Robust Services Core

Coding Guidelines

Version 1.4

September 16, 2017

CONTENTS

FORMATTING

1. Begin each file with the following heading:
```
//==============================================================================
//
//  <FileName>
//
//  Copyright (C) 201n-201n <Name>.  All rights reserved.
//
```
2. Use spaces instead of tabs.
3. Indent a multiple of 3 spaces.
4. Remove unnecessary interior spaces and semicolons. Remove trailing spaces.
5. Use `//` comments instead of `/*`...`*/`.
6. Add blank lines for readability, but avoid multiple blank lines.
7. Limit lines to 80 characters in length. When breaking at punctuation, break after `),` and before `:(`. When breaking at an operator, the break can occur before or after, depending on what reads better.
8. Almost always use Camel case. Use uppercase and underscores only in low-level types and constants. Names that evoke Hungarian notation are an abomination.
9. Keep `*` and `&` with the type instead of the variable (`Type* t` instead of `Type *t`).

INTERFACES

1. Insert an `#include` guard based on the file name (`filename.ext` and `FILENAME_EXT_INCLUDED`) immediately after the standard heading.
2. Sort `#include` statements as follows:
   a. the header that defines the base class of the class defined in the header
   b. C++/C library headers, in alphabetical order
   c. other headers, in alphabetical order
3. Remove an `#include` solely associated with functions inherited from a base class.
4. Remove an `#include` by forward declaring a class that is only named in references or pointers. Use an explicit forward declaration instead of relying on this as a side effect of a friend declaration.
5. Remove `using` declarations and directives. Prefix the namespace directly (i.e. `std::`<symbol>).
6. Initialize global data (static members) in the .cpp if possible.

IMPLEMENTATIONS

1. Order `#include` statements as follows:
   a. the header that defines the functions being implemented
   b. C++/C library headers, in alphabetical order
   c. other headers, in alphabetical order
2. Omit any `#include` or `using` that is already in the header.
3. Put all of the code in the same namespace as the class defined in the header.
4. Implement functions alphabetically, after the constructor(s) and destructor.
5. Separate functions in the same class with a `//----`... that is 80 characters long.
6. Separate private classes (local to the .cpp) with a `//====`... that is 80 characters long.
7. Add a blank line after the `fn_name` that defines a function's name for `Debug::ft`.
8. Name constructors and destructors "<class>`.ctor`" and "<class>`.dtor`" for `Debug::ft`.
9. Fix all compiler warnings.

CLASSES

1. Give a class its own .h and .cpp unless it is trivial, closely related to others, or private to an implementation.
2. A base class should be abstract. Its constructor should therefore be *protected*.
3. Tag a constructor as explicit if it can be invoked with one argument.
4. Make each public function non-virtual, with a one-line invocation of a virtual function if necessary.
5. Make each virtual function private if possible, or protected if derived classes may need to invoke it.
6. Make a base class destructor
   a. virtual and public
   b. non-virtual and protected
   c. virtual and protected, to restrict deletion
7. If a destructor frees a resource, even automatically through a `unique_ptr` member, also define
   a. a copy constructor: `Class(const Class& that);`
   b. a copy assignment operator: `Class& operator=(const Class& that);`

   If the class allows copying, also define
   c. a move constructor: `Class(Class&& that);`
   d. a move assignment operator: `Class& operator=(Class&& that);`
   - In C++11, each of the above function can be suffixed with *delete* to prohibit its use, or *default* to use the compiler-generated default. The pre-C++11 equivalents are to make the function private (*delete*) or not declare it at all (*default*).
   - In a copy assignment operator, create copies of `that`'s resources first, then release `this`'s existing resources, and finally assign the new ones.
   - A "move" function is an optimization that releases the existing resources and then takes over the ones owned by `that`. Its implementation typically uses `std::swap`.
8. To prohibit stack allocation, make constructors private, and/or make the destructor private.
9. To prohibit scalar heap allocation, define *operator new* as private.
10. To prohibit vector heap allocation, define *operator new[]* as private.
11. If a class only has *static* members, convert it to a namespace. If this is not possible, prohibit its creation.
12. Include *virtual* and *override* when overriding a function defined in a base class.
13. Make a function or argument *const* when appropriate.
14. Remove *inline* as a keyword.
15. Avoid *friend* where possible.
16. Override `Display` if a class has data.
17. Override `Patch` except in a trivial leaf class.
18. Avoid invoking virtual functions in the same class hierarchy within constructors and destructors. Provide an implementation for a pure virtual function to highlight the bug of calling it too early during construction or too late during destruction.
19. If a class is large, consider using the PIMPL idiom to move its private members to the .cpp.
20. When only a subset of a class's data should be write-protected, split it into a pair of collaborating classes that use `MemProt` and `MemDyn` (or `MemFirm` and `MemPerm`).
21. Static member data begins with an uppercase letter and ends with an underscore, which may be omitted if it is not returned by a "`Get`" function. Non-static member data begins with a lowercase letter and ends with an underscore, which may be omitted if the field is often used externally, as in a struct.

FUNCTIONS

1. Use the initialization list for constructors. Initialize members in the order that the class declared them.
2. Use `()` instead of `(void)` for an empty argument list.
3. Name each argument. Use the same name in the interface and the implementation.
4. Make the invocation of `Debug::ft` the first line in a function body, and follow it with a blank line.
5. The `fn_name` passed to `Debug::ft` and similar functions should accurately reflect the name of the invoking function.
6. Trace a “`Get`” function only if it is virtual.
7. Left-align the types *and* variable names in a long declaration list.
8. Use `nullptr` instead of `NULL`.
9. Check for `nullptr`, even when an argument is passed by reference. (A reference merely *documents* that `nullptr` is an invalid input.)
10. After invoking `delete`, set a pointer to `nullptr`.
11. Use `unique_ptr` to avoid the need for `delete`.
12. Use `unique_ptr` so that a resource owned by a stack variable will be freed if an exception occurs.
13. Declare loop variables inline (e.g. `for auto i =`).
14. Declare variables of limited scope inline, as close to where they are used as is reasonable.
15. Use `auto` unless specifying the type definitely improves readability.
16. Include `{...}` in all non-trivial `if`, `for`, and `while` statements.
17. Use braces in both or neither clause of an `if-endif`, even if one clause is a single statement.
18. When there is no `else`, use braces for the statement after `if` unless it fits on the same line.
19. Define string constants in a common location to support future localization.
20. To force indentation, even in the face of automated formatting, use `{...}` between function pairs such as
    a. `EnterBlockingOperation` and `ExitBlockingOperation`
    b. `Lock` and `Unlock`
    c. `MakePreemptable` and `MakeUnpreemptable`