C

Overview

This document covers coding guidelines to use when writing C code.

It is expected that code refactoring will follow this ForgeRock C coding standard, unless specific circumstances dictate how the code is to be formatted. Following the FR coding standard should guard against 'bit-rot' or source code unravelling.

Copyright notices

All new source files must begin with the same copyright notice as recommended in the Coding Style and Guidelines

The ForgeRock C coding style

This style is mainly oriented around the desire to make C code more readable. Unless otherwise stated, the Kernighan and Ritchie style (otherwise known as "Egyptian Style") reigns supreme. This basically boils down to:

```
compound-statement {
   statement-1;
   statement-2;
}
```

No other variations should be used, especially not the infinitely ugly GNU C style:

```
compound-statement
   {
       statement-1;
       statement-2;
   }
```

Summary

The ForgeRock C code style rules are based on the Coding Style and Guidelines except that I think we can safely ignore the 120 character column limit on line lengths - we have big screens.

- Source files MUST contain a valid copyright notice (see above).
- Source files MUST NOT contain tab characters - code MUST be formatted using 4 character space indents.
- Curly braces MUST ALWAYS be used with if, while, for and do/while and the opening brace must appear on the SAME LINE as the keyword EVEN IF there is only one statement in the loop body or "if" statement. This is one thing perl got right.
- Checking against NULL should ALWAYS be explicit rather than implied.
- Assignment of the null character (i.e. one byte) should always be done with '\0' as opposed to 0 (which has type int).
- Never declare more than one variable at a time, as this invites an opportunity to add a comment documenting the variable.
- When declaring pointers, the * should be moved towards the type rather than the variable.
- Macros must ALWAYS be defined in UPPERCASE.
- Always include system header files before local header files.
- Try to leave at least one blank line between declarations and code within a function, unless the function is particularly short.
- Document each and every function, giving its purpose, parameters and return value (for non void functions) just like Javadoc. Try to explain WHY the function does what it does, rather than just what it does.
- Document each and every macro, giving its purpose and parameters. More explanation is necessary with macros because there is no type information for the parameters.
- ALWAYS put spaces around binary operators, except for "->" and "."
- Do not put spaces around unary operators. sizeof is a unary operator.
- ALWAYS put space around the three parts of the conditional expression operator.
- ALWAYS separate function and macro parameters with a comma AND A SPACE.
- ALWAYS comment an intended fall through in a case statement.
- ALWAYS AVOID using variables l0 (lowercase-L-zero), l1 (lowercase-L-one), lO (lowercase-L-capital-O) etc. as these can easily be confused with the numeric constants 10, 11, etc.
- Variable names which are wordy and descriptive are ALWAYS TO BE PREFERRED to single or double letter variable names.
- Struct member names which are wordy and descriptive are ALWAYS TO BE PREFERRED to single or double letter member names.
- Single letter variable names should ONLY be used when the meaning is ABSOLUTELY OBVIOUS, e.g. "i" for an integer index is permissible. Such variables should have limited scope and not be used throughout lengthy functions.
- If you need a variable in which to store a function's result, PLEASE take a leaf out of Delphi's book and use **result** as opposed to anything else.
- ALWAYS comment #else and #endif (see "Preprocessor" below).

Details

Consider the following example for a few of the points listed above:

Bad

```
int my_func(int *p,int c)
{
        unsigned int *index = p + c - 1, entry_index=0;
        if (index) *index=entry_index;
}
```

should be written as:

Good

```
/*********************************************************************************
 * The function my_func divides by the number you first thought of, which is useful when
 * parsing HTTP header information.
 * Parameters:
 *    int_base is a pointer to a block of memory containing a series of signed integers
 *    count is the number of signed integers in the block pointed to by "int_base"
 * Return:
 *    If successful, the function returns 1, if unsuccessful, the function returns 0
 *********************************************************************************/
int my_func(int* int_base, int count)
{
        unsigned int  *index = int_base + count - 1;
        unsigned int   entry_index = 0;

        if (index != NULL) {
            *index = entry_index;
        }
    entry_index = sizeof(int);
    // etc. etc.
}
```

Preprocessor

A regrettable feature of the ISO C standard was that preprocessor constants are not allowed after #else and #endif. Things can quickly get confusing, for example:

```
typedef struct {
#ifdef _WIN32
    SOCKET sock;
    HANDLE pw;
    CRITICAL_SECTION lk;
    HANDLE tm;
    HANDLE tm_tick;
#else
    int sock;
    pthread_t pw; /*event loop*/
    pthread_mutex_t lk;
#ifdef __APPLE__
    pthread_t tm_th;
    pthread_mutex_t tm_lk;
    pthread_cond_t tm_cv;
#else
    timer_t tm;
#endif
#endif
}
```

Therefore #else and #endif should ALWAYS be commented as follows:

```
typedef struct {
#ifdef _WIN32
    SOCKET sock;
    HANDLE pw;
    CRITICAL_SECTION lk;
    HANDLE tm;
    HANDLE tm_tick;
#else /* _WIN32 */
    int sock;
    pthread_t pw; /*event loop*/
    pthread_mutex_t lk;
#ifdef __APPLE__
    pthread_t tm_th;
    pthread_mutex_t tm_lk;
    pthread_cond_t tm_cv;
#else /* __APPLE__ */
    timer_t tm;
#endif /* __APPLE__ */
#endif /* _WIN32 */
}
```

Use of the `defined` operator should be considered over the #ifdef and #ifndef directives, because `defined` allows for combinations. This gives a syntax error:

```
#ifdef _WIN32 && !SOLARIS
```

whereas this works as expected:

```
#if defined(_WIN32) && !defined(SOLARIS)
```

Using Doxygen

You could consider using Doxygen, http://www.stack.nl/~dimitri/doxygen/ which allows documentation to be generated from annotated source code, in which case the above would become a much more familiar:

```
/*****************************************************************************
 * The function my_func divides by the number you first thought of, which is useful when
 * parsing HTTP header information.
 * @param int_base is a pointer to a block of memory containing a series of signed integers
 * @param count the number of signed integers in the block pointed to by "int_base"
 * @return If successful, the function returns 1, if unsuccessful, the function returns 0
 *****************************************************************************/
int my_func(int* int_base, int count)
{

    unsigned int  *index = int_base + count - 1;
        unsigned int   entry_index = 0;

        if (index != NULL) {
            *index = entry_index;
        }
    entry_index = sizeof(int);
    // etc. etc.
}
```