



C/C++ Coding Guidelines

Author(s):	Tom Goodale
Document Filename:	C/C++ Coding Guidelines
Work package:	Technical Board
Partner(s):	ALL
Lead Partner:	AEI
Config ID:	
Document classification:	Internal, Informational

Abstract: This Documents presents the C and C++ coding conventions as used by the GridLab project. It is accompanied by similar documents for Java and Perl, and by guidelines for Makefiles, configure scripts and L^AT_EX documents. They should be followed by **all** partners and are subject for reviews by the Quality Assurance manager of the project.





Contents

1 C and C++	2
1.1 File Organization	2
1.2 File Names	2
1.3 Source Files	2
1.4 Header Files	3
1.5 Indentation	4
1.5.1 Line Length	4
1.5.2 Wrapping Lines	4
1.6 Comments	6
1.6.1 Implementation Comment Formats	6
1.6.2 Documentation Comments	7
1.7 Declarations	8
1.7.1 Number Per Line	8
1.7.2 Initialisation	8
1.7.3 Placement	8
1.7.4 Function Declarations	9
1.8 Statements	10
1.8.1 Simple Statements	10
1.8.2 Compound Statements	10
1.8.3 return Statements	10
1.8.4 if, if-else, if else-if else Statements	10
1.8.5 for Statements	11
1.8.6 while Statements	11
1.8.7 do-while Statements	12
1.8.8 switch Statements	12
1.9 White Space	12
1.9.1 Blank Lines	12
1.9.2 Blank Spaces	13
1.10 Naming Conventions	13
1.11 Programming Practices	14
1.11.1 Adherence to Standards	14
1.11.2 Use typedef in preference to #defines	14
1.11.3 Use of const qualifier	14
1.11.4 Global and Static variables	14
1.11.5 Constants	15
1.11.6 Variable Assignments	15
1.11.7 Miscellaneous Practices	15



1 C and C++

These coding conventions are adapted with permission from the Cactus Coding conventions. See <http://www.cactuscode.org/>. These have been formatted in the same way as the Java coding guidelines for consistency and some of those conventions adopted where there was no conflict.

1.1 File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

1.2 File Names

File names should be all lowercase, with the extension `.h` for include files, `.c` for C source files, and `.cpp` for C++ source files. The file names should reflect the functionality defined/implemented in that file. Files with logical connection (e.g. pairs of header and source files) should reflect that connection in their names wherever possible. Files belonging to the same module should reflect that dependency by a short unique prefix to the filename, followed by an underscore. Following examples illustrate these conventions:

<code>GNUmakefile</code>	<code>io_base.cpp</code>
<code>server.c</code>	<code>io_base.h</code>
<code>server_ioplug.h</code>	<code>io_file.h</code>
<code>server_ioplug.c</code>	<code>io_file.cpp</code>
<code>io_cwrapper.h</code>	<code>io_stream.h</code>
<code>io_cwrapper.c</code>	<code>io_stream.cpp</code>

1.3 Source Files

C and C++ source files have the following ordering:

- Beginning comments (see “Beginning Comments” on 3)
- cvs version information
- `#include` statements
- `#defines`.
- local data type definitions.
- local (static) function prototypes.
- local (static) data.
- externally visible functions.
- local (static) functions.



Beginning Comments All source files should begin with a comment that describes the file and its contents along with the date of creation, CVS version information, and a copyright statement.

```
/** @file <filename>
 * Brief description of contents of file.
 *
 * Long description
 *
 * @date <date of creation of file>
 *
 * @version <CVS $ Header $ field>
 * Copyright notice.
 */
```

Note: This is a documentation comment – see section 1.6.2 for details.

CVS Version Information After the documentation header, there should be a single line

```
static const char *rcsid = "$Header$";
```

containing the CVS version information as a static. This can be subsequently extracted from the object file if there is doubt as to the version of the source code which was compiled.

1.4 Header Files

All header files should begin with a comment that describes the file and its contents along with the date of creation, CVS version information, and a copyright statement.

```
/** @file <filename>
 * Brief description of contents of file.
 *
 * Long description
 *
 * @date <date of creation of file>
 *
 * @version <CVS $ Header $ field>
 * Copyright notice.
 */
```

Note: This is a documentation comment – see section 1.6.2 for details.

To protect against multiple inclusion of headers, the contents of a header file should be protected by a `#ifndef ... #endif` pair.

```
#ifndef _NAMEOFHEADERFILEINCAPITALS_H_
#define _NAMEOFHEADERFILEINCAPITALS_H_ 1

...body of header file...
```



```
#endif /* _NAMEOFHEADERFILEINCAPITALS_H_ */
```

Function prototypes in C header files should be protected against C++ linkage by

```
#ifdef __cplusplus
extern "C"
{
#endif

...prototypes...

#ifdef __cplusplus
}
#endif
```

As a general rule header files should not include system header files, but should rather document which system header files they require.

1.5 Indentation

Two spaces should be used as the unit of indentation. Tabs should not be used as inconsistent use of tabs and spaces leads to difficulties when using “diff” or other tools to compare files.

1.5.1 Line Length

Avoid lines longer than 80 characters, since they’re not handled well by many terminals and tools.

Note:

Examples for use in documentation should have a shorter line length-generally no more than 70 characters.

1.5.2 Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking function calls:



C/C++ CODING GUIDELINES

```
Function(longExpression1, longExpression2, longExpression3,  
        longExpression4, longExpression5);  
  
var = Function(longExpression1,  
              Function2(longExpression2,  
                        longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesised expression, which is at a higher level.

```
LongName1 = LongName2 * (LongName3 + LongName4 - LongName5)  
            + 4 * longname6; /* PREFER */  
  
LongName1 = LongName2 * (LongName3 + LongName4  
                        - LongName5) + 4 * Longname6; /* AVOID */
```

Following are two examples of indenting function declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
/* CONVENTIONAL INDENTATION */  
Function(int AnArg, double AnotherArg, char *YetAnotherArg,  
        int *AndStillAnother)  
{  
    ...  
}  
  
/*INDENT 8 SPACES TO AVOID VERY DEEP INDENTS */  
static ReallyLongFunctionName(int AnArg,  
    double anotherArg, char *YetAnotherArg,  
    int *AndStillAnother)  
{  
    ...  
}  
  
/* OR, PUT EACH ARG ON OWN LINE */  
static ReallyLongFunctionName(int AnArg,  
    double anotherArg,  
    char *YetAnotherArg,  
    int *AndStillAnother)  
{  
    ...  
}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;  
  
alpha = (aLongBooleanExpression) ? beta  
      : gamma;
```



```
alpha = (aLongBooleanExpression)
    ? beta
    : gamma;
```

1.6 Comments

Programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C, which are delimited by `/*...*/`. Documentation comments are delimited by `/**...*/`. Doc comments can be extracted to HTML files using the doxygen tool.

Implementation comments are meant for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective, to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how a corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or nonobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note:

The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

1.6.1 Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

Block Comments Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within functions. Block comments inside a function should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

See also “Documentation Comments” on page 7.



Single-Line Comments Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see section 1.6.1). A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in C code (also see "Documentation Comments" on page 7):

```
    if (condition)
    {
        /* Handle the condition. */
        ...
    }
```

Trailing Comments Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in C code:

```
    if (a == 2)
    {
        a = TRUE;           /* special case */
    }
    else
    {
        a = isPrime(a);    /* works only for odd a */
    }
```

1.6.2 Documentation Comments

For further details, see "The Doxygen Manual" which includes information on the doc comment tags (@return, @param, @see):

<http://www.doxygen.org>

Doxygen comments describe C functions, structures, enums, unions, etc. Each doc comment is set inside the comment delimiters `/**...*/`, with one comment per class, interface, or member. This comment should appear just before the declaration:

```
/**
 * The Example function provides ...
 */
static Example(void)
{ ...
```

The first line of doc comment (`/**`) is not indented relative to the surrounding block; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks).

If you need to give information that isn't appropriate for documentation, use an implementation block comment (see section 1.6.1) or single-line (see section 1.6.1) comment immediately *before* the declaration. For example, details about the implementation of a function should go in in such an implementation block comment *following* the doc comment for the function, not in the function doc comment.



1.7 Declarations

1.7.1 Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; /* indentation level */
int size; /* size of table      */
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo, fooarray[SIZE]; /* WRONG! */
```

Note:

The examples above use one space between the type and the identifier. Another acceptable alternative is to line up a group of declarations (with spaces, not tabs), e.g.:

```
int    level;          /* indentation level          */
int    size;           /* size of table            */
Object currentEntry; /* currently selected table entry */
```

1.7.2 Initialisation

Initialisation of a variable should be done in a separate statement to its declaration, but as soon after the declaration as possible. The only reason not to initialise a variable straight after it is declared is if the initial value depends on some computation occurring first.

1.7.3 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces “{“ and “}”.) Don’t wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void MyFunction(...)
{
    int int1;

    int1 = 0;          /* beginning of block */

    if (condition)
    {
        int int2;

        int2 = 0;     /* beginning of "if" block */
        ...
    }
}
```



The one exception to the rule is indices of for loops in C++, which can be declared in the for statement:

```
for (int i = 0; i < MaxLoops; i++)
{
    ...
}
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
int MyFunction(...)
{
    if (condition)
    {
        int count; /* AVOID! */
        ...
    }
    ...
}
```

1.7.4 Function Declarations

When coding C functions the following formatting rules should be followed:

- All functions should be preceded by a documentation comment describing the function, its arguments and return code(s).
- Open brace “{” appears at the beginning of the line following the declaration statement
- Closing brace “}” starts a line by itself.

```
/** The sample function.
 * This function calculates the sample thingy.
 * Some more description.
 *
 * @param i The first argument.
 * @param j the second argument.
 *
 * @return The sample value.
 */
int Sample(int i, int j)
{
    return i+j;
}
```



1.8 Statements

1.8.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++;          /* Correct */
argc--;         /* Correct */
argv++; argc--; /* AVOID! */
```

1.8.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces “{ statements }“. See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be on a line by itself following the line that begins the compound statement, indented to the same level as that line; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

1.8.3 return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return MyDisk_size();

return (size ? size : defaultSize);
```

1.8.4 if, if-else, if else-if else Statements

The if-else class of statements should have the following form:

```
if (condition)
{
    statements;
}

if (condition)
{
    statements;
}
```



```
else
{
    statements;
}

if (condition)
{
    statements;
}
else if (condition)
{
    statements;
}
else
{
    statements;
}
```

Note:

if statements always use braces {}. Avoid the following error-prone form:

```
if (condition) /*AVOID! THIS OMITTS THE BRACES {}! */
    statement;
```

1.8.5 for Statements

A for statement should have the following form:

```
for (initialization; condition; update)
{
    statements;
}
```

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

1.8.6 while Statements

A while statement should have the following form:

```
while (condition)
{
    statements;
}
```



An empty while statement should have the following form:

```
while (condition);
```

1.8.7 do-while Statements

A do-while statement should have the following form:

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

1.8.8 switch Statements

A switch statement should have the following form:

```
switch (condition)
{
    case ABC:
        statements;
        /* falls through */
    case DEF:
        statements;
        break;
    case XYZ:
        statements;
        break;
    default:
        statements;
        break;
}
```

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later the default is changed to a specific case and a new default is introduced.

1.9 White Space

1.9.1 Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file

One blank line should always be used in the following circumstances:

- Between functions



- Between the local variable declarations in a block and its first statement
- Before a block (see section 1.6.1) or single-line (see section 1.6.1) comment
- Between logical sections inside a function to improve readability

1.9.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A blank space should appear after commas in argument lists.
- All binary operators should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement (“-”) from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++)
{
    n++;
}
```

- The expressions in a for statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank space. Examples:

```
MyFunction1((char) aNum, (double) x);
MyFunction2((int) (cp + 5), ((int) (i + 3) + 1);
```

1.10 Naming Conventions

Naming conventions make programs more understandable by making them easier to read.



Externally visible functions	should have a short prefix uniquely identifying the module, followed by an underscore, followed by the rest of the identifier which should consist of words, each of which begins with a capital letter, with no underscores. E.g. <i>GAT_FindResource</i> .
Static functions	should follow the same convention as externally visible functions, but need not have the prefix.
Variable names	should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary “throw-away” variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.
#defines	should be all uppercase.
typedefs and structures	should have a short prefix, followed by underscore, followed by the rest of the identifier which should consist of words, each of which begins with a capital letter, with no underscores. E.g. <i>GAT_PropList</i> .

1.11 Programming Practices

1.11.1 Adherence to Standards

All code should adhere to the ISO C or C++ standards. The presence or absence of library functions not specified by the Posix standard on a particular platform should be detected by use of Autoconf and appropriate logic emplaced to either work around the absence or provide a good error message.

1.11.2 Use typedef in preference to #defines

New types should be introduced via typedefs rather than by #defines – these types are then visible in debuggers and the compiler can do stronger type-checking.

1.11.3 Use of const qualifier

Where possible pointers should be passed using the const qualifier. This is especially important for strings.

1.11.4 Global and Static variables

Don't make any variable global or static without good reason. Access to module level statics in other files can often be granted via a function call rather than by making the variable global.



1.11.5 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, or other numbers in loop counters.

1.11.6 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fchar = lchar = 'c'; /* AVOID! */
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if (file = fopen(...))
{
    /* AVOID! */
    ...
}
```

should be written as

```
if ((file = fopen(...)) != NULL)
{
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
d = (a = b + c) + r; /* AVOID! */
```

should be written as

```
a = b + c;
d = a + r;
```

1.11.7 Miscellaneous Practices

Parentheses It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others—you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d) /* AVOID! */
if ((a == b) && (c == d)) /* RIGHT */
```



Returning Values Try to make the structure of your program match the intent. Example:

```
if (booleanExpression)
{
    return true;
}
else
{
    return false;
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition)
{
    return x;
}
return y;
```

should be written as

```
return (condition ? x : y);
```

There should only be one return statement in a function. Multiple returns can lead to confusion when reading the code and is a frequent source of errors.

```
if(foo)
{
    return a;
}

...
lots of code
...

return b;
```

should be written as



```
if(foo)
{
    retval = a;
}
else
{
    ...
    lots of code
    ...

    retval = b;
}

return retval;
```

Expressions before ‘?’ in the Conditional Operator If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesised. Example:

```
(x >= 0) ? x : -x;
```

Special Comments Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.

Compilation with warnings enabled It is recommended that developers compile with all warnings enabled. Compiler warnings often flag dubious practices and common coding errors.