Revised draft language-specific annex for the programming language C

Date	14-February-2011
Contributed by	Larry Wagoner
Original file name	c_language_annex_052710.docx
Notes	Replaces N0295 Modified – Benito, July 2010 Modified – Keaton and Benito, Sept 2010 Modified – Benito, 2010 Modified – Benito December, 2010 Modified – Benito February, 2011, based on comments from meeting #16 and feedback from David Keaton

Annex C
(*informative*)

C. Vulnerability descriptions for the language C

C.1 Identification of standards and associated documents

ISO/IEC 9899:1999 — *Programming Languages—C*

ISO/IEC TR 24731-1:2007 — *Extensions to the C library — Part 1: Bounds-checking interfaces*

ISO/IEC TR 24731-2:2010 — *Extensions to the C library — Part 2: Dynamic Allocation Functions*

ISO/IEC 9899:1999/Cor. 1:2001 — *Programming languages —C*

ISO/IEC 9899:1999/Cor. 2:2004 — *Programming languages —C*

ISO/IEC 9899:1999/Cor. 3:2007 — *Programming languages —C*

Seacord, Robert C. *The CERT C Secure Coding Standard*. Boston: Addison-Wesley, 2008.

GNU Project. GCC Bugs "Non-bugs" http://gcc.gnu.org/bugs.html#nonbugs_c (2009).

C.2 General terminology and concepts

access: An execution-time action, to read or modify the value of an object. Where only one of two actions is meant, *read* or *modify*. Modify includes the case where the new value being stored is the same as the previous value. Expressions that are not evaluated do not access objects.

alignment: The requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address.

argument:
actual argument: The expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation.

behaviour: An external appearance or action.

implementation-defined behaviour: The *unspecified behaviour* where each implementation documents how the choice is made. An example of implementation-defined behaviour is the propagation of the high-order bit when a signed integer is shifted right.

locale-specific behaviour: The behaviour that depends on local conventions of nationality, culture, and language that each implementation documents. An example, locale-specific behaviour is whether the `islower()` function returns true for characters other than the 26 lower case Latin letters.

undefined behaviour: The use of a non-portable or erroneous program construct or of erroneous data, for which the C standard imposes no requirements. Undefined behaviour ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). An example of, undefined behaviour is the behaviour on integer overflow.

unspecified behaviour: The use of an unspecified value, or other behaviour where the C Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. For example, unspecified behaviour is the order in which the arguments to a function are evaluated.

bit: The unit of data storage in the execution environment large enough to hold an object that may have one of two values. It need not be possible to express the address of each individual bit of an object.

byte: The addressable unit of data storage large enough to hold any member of the basic character set of the execution environment. It is possible to express the address of each individual byte of an object uniquely. A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit.

character: An abstract member of a set of elements used for the organization, control, or representation of data.

single-byte character: The bit representation that fits in a byte.

multibyte character: The sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set.

wide character: The bit representation that will fit in an object capable of representing any character in the current locale. The C Standard uses the type name `wchar_t` for this object.

correctly rounded result: The representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision.

diagnostic message: The message belonging to an implementation-defined subset of the implementation's message output. The C Standard requires diagnostic messages for all constraint violations.

implementation: A particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment.

implementation limit: The restriction imposed upon programs by the implementation.

memory location: Either an object of scalar[1] type, or a maximal sequence of adjacent bit-fields all having nonzero width. A bit-field- and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields-fi, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two bit-field-fi in the same structure if all members declared between them are also bit-fields, no matter what the sizes of those intervening bit-fields happen to be. For example a structure declared as

```
struct {
char a;
int b:5, c:11, :0, d:8;
struct { int ee:8; } e;
}
```

contains four separate memory locations: The member `a`, and bit-fields `d` and `e.ee` are separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields `b` and `c` together constitute the fourth memory location. The bit-fields `b` and `c` can't be concurrently modified, but `b` and `a`, can be concurrently modified.

object: The region of data storage in the execution environment, the contents of which can represent values. When referenced, an object may be interpreted as having a particular type.

parameter:

formal parameter: The object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition.

recommended practice: A specification that is strongly recommended as being in keeping with the intent of the C Standard, but that may be impractical for some implementations.

runtime-constraint: A requirement on a program when calling a library function.

value: The precise meaning of the contents of an object when interpreted as having a specific type.

implementation-defined value: An unspecified value where each implementation documents how the choice for the value is selected.

indeterminate value: Is either an unspecified value or a trap representation.

unspecified value: The valid value of the relevant type where the C Standard imposes no requirements on which value is chosen in any instance. An unspecified value cannot be a trap representation.

trap representation: An object representation that need not represent a value of the object type.

block-structured language: A language that has a syntax for enclosing structures between bracketed keywords, such as an `if` statement bracketed by `if` and `endif`, as in FORTRAN, or a code section bracketed by `BEGIN` and `END`, as in PL/1.

comb-structured language: A language that has an ordered set of keywords to define separate sections within a block, analogous to the multiple teeth or prongs in a comb separating sections of the comb. For example, in Ada, a block is a 4-pronged comb with keywords `declare`, `begin`, `exception`, `end`, and the `if` statement in Ada is a 4-pronged comb with keywords `if`, `then`, `else`, `end if`.

[1] Integer types, Floating types and Pointer types are collectively called *scalar* types in the C Standard.

C.3 Type System [IHN]

C.3.1 Applicability to language

C is a statically typed language. In some ways C is both strongly and weakly typed as it requires all variables to be typed, but sometimes allows implicit or automatic conversion between types. For example, C will implicitly convert a `long int` to an `int` and potentially discard many significant digits. Note that integer sizes are implementation defined so that in some implementations, the conversion from a `long int` to an `int` cannot discard any digits since they are the same size. In some implementations, all integer types could be implemented as the same size.

C allows implicit conversions as in the following example:

```
short a = 1023;
int b;
b = a;
```

If an implicit conversion could result in a loss of precision such as in a conversion from a 32 bit `int` to a 16 bit `short int`:

```
int a = 100000;
short b;
b = a;
```

most compilers will issue a warning message.

C has a set of rules to determine how conversion between data types will occur. For instance, every integer type has an integer conversion rank that determines how conversions are performed. The ranking is based on the concept that each integer type contains at least as many bits as the types ranked below it.

The integer conversion rank is used in the usual arithmetic conversions to determine what conversions need to take place to support an operation on mixed integer types.

Other conversion rules exist for other data type conversions. So even though there are rules in place and the rules are rather straightforward, the variety and complexity of the rules can cause unexpected results and potential vulnerabilities. For example, though there is a prescribed order in which conversions will take place, determining how the conversions will affect the final result can be difficult as in the following example:

```
long foo (short a, int b, int c, long d, long e, long f) {
      return (((b + f) * d – a + e) / c);
  }
```

The implicit conversions performed in the `return` statement can be nontrivial to discern, but can greatly impact whether any of the intermediate values wrap around during the computation.

C.3.2 Guidance to language users

- Consideration of the rules for typing and conversions will assist in avoiding vulnerabilities.
- Make casts explicit to give the programmer a clearer vision and expectations of conversions.

C.4 Bit Representations [STR]

C.4.1 Applicability to language

C supports a variety of sizes for integers such as `short int`, `int`, `long int` and `long long int`. Each may either be signed or unsigned. C also supports a variety of bitwise operators that make bit manipulations easy such as left and right shifts and bitwise operators. These bit manipulations can cause unexpected results or vulnerabilities through miscalculated shifts or platform dependent variations.

Bit manipulations are necessary for some applications and may be one of the reasons that a particular application was written in C. Although many bit manipulations can be rather simple in C, such as masking off the bottom three bits in an integer, more complex manipulations can cause unexpected results. For instance, right shifting a signed integer is implementation defined in C, while shifting by an amount greater than or equal to the size of the data type is undefined behaviour. For instance, on a host where an `int` is of size 32 bits,

```
unsigned int foo(const int k) {
      unsigned int i = 1;
      return i << k;
}
```

is undefined for values of `k` greater than or equal to 32.

The storage representation for interfacing with external constructs can cause unexpected results. Byte orders may be in little endian or big endian format and unknowingly switching between the two can unexpectedly alter values.

C.4.2 Guidance to language users

- Only use bitwise operators on unsigned integer values as the results of some bitwise operations on signed integers are implementation defined.
- Use commonly available functions such as `htonl()`, `htons()`, `ntohl()` and `ntohs()` to convert from host byte order to network byte order and vice versa. This would be needed to interface between an i80x86 architecture where the Least Significant Byte is first with the network byte order, as used on the Internet, where the Most Significant Byte is first. ***Note:*** *functions such as these are not part of the C standard and can vary somewhat among different platforms.*
- In cases where there is a possibility that the shift is greater than the size of the variable, perform a check as the following example shows, or a modulo reduction before the shift:

```
unsigned int i;
unsigned int k;
unsigned int shifted_i;
…
if (k < sizeof(unsigned int)*CHAR_BIT)
  shifted_i = i << k;
else
  // handle error condition
  …
```

C.5 Floating-point Arithmetic [PLF]

C.5.1 Applicability to language

C permits the floating-point data types float, double and long double. Due to the approximate nature of floating-point representations, the use of float and double data types in situations where equality is needed or where

rounding could accumulate over multiple iterations could lead to unexpected results and potential vulnerabilities in some situations.

As with most data types, C is flexible in how `float`, `double` and `long double` can be used. For instance, C allows the use of floating-point types to be used as loop counters and in equality statements. Even though a loop may be expected to only iterate a fixed number of times, depending on the values contained in the floating-point type and on the loop counter and termination condition, the loop could execute forever. For instance iterating a time sequence using 10 nanoseconds as the increment:

```
float f;
for (f=0.0; f!=1.0; f+=0.00000001)
…
```

may or may not terminate after 10,000,000 iterations. The representations used for `f` and the accumulated effect of many iterations may cause `f` to not be identical to 1.0 causing the loop to continue to iterate forever.

Similarly, the Boolean test

```
float f=1.336f;
float g=2.672f;
if (f == (g/2))
…
```

may or may not evaluate to true. Given that `f` and `g` are constant values, it is expected that consistent results will be achieved on the same platform. However, it is questionable whether the logic performs as expected when a float that is twice that of another is tested for equality when divided by 2 as above. This can depend on the values selected due to the quirks of floating-point arithmetic.

C.5.2 Guidance to language users

- Do not use a floating-point expression in a Boolean test for equality. In C, implicit casts may make an expression floating-point even though the programmer did not expect it.
- Check for an acceptable closeness in value instead of a test for equality when using floats and doubles to avoid rounding and truncation problems.
- Do not convert a floating-point number to an integer unless the conversion is a specified algorithmic requirement or is required for a hardware interface.

C.6 Enumerator Issues [CCB]

C.6.1 Applicability to language

The enum type in C comprises a set of named integer constant values as in the example:

```
enum abc {A,B,C,D,E,F,G,H} var_abc;
```

The values of the contents of `abc` would be `A`=0, `B`=1, `C`=2, etc. C allows values to be assigned to the enumerated type as follows:

```
enum abc {A,B,C=6,D,E,F=7,G,H} var_abc;
```

This would result in:

`A`=0, `B`=1, `C`=6, `D`=7, `E`=8, `F`=7, `G`=8, `H`=9

yielding both gaps in the sequence of values and repeated values.

If a poorly constructed `enum` type is used in loops, problems can arise. Consider the enumerated type `var_abc`

defined above used in a loop:

```
int x[8];
…
for (i=A; i<=H; i++)
{
  t = x[i];
…
}
```

Because the enumerated type `abc` has been renumbered and because some numbers have been skipped, the array will go out of bounds and there is potential for unintentional gaps in the use of `x`.

C.6.2 Guidance to language users

- Use enumerated types in the default form starting at 0 and incrementing by 1 for each member if possible. The use of an enumerated type is not a problem if it is well understood what values are assigned to the members.
- Use an enumerated type to select from a limited set of choices to make possible the use of tools to detect omissions of possible values such as in switch statements.
- Use the following format if the need is to start from a value other than 0 and have the rest of the values be sequential:

```
enum abc {A=5,B,C,D,E,F,G,H} var_abc;
```

- Use the following format if gaps are needed or repeated values are desired and so as to be explicit as to the values in the `enum`, then:

```
enum abc {
    A=0,
    B=1,
    C=6,
    D=7,
    E=8,
    F=7,
    G=8,
    H=9
} var_abc;
```

C.7 Numeric Conversion Errors [FLC]

C.7.1 Applicability to language

C permits implicit conversions. That is, C will automatically perform a conversion without an explicit cast. For instance, C allows

```
int i;
float f=1.25f;
i = f;
```

This implicit conversion will discard the fractional part of `f` and set `i` to 1. If the value of `f` is greater than `INT_MAX`, then the assignment of `f` to `i` would be undefined.

The rules for implicit conversions in C are defined in the C standard. For instance, integer types smaller than `int` are promoted when an operation is performed on them. If all values of Boolean, character or integer type can be represented as an `int`, the value of the smaller type is converted to an `int`; otherwise, it is converted to an

unsigned int.

Integer promotions are applied as part of the usual arithmetic conversions to certain argument expressions; operands of the unary +, −, and ~ operators, and operands of the shift operators. The following code fragment shows the application of integer promotions:

```
char c1, c2;
c1 = c1 + c2;
```

Integer promotions require the promotion of each variable (c1 and c2) to int size. The two int values are added and the sum is truncated to fit into the char type.

Integer promotions are performed to avoid arithmetic errors resulting from the overflow of intermediate values. For example:

```
signed char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3;
```

In this example, the value of c1 is multiplied by c2. The product of these values is then divided by the value of c3 (according to operator precedence rules). Assuming that signed char is represented as an 8-bit value, the product of c1 and c2 (300) cannot be represented. Because of integer promotions, however, c1, c2, and c3 are each converted to int, and the overall expression is successfully evaluated. The resulting value is truncated and stored in cresult. Because the final result (75) is in the range of the signed char type, the conversion from int back to signed char does not result in lost data. It is possible that the conversion could result in a loss of data should the data be larger than the storage location.

A loss of data (truncation) can occur when converting from a signed type to a signed type with less precision. For example, the following code can result in truncation:

```
signed long int sl = LONG_MAX;
signed char sc = (signed char)sl;
```

The C standard defines rules for integer promotions, integer conversion rank, and the usual arithmetic conversions. The intent of the rules is to ensure that the conversions result in the same numerical values, and that these values minimize surprises in the rest of the computation.

C.7.2 Guidance to language users

- Check the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type. Any conversion from a type with larger precision to a smaller precision type could potentially result in a loss of data. In some instances, this loss of precision is desired. Such cases should be explicitly acknowledged in comments. For example, the following code could be used to check whether a conversion from an unsigned integer to an unsigned character will result in a loss of precision:

```
unsigned int i;
unsigned char c;
…
if  (i <= UCHAR_MAX) {  // check against the maximum value for an object of type unsigned char
  c = (unsigned char) i;
 }
else
 {
   // handle error condition
  }
```

…

- Close attention should be given to all warning messages issued by the compiler regarding multiple casts. Making a cast in C explicit will both remove the warning and acknowledge that the change in precision is on purpose.

C.8 String Termination [CJM]

C.8.1 Applicability to language

A string in C is composed of a contiguous sequence of characters terminated by and including a null character (a byte with all bits set to 0). Therefore strings in C cannot contain the null character except as the terminating character. Inserting a null character in a string either through a bug or through malicious action can truncate a string unexpectedly. Alternatively, not putting a null character terminator in a string can cause actions such as string copies to continue well beyond the end of the expected string. Overflowing a string buffer through the intentional lack of a null terminating character can be used to expose information or to execute malicious code.

C.8.2 Guidance to language users

- Use safer and more secure functions for string handling from the ISO TR24731-1, Extensions to the C library – *Part 1: Bounds-checking interfaces*[2] or the ISO TR24731-2 — *Part II: Dynamic allocation functions*. Both of these Technical Reports define alternative string handling library functions to the existing Standard C Library. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated. One implementation of these functions has been released as the Safe C Library.

C.9 Buffer Boundary Violation [HCB]

C.9.1 Applicability to language

A buffer boundary violation condition occurs when an array is indexed outside its bounds, or pointer arithmetic results in an access to storage that occurs outside the bounds of the object accessed.

In C, the subscript operator `[]` is defined such that `E1[E2]` is identical to `(*((E1)+(E2)))`, so that in either representation, the value in location `(E1+E2)` is returned. C does not perform bounds checking on arrays, so the following code:

```
int foo(const int i) {
      int x[] = {0,0,0,0,0,0,0,0,0,0};
      return x[i];
}
```

will return whatever is in location `x[i]` even if, `i` were equal to -10 or 10 (assuming either subscript was still within the address space of the program). This could be sensitive information or even a return address, which if altered by changing the value of `x[-10]` or `x[10]`, could change the program flow.

[2] Currently this is an optionally normative annex in the WG 14 working paper.

The following code is more appropriate and would not violate the boundaries of the array `x`:

```
int foo( const int i) {
      int x[X_SIZE] = {0};
      if (i < 0 || i >= X_SIZE) {
        return ERROR_CODE;
      }
      else {
            return x[i];
      }
}
```

A buffer boundary violation may also occur when copying, initializing, writing or reading a buffer if attention to the index or addresses used are not taken. For example, in the following move operation there is a buffer boundary violation:

```
char buffer_src[]={"abcdefg"};
char buffer_dest[5]={0};

strcpy(buffer_dest, buffer_src);
```

the `buffer_src` is longer than the `buffer_dest`, and the code does not check for this before the actual copy operation is invoked. A safer way to accomplish this copy would be:

```
char buffer_src[]={"abcdefg"];
char buffer_dest[5]={0};

strncpy(buffer_dest, buffer_src, sizeof(buffer_dest) -1);
```

this would not cause a buffer bounds violation, however, because the destination buffer is smaller than the source buffer, the destination buffer will now hold "abcd", the 5th element of the array would hold the null character.

C.9.2 Guidance to language users

- Validate all input values.
- Check any array index before use if there is a possibility the value could be outside the bounds of the array.
- Use length restrictive functions such as `strncpy()` instead of `strcpy()`.
- Use stack guarding add-ons to detect overflows of stack buffers.
- Do not use the deprecated functions or other language features such as `gets()`.
- Be aware that the use of all of these measures may still not be able to stop all buffer overflows from happening. However, the use of them can make it much rarer for a buffer overflow to occur and much harder to exploit it.
- Use alternative functions as specified in ISO/IEC TR 24731-1:2007 or TR 24731-2:2010. These Technical Reports provides alternative functions for the C Library (as defined in ISO/IEC 9899:1999) that promotes safer, more secure programming. The functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Optionally, failing functions call a""runtime-constraint handler"" to report the error. Data is never written past the end of an array. All string results are null terminated. In addition, the functions in ISO/IEC TR 24731-1:2007 are re-entrant: they never return pointers to static objects owned by the function. ISO/IEC TR 24731-1:2007 also contains functions that address insecurities with the C input-output facilities.

C.28 Switch Statements and Static Analysis [CLL]

C.28.1 Applicability to language

Because of the way in which the switch-case statement in C is structured, it can be relatively easy to unintentionally omit the `break` statement between cases causing unintended execution of statements for some cases.

C contains a `switch` statement of the form:

```
char abc;
/* … */
switch (abc)
{
   case 1:
      sval = "a";
      break;
   case 2:
      sval = "b";
      break;
   case 3:
      sval = "c";
      break;
   default:
      printf ("Invalid selection\n");
```

If there isn't a default case and the switched expression doesn't match any of the cases, then control simply shifts to the next statement after the switch statement block. Unintentionally omitting a `break` statement between two cases will cause subsequent cases to be executed until a `break` or the end of the switch block is reached. This could cause unexpected results.

C.28.2 Guidance to language users

- Only a direct fall through should be allowed from one case to another. That is, every nonempty `case` statement should be terminated with a `break` statement as illustrated in the following example:

```
int i;
/* … */
switch (i)
 {
   case 1:
   case 2:
      i++;          /* fall through from case 1 to 2 is permitted */
      break;
   case 3:
      j++;
   case 4:          /* fall through from case 3 to 4 is not permitted */
                    /* as it is not a direct fall through due to the */
                    /* j++ statement */
  }
```

- All `switch` statements should have a default value if only to indicate that there could exist a case that was unanticipated and thought impossible by the developers. The only exception is for switches on an enumerated type where all possible values can be exhausted. Even in the case of enumerated types, it is suggested that a default be inserted in anticipation of possible code changes to the enumerated type.

C.29 Demarcation of Control Flow [EOJ]

C.29.1 Applicability to language

C is a block-structured language, while languages such as Ada and Pascal are comb-structured languages. Therefore, it may not be readily apparent which statements are part of a loop construct or an `if` statement.

Consider the following section of code:

```
int foo(int a, const int *b) {
      int i=0;

      /* … */
      a = 0;
      for (i=0; i<10; i++);
        {
         a = a + b[i];
        }
}
```

At first it may appear that a will be a sum of the numbers `b[0]` to `b[9]`. However, even though the code is structured so that the "`a = a + b[i]`" code is structured to appear within the `for` loop, the "`;`" at the end of the `for` statement causes the loop to be on a null statement (the "`;`") and the "`a = a + b[i];`" statement to only be executed once. In this case, this mistake may be readily apparent during development or testing. More subtle cases may not be as readily apparent leading to unexpected results.

`If` statements in C are also susceptible to control flow problems since there isn't a requirement in C for there to be an `else` statement for every `if` statement. An `else` statement in C always belong to the most recent `if` statement without an `else`. However, the situation could occur where it is not readily apparent to which if statement an else due to the way the code is indented or aligned.

C.29.2 Guidance to language users

- Enclose the bodies of `if`, `else`, `while`, `for`, etc. in braces. This will reduce confusion and potential problems when modifying the software. For example:

```
int a,b,i;

/* … */

if (i = 10)
 {
   a = 5;          /* this is correct */
   b = 10;
  }
else
    a = 10;        /* this is incorrect -- the assignments to b */
                   /* were added later and were expected to */
    b = 5;         /* be part of the if and else and indented */
                   /* as such, but did not become part of the else */
```

- Use a final `else` statement or a comment stating why the final `else` isn't necessary in all `if` and `else if` statements.

C.30 Loop Control Variables [TEX]

C.30.1 Applicability to language

C allows the modification of loop control variables within a loop. Though this is usually not considered good programming practice as it can cause unexpected problems, the flexibility of C expects the programmer to use this capability responsibly.

Since the modification of a loop control variable within a loop is infrequently encountered, reviewers of C code may not expect it and hence miss noticing the modification. Modifying the loop control variable can cause unexpected results if not carefully done. In C, the following is valid:

```
int a,i;
for (i=1; i<10; i++)
{
   …
   if (a > 7)
     i = 10;
  …
}
```

which would cause the `for` loop to exit once `a` is greater than 7 regardless of the number of loops that have occurred.

C.30.2 Guidance to language users

- Do not modify a loop control variable within a loop. Even though the capability exists in C, it is still considered to be a poor programming practice.

C.31 Off-by-one Error [XZH]

C.31.1 Applicability to language

Arrays are a common place for off by one errors to manifest. In C, arrays are indexed starting at 0, causing the common mistake of looping from 0 to the size of the array as in:

```
int foo() {
      int a[10];
      int i;
      for (i=0, i<=10, i++)
      …
      return (0);
      }
```

Strings in C are also another common source of errors in C due to the need to allocate space for and account for the string sentinel value. A common mistake is to expect to store an n length string in an n length array instead of length n+1 to account for the sentinel '\0'. Interfacing with other languages that do not use sentinel values in strings can also lead to an off by one error.

C does not flag accesses outside of array bounds, so an off by one error may not be as detectable in C as in some other languages. Several good and freely available tools for C can be used to help detect accesses beyond the bounds of arrays that are caused by an off by one error. However, such tools will not help in the case where only a portion of the array is used and the access is still within the bounds of the array.

Looping one more or one less is usually detectable by good testing. Due to the structure of the C language, this

may be the main way to avoid this vulnerability. Unfortunately some cases may still slip through the development and test phase and manifest themselves during operational use.

C.31.2 Guidance to language users

- Use careful programming, testing of border conditions and static analysis tools to detect off by one errors in C.

C.32 Structured Programming [EWD]

C.32.1 Applicability to language

It is as easy to write structured programs in C as it is not to. C contains the `goto` statement, which can create unstructured code. Also, C has `continue`, `break`, and `return` that can create a complicated control flow, when used in an undisciplined manner. Spaghetti code can be more difficult for C static analyzers to analyze and is sometimes used on purpose to intentionally obfuscate the functionality of software. Code that has been modified multiple times by an assortment of programmers to add or remove functionality or to fix problems can be prone to become unstructured.

Because unstructured code in C can cause problems for analyzers (both automated and human) of code, problems with the code may not be detected as readily or at all as would be the case if the software was written in a structured manner.

C.32.2 Guidance to language users

- Write clear and concise structured code to make code as understandable as possible.
- Restrict the use of `goto`, `continue`, `break` and `return` to encourage more structured programming.
- Encourage the use of a single exit point from a function. At times, this guidance can have the opposite effect, such as in the case of an `if` check of parameters at the start of a function that requires the remainder of the function to be encased in the `if` statement in order to reach the single exit point. If, for example, the use of multiple exit points can arguably make a piece of code clearer, then they should be used. However, the code should be able to withstand a critique that a restructuring of the code would have made the need for multiple exit points unnecessary.

C.33 Passing Parameters and Return Values [CSJ]

C.33.1 Applicability to language

C uses *call by value* parameter passing. The parameter is evaluated and its value is assigned to the formal parameter of the function that is being called. A formal parameter behaves like a local variable and can be modified in the function without affecting the actual argument. An object can be modified in a function by passing the address to the object to the function, for example

```
void swap(int *x, int *y)
{
   int t = *x;
   *x = *y;
   *y = t;
   }
```

Where `x` and `y` are integer pointer formal parameters, and `*x` and `*y` in the `swap()` function body dereference

the pointers to access the integers.

C macros use a *call by name* parameter passing; a call to the macro replaces the macro by the body of the macro. This is call *macro expansion*. Macro expansion is applied to the program source text and amounts to the substitution of the formal parameters with the actual parameter expressions. Formal parameters are often parenthesized to avoid syntax issues after the expansion. Call by name parameter passing reevaluates the actual parameter expression each time the formal parameter is read.

C.33.2 Guidance to language users

- Use caution for reevaluation of function calls in parameters with macros.
- Use caution when passing the address of an object. The object passed could be an alias[3].

C.34 Dangling References to Stack Frames [DCM]

C.34.1 Applicability to language

C allows the address of a variable to be stored in a variable. Should this variable's address be, for example, the address of a local variable that was part of a stack frame, then using the address after the local variable has been deallocated can yield unexpected behaviour as the memory will have been made available for further allocation and may indeed been allocated for some other use. Any use of perishable memory after it has been deallocated can lead to unexpected results.

C.34.2 Guidance to language users

- Do not assign the address of an object to any entity which persists after the object has ceased to exist. This is done in order to avoid the possibility of a dangling reference. Once the object ceases to exist, then so will the stored address of the object preventing accidental dangling references.
- Long lived pointers that contain block-local addresses should be assigned the null pointer value before executing a return from the block.

C.35 Subprogram Signature Mismatch [OTR]

C.35.1 Applicability to language

Functions in C may be called with more or less than the number of parameters the receiving function expects. However, most C compilers will generate a warning or an error about this situation. If the number of arguments does not equal the number of parameters, the behaviour is undefined. This can lead to unexpected results when the count or types of the parameters differs from the calling to the receiving function. If too few arguments are sent to a function, then the function could still pop the expected number of arguments from the stack leading to unexpected results.

C allows a variable number of arguments in function calls. A good example of an implementation of this is the `printf()` function. This is specified in the function call by terminating the list of parameters with an ellipsis (`,`

---

[3] An alias is a variable or formal parameter that refers to the same location as another variable or formal parameter.

`...`). After the comma, no information about the number or types of the parameters is supplied. This can be a useful feature for situations such as `printf()`, but the use of this feature outside of special situations can be the basis for vulnerabilities.

Functions may or may not be defined with a function definition. The function definition may or may not contain a parameter type list. If a function that accepts a variable number of arguments is defined without a parameter

type list that ends with the ellipsis notation, the behaviour is undefined.

If the calling and receiving functions differ in the type of parameters, C will, if possible, do an implicit conversion such as the call to `sqrt()` that expects a double:

```
double sqrt(double)
```

the call:

```
root2 = sqrt(2);
```

coerces the integer 2 into the double value 2.0.

C.35.2 Guidance to language users

- Use a function prototype to declare a function with its expected parameters to allow the compiler to check for a matching count and types of the parameters. The prototype contains just the name of the function and its parameters without the body of code that would normally follow.
- Do not use the variable argument feature except in rare instances. The variable argument feature such as is used in `printf()` is difficult to use in a type safe manner.

C.36 Recursion [GDL]

C.36.1 Applicability to language

C permits recursive calls both directly and indirectly through any chain of other functions. However, recursive functions must be implemented carefully in C,C does not have protective mechanisms that could avert serious problems such as an overly large consumption of resources or an overrun of buffers. Since C is frequently cited for its high performance efficiency, the use of recursion in C can be counter to this as recursion can be inefficient both in execution time and memory usage. Some of the modern compilers perform tail-call optimization to make recursion efficient and resource-friendly.

As with many languages, the high consumption of resources for recursive calls can apply to C. It is difficult to predict the complete range of values that a recursive function can execute that will lead to a manageable consumption of resources. Part of this difficulty is that the range of values can change depending on the current load of the host. Manipulation of the input values to a recursive function can result in an intentional exhaustion of system resources leading to a denial of service.

C.36.2 Guidance to language users

- Only use recursion in rare instances. Although recursion can shorten programs considerably, there is a high performance penalty which is contrary to the usual high efficiency of C.
- Only use recursion if it can be proven that adequate resources exist to support the maximum level of recursion possible.

C.37 Returning Error Status [NZN]

C.37.1 Applicability to language

C provides the include file <errno.h> that defines the macros EDOM, EILSEQ and ERANGE, which expand to integer constant expressions with type int, distinct positive values and which are suitable for use in #if preprocessing directives. C also provides the integer errno that can be set to a nonzero value by any library function (if the use of errno is not documented in the description of the function in the C Standard, errno could be used whether or not there is an error). Though these values are defined, inconsistencies in responding to error conditions can lead to vulnerabilities.

C.37.2 Guidance to language users

- Check the returned error status upon return from a function. The C standard library functions provide an error status as the return value and sometimes in an additional global error value.
- Set errno to zero before a library function call in situations where a program intends to check errno before a subsequent library function call.
- Use errno_t to make it readily apparent that a function is returning an error code. Often a function that returns an errno error code is declared as returning a value of type int. Although syntactically correct, it is not apparent that the return code is an errno error code. TR 24731-1 introduced the new type errno_t in <errno.h> that is defined to be type int.

C.38 Termination Strategy [REU]

C.38.1 Applicability to language

Choosing when and where to exit is a design issue, but choosing how to perform the exit may result in the host being left in an unexpected state. C provides several ways of terminating a program including exit(), _Exit(), and abort(). A return from the initial call to the main function is equivalent to calling the exit() function with the value returned by the main function as its argument (this is if the return type of the main function is a type compatible with int, otherwise the termination status returned to the host environment is unspecified) or simply reaching the "}" that terminates the main function returns a value of 0.

All of the termination strategies in C have undefined, unspecified, and/or implementation defined behaviour associated with them. For example, if more than one call to the exit() function is executed by a program, the behaviour is undefined. The amount of clean-up that occurs upon termination such as the removal of temporary files or the flushing of buffers varies and may be implementation defined.

A call to exit() or _Exit() will terminate a program normally. Abnormal program termination will occur when abort() is used to exit a program (unless the signal SIGABRT is caught and the signal handler does not return). Unlike a call to exit(), when either _Exit() or abort() are used to terminate a program, it is implementation defined as to whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed. This can leave a system in an unexpected state.

C provides the function atexit() that allows functions to be registered so that at normal program termination, the registered functions will be executed to perform desired functions. C99 requires the capability to register *at least* 32 functions. Implementations expecting more than 32 registered functions may yield unexpected results.

C.38.2 Guidance to language users

- Use a return from the main() program as it is the cleanest way to exit a C program.
- Use exit() to quickly exit from a deeply nested function.

- Use `abort()` in situations where an abrupt halt is needed. If `abort()` is necessary, the design should protect critical data from being exposed after an abrupt halt of the program.
- Become familiar with the undefined, unspecified and/or implementation aspects of each of the termination strategies.

C.39 Type-breaking Reinterpretation of Data [AMV]

C.39.1 Applicability to language

The primary way in C that a reinterpretation of data is accomplished is through a `union` which may be used to interpret the same piece of memory in multiple ways. If the use of the union members is not managed carefully, then unexpected and erroneous results may occur.

C allows the use of pointers to memory so that an integer pointer could be used to manipulate character data. This could lead to a mistake in the logic that is used to interpret the data leading to unexpected and erroneous results.

C.39.2 Guidance to language users

- Avoid the use of unions as it is relatively easy for there to exist an unexpected program flow that leads to a misinterpretation of the union data.

C.40 Memory Leak [XYL]

C.40.1 Applicability to language

C can allow memory leaks as many programs use dynamically allocated memory. C relies on manual memory management rather than a built in garbage collector primarily since automated memory management can be unpredictable, impact performance and is limited in its ability to detect unused memory such as memory that is still referenced by a pointer, but is never used.

Memory is dynamically allocated in C using the library calls `malloc()`, `calloc()`, and `realloc()`. When the program no longer needs the dynamically allocated memory, it can be released using the library call `free()`. Should there be a flaw in the logic of the program, memory continues to be allocated but is not freed when it is no longer needed. A common situation is where memory is allocated while in a function, the memory is not freed before the exit from the function and the lifetime of the pointer to the memory has ended upon exit from the function.

C.40.2 Guidance to language users

- Use debugging tools such as leak detectors to help identify unreachable memory.
- Allocate and free memory in the same module and at the same level of abstraction to make it easier to determine when and if an allocated block of memory has been freed.
- Use `realloc()` only to resize dynamically allocated arrays.
- Use garbage collectors that are available to replace the usual C library calls for dynamic memory allocation which allocate memory to allow memory to be recycled when it is no longer reachable. The use of garbage collectors may not be acceptable for some applications as the delay introduced when the allocator reclaims memory may be noticeable or even objectionable leading to performance degradation.

C.41 Templates and Generics [SYM]

Does not apply to C.

C.42 Inheritance [RIP]

Does not apply to C.

C.43 Extra Intrinsics [LRM]

Does not apply to C.

C.44 Argument Passing to Library Functions [TRJ]

C.44.1 Applicability to language

Parameter passing in C is either pass by reference or pass by value. There isn't a guarantee that the values being passed will be verified by either the calling or receiving functions. So values outside of the assumed range may be received by a function resulting in a potential vulnerability.

A parameter may be received by a function that was assumed to be within a particular range and then an operation or series of operations is performed using the value of the parameter resulting in unanticipated results and even a potential vulnerability.

C.44.2 Guidance to language users

- Do not make assumptions about the values of parameters.
- Do not assume that the calling or receiving function will be range checking a parameter. It is always safest to not make any assumptions about parameters used in C libraries. Because performance is sometimes cited as a reason to use C, parameter checking in both the calling and receiving functions is considered a waste of time. Since the calling routine may have better knowledge of the values a parameter can hold, it may be considered the better place for checks to be made as there are times when a parameter doesn't need to be checked since other factors may limit its possible values. However, since the receiving routine understands how the parameter will be used and it is good practice to check all inputs, it makes sense for the receiving routine to check the value of parameters. Therefore, in C it is difficult to create a blanket statement as to where the parameter checks should be made and as a result, parameter checks are recommended in both the calling and receiving routines unless knowledge about the calling or receiving routines dictates that this isn't needed.

C.45 Dynamically-linked Code and Self-modifying Code [NYY]

C.45.1 Applicability to language

Most loaders allow dynamically linked libraries also known as shared libraries. Code is designed and tested using a suite of shared libraries which are loaded at execution time. The process of linking and loading is outside the scope of the C standard.

C can allow self-modifying code. In C there isn't a distinction between data space and code space, executable commands can be altered as desired during the execution of the program. Although self-modifying code may be easy to do in C, it can be difficult to understand, test and fix leading to potential vulnerabilities in the code.

Self-modifying code can be done intentionally in C to obfuscate the effect of a program or in some special situations to increase performance. Because of the ease with which executable code can be modified in C, accidental (or maliciously intentional) modification of C code can occur if pointers are misdirected to modify code space instead of data space or code is executed in data space. Accidental modification usually leads to a program crash. Intentional modification can also lead to a program crash, but used in conjunction with other vulnerabilities can lead to more serious problems that affect the entire host.

C.45.2 Guidance to language users

- Use signatures to verify that the shared libraries used are identical to the libraries with which the code was tested.
- Do not use self-modifying code except in rare instances. In those rare instances, self-modifying code in C can and should be constrained to a particular section of the code and well commented.

C.46 Library Signature [NSQ]

C.46.1 Applicability to language

Integrating C and another language into a single executable relies on knowledge of how to interface the function calls, argument lists and data structures so that symbols match in the object code during linking. Byte alignments can be a source of data corruption.

For instance, when calling Fortran from C, several issues arise. Neither C nor Fortran check for mismatch argument types or even the number of arguments. C passes arguments by value and Fortran passes arguments by reference, so addresses must be passed to Fortran rather than values in the argument list. Multidimensional arrays in C are stored in row major order, whereas Fortran stores them in column major order. Strings in C are terminated by a null character, whereas Fortran uses the declared length of a string. These are just some of the issues that arise when calling Fortran programs from C. Each language has its differences with C, so different issues arise with each interface.

Writing a library wrapper is the traditional way of interfacing with code from another language. However, this can be quite tedious and error prone.

C.46.2 Guidance to language users

- Use a tool, if possible, to automatically create the interface wrappers.
- Minimize the use of those issues known to be error prone when interfacing from C, such as passing character strings, passing multi-dimensional arrays to a column major language, interfacing with other parameter formats such as call by reference or name and receiving return codes.

C.47 Unanticipated Exceptions from Library Routines [HJW]

C.47.1 Applicability to language

Calling software routines produced outside of the control of the main application developer puts all of the code at the mercy of the called routines. An unanticipated exception generated from a library routine could have devastating consequences.

C.47.2 Guidance to language users

- Check the values of parameters to ensure appropriate values are passed to libraries in order to reduce or eliminate the chance of an unanticipated exception

---

C.48 Pre-processor Directives [NMP]

C.48.1 Applicability to language

The C pre-processor allows the use of macros that are text-replaced before compilation.

Function-like macros look similar to functions but have different semantics. Because the arguments are text-replaced, expressions passed to a function-like macro may be evaluated multiple times. This can result in unintended and undefined behaviour if the arguments have side effects or are pre-processor directives as described by C99 §6.10 [1]. Additionally, the arguments and body of function-like macros should be fully parenthesized to avoid unintended and undefined behaviour [2].

The following code example demonstrates undefined behaviour when a function-like macro is called with arguments that have side-effects (in this case, the increment operator) [2]:

```
#define CUBE(X) ((X) * (X) * (X))
/* ... */
int i = 2;
int a = 81 / CUBE(++i);
```

The above example could expand to:

```
int a = 81 / ((++i) * (++i) * (++i));
```

this is undefined behaviour so this macro expansion is difficult to predict.

Another mechanism of failure can occur when the arguments within the body of a function-like macro are not fully parenthesized. The following example shows the `CUBE` macro without parenthesized arguments [2]:

```
#define CUBE(X) (X * X * X)
/* ... */
int a = CUBE(2 + 1);
```

This example expands to:

```
int a = (2 + 1 * 2 + 1 * 2 + 1)
```

which evaluates to 7 instead of the intended 27.

C.48.2 Guidance to language users

This vulnerability can be avoided or mitigated in C in the following ways:

- Replace macro-like functions with inline functions where possible. Although making a function inline only suggests to the compiler that the calls to the function be as fast as possible, the extent to which this is done is implementation-defined. Inline functions do offer consistent semantics and allow for better analysis by static analysis tools.
- Ensure that if a function-like macro must be used, that its arguments and body are parenthesized.
- Do not embed pre-processor directives or side-effects such as an assignment, increment/decrement, volatile access, or function call in a function-like macro.

C.49 Obscure Language Features [BRS]

C.49.1 Applicability to language

C is a relatively small language with a limited syntax set lacking many of the complex features of some other languages. Many of the complex features in C are not implemented as part of the language syntax, but rather implemented as library routines. As such, most of the available features in C are used relatively frequently.

Common use across a variety of languages may make some features less obscure. Because of the unstructured code that is frequently the result of using `goto`'s, the `goto` statement is frequently restricted, or even outright banned, in some C development environments. Even though the `goto` is encountered infrequently and the use of it considered obscure, because it is fairly obvious as to its purpose and since its use is common to many other languages, the functionality of it is easily understood by even the most junior of programmers.

The use of a combination of features adds yet another dimension. Particular combinations of features in C may be used rarely together or fraught with issues if not used correctly in combination. This can cause unexpected results and potential vulnerabilities.

C.49.2 Guidance to language users

- Organizations should specify coding standards that restrict or ban the use of features or combinations of features that have been observed to lead to vulnerabilities in the operational environment for which the software is intended.

C.50 Unspecified Behaviour [BQF]

C.50.1 Applicability to language

The C standard has documented, in Annex J.1, 54 instances of unspecified behaviour. Examples of unspecified behaviour are:

- The order in which the operands of an assignment operator are evaluated
- The order in which any side effects occur among the initialization list expressions in an initializer
- The layout of storage for function parameters

Reliance on a particular behaviour that is unspecified leads to portability problems because the expected behaviour may be different for any given instance. Many cases of unspecified behaviour have to do with the order of evaluation of subexpressions and side effects. For example, in the function call

```
f1(f2(x), f3(x));
```

the functions `f2` and `f3` may be called in any order possibly yielding different results depending on the order in which the functions are called.

C.50.2 Guidance to language users

- Do not rely on unspecified behaviour because the behaviour can change at each instance. Thus, any code that makes assumptions about the behaviour of something that is unspecified should be replaced to make it less reliant on a particular installation and more portable.

C.51 Undefined Behaviour [EWF]

C.51.1 Applicability to language

The C standard does not impose any requirements on undefined behaviour. Typical undefined behaviours include doing nothing, producing unexpected results, and terminating the program.

The C standard has documented, in Annex J.2, 191 instances of undefined behaviour that exist in C. One example of undefined behaviour occurs when the value of the second operand of the `/` or `%` operator is zero. This is generally not detectable through static analysis of the code, but could easily be prevented by a check for a zero divisor before the operation is performed. Leaving this behaviour as undefined lessens the burden on the implementation of the division and modulo operators.

Other examples of undefined behaviour are:

- Referring to an object outside of its lifetime
- The conversion to or from an integer type that produces a value outside of the range that can be represented
- The use of two identifiers that differ only in non-significant characters

Relying on undefined behaviour makes a program unstable and non-portable. While some cases of undefined behaviour may be consistent across multiple implementations, it is still dangerous to rely on them. Relying on undefined behaviour can result in errors that are difficult to locate and only present themselves under special circumstances. For example, accessing memory deallocated by `free()` or `realloc()` results in undefined behaviour, but it may work most of the time.

C.51.2 Guidance to language users

- Eliminate to the extent possible all cases of undefined behaviour from a program

---

C.52 Implementation-defined Behaviour [FAB]

C.52.1 Applicability to language

The C standard has documented, in Annex J.3, 112 instances of implementation-defined behaviour. Examples of implementation-defined behaviour are:

- The number of bits in a byte
- The direction of rounding when a floating-point number is converted to a narrower floating-point number
- The rules for composing valid file names

Relying on implementation-defined behaviour can make a program less portable across implementations. However, this is less true than for unspecified and undefined behaviour.

The following code shows an example of reliance upon implementation-defined behaviour:

```
unsigned int x = 50;
x += (x << 2) + 1;  // x = 5x + 1
```

Since the bitwise representation of integers is implementation-defined, the computation on `x` will be incorrect for implementations where integers are not represented in two's complement form.

C.52.2 Guidance to language users

- Eliminate to the extent possible any reliance on implementation-defined behaviour from programs in order to increase portability. Even programs that are specifically intended for a particular implementation may in the future be ported to another environment or sections reused for future implementations.

C.53 Deprecated Language Features [MEM]

C.53.1 Applicability to language

C has deprecated one function, the function `gets()`. The `gets()` function copies a string from standard input into a fixed-size array. There is no safe way to use `gets()` because it performs an unbounded copy of user input. Thus, every use of gets constitutes a buffer overflow vulnerability.

C has deprecated several language features primarily by tightening the requirements for the feature:

- Implicit `int` declarations are no longer allowed.
- Functions cannot be implicitly declared. They must be defined before use or have a prototype.
- The use of the function `ungetc()` at the beginning of a binary file is deprecated.
- The deprecation of aliased array parameters has been removed.
- A `return` without expression is not permitted in a function that returns a value (and vice versa).

Violating any of these features will generate a diagnostic message.

C.53.2 Guidance to language users

- Do not use the function `gets()` as there isn't a safe and secure way to use it.
- Although backward compatibility is sometimes offered as an option for compilers so one can avoid changes to code to be compliant with current language specifications, updating the legacy software to the current standard is a better option.

C.54 Implications for standardization

Future standardization efforts should consider:

- Moving in the direction over time to being a more strongly typed language. Much of the use of weak typing is simply convenience to the developer in not having to fully consider the types and uses of variables. Stronger typing forces good programming discipline and clarity about variables while at the same time removing many unexpected run time errors due to implicit conversions. This is not to say that C should be strictly a strongly typed language – some advantages of C are due to the flexibility that weaker typing provides. It is suggested that when enforcement of strong typing does not detract from the good flexibility that C offers (e.g. adding an integer to a character to step through a sequence of characters) and is only a convenience for programmers (e.g. adding an integer to a floating-point number), then the standard should specify the stronger typed solution.
- A common warning in Annex I should be added for floating-point expressions being used in a Boolean test for equality.
- Adopting the two TRs on safer C library functions, Extensions to the C Library (TR 24731-1: *Part I: Bounds-checking interfaces (this TR has been included in the WG 14 Working Paper as an optionally normative annex)* and TR 24731-2: *Part II: Dynamic allocation functions*, that are currently under consideration by

ISO SC22 WG14).

- Modifying or deprecating many of the C standard library functions that make assumptions about the occurrence of a string termination character.
- Define a string construct that does not rely on the null termination character.
- Defining an array type that does automatic bounds checking.
- Deprecating less safe functions such as `strcpy()` and `strcat()` where a more secure alternative is available.
- Defining safer and more secure replacement functions such as `memncpy()` and `memncmp()` to complement the `memcpy()` and `memcmp()` functions (see in Implications for standardization.XYW).
- Adopting one of the Technical Reports on safer C library functions, Extensions to the C Library (TR 24731-1: Part I: *Bounds-checking interfaces* or TR 24731-2: Part II: *Dynamic allocation functions*, that have been developed by WG 14.[4]
- Defining an array type that does automatic bounds checking.
- Defining functions that contain an extra parameter in `memcpy()` and `memmove()` for the maximum number of bytes to copy. In the past, some have suggested that the size of the destination buffer be used as an additional parameter. Some critics state that this solution is easy to circumvent by simply repeating the parameter that was used for the number of bytes to copy as the parameter for the size of the destination buffer. This analysis and criticism is correct. What is needed is a failsafe check as to the maximum number of bytes to copy. There are several reasons for creating new functions with an additional parameter. This would make it easier for static analysis to eliminate those cases where the memory copy could not be a problem (such as when the maximum number of bytes is demonstrably less than the capacity of the receiving buffer). Manual analysis or more involved static analysis could then be used for the remaining situations where the size of the destination buffer may not be sufficient for the maximum number of bytes to copy. This extra parameter may also help in determining which copies could take place among objects that overlap. Such copying is undefined according to the C standard. It is suggested that safer versions of functions that include a restriction `max_n` on the number of bytes `n` to copy (e.g. `void *memncpy(void * restrict s1,const void * restrict s2,size_t n), const size_t max_n`) be added to the standard in addition to retaining the current corresponding functions (e.g. `memcpy(void * restrict s1,const void * restrict s2,size_t n))`). The additional parameter would be consistent with the copying function pairs that have already been created such as `strcpy()`/`strncpy()` and `strcat()`/`strncat()`. This would allow a safer version of memory copying functions for those applications that want to use them in to facilitate both safer and more secure code and more efficient and accurate static code reviews.[5]
- Restrictions on pointer arithmetic that could eliminate common pitfalls. Pointer arithmetic is error prone and the flexibility that it offers is useful, but some of the flexibility is simply a shortcut that if restricted could lessen the chance of a pointer arithmetic based error.
- Modifying the library `free(void *ptr)` so that it sets `ptr` to `NULL` to prevent reuse of `ptr`.
- Defining a standard way of declaring an attribute to indicate that a variable is intentionally unused.
- A common warning in Annex I should be added for variables with the same name in nested scopes.
- Creating a few standardized precedence orders. Standardizing on a few precedence orders will help to eliminate the confusing intricacies that exist between languages. This would not affect current languages as altering precedence orders in existing languages is too onerous. However, this would set a basis for the future as new languages are created and adopted. Stating that a language uses “ISO precedence order A” would be useful rather than having to spell out the entire precedence order that differs in a conceptually minor way from some other languages, but in a major way when programmers attempt to switch between languages.

[4] TR 24731-1 has been added to the WG 14 working paper as an optionally normative Annex.

[5] This has been addressed by WG 14 in an optionally normative annex in the current working paper

- Deprecating the `goto` statement. The use of the `goto` construct is often spotlighted as the antithesis of good structured programming. Though its deprecation will not instantly make all C code structured, deprecating the `goto` and leaving in place the restricted `goto` variations (e.g. `break` and `continue`) and possibly adding other restricted `goto`'s could assist in encouraging safer and more secure C programming in general.
- Defining a "fallthru" construct that will explicitly bind multiple switch cases together and eliminate the need for the `break` statement. The default would be for a case to break instead of falling through to the next case. Granted this is a major shift in concept, but if it could be accomplished, less unintentional errors would occur.
- Defining an identifier type for loop control that cannot be modified by anything other than the loop control construct would be a relatively minor addition to C that could make C code safer and encourage better structured programming.
- Defining a standardized interface package for interfacing C with many of the top programming languages and a reciprocal package should be developed of the other top languages to interface with C.
- Joining with other languages in developing a standardized set of mechanisms for detecting and treating error conditions so that all languages to the extent possible could use them. Note that this does not mean that all languages should use the same mechanisms as there should be a variety (e.g. label parameters, auxiliary status variables), but each of the mechanisms should be standardized.
- Since fault handling and exiting of a program is common to all languages, it is suggested that common terminology such as the meaning of fail safe, fail hard, fail soft, etc. along with a core API set such as `exit`, `abort`, etc. be standardized and coordinated with other languages.
- Deprecating unions. The primary reason for the use of unions to save memory has been diminished considerably as memory has become cheaper and more available. Unions are not statically type safe and are historically known to be a common source of errors, leading to many C programming guidelines specifically prohibiting the use of unions.
- Creating a recognizable naming standard for routines such that one version of a library does parameter checking to the extent possible and another version does no parameter checking. The first version would be considered safer and more secure and the second could be used in certain situations where performance is key and the checking is assumed to be done in the calling routine. A naming standard could be made such that the library that does parameter checking could be named as usual, say "library_xyz" and an equivalent version that does not do checking could have a "_p" appended, such as "library_xyz_p". Without a naming standard such as this, a considerable number of wasted cycles will be conducted doing a double check of parameters or even worse, no checking will be done in both the calling and receiving routines as each is assuming the other is doing the checking.
- Making the declarations of undefined behaviour more definitive[6]. The collection of undefined behaviour in Annex J.2 is well done with cross references to sections in the standard. Most of the entries are well defined, but the few that use words such as "proper" or "inappropriately" should be better defined.
- Creating an Annex that lists deprecated features.

[6] This is actually being addressed in Annex L of the C Working Paper, see WG 14/N1494.