VTS Coding Style

Md. Mahmudul Hasan Sumon

08 August 2018

Contents

Introduction This document is intended for new developer who started to work in VTS project. It describe the coding style and basic coding algorithm used and followed in VTS firmware.

1 Compile cleanly at high warning level

1.1 Summary

Take warnings to heart: Use your compiler's highest warning level. Require clean (warning-free) builds. Understand all warnings. Eliminate warnings by changing your code, not by reducing the warning level.

1.2 Discussion

Your compiler is your friend. If it issues a warning for a certain construct, often there's a potential problem in your code. Successful builds should be silent (warning-free). If they aren't, you'll quickly get into the habit of skimming the output, and you will miss real problems. (See Item 2.) To get rid of a warning: a) understand it; and then b) rephrase your code to eliminate the warning and make it clearer to both humans and compilers that the code does what you intended. Do this even when the program seemed to run correctly in the first place. Do this even when you are positive that the warning is benign. Even benign warnings can obscure later warnings pointing to real dangers.

1.3 Examples

Below some example to illustrate some issues.

1.3.1 Manage third-party header

A library header file that you cannot change could contain a construct that causes (probably benign) warnings. Then wrap the file with your own version that includes the original header and selectively turns off the noisy warnings for that scope only, and then include your wrapper throughout the rest of your project. Example (note that the warning control syntax will vary from compiler to compiler):

```
//File: myproj/myjambda.h -- wraps Boost's lambda.hpp
// Always include this file; don't use lambda.hpp directly.
// NOTE: Our build now automatically checks "grep lambda.hpp <srcfile>".
// Boost.Lambda produces noisy compiler warnings that we know are
↪  innocuous.
// When they fix it we'll remove the pragmas below, but this header will
↪  still exist.
#pragma warning(push)
//disable for this header only
#pragma warning(disable:4512)
#pragma warning(disable:4180)
#include <boost/lambda/lambda.hpp>
#pragma warning(pop)
//restore original warning level
```

1.3.2 Unused function parameter

Check to make sure you really didn't mean to use the function parameter (e.g., it might be a placeholder for future expansion, or a required part of a standardized signature that your code has no use for). If it's not needed, simply delete the name of a function parameter.

```
//... inside a user-defined allocator that has no use for the hint...
// warning: "unused parameter 'localityHint'"
pointer allocate( sizejype numObjects, const void *localityHint = 0 )
{
        return static_cast<pointer>( mallocShared( numObjects *
         ↪  sizeof(T)));
}

// new version: eliminates warning
pointer allocate( sizejype numObjects, const void * /* localityHint */ =
↪  0 )
{
        return static_cast<pointer>( mallocShared( numObjects *
         ↪  sizeof(T)) ); }
```

1.3.3 Unused variable defined

Check to make sure you really didn't mean to reference the variable. (An RAII stack-based object often causes this warning spuriously; see Item 13.) If it's not needed, often you can silence the compiler by inserting an evaluation of the variable itself as an expression (this evaluation won't impact run-time speed).

```
// warning: "variable lock' is defined but never used"
void Fun()
{
        Lock lock;
}
// new version: probably eliminates warning
void Fun()
{
        Lock lock;
        lock;
}
```

1.3.4 Uninitialized variable used

Must initialize variable before use.

1.3.5 Missing return statement

Sometimes the compiler asks for a return statement even though your control flow can never reach the end of the function (e.g., infinite loop, throw statements, other returns). This can be a good thing, because sometimes you only think that control can't run off the end. For example, switch statements that do not have a default are not resilient to change and should have a default case that does assert(false).

```
//warning: missing "return"
int Fun( Color c )
{
        switch( c)
        {
                case Red: return 2;
                case Green: return 0;
                case Blue:
                case Black: return 1;
        }
}


// new version: eliminates warning int
Fun( Color c )
{
        switch( c)
        {
                case Red: return 2;
                case Green: return 0;
                case Blue:
                case Black: return 1;
                default:
                assert( {"should never get here!" );
                return -1;
        }
}
```

2 Use an automated build system

A one-action build process is essential. It must produce a dependable and repeatable translation of your source files into a deliverable package. There is a broad range of automated build tools available, and no excuse not to use one. We are going to use make to build entire project and kconfig to maintain configuration of the project.

2.0.1 Summary

Push the (singular) button: Use a fully automatic ("one-action") build system that builds the whole project without user intervention.

2.0.2 Discussion

We've seen organizations that neglect the "one-action" requirement. Some consider that a few mouse clicks here and there, running some utilities to register COM/CORBA servers, and copying some files by hand constitute a reasonable build process. But you don't have time and energy to waste on something a machine can do faster and better. You need a one-action build that is automated and dependable. Successful builds should be silent, warning-free. The ideal build produces no noise and only one log message: "Build succeeded." Have two build modes: Incremental and full. An incremental build rebuilds only what has changed since the last incremental or full build. Corollary: The second of two successive incremental builds should not write any output files; if it does, you probably have a dependency cycle, or your build system performs unnecessary operations (e.g., writes spurious temporary files just to discard them). A project can have different forms of full build. Consider parameterize your build by a number of essential features; likely candidates are target architecture, debug vs. release, and breadth (essential files vs. all files vs. full installer). One build setting can create the product's essential executable and libraries, another might also create ancillary files, and a full-fledged build might create an installer that comprises all your files, third-party redistributables, and installation code. As projects grow over time, so does the cost of not having an automated build. If you don't use one from the start, you will waste time and resources. Worse still, by the time the need for an automated build becomes overwhelming, you will be under more pressure than at the start of the project.

Large projects might have a "build master" whose job is to care for the build system.