ARR37-C. Do not add or subtract an integer to a pointer to a non-array object

Pointer arithmetic must be performed only on pointers that reference elements of array objects.

The C Standard, 6.5.6 [ISO/IEC 9899:2011], states the following about pointer arithmetic:

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

Noncompliant Code Example

This noncompliant code example attempts to access structure members using pointer arithmetic. This practice is dangerous because structure members are not guaranteed to be contiguous.

```c
struct numbers {
    short num_a, num_b, num_c;
};

int sum_numbers(const struct numbers *numb) {
    int total = 0;
    const short *numb_ptr;

    for (numb_ptr = &numb->num_a; 
        numb_ptr <= &numb->num_c; 
        numb_ptr++) {
        total += *(numb_ptr);
    }
    return total;
}

int main(void) {
    struct numbers my_numbers = { 1, 2, 3 }; 
    sum_numbers(&my_numbers); 
    return 0;
}
```

Compliant Solution

It is possible to use the `->` operator to dereference each structure member:

```c
total = numb->num_a + numb->num_b + numb->num_c;
```

However, this solution results in code that is hard to write and hard to maintain (especially if there are many more structure members), which is exactly what the author of the noncompliant code example was likely trying to avoid.

Compliant Solution

A better solution is to define the structure to contain an array member to store the numbers in an array rather than a structure, as in this compliant solution:
```c
#include <stddef.h>

struct numbers {
    short a[3];
};

int sum_numbers(const short *numb, size_t dim) {
    int total = 0;
    for (size_t i = 0; i < dim; ++i) {
        total += numb[i];
    }
    return total;
}

int main(void) {
    struct numbers my_numbers = { .a[0]= 1, .a[1]= 2, .a[2]= 3};
    sum_numbers(
        my_numbers.a,
        sizeof(my_numbers.a)/sizeof(my_numbers.a[0])
    );
    return 0;
}
```

Array elements are guaranteed to be contiguous in memory, so this solution is completely portable.

**Exceptions**

**ARR37-C-EX1**: Any non-array object in memory can be considered an array consisting of one element. Adding one to a pointer for such an object yields a pointer one element past the array, and subtracting one from that pointer yields the original pointer. This allows for code such as the following:

```c
#include <stdlib.h>
#include <string.h>

struct s {
    char *c_str;
    /* Other members */
};

struct s *create_s(const char *c_str) {
    struct s *ret;
    size_t len = strlen(c_str) + 1;
    ret = (struct s *)malloc(sizeof(struct s) + len);
    if (ret != NULL) {
        ret->c_str = (char *)(ret + 1);
        memcpy(ret + 1, c_str, len);
    }
    return ret;
}
```

A more general and safer solution to this problem is to use a flexible array member that guarantees the array that follows the structure is properly aligned by inserting padding, if necessary, between it and the member that immediately precedes it.

**Risk Assessment**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARR37-C</td>
<td>Medium</td>
<td>Probable</td>
<td>Medium</td>
<td>P8</td>
<td>L2</td>
</tr>
</tbody>
</table>

**Automated Detection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>19.04</td>
<td></td>
<td>Supported indirectly via MISRA C:2004 Rule 17.4.</td>
</tr>
<tr>
<td>Tool</td>
<td>Version</td>
<td>Rule/Feature</td>
<td>Implementation Note</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------</td>
<td>---------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-ARR37</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>ARRAY_VS_SINGLETON</td>
<td>Implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>567 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-ARR37-a</td>
<td>Pointer arithmetic shall not be applied to pointers that address variables of non-array type</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule ARR37-C</td>
<td>Checks for invalid assumptions about memory organization (rule partially covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>2930, 2931, 2932, 2933</td>
<td></td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>2930, 2931, 2932, 2933, 3705, 3706, 3707</td>
<td></td>
</tr>
</tbody>
</table>

### Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

### Related Guidelines

Key here (explains table format and definitions)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
</table>

### Bibliography

- Banahan 2003: Section 5.3, "Pointers"; Section 5.7, "Expressions Involving Pointers"
- ISO/IEC 9899:2011: 6.5.6, "Additive Operators"
- VU#162289