Proposed top C specific guidance:

1. Make casts explicit in the return value of malloc. (ref to HFC??)
   Example: `s = (struct foo*)malloc(sizeof(struct foo));`
   uses the C type system to enforce that the pointer to the allocated space will be of a type that is appropriate for the size. Because malloc returns a void *, without the cast, "s" could be of any random pointer type; with the cast, that mistake will be caught.
2. Use length restrictive functions such as strncpy(), strncmp(), and strncat(), snprintf(), instead of strcpy(), strcmp and strcat, sprintf(), respectively. When substituting strncpy for strcpy, ensure that the result will always be null-terminated. Use the safer and more secure functions for string handling from the normative annex K of C11 [4], Bounds-checking interfaces.
3. Use commonly available functions such as htonl(), htons(), ntohl() and ntohs() to convert from host byte order to network byte order and vice versa. [6.3]
4. Use stack guarding add-ons to detect overflows of stack buffers.
5. Perform range checking before accessing an array or before calling a memory copying function such as memcpy() and memmove() since bounds checking is not performed automatically.
   In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
6. Create a specific check that a pointer is not null before dereferencing it.
   As this can be expensive in some cases (such as in a `for` loop that performs operations on each element of a large segment of memory), judicious checking of the value of the pointer at key strategic points in the code is recommended.
7. Set a freed pointer to null immediately after a free() call, as illustrated in the following code:
   i. `free (ptr);`
   ii. `ptr = NULL;`
8. Do not use memory allocated by functions such as malloc() before the memory is initialized as the memory contents are indeterminate.
9. Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. These techniques can be omitted if it can be shown at compile time that overflow or underflow is not possible. Any of the following operators have the potential to wrap or have undefined behavior in C:
   ```
   a + b     a - b     a * b     a++       a--
   a += b    a -= b    a *= b    a << b    a >> b  -a
   ```
10. Do not modify a loop control variable within a loop. Even though the capability exists in C, it is still considered to be a dangerous programming practice.
11. Check the value of a larger type before converting to a smaller type to see if the value in the larger type is within the range of the smaller type.

For reference, here is all of the C specific guidance from 24773 (N0461):

- Be aware of the rules for typing and conversions to avoid vulnerabilities.
- Make casts explicit to give the programmer a clearer vision and expectations of conversions.
- 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
```

- 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.
- 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.
- Avoid using loops that iterate over an enum that has representation specified for the enums, unless it can be guaranteed that there are no gaps or repetition of representation values within the enum definition.
- 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;
```

- 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.
- Use the safer and more secure functions for string handling that are defined in normative Annex K from ISO/IEC 9899:2011 [4] or the ISO TR24731-2 — *Part II: Dynamic allocation functions*. Both of these define alternative string handling library functions to the current 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.
- 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 the safer and more secure functions for string handling from the normative annex K of C11 [4], Bounds-checking interfaces. 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, these functions are re-entrant: they never return pointers to static objects owned by the function. Annex K also contains functions that address insecurities with the C input-output facilities.
- Perform range checking before accessing an array since C does not perform bounds checking automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
- Use the safer and more secure functions for string handling from the normative annex K of C11 [4], *Bounds-checking interfaces*. These are alternative string handling library functions. 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.
- Perform range checking before calling a memory copying function such as memcpy() and memmove(). These functions do not perform bounds checking automatically. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
- Use the safer and more secure functions for string handling from the normative annex K of C11 [4], Bounds-checking interfaces.
- Maintain the same type to avoid errors introduced through conversions.
- Heed compiler warnings that are issued for pointer conversion instances. The decision may be made to avoid all conversions so any warnings must be addressed. Note that casting into and out of “void *” pointers will most likely not generate a compiler warning as this is valid in C.

- Consider an outright ban on pointer arithmetic due to the error-prone nature of pointer arithmetic.
- Verify that all pointers are assigned a valid memory address for use.
- Create a specific check that a pointer is not null before dereferencing it. As this can be expensive in some cases (such as in a `for` loop that performs operations on each element of a large segment of memory), judicious checking of the value of the pointer at key strategic points in the code is recommended.
- Set a freed pointer to null immediately after a free() call, as illustrated in the following code:

```
free (ptr);
ptr = NULL;
```

- Do not create and use additional pointers to dynamically allocated memory.
- Only reference dynamically allocated memory using the pointer that was used to allocate the memory.
- Be aware that any of the following operators have the potential to wrap in C:

```
a + b     a - b     a * b     a++       a--
a += b    a -= b    a *= b    a << b    a >> b   -a
```

- Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. These techniques can be omitted if it can be shown at compile time that overflow or underflow is not possible.
- Only conduct bit manipulations on unsigned data types. The number of bits to be shifted by a shift operator should lie between 1 and (n-1), where n is the size of the data type.
- Use names that are clear and non-confusing.
- Use consistency in choosing names.
- Keep names short and concise in order to make the code easier to understand.
- Choose names that are rich in meaning.
- Keep in mind that code will be reused and combined in ways that the original developers never imagined.
- Make names distinguishable within the first few characters due to scoping in C. This will also assist in averting problems with compilers resolving to a shorter name than was intended.
- Do not differentiate names through only a mixture of case or the presence/absence of an underscore character.
- Avoid differentiating through characters that are commonly confused visually such as ‘O’ and ‘0’, ‘l’ (lower case ‘L’), ‘l’ (capital ‘I’) and ‘1’, ‘S’ and ‘5’, ‘Z’ and ‘2’, and ‘n’ and ‘h’.
- Develop coding guidelines to define a common coding style and to avoid the above dangerous practices.
- Use compilers and analysis tools to identify dead stores in the program.
- Declare variables as volatile when they are intentional targets of a store whose value does not appear to be used.
- 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.
- 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.
- 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.
- Heed compiler warning messages about uninitialized variables. These warnings should be

resolved as recommended to achieve a clean compile at high warning levels.

- Do not use memory allocated by functions such as malloc() before the memory is initialized as the memory contents are indeterminate.
- Use parentheses any time arithmetic operators, logical operators, and shift operators are mixed in an expression.
- Expressions should be written so that the same effects will occur under any order of evaluation that the C standard permits since side effects can be dependent on an implementation specific order of evaluation.
- Simplify statements with interspersed comments to aid in accurately programming functionality and help future maintainers understand the intent and nuances of the code. The flexibility of C permits a programmer to create extremely complex expressions.
- Avoid assignments embedded within other statements, as these can be problematic. Each of the following would be clearer and have less potential for problems if the embedded assignments were conducted outside of the expressions:

```
int a,b,c,d;
/* … */
if ((a == b) || (c = (d-1)))  /* the assignment to c may not
                                 occur if a is equal to b */
```

or:

```
int a,b,c;
/* … */
foo (a=b, c);
```

Each is a valid C statement, but each may have unintended results.

- Give null statements a source line of their own. This, combined with enforcement by static analysis, would make clearer the intention that the statement was meant to be a null statement.
- Consider the adoption of a coding standard that limits the use of the assignment statement within an expression.
- Eliminate dead code to the extent possible from C programs.
- Use compilers and analysis tools to assist in identifying unreachable code.
- Use "//" comment syntax instead of "/*…*/" comment syntax to avoid the inadvertent commenting out sections of code.
- Delete deactivated code from programs due to the possibility of accidentally activating it.
- 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 */
   }
```

- Adopt a style that permits your language processor and analysis tools to verify that all cases are covered. Where this is not possible, use a default clause that diagnoses the error.
- Enclose the bodies of if, else, while, for, and similar 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;
      b = 5;

/* If the assignments to b were added later and were expected to be part
of */
/* each if and else clause (they are indented as such), the above code is
*/ /* incorrect: the assignment to b that was intended to be in the else
clause */
/* is not. */
```

- 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.
- Use careful programming, testing of border conditions and static analysis tools to detect off by one errors in C.
- Write clear and concise structured code to make code as understandable as possible.
- Restrict the use of goto, continue, break, return and longjmp 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.
- 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[1]. Aliases can be avoided by following the respective guidelines of TR 24772-1 Clause 6.32.5.
- 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. In particular, never return the address of a local variable as the result of a function call.
- Long lived pointers that contain block-local addresses should be assigned the null pointer value before executing a return from the block.
- 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.

---

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

- 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.
- 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. The normative Annex K from ISO/IEC 9899:2011 [4] introduces the new type errno_t in <errno.h> that is defined to be type int.
- Handle an error as close as possible to the origin of the error but as far out as necessary to be able to deal with the error.
- For each routine, document all error conditions, matching error detection and reporting needs, and provide sufficient information for handling the error situation.
- Use static analysis tools to detect and report missing or ineffective error detection or handling.
- When execution within a particular context encounters an error, finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.
- 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.
- When using unions, implement an explicit discriminant and check its value before accessing the data in the union.
- 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.
- Do not make assumptions about the values of parameters.
- Do not assume that the calling or receiving function will be range checking a parameter. Therefore, check parameters in both the calling and receiving routines unless knowledge about the calling or receiving routines indicates that this is not needed.
  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.

- 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. (Clive to verify)
- 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. In those extremely rare instances where its use is justified, limit the amount of self-modifying code and heavily document it.
- Verify that the dynamically linked or shared code being used is the same as that which was tested.
- Retest when it is possible that the dynamically linked or shared code has changed before using the application.
- Use signatures to verify that the shared libraries used are identical to the libraries with which the code was tested.
- Use a tool, if possible, to automatically create the interface wrappers.
- 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.
- (Organizations) 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.
- Use tool-based static analysis to find incorrect usage of obscure language features where possible.
- 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.
- 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.
- 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.