# 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

# 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.

## Formatting, Indentation and Layout

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

### Open Brace at Start of Function

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

### Open Braces Inside Function

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

### Function Names Column

Function names should start on column one.

### Break Long Function Definitions

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

### Function Header

When a reader starts to read the function code, he should already have a very good idea of what it is does, how it does it, and what problems he might meet. The name of the function is probably the most succinct description of its purpose.

A function usually processes inputs to create outputs. Describing these inputs and outputs is a clear hint as to what is being done in the function. Inputs include not only parameters, but also global variables and (possibly) constants. Outputs include parameters, global variables and function returns.

### 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)

### 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”.

### 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.

### 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)

### Extra Parenthesis For Indentation

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

### Extra Parenthesis To Remove Ambiguity

Insert extra parenthesis to remove ambiguity in code

### Do-While Loop Example

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

do {

...

} while (condition);

### 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:

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();

}

### 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;

### 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.

### 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.

### 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.

### 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.

### 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.

## Arrays

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

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

## Characters and Strings

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

### 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.

### 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.

### 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.