StyleGuide

This page is obsolete. It is replaced by the updated HotSpot Style Guide.

Hotspot Coding Style

How will my new code best fit in with the Hotspot code base? Here are some guidelines.

The Top Ten List for Writing Good HotSpot Code

1. #Classes Encapsulate code within classes, factor the code, and make it easy to understand.
2. #Accessors Use public accessor functions for instance variables accessed outside the class.
3. #Arrays Use arrays with abstraction supporting range checks.
4. #Switches Always enumerate all cases in a switch statement or specify default case. It is ok to have an empty default with comment.
5. #Asserts Use `assert(...)`, `guarantee(...)`, `ShouldNotReachHere()`, `Unimplemented()` and comments wherever needed. (Performance is almost never a reason to omit asserts.)
6. #SimpleC Use single inheritance, no operator overloading, no C++ exception handling, and no goto statements. (There are a few uses of operator overloading, but these should be rare cases.) Be sparing with templates. Use only C++ features which will work correctly on all of our platforms.
7. #NamedCons Assign names to constant literals and use the names instead.
8. #Booleans Use `bool` for booleans (not int), use `true` & `false` (not 1 & 0); use `NULL` for pointers (not 0).
9. #Names Instance variable names start with underscore "_", classes start with upper case letter, local functions are all lower case, all must have meaningful names.
10. #Ifdefs Ifdefs should not be used to introduce platform-specific code into shared code (except for `LP64`). They must be used to manage header files, in the pattern found at the top of every source file. They should be used mainly for major build features, including `PRODUCT`, `ASSERT`, `_LP64`, `SERIALGC`, `COMPILER1`, etc.

Why Care About Style?

Some programmers seem to have lexers and even C preprocessors installed directly behind their eyeballs. The rest of us require code that is not only functionally correct but also easy to read. More than that, since there is no one style for easy-to-read code, and since a mashup of many styles is just as confusing as no style at all, it is important for coders to be conscious of the many implicit stylistic choices that historically have gone into the Hotspot code base.

Nearly all of the guidelines mentioned below have many counter-examples in the Hotspot code base. Finding a counterexample is not sufficient justification for new code to follow the counterexample as a precedent, since readers of your code will rightfully expect your code to follow the greater bulk of precedents documented here. For more on counterexamples, see the section at the bottom of this page.

When changing pre-existing code, it is reasonable to adjust it to match these conventions. Exception: If the pre-existing code clearly conforms locally to its own peculiar conventions, it is not worth reformatting the whole thing.

Whitespace

- Indentation levels are two columns.
- There is no hard line length limit.
- Tabs are not allowed in code. Set your editor accordingly. (Emacs: `(setq-default indent-tabs-mode nil)`.)
- Use braces around substatements. (Relaxable for extremely simple substatements on the same line.)
- Use good taste to break lines and align corresponding tokens on adjacent lines.
- Use spaces around operators, especially comparisons and assignments. (Relaxable for boolean expressions and high-precedence operators in classic math-style formulas.)
- Put spaces on both sides of control flow keywords `if`, `else`, `for`, `switch`, etc.
- Use extra parentheses in expressions whenever operator precedence seems doubtful. Always use parentheses in shift/mask expressions (<<, &, |, ~).
- Use more spaces and blank lines between larger constructs, such as classes or function definitions.
- If the surrounding code has any sort of vertical organization, adjust new lines horizontally to be consistent with that organization. (E.g., trailing backslashes on long macro definitions often align.)
- Otherwise, use normal conventions for whitespace in C.
- Try not to change whitespace unless it improves readability or consistency. (Different editors indent differently, and spurious indentation changes will just make integrations more difficult.)

Naming

- Type names and global names are mixed-case (`FooBar`).
- Local names (fields, variables) and function names are lower-case (`foo_bar`). (For these, avoid mixing in upper case letters.)
- Constant names in upper-case or mixed-case are tolerated, according to historical necessity.
- Constant names should follow an existing pattern, and must have a distinct appearance from other names in related APIs.
    - Inside class definitions, integer constants can be defined with "`static const`". (Historically, enums have been also been used.)
- Class and type names are noun phrases. Consider an "er" suffix for a class that represents an action.
- Getter accessor names are noun phrases, with no "`get_`" noise word. Boolean getters can also begin with "`is_`" or "`has_`".
- Setter accessor names prepend "`set_`" to the getter name.
- Other method names are verb phrases, as if commands to the receiver.
- Avoid leading underscores (as "`_oop`") except in cases required above. (Names with leading underscores can cause portability problems.)

Commenting

- Clearly comment subtle fixes.

- Clearly comment tricky classes and functions.
- If you have to choose between commenting code and writing wiki content, comment the code. Link from the wiki to the source file if it makes sense.
- As a general rule don't add bug numbers to comments (they would soon overwhelm the code). But if the bug report contains significant information that can't reasonably be added as a comment, then refer to the bug report.
- Personal names are discouraged in the source code, which is a team product.

Macros

- You can almost always use an inline function or class instead of a macro. Use a macro only when you really need it.
- Templates may be preferable to multi-line macros. (There may be subtle performance effects with templates on some platforms; revert to macros if absolutely necessary.)
- For build features such as `PRODUCT`, use `#ifdef PRODUCT` for multiple-line inclusions or exclusions.
- For short inclusions or exclusions based on build features, use macros like `PRODUCT_ONLY` and `NOT_PRODUCT`. But avoid using them with multiple-line arguments, since debuggers do not handle that well.
- Use `CATCH`, `THROW`, etc. for HotSpot-specific exception processing.

Grouping

- Group related code together, so readers can concentrate on one section of one file.
- Avoid making functions too large, more than a screenful of text; split out chunks of logic into file-local classes or static functions if needed.
- If a class `FooBar` is going to be used in more than one place, put it a file named `fooBar.hpp` and `fooBar.cpp`. If the class is a sidekick to a more important class `BazBat`, it can go in `bazBat.hpp`.
- Put a member function `FooBar::bang` into the same file that defined `FooBar`, or its associated `*.cpp` file.

Miscellaneous

- Conform new code to style conventions. Avoid unnecessary "esthetic" variations, which are distracting.
- Use the C++ RAII design pattern to manage bracketed critical sections. See class `ResourceMark` for an example.
- `+Verbose` is used to provide additional output for another flag, but does not enable output by itself.
- Do not use ints or pointers as booleans with `&&`, `||`, `if`, `while`. Instead, compare explicitly `!= 0` or `!= NULL`, etc. (See #8 above.)
- Use functions from `globalDefinitions.hpp` when performing bitwise operations on integers. Do not code directly as C operators, unless they are extremely simple. (Examples: `round_to`, `is_power_of_2`, `exact_log2`.)
- Naming JTreg tests
- More suggestions on factoring
- Test Development Guidelines

Files

- Do not put non-trivial function implementations in .hpp files. If the implementation depends on other .hpp files, put it in a .cpp or a .inline.hpp file.
- .inline.hpp files should only be included in .cpp or .inline.hpp files.
- All .cpp files include precompiled.hpp as the first include line.
- precompiled.hpp is just a build time optimization, so don't rely on it to resolve include problems.
- Keep the include lines sorted.
- Put conditional inclusions (#if ...) at the end of the include list.

Counterexamples

Occasionally a guideline mentioned here may be just out of synch with the actual Hotspot code base. That's why we're using a wiki to document the guidelines. If you find that a guideline is consistently contradicted by a large number of counterexamples, please mention it here, to assist the rest of us coders with making an informed decision about coding style. The architectural rule, of course, is "When in Rome do as the Romans". Sometimes in the suburbs of Rome the rules are a little different; these differences can be pointed out here.

There may also be corrections needed. Please correct in a cautious and incremental fashion, because other Hotspot coders have been using these guidelines for years.