Scout Coding Style

September 21, 1999

Because a lot of people write code for Scout, it is important to adhere to a consistent set of programming conventions. This document outlines Scout's coding rules. It is adapted from the style guide used by the Sprite project.

1 Naming Conventions

1. Globally visible names use capitalization, not underscores. (`thisIsAGlobalName`.)
2. For names visible within a single file only, use underscores, not capitalization. (`this_is_a_local_name`.)
3. The first component of a global procedure name identifies the module to which the procedure belongs. The rest of the name identifies the operation within that module. For example, `msgPush` is the push operation in the message module.
4. Names of `#define`d macros should be all uppercase except for macros used as functions.
5. Type names start with a capital letter, variable and function names with a lower-case letter. (`TypeName`, `objName`.)
6. Structure variables are declared as "`struct` *TypeName*"; the type that is a pointer to such a structure is named "*TypeName*."

2 Low-Level Coding Rules

1. Source code is no wider than 79 characters.
2. Indents are four spaces.
3. Code comments occupy full lines; they are not tacked to the end of lines. Code comments should be indented to the same level as the code they document. Declaration comments are side-by-side.
4. Open curly braces do not normally stand alone. Instead, they appear at the end of the preceding line. The only exception is when a controlling conditional needs to be broken into multiple lines (e.g., because of a long `if`-conditional). In such a case, the opening curly brace should stand on its own line and be indented to the same level as the outer code. Close curly braces are indented to the same level as the outer code. Always use curly braces around compound statements, even if there is only one statement in the block.

5. Continuation lines should be indented to line up with the code that they continue. The Emacs tab key in C-mode does that automatically for you. Use clean places to break lines for continuation.

6. Use macros only where strictly necessary. Inlined functions are just as fast, safer to use, and easier to debug.

7. Use inlined functions only where strictly necessary. In particular, functions expanding into non-trivial code probably should not be inlined (unless they are used at most once).

8. All functions are declared and defined with ANSI-C style prototypes.

9. Don't place revision-control strings in source files (i.e., no `rcsid` or `sccsid` strings).

3 Header Files

1. Header files are (normally) protected against multiple inclusion. A Scout infrastructure header file of name *ifile* is protected by CPP name __*ifile*_h__. Any other header file of name *rfile* is protected by CPP name *rfile*_h. For example, header files for protocol modules are protected in this manner.

2. Any declaration/definition that should be conceptually invisible to a programmer but needs to be exposed in a header-file for practical reasons (e.g., performance) is isolated in a header file that has a suffix _p (for "private"). These header files are not protected against multiple inclusion as the only place that should ever include them is the public header file.

3. Scout infrastructure header files are included as "#include <scout/*name*.h>"; (public) protocol header files are included as "#include <router/*name*.h>"; private header files are included as "#include <*name*.h>".

3.1 Example

Sample code showing the recommended formatting style is given below:

```
#include <stdio.h>

#include <scout/path.h>
#include <scout/thread.h>

#include <local.h>

/*
 * Following is a local function doing not much.
 */
static void
a_local_function (int first_arg, const char * second_arg)
{
     printf("This is a_local_function speaking.\n");
}

long
```

```
globalFunction (long anotherArg, char * string)
{
    int ch;

    if (anotherArg > 0) {
        /*
         * Replace lower-case letters and digits by
         * 'x's.
         */
        while ((ch = *string)) {
            if ((ch >= 'a' && ch <= 'a')
                || (ch >= '0' && ch <= '9'))
            {
                *string = 'x';
            }
            ++string;
        }
    }
    return 0;
}
```