allocated.

Coding Guidelines

Having a pointer to an object behaves the same as if it pointed to the first element of an array; it is part of the C model of pointer handling. It fits in with behavior specified in other parts of the standard, and modifying this single behavior creates a disjoint pointer model (not a more strongly typed model).

-
- 1166 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand.

pointer arithmetic
type

Commentary

The additive operation returns a modified pointer value. This pointer value denotes a location having the pointer's pointed-to type.

-
- 1167 If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression.

pointer arithmetic
addition result

Commentary

This specification allows pointers to be treated, by developers, like array indexes. This would not be possible if adding/subtracting integer values to/from pointer values operated at the byte level (unless the operand had pointer to character type).

Common Implementations

Because every byte of an object has a unique address, adding one to a pointer to `int`, for instance, requires adding $(1 * \text{sizeof}(\text{int}))$ to the actual pointer value. Adding, or subtracting, x to a pointer to `type` requires that the pointer value be offset by $(x * \text{sizeof}(\text{type}))$. Subtracting two pointers involves a division operation. Scalar type sizes are often a power of two and optimization techniques, described elsewhere, can be applied. Structure types, may require an actual divide.

byte
address unique
pointer 1174
arithmetic
subtraction result
binary *
result

In other words, if the expression `P` points to the i -th element of an array object, the expressions $(P)+N$ (equivalently, $N+(P)$) and $(P)-N$ (where N has the value n) point to, respectively, the $i+n$ -th and $i-n$ -th elements of the array object, provided they exist. 1168

Commentary

Adding to a pointer value can be compared to adding to an array index variable. The storage, or array, is not accessed, until the dereference operator is applied to it.

array sub-
script
identical to

Moreover, if the expression `P` points to the last element of an array object, the expression $(P)+1$ points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression $(Q)-1$ points to the last element of the array object. 1169

Commentary

The concept of *one past the last object* has a special meaning in C. When array objects are indexed in a loop (for the purposes of this discussion the array index may be an object having pointer type rather than the form `a[i]`), it is common practice to start at the lowest index and work up. This often means that after the last iteration of the loop to the array the index has a value that is one greater than the number of elements it contains (however, the array object is never indexed with this value). This common practice makes one past the end of the array object special, rather than the one before the start of the array object (which has no special rules associated with it).

This specification requires that a pointer be able to point one past the end of an object, but it does not permit storage to be accessed using such a value. There is an implied requirement that all pointers to one past the end of the same object compare equal.

The response to DR #221 pointed out that:

DR #221 *Simply put, $10 == 9+1$. Based on the “as-if” rule, there is no semantic distinction among any of the following:*

```
v+9+1
(v+9)+1
v+(9+1)
v+10
```

This means that there are many ways of creating a pointer to *one past the last element*, other than adding one to a pointer-to the last element.

If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; 1170

Commentary

This is a requirement on the implementation. *Overflow* is not a term usually associated with pointer values. Here it is used in the same sense as that used for arithmetic values (i.e., denoting a value that is outside of the accepted bounds).

Common Implementations

The standard does not impose any representation requirements on the value of “one past the last element of an array object”. There is a practical problem on host processors that use a segmented memory architecture.

pointer
segmented
architecture

If an object occupies an entire segment, there is a unique location for the one past the end pointer, to point. *One past the end* could be represented by adding one to the current pointer value (which for a multiple byte element type would not be a pointer to the next element).

Example

```

1  int a[10];
2
3  void f(void)
4  {
5  int *loc_p1 = (a + 10) - 9; /* Related discussion in DR #221. */
6
7  /*
8   * (a+20) may produce an overflow, any subsequent
9   * operation on the value returned is irrelevant.
10  */
11 int *loc_p2 = (a + 20) - 19 ;
12 int *loc_p3 = a + 20 - 19 ; /* Equivalent to line above. */
13 int *loc_p4 = a +(20 - 19); /* Defined behavior.          */
14 }

```

1171 otherwise, the behavior is undefined.

pointer arithmetic
undefined

Commentary

Any pointer arithmetic that takes a pointer outside of the pointed-to object (apart from the one past exception) is undefined behavior. There is no requirement that the pointer be dereferenced; creating it within a subexpression is sufficient.

Common Implementations

Most implementations treat storage as a contiguous sequence of bytes. Incrementing a pointer simply causes it to point at different locations. On a segmented architecture the value usually wraps from either end of the segment to point at the other end. Some processors^[1] support circular buffers by using modulo arithmetic for operations on pointer values.

pointer
segmented
architecture
Motorola
56000

Some implementations perform checks on the results of pointer arithmetic operations during program execution.^[9,10] (Most implementations that perform pointer checking only perform checks on accesses to storage.)

Coding Guidelines

Expressions whose intermediate results fall outside the defined bounds that can be pointed out (such as the $(a+20)-19$ example above), but whose final result is within those bounds, do occur in practice. In practice, the behavior of the majority of processors results in the expected final value of the expression being returned. Given the behavior of existing processors and the difficulty of enforcing any guideline recommendation (the operands are rarely translation-time constants, meaning the check could only be made during program execution), no recommendation is made here.

1172 If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated.

one past the end
accessing

Commentary

Accessing storage via such a result has undefined behavior. The special case of $\&*p$ is discussed elsewhere. **&***

C++

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

Common Implementations

This is one of the checks that has to be performed by runtime checking tools that claim to do pointer checking.^[2,7,9,10,13,16] The granularity of checking actually performed varies between these tools. Some tools try to minimize the runtime overhead by only performing a small number of checks (e.g., does the pointer point within the stack or within the storage area currently occupied by allocated storage). A few tools check that storage is allocated for objects at the referenced location. Even fewer tools detect a dereference of *a one past the last object* pointer value.

Coding Guidelines

Developers rarely intend to reference storage via such a pointer value. These coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines
not faults

When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object;

1173

Commentary

The standard does not contain any requirements on the relative positions, in storage, of different objects. (Although there is a relational requirement for the members of structure types.) Subtracting two pointers that do not point at elements of the same object (or one past the last element) is undefined behavior. This requirement renders P-Q undefined behavior when both operands have a value that is equal to the null pointer constant.

C90

The C90 Standard did not include the wording “or one past the last element of the array object;”. However, all implementations known to your author handled this case according to the C99 specification. Therefore, it is not listed as a difference.

Common Implementations

Processors do not usually contain special arithmetic instructions for operands having pointer types. The same instruction used for subtracting two integer values is usually used for subtracting two pointer values. This means that checking pointer operands, to ensure that they both point to elements of the same object, incurs a significant runtime time-performance overhead. Only a few of the tools that perform some kind of runtime pointer checking perform this check.^[2,9,10,16] The behavior of most implementations is to treat all storage as a single array object of the appropriate type. Whether the two pointers refer to declared objects does not usually affect behavior.

Coding Guidelines

Developers sometimes do intend to subtract pointers to different array objects and such usage makes use of information on the layout of objects in storage, which is discussed elsewhere.

storage
layout

Example

```

1  #include <stddef.h>
2
3  extern void zero_storage(char *, size_t);
4
5  void f(void)
6  {
7  char start;
8  /* ... */
9  char end;
10
11  if (&start < &end)
12      zero_storage(&start, &end-&start);
13  else

```

```

14     zero_storage(&end, &start-&end);
15 }

```

1174 the result is the difference of the subscripts of the two array elements.

pointer arithmetic
subtraction result

Commentary

This specification matches the behavior for adding/subtracting integer values to/from pointer values. The calculated difference could be added to the value of the right operand to yield the value of the left operand.

1167 pointer
arithmetic
addition result

Common Implementations

Because every byte of an object has a unique address, subtracting two pointers to **int**, for instance, requires dividing the result of the subtraction by `sizeof(int)`. Subtracting two pointers to type requires that the subtracted value be divided by `sizeof(type)`.

byte
address unique

The size of scalar types is often a power of two and an obvious optimization technique is to map the divide to a shift instruction. However, even if the size of the element type is not a power of two, a divide may not need to be performed. A special case of the technique that allows a divide to be replaced by a multiply instruction applies when it is known that the division is exact (i.e., zero remainder).^[18] For instance, on a 32 bit processor, division by 7 can be mapped to multiplication by 0xB6DB6DB7 and extracting the least significant 32 bits of the result.

binary /
result

Note: this technique produces widely inaccurate results if the numerator is not exactly divisible, which can occur if the value of one of the pointers involved in the original subtraction has been modified so that it no longer points to the start of an element.

Example

In the following the first `printf` should yield 10 for both differences. The second `printf` should yield the value `sizeof(int)*10`.

```

1  #include <stdio.h>
2
3  void f(void)
4  {
5  char a1[20], *p11 = a1, *p12 = a1+10;
6  int  a2[20], *p21 = a2, *p22 = a2+10;
7
8  printf("a1 pointer difference=%d, a2 pointer difference=%d\n",
9         (p12 - p11), (p22 - p21));
10 printf("char * cast a2 pointer difference=%d\n",
11        ((char *)p22 - (char *)p21));
12 }

```

1175 The size of the result is implementation-defined, and its type (a signed integer type) is **ptrdiff_t** defined in the `<stddef.h>` header.

pointer subtract
result type

Commentary

Here the word *size* is referring to the numeric value of the result. The implementation-defined value will be greater than or equal to the value of the `PTRDIFF_MIN` macro and less than or equal to the value of the `PTRDIFF_MAX` macro. There is no requirement that `SIZE_MAX == PTRDIFF_MAX` or that `ptrdiff_t` be able to represent a difference in pointers into an array object containing the maximum number of bytes representable in the type `size_t`.

sizeof
result type

It is the implementation's responsibility to ensure that the type it uses for `ptrdiff_t` internally is the same as the typedef definition of `ptrdiff_t` in the supplied header, `<stddef.h>`. A developer can define a

typedef whose name is `ptrdiff_t` (provided the name is unique in its scope). Such a declaration does not affect the type used by a translator as the result type for the subtraction operator when both operands have a pointer type.

Other Languages

Most languages have a single integer type, if they support pointer subtraction, there is often little choice for the result type.

Coding Guidelines

While the type `ptrdiff_t` may cause a change in type of subsequent operations within the containing full expression, the effect is not likely to be significant.

`sizeof`
result type

If the result is not representable in an object of that type, the behavior is undefined.

1176

Commentary

Most processors have an unsigned address space, represented using the same number of bits as the width of the largest integer type supported in hardware. (Operations on values having pointer and integer types are usually performed using the same instructions.) Given that `ptrdiff_t` is a signed type, an implementation only needs to support the creation of an object occupying more than half the total addressable storage before nonrepresentable results can be obtained. (The object need contain only a single element since a pointer to one past the last element can be the second operand.)

`width`
integer type

C90

As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined.

Common Implementations

For the majority of processors that use the same instructions for integer and pointer operands the behavior for nonrepresentable results is likely to be the same in both cases. The IBM AS/400 hardware uses a 16-byte pointer. The ILE C documentation^[8] is silent on the issue of this result not being representable.

Coding Guidelines

Programs creating objects that are sufficiently large for this undefined behavior to be a potential issue are sufficiently rare that these coding guidelines say nothing about the issue.

In other words, if the expressions `P` and `Q` point to, respectively, the i -th and j -th elements of an array object, the expression `(P)-(Q)` has the value $i-j$ provided the value fits in an object of type `ptrdiff_t`.

1177

Commentary

This sentence specifies the behavior described in the previous C sentences in more mathematical terms.

Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object.⁸⁹⁾

1178

Commentary

The equivalence between these expressions requires that pointer subtraction produce the same result when one or more of the operands point within the same array object or one past the last element of the same array object.

1179 EXAMPLE Pointer arithmetic is well defined with pointers to variable length array types.

```
{
    int n = 4, m = 3;
    int a[n][m];
    int (*p)[m] = a;      // p == &a[0]
    p += 1;              // p == &a[1]
    (*p)[2] = 99;        // a[1][2] == 99
    n = p - a;           // n == 1
}
```

If array `a` in the above example were declared to be an array of known constant size, and pointer `p` were declared to be a pointer to an array of the same known constant size (pointing to `a`), the results would be the same.

Commentary

The machine code generated by a translator to perform the pointer arithmetic needs to know the size of the type pointed-to, which in this example will not be a constant value that is known at translation time.

known constant size

C90

This example, and support for variable length arrays, is new in C99.

Common Implementations

When the pointed-to type is a variable length array, its size is not known at translation time. A variable size on the second or subsequent subscript prevents optimization of any of the implicit multiplication operations needed when performing pointer arithmetic.

1167 pointer arithmetic addition result

At the time of this writing your author is not aware of any C based pointer-checking tools that support checking on pointers to variably sized arrays.

1180 **Forward references:** array declarators (6.7.5.2), common definitions `<stddef.h>` (7.17).

References

1. Agere Systems. *DSP16000 Digital Signal Processor Core Information Manual*. Agere Systems, mn02-026winf edition, June 2002.
2. T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.
3. A. Cuyt and B. Verdonk. A remarkable example of catastrophic cancellation unraveled. *Computing*, 66(3):309–320, 2001.
4. J. Demmel and Y. Hilda. Accurate floating-point summation. Technical Report UCB//CSD-02-1180, University of California, Berkeley, May 2002.
5. A. Feldstein and P. Turner. Overflow, underflow, and severe loss of significance in floating-point addition and subtraction. *IMA Journal of Numerical Analysis*, 6:241–251, 1986.
6. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.
7. R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, pages 125–136, Jan. 1992.
8. IBM. *WebSphere Development Studio ILE C/C++ Programmer's Guide*. IBM Canada Ltd, Ontario, Canada, sc09-27 12-02 edition, May 2001.
9. D. M. Jones. *The Model C Implementation*. Knowledge Software Ltd, 1992.
10. R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In M. Kamkar and D. Byers, editors, *Third International Workshop on Automated Debugging*. Linkoping University Electronic Press, 1997.
11. M. A. Malcolm. On accurate floating-point summation. *Communications of the ACM*, 14(11):731–736, 1971.
12. E. Mizukami. The accuracy of floating-point summation for cg-like methods. Technical Report Technical Report 486, Indiana University, July 1997.
13. Y. Oiwa, T. Sekiguichi, E. Sumii, and A. Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security – Theories and Systems*, pages 133–153. Springer-Verlag, Apr. 2003.
14. J. M. Parkman and G. J. Groen. Temporal aspects of simple addition and comparison. *Journal of Experimental Psychology*, 89(2):335–342, 1971.
15. T. G. Robertazzi and S. C. Schwartz. Best "ordering" for floating point addition. *ACM Transactions on Mathematical Software*, 14(1):101–110, 1988.
16. J. L. Steffen. Adding run-time checking to the portable C compiler. *Software—Practice and Experience*, 22(4):305–316, 1992.
17. D. W. Sweeney. An analysis of floating-point addition. *IBM Systems Journal*, 4(1):31–42, 1965.
18. J. H. S. Warren. *Hacker's Delight*. Addison–Wesley, 4th edition, 2003.
19. J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover, 1994.
20. N. J. Zbrodoff. Why is $9 + 7$ harder than $2 + 3$? Strength and interference as explanations of the problem-size effect. *Memory & Cognition*, 23(6):689–700, 1995.