Below the dashed line on the fourth page, this document contains all of the recommendations from section 6.x.5 *Avoiding the vulnerability or mitigating its effects* of N0436. For each item, a summary headline was created in bold. From the set of all recommendations, the following list of primary recommendations was extracted as the mitigations that appear frequently, apply to a large number of languages or are judged the most effective. Items that were extracted from the complete list are annotated in red.

List of primary recommendations (in no particular order)

Check for or enforce type compatibility
Take advantage of any facility offered by the programming language to declare distinct types and use any mechanism provided by the language processor and related tools to check for or enforce type compatibility.

Do not use undefined language constructs
Ensuring that undefined language constructs are not used.

Use language constructs that have specified behaviour.
Use language constructs that have specified behaviour.

Document uses of implementation-defined features
Document the set of implementation-defined features an application depends upon, so that upon a change of translator, development tools, or target configuration it can be ensured that those dependencies are still met.

Avoid the use of deprecated features of a language.
Avoid the use of deprecated features of a language.

Do not use floating point for exact values
Do not use floating-point for exact values such as monetary amounts. Use floating-point only when necessary such as for fundamentally inexact values such as measurements.

Use static analysis checking

- **Use static analysis tools to detect inappropriate use of enumerators**

Use static analysis tools that will detect inappropriate use of enumerators, such as using them as integers or bit maps, and that detect enumeration definition expressions that are incomplete or incorrect. For languages with a complete enumeration abstraction this is the compiler.

Many entries state to use static analysis checking
As a sub-category of "Use static analysis checking":

- **Enable compiler static analysis checking**

If the default behaviour of the compiler or the language is to suppress checks, then enable them.

- **Fix compiler pointer conversion warnings**

Treat the compiler's pointer-conversion warnings as serious errors.

Perform range checking

- **Perform range checking of integer values**

The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. All integer values originating from a source that is not trusted should be validated

for correctness. However, it is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program [30].

- **Perform range checking prior to calling functions**

Only use library functions that perform checks on the arguments to ensure no buffer overrun can occur (perhaps by writing a wrapper for the Standard provided functions). Perform checks on the argument expressions prior to calling the Standard library function to ensure that no buffer overrun will occur.

- *(this occurs in several other entries)*

Perform thorough testing

A systematic development process, use of development/analysis tools and thorough testing are all common ways of preventing errors, and in this case, off-by-one errors.

Check return values from subprograms

Checking error return values or auxiliary status variables following a call to a subprogram is mandatory unless it can be demonstrated that the error condition is impossible.

Check that a pointer is non-NULL before dereferencing

Before dereferencing a pointer, ensure it is not equal to NULL.

Use only those features of the programming language that enforce a logical structure on the program

Use only those features of the programming language that enforce a logical structure on the program. The program flow follows a simple hierarchical model that employs looping constructs such as for, repeat, do, and while.

Use consistency in fault handling

A strategy for fault handling should be decided. Consistency in fault handling should be the same with respect to critically similar parts.

Allocate and free memory at the same level of abstraction

- **Allocate and free memory at the same level of abstraction**

Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or dereferencing NULL pointers or pointers that are not initialized. To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

- **Allocate and free memory at the same level of abstraction**

Allocating and freeing memory in different modules and levels of abstraction may make it difficult for developers to match requests to free storage with the appropriate storage allocation request. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to memory leaks. To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

Use library file wrappers

All library calls should be wrapped within a 'catch-all' exception handler (if the language supports such a construct), so that any unanticipated exceptions can be caught and handled appropriately. This wrapping may be done for each library function call or for the entire behaviour of the program, for

example, having the exception handler in main for C++. However, note that the latter isn't a complete solution, as static objects are constructed before main is entered and are destroyed after it has been exited. Consequently, MISRA C++ [16] bars class constructors and destructors from throwing exceptions (unless handled locally).

- **Develop wrappers around library functions that check the parameters before calling the function.**

Check parameters

- **Libraries should be defined to validate any values passed to the library before the value is used.**

Perform graceful exits

When execution within a particular context is abandoned due to an exception or error condition, it is important to finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.

Use branch coverage measurement tools

The developer should apply standard branch coverage measurement tools and ensure by 100% coverage that all branches are neither dead nor deactivated.

Use dynamic allocation of memory carefully

Memory leaks can be eliminated by avoiding the use of dynamically allocated storage entirely, or by doing initial allocation exclusively and never allocating once the main execution commences. For safety-critical systems and long running systems, the use of dynamic memory is almost always prohibited, or restricted to the initialization phase of execution.

Use proofs of correctness for critical code

Other means of mitigation include manual review, bounds testing, tool analysis, verification techniques, and proofs of correctness.
(this is suggested in many entries)

Develop and use programming guidelines/standards…

- Adopt programming guidelines (preferably augmented by static analysis) that restrict pointer conversions. For example, consider the rules itemized above from JSF AV [15], CERT C [11], Hatton [18], or MISRA C [12].
- Some of the areas that programming guidelines/standards are suggested:
  - **Use parenthesis to make operations explicit**
  - **Simplify expressions**
  - **Do not permit deallocation**
  - **Name selection and use**
  - **Avoiding similar names or identical names for different entities**
  - **Number of significant characters needed for names**
  - **On initialization of variables**
  - **On parenthesis use around binary operations**
  - **To simplify expressions**
  - **Use parenthesis to simplify expressions**
  - **Prohibit unspecified or undefined behavior**
  - **Annotate no-ops**

- **All statements should have a side effect or cause control flow to change**
- **Mark closing of a construct**
- **Put in a final else statement**
- **Use parenthesis to segment blocks of code**
- **Document uses of implementation defined behavior**

List of all recommendations

Here is the complete list from section 6. Items used in primary list are marked in red.

check for or enforce type compatibility
declare distinct types
Take advantage of any facility offered by the programming language to declare distinct types and use any mechanism provided by the language processor and related tools to check for or enforce type compatibility.

Preclude or detect the occurrence of coercion
Use available language and tools facilities to preclude or detect the occurrence of coercion. If it is not possible, use human review to assist in searching for coercions.

Avoid casting data values
Avoid casting data values except when there is no alternative. Document such occurrences so that the justification is made available to maintainers.

Use the most restricted data type
Use the most restricted data type that suffices to accomplish the job. For example, use an enumeration type to select from a limited set of choices (such as, a switch statement or the discriminant of a union type) rather than a more general type, such as integer. This will make it possible for tooling to check if all possible choices have been covered.

Resolve diagnostics/warnings
Treat every compiler, tool, or run-time diagnostic concerning type compatibility as a serious issue. Do not resolve the problem by modifying the code by inserting an explicit cast, without further analysis; instead examine the underlying design to determine if the type error is a symptom of a deeper problem.

Use casts
Never ignore instances of coercion; if the conversion is necessary, change it to a cast and document the rationale for use by maintainers.

Analyze the problem to be solved to learn the magnitudes and/or the precisions of the quantities needed as auxiliary variables, partial results and final results.

Explicitly document bit ordering
Any assumption about bit ordering should be explicitly documented.

The way bit ordering is done on the host system and on the systems with which the bit manipulations will be interfaced should be understood.

Use bit fields if available
Bit fields should be used in languages that support them.

Do not use bit operands on signed operands
Bit operators should not be used on signed operands.

Localize bit manipulation code
Localize and document the code associated with explicit manipulation of bits and bit fields.

Do not use floating-point expression in a Boolean test
Do not use a floating-point expression in a Boolean test for equality. Instead, use coding that determines the difference between the two values to determine whether the difference is acceptably small enough so that two values can be considered equal. Note that if the two values are very large, the "small enough" difference can be a very large number.

Use library functions for calculations (generalized this one)
Use library functions with known numerical characteristics whenever possible.

Unless the use of floating-point is simple, an expert in numerical analysis should check the stability and accuracy of the algorithm employed.

Avoid the use of a floating point variable as a loop counter
Avoid the use of a floating-point variable as a loop counter. If it is necessary to use a floating-point value as a loop control, use inequality to determine the loop control (that is, <, <=, > or >=).

Understand the floating-point format used to represent the floating-point numbers. This will provide some understanding of the underlying idiosyncrasies of floating-point arithmetic.

Only use language operators and functions to manipulate floating point numbers
Manipulating the bit representation of a floating-point number should not be done except with built-in language operators and functions that are designed to extract the mantissa and exponent.

Do not use floating point for exact values
Do not use floating-point for exact values such as monetary amounts. Use floating-point only when necessary such as for fundamentally inexact values such as measurements.

Consider the use of decimal floating-point facilities when available.

Use static analysis tools to detect inappropriate use of enumerators
Use static analysis tools that will detect inappropriate use of enumerators, such as using them as integers or bit maps, and that detect enumeration definition expressions that are incomplete or incorrect. For languages with a complete enumeration abstraction this is the compiler.

Perform range checking of integer values

The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. All integer values originating from a source that is not trusted should be validated for correctness. However, it is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program [30].

Perform range checking of integer values
An alternative or ancillary approach is to protect each operation. However, because of the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, this approach can be prohibitively labor intensive and expensive to implement.

Select a language with built in protections on data conversions
A language that generates exceptions on erroneous data conversions might be chosen. Design objects and program flow such that multiple or complex casts are unnecessary. Ensure that any data type casting that you must use is entirely understood to reduce the plausibility of error in use.

Use static analysis to identify questionable numeric conversions
The use of static analysis can often identify whether or not unacceptable numeric conversions will occur. Verifiably in-range operations are often preferable to treating out of range values as an error condition because the handling of these errors has been repeatedly shown to cause denial-of-service problems in actual applications. Faced with a numeric conversion error, the underlying computer system may do one of two things: (a) signal some sort of error condition, or (b) produce a numeric value that is within the range of representable values on that system. The latter semantics may be preferable in some situations in that it allows the computation to proceed, thus avoiding a denial-of-service attack. However, it raises the question of what numeric result to return to the user.

When using C, use rsize_t
A recent innovation from ISO/IEC TR 24731-1 [13] that has been added to the C standard 9899:2011 [4] is the definition of the rsize_t type for the C programming language. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like size_t. Also, some implementations do not support objects as large as the maximum value that can be represented by type size_t. For these reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that RSIZE_MAX be defined as the smaller of the size of the largest object supported or (SIZE_MAX >> 1), even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define RSIZE_MAX as SIZE_MAX, which means that there is no object size that is considered a runtime-constraint violation.

Do not rely solely on the string termination character.
Do not rely solely on the string termination character.

Use library calls that do not rely on string termination characters
Use library calls that do not rely on string termination characters such as strncpy instead of strcpy in the standard C library.

Perform range checking of integer values

Use of implementation-provided functionality to automatically check array element accesses and prevent out-of-bounds accesses.

Use static analysis…to verify array access bounds
Use of static analysis to verify that all array accesses are within the permitted bounds. Such analysis may require that source code contain certain kinds of information, such as, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified.

Perform range checking of integer values
Sanity checks should be performed on all calculated expressions used as an array index or for pointer arithmetic.

Some guideline documents recommend only using variables having an unsigned data type when indexing an array, on the basis that an unsigned data type can never be negative. This recommendation simply converts an indexing underflow to an indexing overflow because the value of the variable will wrap to a large positive value rather than a negative one. Also some languages support arrays whose lower bound is greater than zero, so an index can be positive and be less than the lower bound. Some languages support zero-sized arrays, so any reference to a location within such an array is invalid.
In the past the implementation of array bound checking has sometimes incurred what has been considered to be a high runtime overhead (often because unnecessary checks were performed). It is now practical for translators to perform sophisticated analysis that significantly reduces the runtime overhead (because runtime checks are only made when it cannot be shown statically that no bound violations can occur).

Perform range checking of integer values
Include sanity checks to ensure the validity of any values used as index variables.

The choice could be made to use a language that is not susceptible to these issues.

Use whole array operations when possible
When available, use whole array operations whenever possible.

Use library functions that contain range checking
Perform range checking prior to calling functions
Only use library functions that perform checks on the arguments to ensure no buffer overrun can occur (perhaps by writing a wrapper for the Standard provided functions). Perform checks on the argument expressions prior to calling the Standard library function to ensure that no buffer overrun will occur.

Use static analysis…to detect buffer overruns in calls to library functions
Use static analysis to verify that the appropriate library functions are only called with arguments that do not result in a buffer overrun. Such analysis may require that source code contain certain kinds of information, for example, that the bounds of all declared arrays be explicitly specified, or that pre- and post-conditions be specified as annotations or language constructs.

Fix compiler pointer conversion warnings
Treat the compiler's pointer-conversion warnings as serious errors.

Develop and use programming guidelines/standards

Adopt programming guidelines (preferably augmented by static analysis) that restrict pointer conversions. For example, consider the rules itemized above from JSF AV [15], CERT C [11], Hatton [18], or MISRA C [12].

Other means of assurance might include proofs of correctness, analysis with tools, verification techniques, or other methods.

Only use pointers for composite types
Avoid using pointer arithmetic for accessing anything except composite types.

Use indexing to access array types instead of pointers
Prefer indexing for accessing array elements rather than using pointer arithmetic.

Limit pointer arithmetic calculations to the addition and subtraction of integers
Limit pointer arithmetic calculations to the addition and subtraction of integers.

Check that a pointer is non-NULL before dereferencing
Before dereferencing a pointer, ensure it is not equal to NULL.

Use an implementation that checks whether a pointer is used that designates a memory location that has already been freed.

Develop and use programming guidelines/standards…
Use a coding style that does not permit deallocation.
Use a coding style that does not permit deallocation.

In complicated error conditions, be sure that clean-up routines respect the state of allocation properly. If the language is object-oriented, ensure that object destructors delete each chunk of memory only once.

Ensuring that all pointers are set to NULL once the memory they point to have been freed can be an effective strategy. The utilization of multiple or complex data structures may lower the usefulness of this strategy.

Use static analysis…
Use a static analysis tool that is capable of detecting some situations when a pointer is used after the storage it refers to is no longer a pointer to valid memory location.

Allocate and free memory at the same level of abstraction
Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or dereferencing NULL pointers or pointers that are not initialized. To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

Perform range checking…

Determine applicable upper and lower bounds for the range of all variables and use language mechanisms or static analysis to determine that values are confined to the proper range.

Use static analysis…
Analyze the software using static analysis looking for unexpected consequences of arithmetic operations.

Perform range checking…
Determine applicable upper and lower bounds for the range of all variables and use language mechanisms or static analysis to determine that values are confined to the proper range.

Use static analysis…
Analyze the software using static analysis looking for unexpected consequences of shift operations.

Avoid using shift operations as a surrogate for multiplication and division
Avoid using shift operations as a surrogate for multiplication and division. Most compilers will use the correct operation in the appropriate fashion when it is applicable.

Use a sign extension library, standard function, or appropriate language-specific coding methods to extend signed values.

Use static analysis…
Use static analysis tools to help locate situations in which the conversion of variables might have unintended consequences.

Develop and use programming guidelines/standards
Implementers can create coding standards that provide meaningful guidance on name selection and use.

Develop and use programming guidelines/standards
Good language specific guidelines could eliminate most problems.

Use static analysis…
Use static analysis tools to show the target of calls and accesses and to produce alphabetical lists of names. Human review can then often spot the names that are sorted at an unexpected location or which look almost identical to an adjacent name in the list.

Use static analysis…
Use static tools (often the compiler) to detect declarations that are unused.

Use languages with a requirement to declare names before use or use available tool or compiler options to enforce such a requirement.

Use static analysis…
Use static analysis to identify any dead stores in the program, and ensure that there is a justification for them.

Mark appropriate variables as volatile

If variables are intended to be accessed by other execution threads or external devices, mark them as volatile.

Develop programming guidelines…
Avoid declaring variables of compatible types in nested scopes with similar names.

Use static analysis…
Enable detection of unused variables in the compiler.

Develop programming guidelines…
Use static analysis…
Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and can be used in the same context. A language-specific project coding convention can be used to ensure that such errors are detectable with static analysis.

Develop programming guidelines…
Use static analysis…
Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and has a type that permits it to occur in at least one context where the first entity can occur.

Use language features, if any, which explicitly mark definitions of entities that are intended to hide other definitions.

Use static analysis…
Develop or use tools that identify name collisions or reuse when truncated versions of names cause conflicts.

Develop programming guidelines…
Ensure that all identifiers differ within the number of characters considered to be significant by the implementations that are likely to be used, and document all assumptions.

Avoiding “wholesale” import directives

Using only selective “single name” import directives or using fully qualified names (in both cases, provided that the language offers the respective capabilities)

The general problem of showing that all objects are initialized is intractable; hence developers must carefully structure programs to show that all variables are set before first read on every path throughout the subprogram. Where objects are visible from many modules, it is difficult to determine where the first read occurs, and identify a module that must set the value before that read. When concurrency, interrupts and coroutines are present, it becomes especially imperative to identify where early initialization occurs and to show that the correct order is set via program structure, not by timing, OS precedence, or chance.

Develop programming guidelines…
Initialize variables

The simplest method is to initialize each object at elaboration time, or immediately after subprogram execution commences and before any branches. If the subprogram must commence with conditional statements, then the programmer is responsible to show that every variable declared and not initialized earlier is initialized on each branch. However, the initial value must be a sensible value for the logic of the program. So-called "junk initialization", for example, setting every variable to zero, prevents the use of tools to detect otherwise uninitialized variables.

Applications can consider defining or reserving fields or portions of the object to only be set when fully initialized. However, this approach has the effect of setting the variable to possibly mistaken values while defeating the use of static analysis to find the uninitialized variables.

Use static analysis…
It should be possible to use static analysis tools to show that all objects are set before use in certain specific cases, but as the general problem is intractable, programmers should keep initialization algorithms simple so that they can be analyzed.

Use static analysis…
When declaring and initializing the object together, if the language does not require that the compiler statically verify that the declarative structure and the initialization structure match, use static analysis tools to help detect any mismatches.

Use static analysis…
Initialize components together
When setting compound objects, if the language provides mechanisms to set all components together, use those in preference to a sequence of initializations as this helps coverage analysis; otherwise use tools that perform such coverage analysis and document the initialization. Do not perform partial initializations unless there is no choice, and document any deviations from 100% initialization.

Where default assignments of multiple components are performed, explicit declaration of the component names and/or ranges helps static analysis and identification of component changes during maintenance.

Use static analysis…
Some languages have named assignments that can be used to build reviewable assignment structures that can be analyzed by the language processor for completeness. Languages with positional notation only can use comments and secondary tools to help show correct assignment.

Develop programming guidelines…
Adopt programming guidelines (preferably augmented by static analysis). For example, consider the rules itemized above from JSF AV [15], CERT C [11] or MISRA C [12].

Develop programming guidelines…
Use parenthesis to make operations explicit
Use parentheses around binary operator combinations that are known to be a source of error (for example, mixed arithmetic/bitwise and bitwise/relational operator combinations).

Develop programming guidelines…
Simplify expressions

Break up complex expressions and use temporary variables to make the order clearer.

Develop programming guidelines…
Make use of one or more programming guidelines which (a) prohibit these unspecified or undefined behaviours, and (b) can be enforced by static analysis. (See JSF AV and MISRA rules in Cross reference clause [SAM])

Develop programming guidelines…
Simplify expressions.
Keep expressions simple. Complicated code is prone to error and difficult to maintain.

Develop programming guidelines…
Simplify expressions.
Simplify expressions.

Do not use assignment expressions as function parameters
Do not use assignment expressions as function parameters. Sometimes the assignment may not be executed as expected. Instead, perform the assignment before the function call.

Do not perform assignments within a Boolean expression
Do not perform assignments within a Boolean expression. This is likely unintended, but if it is not, then move the assignment outside of the Boolean expression for clarity and robustness.

Annotate no-ops
On some rare occasions, some statements intentionally do not have side effects and do not cause control flow to change. These should be annotated through comments and made obvious that they are intentionally no-ops with a stated reason. If possible, such reliance on null statements should be avoided.

All statements should have a side effect or cause control flow to change
In general, except for those rare instances, all statements should either have a side effect or cause control flow to change.

Remove dead code unless it is justified
The developer should endeavor to remove dead code from an application unless its presence serves a purpose. When a developer identifies code that is dead because a conditional consistently evaluates to the same value, this could be indicative of an earlier bug or it could be indicative of inadequate path coverage in the test regimen. Additional investigation may be needed to ascertain why the same value is occurring.

Document and justify dead code
The developer should identify any dead code in the application, and provide a justification (if only to themselves) as to why it is there.

Document and justify dead code
The developer should also ensure that any code that was expected to be unused is actually documented as dead code.

Use branch coverage measurement tools…
The developer should apply standard branch coverage measurement tools and ensure by 100% coverage that all branches are neither dead nor deactivated.

Use static analysis…
The developer should use analysis tools to identify unreachable code.

Statically enumerate switched types if possible
Base the switch choice upon the value of an expression that has a small number of potential values that can be statically enumerated. In languages that provide them, a variable of an enumerated type is to be preferred because a possible set of values is known statically and is small in number (as compared, for example, to the value set of an integer variable). Where it is practical to statically enumerate the switched type, it is preferable to omit the default case, because the static analysis is simplified and because maintainers can better understand the intent of the original programmer. When one must switch based upon the value of an instance of some other type, it is necessary to have a default case, preferably to be regarded as a serious error condition.

Avoid flow through from one case to another
Avoid "flowing through" from one case to another. Even if correctly implemented, it is difficult for reviewers and maintainers to distinguish whether the construct was intended or is an error of omission. In cases where flow-through is necessary and intended, an explicitly coded branch may be preferable to clearly mark the intent. Providing comments regarding intention can be helpful to reviewers and maintainers.

Use static analysis…
Perform static analysis to determine if all cases are, in fact, covered by the code. (Note that the use of a default case can hamper the effectiveness of static analysis since the tool cannot determine if omitted alternatives were or were not intended for default treatment.)

Perform manual review
Bounds testing
Use static analysis tools…
Proofs of correctness
Other means of mitigation include manual review, bounds testing, tool analysis, verification techniques, and proofs of correctness.

Develop and use programming guidelines/standards
Adopt a convention for marking the closing of a construct that can be checked by a tool, to ensure that program structure is apparent.

Develop and use programming guidelines/standards
Adopt programming guidelines (preferably augmented by static analysis). For example, consider the rules itemized above from JSF AV, MISRA C, MISRA C++ or Hatton.

Proofs of correctness
Use static analysis tools…
Other means of assurance might include proofs of correctness, analysis with tools, verification techniques, or other methods.

Perform manual review…
Pretty-printers and syntax-aware editors may be helpful in finding such problems, but sometimes disguise them.

Include a final else statement to avoid confusion
Include a final else statement at the end of if-…-else-if constructs to avoid confusion.

Use parenthesis to segment blocks of code
Always enclose the body of statements of an if, while, for, do, or other statements potentially introducing a block of code in braces ("{}") or other demarcation indicators appropriate to the language used.

Do not modify a loop control variable in the body of a loop
Not modifying a loop control variable in the body of its associated loop body.

Some languages, such as C and C++ do not explicitly specify which of the variables appearing in a loop header is the control variable for the loop. MISRA C [12] and MISRA C++ [16] have proposed algorithms for deducing which, if any, of these variables is the loop control variable in the programming languages C and C++ (these algorithms could also be applied to other languages that support a C-like for-loop).

Use static analysis tools…
Perform thorough testing
A systematic development process, use of development/analysis tools and thorough testing are all common ways of preventing errors, and in this case, off-by-one errors.

When available, use language features specify the whole structure or the starting and ending indices explicitly
Where references are being made to structure indices and the languages provide ways to specify the whole structure or the starting and ending indices explicitly (for example, Ada provides xxx'First and xxx'Last for each dimension), these should be used always. Where the language doesn't provide these, constants can be declared and used in preference to numeric literals.

Where the language doesn't encapsulate variable length arrays, encapsulation should be provided through library objects and a coding standard developed that requires such arrays to only be used via those library objects, so the developer does not need to be explicitly concerned with managing bounds values.

Use only those features of the programming language that enforce a logical structure on the program
Use only those features of the programming language that enforce a logical structure on the program. The program flow follows a simple hierarchical model that employs looping constructs such as for, repeat, do, and while.

Avoid using language features such as goto.

Avoid using language features such as continue and break in the middle of loops.

Avoid using language features that transfer control of the program flow via a jump.

Avoid multiple exit points to a function/procedure/method/subroutine.

Avoid multiple entry points to a function/procedure/method/subroutine.

Use available mechanisms to label parameters as constants or with modes like in, out, or inout.

When a choice of mechanisms is available, pass small simple objects using call by copy.

Use call by copy
When a choice of mechanisms is available and the computational cost of copying is tolerable, pass larger objects using call by copy.

Minimize side-effects of subprograms on non-local objects
Avoid using expressions or functions as actual arguments
Use static analysis
Ensure that called subprograms assign values to all output parameters
When the choice of language or the computational cost of copying forbids using call by copy, then take safeguards to prevent aliasing:

- Minimize side-effects of subprograms on non-local objects; when side-effects are coded, ensure that the affected non-local objects are not passed as parameters using call by reference.
- To avoid unintentional aliasing, avoid using expressions or functions as actual arguments; instead assign the result of the expression to a temporary local and pass the local.
- Utilize tools or other forms of analysis to ensure that non-obvious instances of aliasing are absent.
- Perform reviews or analysis to determine that called subprograms fulfill their responsibilities to assign values to all output parameters.

Do not use the address of locally declared entities as storable, assignable or returnable value
Do not use the address of locally declared entities as storable, assignable or returnable value (except where idioms of the language make it unavoidable).

Ensure that the lifetime of the variable containing the address is completely enclosed by the lifetime of the designated object
Where unavoidable, ensure that the lifetime of the variable containing the address is completely enclosed by the lifetime of the designated object.

Never return the address of a local variable as the result of a function call.

Ensure that subprogram signatures match
Take advantage of any mechanism provided by the language to ensure that subprogram signatures match.

Avoid any language features that permit variable numbers of actual arguments without a method of enforcing a match for any instance of a subprogram call.

Ensure that subprogram signatures match

Take advantage of any language or implementation feature that would guarantee matching the subprogram signature in linking to other languages or to separately compiled modules.

Perform manual review…
Intensively review subprogram calls where the match is not guaranteed by tooling.

Ensure that only a trusted source is used when using non-standard imported modules.

Minimize the use of recursion.
Converting recursive calculations to the corresponding iterative calculation. In principle, any recursive calculation can be remodeled as an iterative calculation which will have a smaller impact on some computing resources but which may be harder for a human to comprehend. The cost to human understanding must be weighed against the practical limits of computing resource.

Document instances of recursion
In cases where the depth of recursion can be shown to be statically bounded by a tolerable number, then recursion may be acceptable, but should be documented for the use of maintainers.
It should be noted that some languages or implementations provide special (more economical) treatment of a form of recursion known as tail-recursion. In this case, the impact on computing economy is reduced. When using such a language, tail recursion may be preferred to an iterative calculation.

Given the variety of error handling mechanisms, it is difficult to provide general guidelines. However, dealing with exception handling in some languages can stress the capabilities of static analysis tools and can, in some cases, reduce the effectiveness of their analysis. Inversely, the use of error status variables can lead to confusingly complicated control structures, particularly when recovery is not possible locally. Therefore, for situations where the highest of reliability is required, the decision for or against exception handling deserves careful thought. In any case, exception-handling mechanisms should be reserved for truly unexpected situations and other situations where no local recovery is possible. Situations which are merely unusual, like the end of file condition, should be treated by explicit testing—either prior to the call which might raise the error or immediately afterward.

Error detection, reporting, correction, and recovery should be built in, not bolted on
In general, error detection, reporting, correction, and recovery should not be a late opportunistic add-on, but should be an integral part of a system design.

Check return values from subprograms
Checking error return values or auxiliary status variables following a call to a subprogram is mandatory unless it can be demonstrated that the error condition is impossible.

Handle errors as close as possible to origin of the exception
Equally, exceptions need to be handled by the exception handlers of an enclosing construct as close as possible to the origin of the exception but as far out as necessary to be able to deal with the error.

Document error conditions
For each routine, all error conditions need to be documented and matching error detection and reporting needs to be implemented, providing sufficient information for handling the error situation.

Perform graceful exits
When execution within a particular context is abandoned due to an exception or error condition, it is important to finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.

It is often not appropriate to repair an error situation and retry the operation. It is usually a better solution to finalize and terminate the current context and retreat to a context where the fault can be handled completely.

Leverage available error checking
Error checking provided by the language, the software system, or the hardware should never be disabled in the absence of a conclusive analysis that the error condition is rendered impossible.

Perform manual review…
Because of the complexity of error handling, careful review of all error handling mechanisms is appropriate.

Use defense in depth
In applications with the highest requirements for reliability, defense-in-depth approaches are often appropriate, for example, checking and handling errors even if thought to be impossible.

Use consistency in fault handling
A strategy for fault handling should be decided. Consistency in fault handling should be the same with respect to critically similar parts.

A multi-tiered approach of fault prevention, fault detection and fault reaction should be used.

Use system defined components to assist in uniformity in fault handling
System-defined components that assist in uniformity of fault handling should be used when available. For one example, designing a "runtime constraint handler" (as described in Annex K of 9899:2012 [4]) permits the application to intercept various erroneous situations and perform one consistent response, such as flushing a previous transaction and re-starting at the next one.

Use consistency in fault handling
When there are multiple tasks, a fault-handling policy should be specified whereby a task may

- Halt, and keep its resources available for other tasks (perhaps permitting restarting of the faulting task).
- Halt, and remove its resources (perhaps to allow other tasks to use the resources so freed, or to allow a recreation of the task).
- Halt, and signal the rest of the program to likewise halt.

Use garbage detection
Use of garbage collectors that reclaim memory that will never be used by the application again. Some garbage collectors are part of the language while others are add-ons.

In systems with garbage collectors, set all non-local pointers or references to null, when the designated data is no longer needed, since the data will not be garbage-collected otherwise. In systems without garbage collectors, cause deallocation of the data before the last pointer or reference to the data is lost.

Allocate and free memory at the same level of abstraction
Allocating and freeing memory in different modules and levels of abstraction may make it difficult for developers to match requests to free storage with the appropriate storage allocation request. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to memory leaks. To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

Storage pools are a specialized memory mechanism where all of the memory associated with a class of objects is allocated from a specific bounded region. When used with strong typing one can ensure a strong relationship between pointers and the space accessed such that storage exhaustion in one pool does not affect the code operating on other memory.

Use dynamic allocation of memory carefully
Memory leaks can be eliminated by avoiding the use of dynamically allocated storage entirely, or by doing initial allocation exclusively and never allocating once the main execution commences. For safety-critical systems and long running systems, the use of dynamic memory is almost always prohibited, or restricted to the initialization phase of execution.

Use static analysis…
Use static analysis, which can sometimes detect when allocated storage is no longer used and has not been freed.

Document the properties of an instantiating type necessary for a generic to be valid.

If an instantiating type has the required properties, the whole of the generic should be ensured to be valid, whether actually used in the program or not.

Preferably avoid, but at least carefully document, any 'special cases' where a generic is instantiated with a specific type doesn't behave as it does for other types.

Avoid the use of multiple inheritance whenever possible.

Provide complete documentation of all encapsulated data, and how each method affects that data for each object in the hierarchy.

Inherit only from trusted sources
Inherit only from trusted sources, and, whenever possible, check the version of the parent classes during compilation and/or initialization.

Provide a method that provides versioning information for each class.

Use whatever language features are available to mark a procedure as language defined or application defined.

Be aware of the documentation for every translator in use and avoid using procedure signatures matching those defined by the translator as extending the standard set.

Libraries should be defined to validate any values passed to the library before the value is used.

Develop wrappers around library functions that check the parameters before calling the function.

Demonstrate statically that the parameters are never invalid.

Use only libraries known to have been developed with consistent and validated interface requirements.
Use only libraries known to have been developed with consistent and validated interface requirements. It is noted that several approaches can be taken, some work best if used in conjunction with each other.

Use the inter-language methods and syntax specified by the applicable language standard(s)
Use the inter-language methods and syntax specified by the applicable language standard(s). For example, Fortran and Ada specify how to call C functions.

Understand the calling conventions of all languages used.

For items comprising the inter-language interface:

- Understand the data layout of all data types used.
- Understand the return conventions of all languages used.
- Ensure that the language in which error check occurs is the one that handles the error.
- Avoid assuming that the language makes a distinction between upper case and lower case letters in identifiers.
- Avoid using a special character as the first character in identifiers.
- Avoid using long identifier names.

Verify that the dynamically linked or shared code being used is the same as that which was tested.
Verify that the dynamically linked or shared code being used is the same as that which was tested.

Do not write self-modifying code
Do not write self-modifying code except in extremely rare instances. Most software applications should never have a requirement for self-modifying code.

Document uses of self-modifying code
In those extremely rare instances where its use is justified, self-modifying code should be very limited and heavily documented.

Use tools to create the signatures.

Avoid using translator options or language features to reference library subprograms without proper signatures.

Use library file wrappers
All library calls should be wrapped within a 'catch-all' exception handler (if the language supports such a construct), so that any unanticipated exceptions can be caught and handled appropriately. This wrapping may be done for each library function call or for the entire behaviour of the program, for example, having the exception handler in main for C++. However, note that the latter isn't a complete solution, as static objects are constructed before main is entered and are destroyed after it has been

exited. Consequently, MISRA C++ [16] bars class constructors and destructors from throwing exceptions (unless handled locally).

An alternative approach would be to use only library routines for which all possible exceptions are specified.

Avoid pre-processor directives
Where it is possible to achieve the desired functionality without the use of pre-processor directives, this should be done in preference to the use of pre-processor directives.

Enable compiler static analysis checking
Do not suppress checks at all or restrict the suppression of checks to regions of the code that have been proved to be performance-critical.

Enable compiler static analysis checking
If the default behaviour of the compiler or the language is to suppress checks, then enable them.

Where checks are suppressed, verify that the suppressed checks could not have failed.

Clearly identify code sections where checks are suppressed.

Do not assume that checks in code verified to satisfy all checks could not fail nevertheless due to hardware faults.

Enable compiler static analysis checking
Restrict the suppression of compile-time checks to where the suppression is functionally essential.

Use inherently unsafe operations only when they are functionally essential.

Clearly identify program code that suppresses checks or uses unsafe operations. This permits the focusing of review effort to examine whether the function could be performed in a safer manner.

Avoid obscure or error prone language features
Individual programmers should avoid the use of language features that are obscure or difficult to use, especially in combination with other difficult language features. Organizations should adopt coding standards that discourage use of such features or show how to use them correctly.

Use metrics to track which language features are error prone
Organizations developing software with critically important requirements should adopt a mechanism to monitor which language features are correlated with failures during the development process and during deployment.

Develop coding guidelines…
Organizations should adopt or develop stereotypical idioms for the use of difficult language features, codify them in organizational standards, and enforce them via review processes.

Avoid complicated language features
Avoid the use of complicated features of a language.

Avoid obscure or error prone language features
Avoid the use of rarely used constructs that could be difficult for entry-level maintenance personnel to understand.

Use static analysis…
Static analysis can be used to find incorrect usage of some language features.
It should be noted that consistency in coding is desirable for each of review and maintenance. Therefore, the desirability of the particular alternatives chosen for inclusion in a coding standard does not need to be empirically proven.

Use language constructs that have specified behaviour.
Use language constructs that have specified behaviour.

Test constructs that have unspecified behavior for all possible behaviours
Ensure that a specific use of a construct having unspecified behaviour produces a result that is the same for all of the possible behaviours permitted by the language specification.

Develop coding guidelines…
When developing coding guidelines for a specific language all constructs that have unspecified behaviour should be documented and for each construct the situations where the set of possible behaviours can vary should be enumerated.

Do not use undefined language constructs
Ensuring that undefined language constructs are not used.

Ensuring that a use of a construct having undefined behaviour does not operate within the domain in which the behaviour is undefined. When it is not possible to completely verify the domain of operation during translation a runtime check may need to be performed.

Develop/Use coding guidelines…
When developing coding guidelines for a specific language all constructs that have undefined behaviour should be documented. The items on this list might be classified by the extent to which the behaviour is likely to have some critical impact on the external behaviour of a program (the criticality may vary between different implementations, for example, whether conversion between object and function pointers has well defined behaviour).

Document uses of implementation-defined features
Document the set of implementation-defined features an application depends upon, so that upon a change of translator, development tools, or target configuration it can be ensured that those dependencies are still met.

Review uses of implementation defined behaviours
Ensure that a specific use of a construct having implementation-defined behaviour produces an external behaviour that is the same for all of the possible behaviours permitted by the language specification.

Only use a language implementation whose implementation-defined behaviours are within a known subset of implementation-defined behaviours. The known subset should be chosen so that the 'same external behaviour' condition described above is met.

Document that the default implementation-defined behaviour is changed within the current file
Create highly visible documentation (perhaps at the start of a source file) that the default implementation-defined behaviour is changed within the current file.

Develop/use coding guidelines…
When developing coding guidelines for a specific language all constructs that have implementation defined behaviour shall be documented and for each construct, the situations where the set of possible behaviours can vary shall be enumerated.

Document that the use of implementation-defined behaviour

When applying this guideline on a project the functionality provided by and for changing its implementation-defined behaviour shall be documented.

Use multiple compilers to verify code behaviour
Verify code behaviour using at least two different compilers with two different technologies.

Adhere to the latest published standard for which a suitable complier and development environment is available.
Adhere to the latest published standard for which a suitable complier and development environment is available.

Avoid the use of deprecated features of a language.
Avoid the use of deprecated features of a language.

Stay abreast of language discussions in language user groups and standards groups on the Internet. Discussions and meeting notes will give an indication of problem prone features that should not be used or should be used with caution.