GLOBAL EDGE Intelligence Of Things	Global Edge Software Ltd. Global Village, IT SEZ, Pattanagere, Mylasandra Village, RVCE Post, Off Mysore Road, Bangalore - 560 059, India	
Document Name	GEMS-GDL-CodingGuidelines.doc	
Preparation Time	30Hours	
Review Time	10Hours	
Rework Time	20 Hours	
Version	2.1	
Issued By	Core Team	
Distribution List	SEPG/Projects/SPE Council	
Originally Prepared By	Suneel R. Sanganahal	Date : May 4, 2002
Last Re-Worked By	Ian Carvalho	Date: 27th February, 2009
Reviewed By	SEPG	Date :March 17 2009
Approved By	SEPG	Date : March 17 2009

GEMS-GDL-Coding Guidelines

RECORD OF CHANGES

Version	Date	Brief Description of Changes	Author	Reviewed By	DCIN No/CRF /PCR No. If Applicable
0.1	02/May/2002	All	Suneel R Sanganahal	SEPG	
1.0	04/May/2004	Front Sheet	Suneel R Sanganahal	SEPG	
1.1	16/July/2008	Fixed Formatting	Ian Carvalho	N/A dues to re-work in next release	
2.0	25/July/2008	Re-worked Complete Coding Guidelines taking into account current coding being done at GlobalEdge	Ian Carvalho	SEPG	
2.1	27/Feb/2009	Section 6.2.5 “Standard Function Definition” has been changed.	Ian Carvalho	SEPG	

Table of Contents

1 Introduction....6
2 Purpose....6
3 Scope and Responsibility....6
4 References....6
5 Coding Standards....7
6 Coding Style....7
6.1 Source File Organization....7
6.2 Formatting, Indentation and Layout....8
6.2.1 Open Brace at Start of Function....8
6.2.2 Open Braces Inside Function....8
6.2.3 Function Names Column....8
6.2.4 Break Long Function Definitions....8
6.2.5 Recommended Style for Function Definition....8
6.2.6 Recommended Style for Function Body....9
6.2.7 Space After Commas....9
6.2.8 Split Expressions Before Operator....9
6.2.9 Extra Parenthesis For Indentation....9
6.2.10 Extra Parenthesis To Remove Ambiguity....9
6.2.11 Do-While Loop Example....9
6.2.12 Switch Statements....10
6.2.13 Structure or Variable Declarations and Typedefs....10
6.2.14 Functions Should Be Short....10
6.2.15 Tabs For Indentation....10
6.2.16 Line Length Limit....11
6.2.17 Indent Options....11
6.3 Commenting and Inline Code Documentation....11
6.4 Declarations and Initialization....12
6.4.1 Identifier Declarations....12
6.4.2 Shadow Variables....12
6.4.3 Multiple Variable Declaration Should Not Span Lines....12
6.4.4 Nested 'if' Statements....13
6.4.5 Casts to Void....13
6.4.6 Storage Duration....13
6.4.7 'Restrict' Qualified Pointers Should Not Reference Overlapping Objects13
6.4.8 Volatile Storage for Non-Cacheable Memory....13
6.4.9 Matching Parameter and Return Types for Function Calls....14
6.4.10 Global Variables....14
6.4.11 Use of Constants....14
6.4.12 Never Define Variables in Header Files....14
6.4.13 Do Not Reuse Local Variables....14
6.5 Expressions and Statements....14
6.5.1 Do Not Depend on Order of Evaluation....14
6.5.2 Avoid Side Effects in Assertions....15
6.5.3 Do Not Cast Away Volatile Qualification....15
6.5.4 Do Not Reference Uninitialized Memory....15
6.5.5 Ensure NULL Pointers are not Dereferenced....15

6.5.6 Do Not Return Pointers to Local Storage....15
6.5.7 Pointer Alignment Conversion....15
6.5.8 Offsetof and Bit Fields....16
6.5.9 Use Sizeof to Determine the Size of Types or Variables....16
6.5.10 Typecasting....16
6.5.11 Do Not Access Freed Memory....16
6.5.12 Free Dynamically Allocated Memory Only Once....16
6.5.13 Check For Memory Allocation Errors....16
6.5.14 Use Correct Syntax for Flexible Arrays....17
6.5.15 Only Free Dynamically Allocated Memory....17
6.5.16 Allocate Sufficient Memory....17
6.6 Integer Expressions....18
6.6.1 Wrapping Of Unsigned Integer Variables....18
6.6.2 Loss Of Resolution....18
6.6.3 Operations on Signed Integers Should Not Overflow....18
6.6.4 Divide By Zero....19
6.6.5 Shifting Invalid Number of Bits....19
6.6.6 Evaluate Integers in a Larger Size....19
6.7 Floating Point Expressions....19
6.7.1 Floating Point Variables As Loop Counters....19
6.7.2 Calling Real Value Functions With Complex Values....20
6.7.3 Domain And Range Errors In Math Functions....20
6.7.4 Convert Integer To Floating Point For Operations....20
6.7.5 Floating Point Conversions Should Stay In Range....20
6.8 Arrays....20
6.8.1 Array Indices Are Within Valid Range....20
6.8.2 Array Notation....20
6.8.3 Size Argument For Variable Length Arrays....21
6.8.4 Storage For Copies Of Arrays....21
6.8.5 Compatible Array Types In Expressions....21
6.8.6 Iteration Beyond The End Of An Array....21
6.8.7 Pointer Arithmetic On Pointers To Arrays....22
6.9 Characters and Strings....22
6.9.1 Modifying String Literals....22
6.9.2 Sufficient Space For Storage Of Strings....22
6.9.3 NULL Termination Of Strings....22
6.9.4 Size Wide Character Strings Correctly....23
6.9.5 Casting Characters To Integers....23
6.9.6 Coping Unbounded Data To Fixed Length Arrays....23
6.9.7 Bounds For Character Arrays Initialized With Literal Strings....23
6.9.8 Character Arguments Should Be Unsigned Char....24
6.10 Naming Conventions....24
6.11 Functions....25
6.12 Complexity and Performance....25
7 Compiling Code....26
7.1 Using Make or Using IDE Project Building....26
7.2 Compiler Options....26
7.3 Debug and Release Versions....26
8 Code Checking and Verification Tools....26

9 Release Notes and Readme Files....27
10 Feature Addition and Optional Features....27
11 Using Non-Standard Compiler Features....27

1 Introduction

The goal of these guidelines is to increase productivity, decrease defects, increase portability, ease maintenance and improve clarity of code that is being developed as well as code that is being maintained.

Conforming to the coding guidelines would enable the developers to code in a uniform manner which would help maintainers, testers and reviewers to perform their task easily and faster.

Source-code clarity is enhanced by structured coding techniques, good coding style, appropriate supporting documents, good code commenting practices and by the use of tools and features provided for modern programming languages.

Also, each team member must understand the objectives of implementation. During implementation it should be kept in mind that the programs should not be constructed so that they are easy to write, but that they are easy to read and understand.

2 Purpose

The purpose of this document is to describe coding guidelines to ensure uniformity amongst the developer code so that when the code is released to a customer it has a uniform look and feels like the software has been developed by a single person.

3 Scope and Responsibility

This document covers the coding guidelines that need to be followed while writing a code.

It is assumed that these Coding Guidelines are to be used in the context of writing programs in the 'C' Programming Language using a Standards Compliant ANSI C Compiler (1999 Standard C) or later.

It is the responsibility of the developers and testers to maintain this uniformity by strictly adhering to these coding guidelines.

4 References

- ANSI Standard for C
- This document borrows most of the Rule definitions from the GNU Coding Standards available at http://www.gnu.org/prep/standards/standards.html
- CERT Secure Coding Standards available at http://www.securecoding.cert.org/
- Valgrind available at http://valgrind.org/
- dmalloc available at http://dmalloc.com/
- SPLINT available at http://www.splint.org/
- doxygen available at http://www.doxygen.org/
- GEMS-GDL-CodingGuidelinesSummary is a companion spreadsheet to this Coding Guideline that summarizes the Style, Rules and Recommendations present in this document. Additionally, this summary document contains additional Rules and Recommendations that have not been considered as Rules or Guidelines in this Coding Guidelines document.

5 Coding Standards

Coding standards are specifications for a preferred coding style.

The following conditions are necessary to obtain voluntary adherence to programming guidelines and to ensure that the guidelines are in fact followed.

- The programmer must understand the value of programming guidelines
- Programmers must have the opportunity to participate in establishing the guidelines
- The guidelines must be subject to review and revision when they become burdensome
- There must be a mechanism for allowing violations of the guidelines in special circumstances
- Automated tools must be used to check adherence to the guidelines
- Programming guidelines should be evolved for all the languages in order to enhance the quality of the software

6 Coding Style

Coding Style covers a set of guidelines that are to be followed while writing code. These are not generally not Programming Language related and are more to do with the aesthetics of the code. This may also cover a few rules or guidelines that make the code easier to read. This section lists such guidelines for the “style” of coding that is to be used in projects undertaken at GlobalEdge. It ends with a section giving a set of options that can be given to the GNU “indent” program to generate code that closely resembles the coding guidelines described here.

6.1 Source File Organization

A file consists of various sections that should be separated by blank lines and optionally comment blocks.

A limit on the file size is left for the programmer to decide as most modern computers, editors and compilers are capable of efficiently handling files of any size. The only place where small files are required is when writing libraries where each file would normally consist of a single function or a group of functions that are interdependent.

The suggested order of sections for a program file are as follows:

1. Prologue

   The file starts with a prologue that provides information related to the file. This information will include:

   a) Name
   b) Author
   c) Date of creation of the file
   d) A few lines describing the purpose of the file.
   e) Copyright notice
   f) The the file is the start of a program (contains the main() function), then a brief description of what the program does.
   g) Modification history in the form of Date, Authors Name or Initials and a

brief description of the change

2. System Header Files
3. Project Specific Header Files
4. Declaration of any defined constants, enumerations and structures and type definitions are to be placed next. It is recommended that these definitions be placed in a header file
5. Declaration of any Global variables, if any, along with a comment describing the purpose and use of the variable along with any constraints in using it.
6. Declaration of any Module variables (static), if any, along with a comment describing the purpose and use of the variable along with any constrains in using it.
7. Function prototype declarations of local functions (static) that are referenced within the file
8. External definitions, if any. It is preferred to place external definitions in a header file and include the header file.
9. The functions come last and should be in some sort of meaningful order. Considerable judgment is called for here.

6.2 Formatting, Indentation and Layout

This section lists out the code formatting guidelines that are to be followed.

6.2.1 Open Brace at Start of Function

Open braces that start the body of a function should be placed in column 1.

6.2.2 Open Braces Inside Function

Do not put open-brace, open-parenthesis or open bracket in column one when they are inside a function.

6.2.3 Break Long Function Definitions

If a function definition does not fit on a single line break it up nicely onto multiple lines

6.2.4 Recommended Style for Function Definition

The following is the recommended style for the definition of a function:

Standard function definition.

```
static char *concat(char *s1, char *s2)
{
    ...
}
```

If the arguments don't fit nicely on one line, split it like this:

```
int lots_of_args(int an_integer, long a_long,
                 short a_short, double a_double,
                 float a_float)
```

6.2.5 Recommended Style for Function Body

The following is the recommend style for the body of a function:

```
{
        if (x < foo(y, z)) {
             haha = bar[4] + 5;
        } else {
             while (z) {
                  haha += foo(z, z);
                  z--;
             }

             return ++x + bar();
        }
   }
```

Note the space after the “if” and the “while” as well as the open flower-brackets being located on the same line as the “if” and “while”.

6.2.6 Space After Commas

It is recommended to insert a space after commas when used between parameters to functions, in function prototypes and in other expressions.

6.2.7 Split Expressions Before Operator

When splitting an expression into multiple lines split before the operator.

For example:

```
if (foo_this_is_long && bar > win(x, y, z)
          && remaining_condition)
```

6.2.8 Extra Parenthesis For Indentation

Insert extra parenthesis to help editors and code formatting tools perform better indentation

6.2.9 Extra Parenthesis To Remove Ambiguity

Insert extra parenthesis to remove ambiguity in code

6.2.10 Do-While Loop Example

The following is the recommended style for programming a do-while loop:

```
do {
     ...
} while (condition);
```

6.2.11 Switch Statements

In switch statements, make sure that every case ends with either a break, continue, return, or /* FALL THROUGH */ comment. Don't forget to put a break on the last case of a switch statement. Someone will forget to add one when adding new cases.

```
switch (phase) {
case New:
     printf("don't do any coding tonight\n");
     break;
case Full:
     printf("beware lycanthropes\n");
     break;
case Waxing:
     printf(“Waxing “);
     /* FALL THROUGH */
case Waning:
```

```
          printf("the heavens are neutral\n");
          break;
     default:
          /*
           * Include occasional sanity checks in your code.
           */
          fprintf(stderr, "and here you thought this couldn't
     happen!\n");
          abort();
     }
```

6.2.12 Structure or Variable Declarations and Typedefs

Don't declare both a structure tag and variables or typedefs in the same declaration. Instead, declare the structure tag separately and then use it to declare the variables or typedefs.

6.2.13 Functions Should Be Short

Functions should be short and sweet. Don't be afraid to break functions down into smaller helper functions. This will go a long way in sharing code and help in making sure that there is less chance of an error. There are a few cases where large functions would be necessary – as in the case of a large switch case statement. Even for these cases if an alternate method is possible that will make the function smaller then go with the alternate.

6.2.14 Tabs For Indentation

Tabs will be used for indentation in all source code. -The width of the tab can be either 4 or 8. This selection needs to be done at the start of the project and all code developed for a given project should use the same tab size.

Selecting a tabs size of 8 spaces has the following advantages:

1. It doesn't require a fancy editor which automates indentation
2. It encourages you to break deeply nested code into functions
3. If you use short names and write simple code, your horizontal space goes a long way even with 8 space tab indenting.

6.2.15 Line Length Limit

A single line of code should not exceed a fixed limit. This limit can be selected on a per project basis. The default line size limit is 80.

6.2.16 Indent Options

The following set of options can be used to generate a coding style that is mostly compatible with the Coding Guidelines mentioned in this document. These options can be placed in a file “.indent.pro” in the home directory of a Unix / Linux machine to make it easy to run the indent command with this set of options.

```
-nbad -bap -bbb -nbc -bbo
-br -brs -ndj -nbs -ncdb -cdw
-ce -cp1 -ncs -nfc1 -nfca -hnl
-l80 -ts4 -i4 -ip4 -di8 -ci8 -cli4 -nlp
-psl -saf -sai -saw -nsc -sob -npcs -nprs -nbfda
```

6.3 Commenting and Inline Code Documentation

Some amount of internal documentation in the code can be extremely useful in enhancing the understandability of programs. Internal documentation consists of comments inserted in the code at various points. These include the prologue at the start of the file, comments for the global and static variable being defined and comments that document function usage and return values.

Additional comments maybe added within the body of functions to explain non-trivial concepts being implemented in the code.

It is recommended that the 'doxygen' commenting style be followed for all projects.

All comments are to be written in English.

The following code documentation MUST be present in each module:

- File / Module documentation indicating the name of the module and a description of its function
- Copyright notice for the module
- Commenting for global and static variables defined in the module
- Function documentation in 'doxygen' style before the start of each function which includes the following information. Trivial local functions may be exempt from this rule.
  - Brief description
  - Description
  - Parameter description
  - Return Values
  - See Also (optional)
  - Notes (optional)
- Additional code commenting MUST be inserted for:
  - Complex sections of code within a function
  - Code that is put in a loop
  - For large 'if' statements
  - For large 'switch' statements
  - Endif directives should have a comment unless the start and end directives are a few lines apart (within 15 to 20 lines)
  - For delays added in the code either using any of the “sleep” function calls or delay loops. This comment should indicate why the delay has been added. Ideally there should be no delays in the code.
  - Optimized code for which maintainers should take special care when modifying
  - Modifications made to code that has already been released (maintenance phase) should have the initials of the author, the date and a brief description of the change, this may even include a bug reference number.
- In addition to the above explicit documentation guidelines the following self documenting guidelines should also be followed:
  - The file name chosen should reflect the function of the module contained within that file

- The function names chosen should reflect the task that is being performed by the function
- Parameter and variable names should reflect the use of the parameter or the variable

Refer to the Source File Organization section for more information on documentation required for various sections of a source file

Refer to the Naming Conventions section for more information on selection of function and variable names

6.4 Declarations and Initialization

The following rules are to be followed regarding writing declarations and initialization:

6.4.1 Identifier Declarations

Declare identifiers of functions and variables before using them. These declarations should go in one place at the beginning of the file.

6.4.2 Shadow Variables

Don't use local variables or parameters that shadow global variables

6.4.3 Multiple Variable Declaration Should Not Span Lines

Don't declare multiple variables in one declaration than spans lines. Start a new declaration on each line. For example, instead of this:

```
int  foo,
     bar;
```

write either this:

```
int  foo, bar;
```

or

```
int  foo;
int  bar;
```

6.4.4 Nested 'if' Statements

For an if-else statement nested in another if statement always put braces around the if-else. Thus, never write this:

```
if (foo)
     if (bar)
          win();
     else
          lose();
```

always like this:

```
if (foo) {
     if (bar)
          win();
     else
          lose();
}
```

6.4.5 Casts to Void

Do not insert casts to void. Zero without a cast is perfectly fine as a 'null' pointer constant, except when calling a 'varargs' function.

6.4.6 Storage Duration

An object has a storage duration that determines its lifetime. There are three storage durations: *static*, *automatic*, and *allocated*.

According to C99 [ISO/IEC 9899:1999]:

> *The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is* undefined*. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.*

Attempting to access an object outside of its lifetime can result in an exploitable vulnerability.

6.4.7 'Restrict' Qualified Pointers Should Not Reference Overlapping Objects

The '`restrict`' qualification requires that the pointers do not reference overlapping objects. If the objects referenced by arguments to functions overlap (meaning the objects share some common memory addresses), the behavior is undefined.

6.4.8 Volatile Storage for Non-Cacheable Memory

An object that has a `volatile`-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Asynchronous signal handling falls into this category. Without this type qualifier, unintended optimizations may occur.

The `volatile` keyword eliminates this confusion by imposing restrictions on access and caching.

6.4.9 Matching Parameter and Return Types for Function Calls

Do not invoke a function using parameter types that do not match the function definition. The functions return type should also be used when evaluating the value returned by the function.

6.4.10 Global Variables

Global variables should be avoided wherever possible. If and when used comments should be added indicating the use of the variable. As mentioned above Global variables should be declared near the beginning of the file / module.

6.4.11 Use of Constants

All constants should be hash defined and the definition should be used.

6.4.12 Never Define Variables in Header Files

Global or Static variables should never be defined in header files. This would cause the same variables to be defined twice. 'static' variables variables are more prone to error as the linker does not generate an error for these multiply defined variables. Only 'extern' declarations of variables should be present in header files.

6.4.13 Do Not Reuse Local Variables

It is common practice to use the same local variables (with names like i, j, k) over and over for different purposes within one function. Instead of doing this, it is better to declare separate local variables for each distinct purpose and give it a name which is meaningful. This not only makes programs easier to understand, it also facilitates optimization by good compilers. You can also move the declaration of each local variable into the smallest scope that includes all its uses. This makes the program even cleaner.

6.5 Expressions and Statements

The following rules are to be followed with regard to writing Expressions and Statements:

6.5.1 Do Not Depend on Order of Evaluation

Evaluation of an expression may produce side effects. At specific points during execution called sequence points, all side effects of previous evaluations have completed and no side effects of subsequent evaluations have yet taken place.

This rule means that the following statement can be used:

```
i = i + 1;
a[i] = i;
```

And the following statement cannot be used:

```
/* i is modified twice between sequence points */
i = ++i + 1;

/* i is read other than to determine the value to be stored */
a[i++] = i;
```

6.5.2 Avoid Side Effects in Assertions

Expressions used with the standard `assert` macro should not have side effects. Typically, the behavior of the `assert` macro depends on the status of the `NDEBUG` preprocessor symbol. If `NDEBUG` is undefined, the `assert` macro is defined to evaluate its expression argument and abort if the result of the expression is convertible to `false`. If `NDEBUG` is defined, `assert` is defined to be a no-op. Consequently, any side effects resulting from evaluation of the expression in the assertion are lost in non-debugging versions of the code.

6.5.3 Do Not Cast Away Volatile Qualification

Do not cast away a volatile qualification on a variable type. Casting away the volatile qualification permits the compiler to optimize away operations on the volatile type, consequently negating the use of the volatile keyword in the first place.

6.5.4 Do Not Reference Uninitialized Memory

Local, automatic variables can assume unexpected values if they are used before they are initialized. In the common case, on architectures that make use of a program stack, this value defaults to whichever values are currently stored in stack memory. While uninitialized memory often contains zeroes, this is not guaranteed. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

In most cases, compilers warn about uninitialized variables when set the work at the appropriate warning level. This warning generation should be enabled and the warnings should be resolved.

Additionally, memory allocated by functions such as `malloc()` should not be used before being initialized as its contents are indeterminate.

6.5.5 Ensure NULL Pointers are not Dereferenced

Attempting to dereference a NULL pointer results in undefined behavior, typically abnormal program termination.

6.5.6 Do Not Return Pointers to Local Storage

Returning a pointer to local storage implies that when the calling function tries to access the data storage area its storage duration would have expired. This is result in undefined behavior.

6.5.7 Pointer Alignment Conversion

Do not convert pointers into more strictly aligned pointers types. Different alignments are possible for different types of objects. If the type-checking system is overridden by an explicit cast or the pointer is converted to a void pointer (`void *`) and then to a different type, the alignment of an object may be changed. As a result, if a pointer to one object type is converted to a pointer to a different object type, the second object type must not require stricter alignment than the first.

6.5.8 Offsetof and Bit Fields

Do not call `offsetof()` on bit-field members or invalid types. Behavior is undefined when the member designator parameter of an `offsetof()` macro designates a bit field or is an invalid right operand of the '`.`'operator for the `type` parameter.

6.5.9 Use Sizeof to Determine the Size of Types or Variables

Do not hard code the size of a type into an application. Because of alignment, padding, and differences in basic types (e.g., 32-bit versus 64-bit pointers), the size of most types can vary between compilers and even versions of the same compiler. Using the `sizeof` operator to determine sizes improves the clarity of what is meant and ensures that changes between compilers or versions will not affect the code.

6.5.10 Typecasting

Be very careful when using typecasting. First check if there is an alternate way to fix the problem, by defining variables with the right type. In some cases it is also the right thing to do.

6.5.11 Do Not Access Freed Memory

Accessing memory once it is freed may corrupt the data structures used to manage the heap. References to memory that has been deallocated are referred to as *dangling pointers*. Accessing a dangling pointer can lead to security vulnerabilities.

When memory is freed, its contents may remain intact and accessible. This is because it is at the memory manager's discretion when to reallocate or recycle the freed chunk. The data at the freed location may appear valid. However, this can change unexpectedly, leading to unintended program behavior. As a result, it is necessary to guarantee that memory is not written to or read from once it is freed.

6.5.12 Free Dynamically Allocated Memory Only Once

Freeing memory multiple times has similar consequences to accessing memory after it is freed. The underlying data structures that manage the heap can become corrupted in a way that can introduce security vulnerabilities into a program. These types of issues are referred to as double-free vulnerabilities.

To eliminate double-free vulnerabilities, it is necessary to guarantee that dynamic memory is freed exactly one time. Programmers should be wary when freeing memory in a loop or conditional statement; if coded incorrectly, these constructs can lead to double-free vulnerabilities. It is also a common error to misuse the `realloc()` function in a manner that results in double-free vulnerabilities.

6.5.13 Check For Memory Allocation Errors

The return values for memory allocation routines indicate the failure or success of the allocation. Failure to detect and properly handle memory management errors can lead to unpredictable and unintended program behavior. As a result, it is necessary to check the final status of memory management routines and handle errors appropriately.

6.5.14 Use Correct Syntax for Flexible Arrays

Flexible array members are a special type of array where the last element of a structure with more than one named member has an incomplete array type; that is, the size of the array is not specified explicitly within the structure. A variety of different syntaxes have been used for declaring flexible array members.

This is the syntax used by ISO C89:

```
struct flexArrayStruct {
     int num;
     int data[1];
};
```

This is the syntax used by ISO C99:

```
struct flexArrayStruct {
     int num;
     int data[];
};
```

The C89 example may be the only alternative for compilers that do not yet implement the C99 syntax. As an example Microsoft Visual Studio 2005 does not implement the C99 syntax.

6.5.15 Only Free Dynamically Allocated Memory

Freeing memory that is not allocated dynamically can lead to serious errors. The specific consequences of this error depend on the compiler, but they range from nothing to abnormal program termination. Regardless of the compiler, avoid calling `free()` on anything other than a pointer returned by a dynamic-memory allocation function, such as `malloc()`, `calloc()`, or `realloc()`.

A similar situation arises when `realloc()` is supplied a pointer to non-dynamically allocated memory. The `realloc()` function is used to resize a block of dynamic memory. If `realloc()` is supplied a pointer to memory not allocated by a memory allocation function, such as `malloc()`, the program may terminate abnormally.

6.5.16 Allocate Sufficient Memory

Integer values used as a size argument to `malloc()`, `calloc()`, or `realloc()` must be valid and large enough to contain the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer. The programmer must ensure that size arguments to memory allocation functions allocate sufficient memory.

6.6 Integer Expressions

The following rules are to be followed when dealing with Integer Expressions:

6.6.1 Wrapping Of Unsigned Integer Variables

A computation involving unsigned operands should never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

This behavior is more informally referred to as unsigned integer *wrapping*. Unsigned integer operations can wrap if the resulting value cannot be represented by the underlying representation of the integer.

6.6.2 Loss Of Resolution

Integer conversions, including implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data. This is particularly true for integer values that originate from untrusted sources and are used in any of the following ways:

- as an array index
- in any pointer arithmetic
- as a length or size of an object
- as the bound of an array (for example, a loop counter)
- as an argument to a memory allocation function
- in security critical code

The only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness.

6.6.3 Operations on Signed Integers Should Not Overflow

Integer overflow is undefined behavior. This means that implementations have a great deal of latitude in how they deal with signed integer overflow. An implementation that defines signed integer types as being modulo, for example, does not need to detect integer overflow. Implementations may also trap on signed arithmetic overflows, or simply assume that overflows will never happen and generate object code accordingly. For these reasons, it is important to ensure that operations on signed integers do no result in a signed overflow. Of particular importance, however, are operations on signed integer values that originate from untrusted sources and are used in any of the following ways:

- as an array index
- in any pointer arithmetic
- as a length or size of an object
- as the bound of an array (for example, a loop counter)
- as an argument to a memory allocation function
- in security critical code

Most integer operations can result in overflow if the resulting value cannot be represented by the underlying representation of the integer.

6.6.4 Divide By Zero

Ensure that division and modulo operations do not result in divide-by-zero errors. Division and modulo operations are susceptible to divide-by-zero errors.

6.6.5 Shifting Invalid Number of Bits

Do not shift a negative number of bits or more bits than exist in the operand. Bitwise shifts include left-shift operations of the form:

shift-expression << *additive-expression*

and right-shift operations of the form

shift-expression >> *additive-expression*.

The integer promotions are performed on the operands, each of which has an integer type. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

In almost every case, an attempt to shift by a negative number of bits or by more bits than exist in the operand indicates a bug (logic error). This is different than overflow, where there is simply a representational deficiency.

6.6.6 Evaluate Integers in a Larger Size

Evaluate integer expressions in a larger size before comparing or assigning to that size. If an integer expression is compared to, or assigned to, a larger integer size, that integer expression should be evaluated in that larger size by explicitly casting one of the operands.

6.7 Floating Point Expressions

The following rules are to be followed when dealing with Floating Point Expressions:

6.7.1 Floating Point Variables As Loop Counters

Do not use floating point variables as loop counters. Because floating point numbers can represent fractions, it is often mistakenly assumed that they can represent any simple fraction exactly. In fact, floating point numbers are subject to precision limitations just as integers are, and binary floating point numbers cannot represent all decimal fractions exactly, even if they can be represented in a small number of decimal digits.

In addition, because floating point numbers can represent large values, it is often mistakenly assumed that they can represent all digits of those values. To gain a large dynamic range, floating point numbers maintain a fixed number of bits of precision and an exponent. Incrementing a large floating point value may not change that value within the available precision.

Different implementations have different precision limitations, and to keep code portable, floating point variables should not be used as loop counters.

6.7.2 Calling Real Value Functions With Complex Values

Do not call functions expecting real values with complex values.

6.7.3 Domain And Range Errors In Math Functions

Prevent or detect domain and range errors in math functions.

A *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined.

A *range error* occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude.

Many math errors can be prevented by carefully bound checking the arguments before calling functions and taking alternative action if the bounds are violated.

6.7.4 Convert Integer To Floating Point For Operations

Convert integers to floating point for floating point operations. Using integer arithmetic to calculate a value for assignment to a floating point variable may lead to loss of information. This can be avoided by converting one of the integers in the expression to a floating type.

When converting integers to floating point and vice versa, it is important to carry out proper range checks to avoid undefined behavior.

It may be desirable to have the operation take place as integers before the conversion (obviating the need for a `trunc()` call, for example). If that is done, it should be clearly documented to help future maintainers understand the intent of the code.

6.7.5 Floating Point Conversions Should Stay In Range

Ensure that floating point conversions are within range of the new type. If a floating point value is to be demoted to a floating point value of a smaller range and precision or to an integer type, or if an integer type is to be converted to a floating point type, it must be ensured that this value is representable by the new type.

6.8 Arrays

The following rules are to be followed when dealing with expressions that use arrays:

6.8.1 Array Indices Are Within Valid Range

Guarantee that array indices are within the valid range. Ensuring that array references are within the bounds of the array is almost entirely the responsibility of the programmer.

6.8.2 Array Notation

Use consistent array notation across all source files.

Use consistent notation to declare variables, including arrays, used in multiple files or translation units. This requirement is not always obvious because within the same file, arrays are converted to pointers when passed as arguments to functions. This means that the function prototype definitions

```
void func(char *a);
```

and

```
void func(char a[]);
```

are exactly equivalent.

However, outside of function prototypes, these notations are not equivalent if an array is declared using pointer notation in one file and array notation in a different file.

6.8.3 Size Argument For Variable Length Arrays

Ensure size arguments for variable length arrays are in a valid range.

Variable length arrays (VLAs) are essentially the same as traditional C arrays, the major difference being that they are declared with a size that is not a constant integer expression. A variable length array can be declared as follows:

```
char vla[s];
```

Where the integer s and the declaration are both evaluated at runtime. If a size argument supplied to VLA's is not a positive integer value of reasonable size, then the program may behave in an unexpected way. An attacker may be able to leverage this behavior to overwrite critical program data. The programmer must ensure that size arguments to VLAs are valid and have not been corrupted as the result of an exceptional integer condition.

6.8.4 Storage For Copies Of Arrays

Guarantee that copies are made into storage of sufficient size.

Copying data into an array that is not large enough to hold that data results in a buffer overflow. To prevent such errors, data copied to the destination array must be restricted based on the size of the destination array or, preferably, the destination array must be guaranteed to be large enough to hold the data to be copied.

6.8.5 Compatible Array Types In Expressions

Ensure that array types in expressions are compatible.

Using two or more incompatible arrays in an expression results in undefined behavior.

For two array types to be compatible, both should have compatible underlying element types, and both size specifiers should have the same constant value. If either of these properties is violated, the resulting behavior is undefined.

6.8.6 Iteration Beyond The End Of An Array

Do not allow loops to iterate beyond the end of an array.

Loops are frequently used to traverse arrays to find the position of a particular element. These loops may read or write memory as they traverse the array, or use the position of an element, once discovered, to perform a copy or similar operation. Consequently, when searching an array for a particular element, it is critical that the element be found within the bounds of the array, or that the iteration be otherwise limited, to prevent the reading or writing of data outside the bounds of the array.

6.8.7 Pointer Arithmetic On Pointers To Arrays

Do not add, subtract or compare two pointers that do not refer to the same array.

When two pointers are subtracted, both must point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. This restriction exists because pointer subtraction in C produces the number of objects between the two pointers, not the number of bytes.

Similarly, comparing pointers gives the relative positions of the pointers in term of each other. Subtracting or comparing pointers that do not refer to the same array can result in erroneous behavior.

It is acceptable to compare two member pointers within a single `struct` object, suitably cast, because any object can be treated as an array of `unsigned char`.

6.9 Characters and Strings

The following rules are to be followed when dealing with expressions that contain characters or strings:

6.9.1 Modifying String Literals

Do not attempt to modify string literals.

A string literal is a sequence of zero or more multi-byte characters enclosed in double quotes (`"xyz"`, for example). A wide string literal is the same, except prefixed by the letter 'L' (`L"xyz"`, for example).

At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a null-termination character. It is unspecified whether these arrays are distinct. The behavior is undefined if a program attempts to modify string literals but frequently results in an access violation, as string literals are typically stored in read-only memory.

Do not attempt to modify a string literal. Use a named array of characters to obtain a modifiable string.

6.9.2 Sufficient Space For Storage Of Strings

Guarantee that storage for strings has sufficient space for character data and the null terminator.

Copying data to a buffer that is not large enough to hold the data results in a buffer overflow. While not limited to null-terminated byte strings (NTBS), buffer overflows often occurs when manipulating NTBS data. To prevent such errors, limit copies either through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character.

6.9.3 NULL Termination Of Strings

Null-terminate byte strings as required.

Null-terminated byte strings (NTBS) must contain a null-termination character at or before the address of the last element of the array before they can be safely passed as arguments to standard string-handling functions, such as `strcpy()` or `strlen()`. This is because these functions, as well as other string-handling functions defined by C99, depend on the existence of a null-termination character to determine the length of a string. Similarly, NTBS must be null terminated before iterating on a character array where the termination condition of the loop depends on the existence of a null-termination character within the memory allocated for the string, as in the following example:

```
size_t    i;
char      ntbs[16];
/* ... */
for (i = 0; i < sizeof(ntbs); ++i) {
     if (ntbs[i] == '\0') break;
     /* ... */
}
```

Failure to properly terminate null-terminated byte strings can result in buffer overflows and other undefined behavior.

6.9.4 Size Wide Character Strings Correctly

Wide character strings may be improperly sized when they are mistaken for *narrow* strings or for multi-byte character strings. Incorrect string sizes can lead to buffer overflows when used, for example, to allocate an inadequately sized buffer.

6.9.5 Casting Characters To Integers

Cast characters to unsigned types before converting to larger integer sizes.

Signed character data must be converted to an unsigned type before being assigned or converted to a larger signed type. Because compilers have the latitude to define `char` to have the same range, representation, and behavior as either `signed char` or `unsigned char`, this rule should be applied to both `signed char` and (plain) `char` characters.

This rule is only applicable in cases where the character data may contain values that can be interpreted as negative values. For example, if the `char` type is represented by a two's complement 8-bit value, any character value greater than +127 is interpreted as a negative value.

6.9.6 Coping Unbounded Data To Fixed Length Arrays

Do not copy data from an unbounded source to a fixed-length array.

Functions that perform unbounded copies often rely on external input to be a reasonable size. Such assumptions may prove to be false, causing a buffer overflow to occur. For this reason, care must be taken when using functions that may perform unbounded copies.

6.9.7 Bounds For Character Arrays Initialized With Literal Strings

Do not specify the bound of a character array initialized with a string literal.

The C standard allows an array variable to be declared both with a bound index and with an initialization literal. The initialization literal also implies an array size, in the number of elements specified. For strings, the size specified by a string literal is the number of characters in the literal plus one for the terminating NULL character.

It is common for an array variable to be initialized by a string literal and declared with an explicit bound that matches the number of characters in the string literal. This is one too few characters to hold the string, because it does not account for the terminating NULL character. Such a sequence of characters has limited utility and has the potential to cause vulnerabilities if a null-terminated byte string is assumed.

A better approach is to not specify the bound of a string initialized with a string literal, as the compiler will automatically allocate sufficient space for the entire string literal, including the terminating null character.

Initializing a character array using a string literal to fit exactly without a null byte is **not** allowed in C++.

6.9.8 Character Arguments Should Be Unsigned Char

Arguments to character handling functions must be representable as an unsigned char.

The header `<ctype.h>` declares several functions useful for classifying and mapping characters. In all cases the argument is an `int`, the value of which shall be representable as an `unsigned char` or shall equal the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

Compliance with this rule is complicated by the fact that the `char` data type might, in any implementation, be signed or unsigned.

6.10 Naming Conventions

Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names. Ex _sort, search_

#define constants should be in all CAPS.

Enum constants are Capitalized or in all CAPS

Function, typedef, and variable names, as well as struct, union, and enum tag names should be in lower case.

Avoid names that differ only in case, like foo and Foo. Similarly, avoid foobar and foo_bar. The potential for confusion is considerable.

Similarly, avoid names that look like each other. On many terminals and printers, 'l', '1' and 'I' look quite similar. A variable named `l' is particularly bad because it looks so much like the constant '1'.

For Library Functions

Choose a name prefix for the library, more than two characters long. All external function and variable names should start with this prefix. In addition, there should only be one of these in any given library member. This usually means putting each one in a separate source file. An exception can be made when two external symbols are always used together, so that no reasonable program could use one without the other; then they can both go in the same file.

External symbols that are not documented entry points for the user should have names beginning with '_'. The '_' should be followed by the chosen name prefix for the library, to prevent collisions with other libraries. These can go in the same files with user entry points if you like.

6.11 Functions

Functions should be short and sweet. If a function won't fit on a single screen, it's probably too long. Don't be afraid to break functions down into smaller helper functions. If they are static to the module an optimizing compiler can inline them again, if necessary. Helper functions can also be reused by other functions.

However, sometimes it is hard to break things down. Since functions don't nest, variables have to be communicated through function arguments or global variables. Don't create huge interfaces to enable a decomposition that is just not meant to be.

Check every system call for an error return, unless you know you wish to ignore errors. Include the system error text (from `perror` or equivalent) in *every* error message resulting from a failing system call, as well as the name of the file if any and the name of the utility. Just “cannot open foo.c” or “stat failed” is not sufficient.

Check every call to `malloc` or `realloc` to see if it returned zero. Check `realloc` even if you are making the block smaller; in a system that rounds block sizes to a power of 2, `realloc` may get a different block if you ask for less space.

6.12 Complexity and Performance

There is a temptation to use all sorts of clever techniques to optimize the code. However, this optimization comes at a penalty.

Rule 1

You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

Rule 2

Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

Rule 3

Fancy algorithms are slow when 'n' is small, and 'n' is usually small. Fancy algorithms have big constants. Until you know that 'n' is frequently going to be big, don't get fancy. (Even if 'n' does get big, use Rule 2 first.)

Rule 4

Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.

Rule 5

Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

7 Compiling Code

The following sections details guidelines to be followed while compiling code using any of the Standard C Compilers. If the C Compiler that is being used is not listed below, the appropriate option needs to be enabled that provide the same functionality.

7.1 Using Make or Using IDE Project Building

The 'make' utility MUST be used when compiling code that is being compiled using any GNU Compiler Tool Chain. Even if an IDE is being used which has a Project Building facility, the option to build the code using 'make' MUST also be provided.

The 'make' utility MUST be used if the C compiler does not support an IDE with a Project Building facility.

If any other compiler is being used with an IDE, the IDE Project Builder can be used by configuring it appropriately.

Optionally, the use of the “automake” and “autoconf” tools is recommended. This will help porting the code to other target platforms easier.

7.2 Compiler Options

When compiling code compiler options that enable all warnings and strict prototyping SHOULD be used.

For the GNU GCC Compiler this is enabled using the following options

```
    -Wall -Wstrict-prototypes
```

For the MS Visual C++ Compiler these options are enabled using the GUI in the ... tab. Please select the appropriate options to enable the above mentioned functionality.

7.3 Debug and Release Versions

The compilation process SHOULD support two main modes of compilation, namely, Debug version and Release version.

There MAY be additional compilation modes supported to generate additional types of builds. These could include building with support for:

- Profiling - gprof
- Code Error Checking – lint or splint
- Memory Leak & Profiling – dmalloc or valgrind
- Proprietary Modes for testing, etc.

8 Code Checking and Verification Tools

The following tools are recommended for use with all non-trivial projects to add an extra layer of code checking and verification. These tools when used should be integrated into the build system.

- Memory Leak Testing
  - valgrind
  - dmalloc
- Code Analysis and Checking
  - splint
- Profiling
  - valgrind
  - GNU gprof

9 Release Notes and Readme Files

All releases MUST have the README and RELEASENOTES files. If the released item is an end product, then the README file may be replaced with a User Manual, in the PDF file format, containing information mentioned below explained in language that is suitable for the end user.

The README file should contain the following information:

- Module / program / product information
- Contents of the release – directory structure, files, etc.
- Special unpacking instructions if any
- Build instructions
- Installation instructions
- Usage information explaining all the options of the module / program
- Execution instructions with examples

The RELEASENOTES file should contain the following information:

- Release Version and Date of Release
- Brief description of the item being released
- For the first release of the product AND for every major release, the supported feature set of the release are to be detailed.
- For every incremental releases the following information is to be included:
  - Feature changes between the new release with the previous release
  - Bug Fixes that have gone into this release along with the bug number when available
  - Test Summary detailing the bugs that have been detected during testing
  - Known issues about the release

10 Feature Addition and Optional Features

When adding features to a released code base or adding optional features it is best to use some form of conditional compilation.

This can be either done using condition statements, as shown below,

```
#ifdef HAS_FOO
     ...
#else
     ...
#endif
```

OR

by using C style 'if' statements as shown below

```
if (HAS_FOO)
     ...
else
     ...
```

where HAS_FOO is defined to a value of either 0 or 1.

The second method is preferred as modern compilers, like the GCC, will generate the same code in both cases and the compiler is able to perform more extensive checking of all possible code paths.

While the second method does not solve all portability problems and is not always appropriate, it does save time by eliminating potential code errors at an early stage of development.

11 Using Non-Standard Compiler Features

In general, it is best not to use non-standard extensions if you can straightforwardly do without them, but to use the extensions if they are a big improvement.