# UD2 Coding Standard

2-space indents, no tabs, camelCase variables, ProperCase function/methods, start pointers with a lower case p (eg void \*pBlockData), pointers to pointers have two p's (\*ppData). m\_ prefix for class members, s\_ prefix for statics. Structure members need not be prefixed.

Be minimal wherever possible, design for future expansion but don't code for it now.

Use classes sparingly, but where necessary.

For example, a class can be declared on the stack, a C module cannot.

Do not use C modules for small utility functionality.

Do not use a class for a large system that may require platform-specific declarations.

Generally speaking, a class is useful when operating on one element at a time, this is usually

the slowest possible technique for any problem that has millions of elements.

Treat accessing memory like it's a hard drive, the hard drive like it's the internet, and the internet like it's the post office.

Allocate memory infrequently, and deal with failures. This means allocating memory in a constuctor is unlikely to be worthwhile. See the lead about patterns for dealing with failures, generally it will have exactly one return statement, a goto for each error check, and look something like:

udResult MyAllocerator(void \*\*ppData, size\_t byteCount)

{

udResult result;

<.. Some macros at the top to check passed in parameters (not yet finalised) ..>

void \*pData = nullptr;

result = udR\_MemoryAllocationFailure;

pData = udAlloc(byteCount);

if (pData == nullptr)

goto epilogue;

memset(pData, 0, byteCount);

\*ppData = pData;

pData = nullptr;

result = udR\_Success;

epilogue:

udFree(pData);

return result;

}

Prefer udAllocType to udAllocNew, and udAlloc only when allocating a number of bytes. In other words, Use udAlloc when you have an amount of memory to allocate, use udAllocType when you have a type to allocate, or udNew if you need a constructor called.

Exceptions are not used, if you need to ask why then we probably need to sit down and have a longer chat.

Avoid dependencies unless the avoidance would be counter-productive.

STL/Boost - check with lead. Trying to avoid where possible but where memory allocation can be controlled via customer's functions it's ok, but not in interfaces, and not where it carries a lot of baggage. This not only applies to STL/Boost, but also any 3rd party libraries. Once a dependency is accepted it is unlikely to ever be removed, the company’s previous efforts boot-strapped on the GT library are an excellent example of this.

Keep the code compact, with white-space used to delineate groups of functionally related lines of code. Spaces after key words, not after functions, and no braces on single list ifs, this generates too much wasted space in functions leading to relatively small functions now occupying more than a page. Try to avoid functions using more than a couple of pages.

if (condition)

DoSomething();

Braces for single-line if/else's when the either the if or the else requires them

if (condition)

{

DoSomething();

}

else

{

DoOneThing();

DoAnotherThing();

}

Keep the while from a do-while with its closing brace so it doesn't look like a while.

do

{

DoSomething();

} while (condition);

Space after a comma, not around parentheses.

// FunctionCall(a, b);

The asterisk/ampersand to denote a pointer/reference goes with the variable, not the type, no matter how much we might prefer the language creators chose differently. Do not add spaces after the asterisk/ampersand before the identifier.

void \*pMyData, \*\*ppMyData;

void func(const BigStructure &myReference);

Avoid non-const references unless there's a good reason to use them.

Delineate the start of a function/method with a row of stars (76 to be consistent) if it's an API function or dashes if it's an internal function. Follow the row of stars with A comment indicating the author, and the month/year.

Try to match the example exactly so that if we need to improve on this we can use automated search/replace.This means the date is separated from the author with a comma, and the date is exactly month-name followed by year.

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Author: Dave Pevreal, March 2014

void PublicAPIFunction()

{

DoSometingInternal();

}

// --------------------------------------------------------

// Author: Dave Pevreal, March 2014

// Internal functions have a description in the source file

// whereas external API functions have a description in the header.

void DoSomethingInternal();

When typedefing to simplify pointers to functions, leave the asterisk out so that the declaration still looks like a pointer when it's actually declared. Eg:

typedef void (CallbackFunc)(int a, int b);

void SetCallback(CallbackFunc \*funcPointer);

Forward declare functions for additional safety, for example

static CallbackFunc MyFunction;

void MyFunction(int a, int b)

{

return a \* b;

}

Communicate regularly and bounce proposed interface ideas off other team members. Code is easily rewritten, interfaces are contracts, and sometimes the product can be bound for life to them. Spending an extra day making sure the interface is perfect pays massive dividends. Buggy code behind a great interface is a trivial problem to fix.

The end.