Pan-STARRS IPP C Code Standards

Grant Award No. : F29601-02-1-0268
Prepared For : panstarrs team
Prepared By : Robert Lupton
Document No. : PSDC-430-004-01
Document Date : October 28, 2004
Revision : 01

DISTRIBUTION STATEMENT
Approved for Public Release – Distribution is Unlimited

©Institute for Astronomy, University of Hawaii
2680 Woodlawn Drive, Honolulu, Hawaii 96822
An Equal Opportunity/Affirmative Action Institution
## Revision History

<table>
<thead>
<tr>
<th>Revision Number</th>
<th>Release Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR.00 00</td>
<td>2004/03/01</td>
<td>First draft</td>
</tr>
<tr>
<td>00</td>
<td>2004/04/23</td>
<td>First release</td>
</tr>
<tr>
<td>01</td>
<td>2004/10/28</td>
<td>Title Changed</td>
</tr>
</tbody>
</table>
## Referenced Documents

<table>
<thead>
<tr>
<th>Pan-STARRS ID</th>
<th>Title</th>
<th>Author</th>
</tr>
</thead>
</table>


Contents

1 Introduction ........................................ 1
   1.1 Why Have Code Conventions ....................... 1
   1.2 Acknowledgments .................................. 1
   1.3 To whom should I Complain? ..................... 1

2 File Names ........................................ 1
   2.1 File Suffixes ................................... 1
   2.2 Common File Names ............................... 2

3 File Organization .................................. 2
   3.1 Source Files .................................... 2
      3.1.1 Include Files ................................. 2
      3.1.2 C Source Files ............................... 3
      3.1.3 SWIG Interface Files ........................ 3

4 Indentation ....................................... 3
   4.1 Line Length ..................................... 4
   4.2 Wrapping Lines .................................. 4

5 Comments ......................................... 5
   5.1 Implementation Comment Formats ................. 6
      5.1.1 Block Comments ............................ 6
      5.1.2 Single-Line Comments ..................... 6
      5.1.3 Trailing Comments ......................... 6
      5.1.4 End-Of-Line Comments ..................... 7
   5.2 Documentation Comments ........................ 7

6 Declarations ..................................... 7
   6.1 Number Per Line ................................ 8
   6.2 Initialization .................................. 8
   6.3 Placement ...................................... 8
   6.4 Function Declarations ........................... 9

7 Statements ....................................... 10
   7.1 Simple Statements .............................. 10
   7.2 Compound Statements ............................ 10
   7.3 return Statements ................................ 10
   7.4 if, if-else, if else-if else Statements ....... 10
   7.5 for Statements .................................. 11
   7.6 while Statements ................................ 11
   7.7 do-while Statements ............................. 12
   7.8 switch Statements ................................ 12
   7.9 label Statements ................................ 12

8 White Space ...................................... 12
8.1 Blank Lines ........................................................................................................... 12
8.2 Blank Spaces ........................................................................................................ 13

9 Naming Conventions ............................................................................................... 13

10 Programming Practices ......................................................................................... 14
  10.1 When to Make Symbols Global ......................................................................... 14
  10.2 Constants ........................................................................................................... 15
  10.3 Type Qualifiers (const and restrict) ................................................................. 15
  10.4 structs and typedefs ......................................................................................... 15
  10.5 Variable Assignments ....................................................................................... 15
  10.6 Miscellaneous Practices ................................................................................... 16
    10.6.1 Parentheses .................................................................................................. 16
    10.6.2 Returning Values ......................................................................................... 16
    10.6.3 Expressions before ‘?’ in the Conditional Operator ............................... 17
    10.6.4 Functions that Take No Parameters ......................................................... 17
    10.6.5 Special Comments ....................................................................................... 17

A Code Examples ....................................................................................................... 18
  A.1 Source File Example .......................................................................................... 18

B Departures from Sun Java Coding Standards .................................................... 20
  B.1 Additions to the Sun-Java standards ............................................................... 20
  B.2 Departures from the Sun-Java standards ....................................................... 20

C How to Achieve This Style in Emacs ................................................................... 21

D How to Achieve This Style with Astyle ............................................................... 21
1 Introduction

1.1 Why Have Code Conventions

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

1.2 Acknowledgments

This document is derived from the Sun Microsystems Java language coding standards presented in the Java Language Specification (http://java.sun.com/docs/books/jls/index.html).

Adapted with permission from
CODE CONVENTIONS FOR THE JAVA™ PROGRAMMING LANGUAGE.
Copyright 1995-1999 Sun Microsystems, Inc. All rights reserved.

See the Java Code Conventions Web site (http://java.sun.com/docs/codeconv/) for more details.
Such an adaption is explicitly permitted by the Sun Microsystems copyright notice (Copyright.doc.html).

1.3 To whom should I Complain?

While we’re working on this document complain to rhl@astro.princeton.edu.

2 File Names

This section lists commonly used file suffixes and names.

2.1 File Suffixes

Software uses the following file suffixes:
File Type       Suffix
C source        .c
header files    .h
SWIG interface files .i
Perl files      .pl
Python files    .py

2.2 Common File Names

Frequently used file names include:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
<td>The preferred name for makefiles.</td>
</tr>
<tr>
<td>README</td>
<td>The preferred name for the file that summarizes the contents of a</td>
</tr>
<tr>
<td></td>
<td>particular directory.</td>
</tr>
<tr>
<td>TAGS</td>
<td>The index file (built by etags) used by emacs to follow cross</td>
</tr>
<tr>
<td></td>
<td>references.</td>
</tr>
</tbody>
</table>

3 File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Files longer than 2000 lines are cumbersome and should be avoided.

For an example of a properly formatted program, see Source File Example (§A.1).

3.1 Source Files

3.1.1 Include Files

Include files should have the following order:

- A brief description of the functionality provided by this set of APIs. This comment should not duplicate information available elsewhere (e.g. the filename; log information available from cvs).

- A check for the existence of a CPP symbol of the form FILENAME_H (e.g. STDIO_H). This is used to ensure that an include file may safely be included more than once.

- Any #defined constants. Note: #define should be used rather than e.g. const int; C and C++ const semantics are different.

- Any enums.

- Any typedefs and structs.

- Any function prototypes.

e.g. for file psThing.h:
/** \file psThing.h
 * Support for psThings
 */

#if !defined(PS_THING_H)
define PS_THING_H

#define PS_NMAX 10 // maximum number of subthings

enum {PS_FIRST, PS_SECOND, PS_THRID}; // subthing sequences
typedef struct {
    int nthing; // number of subthings
    int sthing[PS_NMAX]; // the subthings
} psThing;

/// make a new psThing
psThing *psThingAlloc(int nthing);

/// free an existing psThing
void psThingFree(psThing *thing);
#endif

add doxygen comments to this example (TBD)

3.1.2 C Source Files

C Source files have the following ordering:

- Beginning comments, concisely describing the functionality present in the file. This comment should not duplicate information available elsewhere (e.g. the filename; log information available from cvs).

- #include statements

- #defines local to the file

- Typedefs local to the file

- Any needed prototypes for file-static functions. Prototypes need only be provided for functions used before their definition; in practice this means ones that the compiler complains about.

- File-static variable declarations

- The code that actually implements the desired functionality.

3.1.3 SWIG Interface Files

Write me (TBD)

4 Indentation

Four spaces should be used as the unit of indentation. Spaces are recommended over tabs for indentation, but if tabs are used, the tabs stops must be set to 8 spaces, not 4.
4.1 Line Length

Avoid lines longer than 110 characters.

When preparing documents, you should ensure that lines of this length are not wrapped. If you are using the standard PSDC \LaTeX{} class file \texttt{panstarrs.cls}, \begin{verbatim} ... \end{verbatim} will do this for you.

4.2 Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma or operator.

\begin{verbatim}
if(a < b || a > 2*b) {
  x = (a + b + c) + sin(z);
}
\end{verbatim}

- Prefer higher-level breaks to lower-level breaks.

- Align the new line with the beginning of the expression at the same level on the previous line.\footnote{For emacs users, this means the indentation that <tab> produces}

- If the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead.\footnote{This should be very rare! Consider whether you should be e.g. factoring code into a function.}

Here are some examples of breaking function calls:

\begin{verbatim}
someFunc(longExpression1, longExpression2, longExpression3, longExpression4, longExpression5);
var = someFunc1(longExpression1, someFunc2(longExpression2, longExpression3));
\end{verbatim}

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

\begin{verbatim}
longName1 = longName2*(longName3 + longName4 - longName5) + 4*longName6; // PREFER
longName1 = longName2*(longName3 + longName4 - longName5) + 4*longName6; // AVOID
\end{verbatim}

Following are two examples of indenting function declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 4 spaces (note that putting the \texttt{anArg} argument on a line of its own makes this the ‘natural’ emacs indentation).
void someFunc(int anArg, Object anotherArg, String yetAnotherArg, 
           Object andStillAnother)
{
    ...
}

static type honkingLongFunctionName(
    int anArg, 
    Object anotherArg, String yetAnotherArg, 
    Object andStillAnother)
{
    ...
}

Here are three acceptable ways to format ternary expressions:

alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
            : gamma;

alpha = (aLongBooleanExpression)
            ? beta
            : gamma;

5 Comments

C99 supports both types of comments found in C++, namely those delimited by /*...*/, and //.

Comments should not be used to ‘comment out code’; use #if 0 instead:

#if 0
    code that is not wanted just at present
#endif

This has two advantages:

- The code can easily be reinstated.
- Related blocks of code can be removed with

```c
#define IGNORE_RHL 1
#if !IGNORE_RHL
    ...
#endif

#define IGNORE_RHL 1
#if !IGNORE_RHL
    ...
#endif
```

This type of #define may appear within the main body of the file.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For
example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or nonobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

5.1 Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

5.1.1 Block Comments

Block comments are used to provide descriptions of files, functions, data structures and algorithms. Block comments may be used at the beginning of each file and before each function. They can also be used in other places, such as within functions. Block comments inside a function should be indented to the same level as the code they describe.

See also Documentation Comments (§5.2).

5.1.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can’t be written in a single line, it should follow the block comment format (§5.1.1).

5.1.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, and should be indented to column 40[^3]. If the code extends beyond this column, the comment should be separated from the closing semi-colon by a single space.

Here’s an example of a trailing comment in C code:

```c
if (a == 2) {
    return TRUE; /* special case */
} else {
    return isPrime(a); /* works only for odd a */
}
```

[^3]: In emacs, this may be achieved with ESC;
5.1.4 End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line; as for trailing comments, the comment should start in column 40. It shouldn’t be used on consecutive multiple lines for text comments. Examples of both styles follow:

```
if (foo > 1) {
    // Do a double-flip.
    ...
} else {
    return false;  // Explain why here.
}
```

5.2 Documentation Comments

Doxygen\(^4\) will be used to produce documentation of the types, functions and variables without requiring much extra effort for the programmer. Comments starting with particular characters (“tags”) are used by Doxygen to identify the relevant code to be documented.

Functions shall be tagged for Doxygenation by pre-pending them with a block comment (§5.1.1) which starts with a /** instead of the usual /*.

Variables shall be tagged for Doxygenation by appending their declaration with an end-of-line comment which starts with a /**/< instead of the usual //.

An example of a function definition employing Doxygen-compatible comments follows:

```
/** This is a really cool function.
 * It does many really cool things.
 */
int reallyCoolFunction(int aNumber, /**< This is a number
    float aRealNumber /**< This is a real number
    )
{
    char *aString; /**< This is a string used to do stuff.
    /* Do really cool stuff */
    ...
}
```

If you need to give information about a type, interface, or variable that isn’t appropriate for doxygenation, use a normal implementation block comment (§5.1.1) or single-line comment (§5.1.2) immediately before the code (as exampled above). For example, details about the implementation of a type should go in in such an implementation block comment before the start of the implementation, not in the doxygen comment.

6 Declarations

All globally-visible symbols must be declared in a suitable header (.h) file. Global private symbols should not appear in public header files, but rather in separate, private header files.

\(^4\)www.doxygen.org
6.1 Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```c
int level; // indentation level
int size; // size of table
```

is preferred over

```c
int level, size;
```

Do not put different types on the same line. Example:

```c
int foo, fooarray[]; // WRONG!
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```c
int level; // indentation level
int size; // size of table
Object currentEntry; // currently selected table entry
```

(Note that the comments are indented to column 40, as per section 5.1.3).

6.2 Initialization

Try to initialize local variables where they’re declared. The only reason not to initialize a variable where it’s declared is if the initial value depends on some computation occurring first.

In some cases it may be necessary to initialize a variable to suppress a compiler warning; in this case a comment should explain the circumstances.

6.3 Placement

Variables should ordinarily be declared at the top of the block in which they appear, unless there is some reason to declare them later. This allows them to be initialised as they are created, and naturally associates their declaration with their use.

For example, declarations are naturally mixed with assertions:

```c
void function(REGION *reg) // image of the sky
{
    assert (reg != NULL);
    const int nrow = reg->nrow;
    assert (reg->type == psType_U16);
    const psTypeU16 *rows = reg->rows;
    ...
}
```

Indexes of for loops should usually be declared in the for statement (the declarations appear in the scope of the braces):

```c
for (int i = 0; i < nrow; i++)
```
for (int i = 0; i < maxLoops; i++) {
    ...
}

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```c
static int count;
...
void myFunction(void)
{
    if (condition) {
        int count = 0; // AVOID!
        ...
    }
    ...
}
```

### 6.4 Function Declarations

When coding C functions the following formatting rules should be followed:

- Function declarations should be preceded by a short comment describing what the function does. These comment blocks should include doxygen-style comments to provide a brief description as well as other warnings, bugs, etc as needed.
- The function’s type should appear on the same line as the function declaration.
- All arguments should be declared with their types; i.e. no classic-C declarations like:

  ```c
  int main(ac, av)
      int ac;
      char **av;
  ```

  No space between a function name and the parenthesis ( starting its parameter list

- The function body’s open brace { should be in column 1 of the next line:

  ```c
  void function(void)
  {
      ...
  }
  ```

- Closing brace } starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the } should appear immediately after the {

  ```c
  void find_neos(const REGION *sky, const char *descr)
  {
      struct {
          int x, y;
      } work[10];
      int ivar;
      while(isspace(*descr++)) {} // skip white space
      descr--; // one too far. Not great code...
      ...
  }
  ```
7 Statements

7.1 Simple Statements

Each line should usually contain only one statement. Example:

```
x = sqrt(x2); // Correct
i++; // Correct
x = sqrt(x2); i++; // Avoid!
```

An example of a reasonably two-statement line is:

```
argv++; argc--; 
```

where the two actions are intimately related.

7.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces. See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

7.3 return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;
return myDisk.size;
return (size ? size : defaultSize);
```

7.4 if, if-else, if else-if else Statements

The if-else class of statements should have the following form:
if (condition) {
  statements;
}

if (condition) {
  statements;
} else {
  statements;
}

if (condition) {
  statements;
} else if (condition) {
  statements;
} else {
  statements;
}

Note: if statements always use braces. Avoid the following error-prone form:

if (condition) // AVOID! THIS OMITS THE BRACES {}!
  statement;

7.5 for Statements

A for statement should have the following form:

for (initialization; condition; update) {
  statements;
}

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have one of the following forms:

for (initialization; condition; update);
for (initialization; condition; update) {};

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

7.6 while Statements

A while statement should have the following form:

while (condition) {
  statements;
}

An empty while statement should have one of the following forms:

while (condition);
while (condition) {}
7.7  do-while Statements

A do-while statement should have the following form:

```c
do {
    statements;
} while (condition);
```

7.8  switch Statements

A switch statement should have the following form:

```c
switch (condition) {
    case ABC:
        statements;
        /* falls through */
    case DEF:
    case GHI:
        statements;
        break;
    case XYZ:
        statements;
        break;
    default:
        statements;
        break;
}
```

Every time a case falls through (doesn’t include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment. A comment is not required (or expected) when the fall-through is between multiple case statements.

Every switch statement should include a default case, which should come last. The break in the default case is redundant, but it prevents a fall-through error if later another case is later (and illegally) added after the default clause.

When switching on an enumerated type, if all the elements of the type are included in the switch a default clause should still be added (not all compilers diagnose missing elements). In this case, the action in the default clause should be to generate an error and abort.

7.9  label Statements

Code labels should be indented to align with the previous level of indentation.

8  White Space

8.1  Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.
Two blank lines should always be used in the following circumstances:

- Between sections of a source file

One blank line should always be used in the following circumstances:

- Between the local variables in a function and its first statement
- Between logical sections inside a function to improve readability

### 8.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

  ```
  while (true) {
      ...
  }
  ```

  Note that a blank space should not be used between a function name and its opening parenthesis. This helps to distinguish keywords from function calls.

- A blank space should appear after commas in argument lists.

- Binary operators should be separated from their operands by spaces. Blank spaces should never separate unary operators such as a type cast, unary minus, increment (++) and decrement (--) from their operands. Examples:

  ```
  a += c + d;
  a = (a + b) / (c * d);
  ```

  ```
  while (*d++ = *s++) { // Tricky way of copying until '\0'
      n++;
  }
  ```

- The expressions in a `for` statement should be separated by blank spaces. Example:

  ```
  for (expr1; expr2; expr3)
  ```

- The word `assert` should be treated as a keyword, and separated by a space from its logical expression.

### 9 Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier – for example, whether it’s a constant, a function, or a type – which can be helpful in understanding the code. Remember these are guidelines for improving readability; clarity should trump rigid adherence to the guideline.
### Identifier Type

**Typedefs**
- Type names should be nouns, in mixed case with the first letter of each internal word capitalized. Little words, such as ‘for’, ‘to’, ‘at’, etc., may be written in all lower-case or separated with underscores if it makes the name clearer.
- If and only if the type is visible at global scope, the type name should start with `ps`.
- Try to keep your type names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

**Examples**
```
typedef struct ... psRaster;
typedef struct ... psImage;
```

**Functions**
- The names of all externally visible functions (i.e. all those that are not declared `static`) should be verbal phrases, in mixed case.
- Namespaces should be protected by using special naming prefixes to restrict the name space in particular libraries. For example, the PSLib functions are all prefixed with `ps`.

**Examples**
```
psRun(int ID);
psRunFast(float velocity);
psGetBackground(void);
p_psForgeSignature(const char *name);
```

**Variables**
- Except for variables, all instance, class, and class constants are in mixed case with
- Variable names should not start with underscore `_` as these names are, under some circumstances, reserved by the C standard. Non-globally visible words should start with a lowercase first letter; Internal words start with capital letters.
- Variable names should be short yet meaningful. The choice of a variable name should be mnemonic—that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary *throwaway* variables.
- Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.

**Examples**
```
int i;
char c;
float myWidth;
int psNumOTA;
int p_psMyFiddleFactor;
```

**Constants**
- Constants used to e.g. dimension arrays should be set using the `#define; const` variables should not be used, especially in header files. Symbolic values should usually be declared using enumerated types.

**Examples**
```
#define PS_MAXLEN 40
enum { PS_ONE = 1, PS_TWO = 2 };
```

---

### 10 Programming Practices

#### 10.1 When to Make Symbols Global

Declare all functions and top-level variables `static` within a file if they are not needed outside of the file. **Note:** do not confuse this use of `static` with its usage to make auto-variables within a function persistent.

A name (whether of a variable, a function, or a type) shall start `ps` (or `p_ps`) if and only if it is visible at global scope. The distinction is that `p_ps` names are not part of the documented APIs, but need to be exposed for some reason.
10.2 Constants

Numerical constants (literals) should not be coded directly, except for small integers such as -1, 0, and 1 which are permitted to e.g. appear in a for loop as counter values.

10.3 Type Qualifiers (const and restrict)

All interfaces and type definitions should use const and restrict wherever appropriate. For example,

```c
typedef struct {
    int n;
    float *restrict vec;
} psVec;
```

```c
psVec *psVecAdd(const restrict* psVec s1, const restrict* psVec s2);
```

**Note:** compilers are free to ignore the restrict keyword, so all code should be written to explicitly handle aliasing.

10.4 structs and typedefs

All structs should be defined as typedefs.

A struct should not include a struct tag unless it’s self-referential; E.g.

```c
typedef struct myStruct { // Omit "myStruct"
    int x;
} myStruct;
```

```c
typedef struct yourStruct { // OK
    struct yourStruct *next;
    int x;
} yourStruct;
```

10.5 Variable Assignments

Avoid assigning several variables to the same value in a single statement, unless the variables are intimately related.

Acceptable examples:

```c
row0 = col0 = 0;
sum = sumx = sumy = 0;
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```c
if (c++ = d++) { // AVOID!
...
}
```

should be written as

```c
if ((c++ - d++) != 0) {
...
}
```
Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
d = (a = b + c) + r;  // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

### 10.6 Miscellaneous Practices

#### 10.6.1 Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others—you shouldn’t assume that other programmers know precedence as well as you do. In cases where the precedence rules are clear, the parentheses may be omitted.

```
if (a & b || c & d) // AVOID!
if ((a & b) || (c & d)) // RIGHT
```

There are some famous problems with precedence in C. In particular, expressions involving the combinations of `||` and `&` should be fully parenthesised, as should all expressions containing bitwise operators:

```
if ((a == b && c == d) || e == f)
l << (j + k)
(l << j) + k
(i & b) | c
```

#### 10.6.2 Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

should instead be written as

```
return booleanExpression;
```

If you’re concerned that the reader may not know that `booleanExpression` is boolean, use:

```
return (booleanExpression ? true : false);
```

Similarly,
if (condition) {
    return x;
} 
return y;

should be written as

return (condition ? x : y);

10.6.3 Expressions before ‘?’ in the Conditional Operator

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

(x >= 0) ? x : -x;

10.6.4 Functions that Take No Parameters

When a function takes no arguments, it should be explicitly declared as void:

static type myFunction(void)
{
    ...
}

10.6.5 Special Comments

Doxygen has special comments which are used to provide specific notes in the code. These are added in as special entries and sections in the Doxygen-generated documentation. These special comments should be used in addition to the Doxygen usage to make these types of conditions easily searchable.

- Use \warning in a comment to flag something that is bogus but works.
- Use \bug to flag something that is bogus and broken.
- Use \todo to note additional work to be done.
- Use \note to make other general notes.
- Use \test to list test cases.
- Use \notreached to indicate a line of code that cannot be reached, e.g.

```c
if (sqrt(x) < 0) {
    psAbort(); // never returns
    exit(1); // NOTREACHED
}
```

- Use \notused to indicate unused arguments to a function: e.g.
type psWorkHard(const Region *restrict reg, // Region to operate on
    int myConst, // magic value
    int magicNumber // NOTUSED; reserve for next version
  
  )
{
    ...
}

A  Code Examples

A.1  Source File Example

Need to include more of the rules, and shorten the example (TBD)  Here’s psBuffer.h:

```c
#if !defined(PS_BUFFER_H) /* here’s the insides of psBuffer.h */
#define PS_BUFFER_H
#define PS_BUFSIZE 128 // Size for I/O buffers
typedef struct {
    char buf[PS_BUFSIZE]; // buffer
    int n; // number of bytes in buffer
} psBuffer;

psBuffer *psBufferAlloc(void);
void psBufferFree(psBuffer *buf, // buffer to delete
                  int deep); // NOTUSED. Do a deep delete

void psBufferAppend(psBuffer *restrict buf, // psBuffer to append to
                    const char *restrict str); // string to add

#endif
```

And here’s the C source; psUtils.h provides psAlloc/psFree.

```c
/*
 * This file implements an example of formatting a file of C code
 * It isn’t a very good piece of code
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <time.h>
#include "psUtils.h"
#include "psBuffer.h"

typedef enum { CRNL, NL } NLType;

static char *getNewline(NLType type); // return a suitable newline

static psBuffer *buf = NULL; // I/O buffer

/***************************************************************************/
/* Create/destroy psBuffers */
psBuffer *psBufferAlloc(void)
{
  psBuffer *buf = psAlloc(sizeof(psBuffer));
  buf->n = 0;
```
return buf;
};

void psBufferFree(psBuffer *buf, // buffer to delete
    int deep) // NOTUSED. Do a deep delete
{
    if (buf == NULL) {
        return;
    }
    if (buf->n > 0) {
        (void)fputs(buf->buf, stdout);
    }
    psFree(buf);
}

/********************************************************************************
* Append to a string to a psBuffer
*/
void psBufferAppend(psBuffer *restrict buf, // psBuffer to append to
    const char *restrict str) // string to add
{
    assert(str != NULL);
    const int len = strlen(str);
    if (buf->n + len >= PS_BUFSIZE) { // XXX Handle this better
        fprintf(stderr, "Sorry; too many bytes. Bye bye\n");
        abort();
    } // NOTREACHED
    (void)strcat(&buf->buf[buf->n], str);
    buf->n += len;
}

/********************************************************************************
* Now do the work
*/
int main(void)
{
    buf = psBufferAlloc();
    long t; // current time
    while(t = (long)time(NULL), t % 3 == 2) {}
    if (t % 3 == 0) {
        psBufferAppend(buf, "Hello");
    } else if (t % 3 == 1) {
        psBufferAppend(buf, "Aloha");
    }
    psBufferAppend(buf, " World!");
    psBufferAppend(buf, getNewline(NL));
    psBufferFree(buf, 0);
    return 0;
}

/********************************************************************************
* Return a desired line terminator
*/
static char *getNewline(NLType type) // what sort of newline?
B Departures from Sun Java Coding Standards

Apart from changes required by our use of C99 rather than Java, this document differs from the original Sun-Java standard in two sorts of ways; additions and changes.

B.1 Additions to the Sun-Java standards

- Naming convention for include files
- Added naming conventions for constructors, destructors, and conversion functions.
- Specified that the ‘output’ argument should come first.
- Added conventional comments for unused arguments and unreachable statements (conforming to Doxygen’s conventions).
- Specified that functions taking no arguments should be explicitly specified as (void) (avoiding complaints on some compilers, e.g. on SGIs).
- Added rules on typedefing structs, and on struct tags.

B.2 Departures from the Sun-Java standards

- We break lines after (not before) an operator.
- We don’t specify any special indentation for continued logical expressions within an if clause.
- Specify that case/break statements should be indented by half an indent (2 spaces)
- Comments need not be preceded by a blank line
- The restriction on only declaring variables at the top of blocks (including for loops has been somewhat relaxed.
- Relaxed the ‘only one statement per line’ rule a little.
• High-precedence binary operators (*, /, % and above) should not be surrounded by whitespace.

• The line-length limit has been changed from 80 to 110 characters.

• Relaxed wording to allow parentheses to be omitted when the precedence is well known and unambiguous.

C How to Achieve This Style in Emacs

This section is provided for your convenience; it is, of course, not part of the coding standards.

The easiest way to use these conventions while writing code using emacs is to get panstarrs.el from cvs with the command

cvs -d poiserver0.ifa.hawaii.edu:/usr/local/cvs/repositories/pan-starrs co Templates

and then grab Templates/panstarrs.el. Then add:

(load-file "~/home/you/Templates/panstarrs.el")
(add-to-list 'auto-mode-alist (cons "\.[ch]" 'panstarrs-c-mode))

to your .emacs file; this will use panstarrs-c-mode for all .c and .h files.

If you want to

• use C99-style comments (i.e. // to end of line)

• have your emacs window set to 110 characters wide

• somewhat improve (or spoil?) the handling of re-indenting comments

add one or more of these add-hook commands to your .emacs file too:

(add-hook 'panstarrs-c-mode-hook 'panstarrs-c99-comments)
(add-hook 'panstarrs-c-mode-hook 'panstarrs-set-width)
(add-hook 'panstarrs-c-mode-hook
'(lambda ()
 (set (make-variable-buffer-local 'comment-indent-function)
 'panstarrs-comment-indent)))

D How to Achieve This Style with Astyle

The purpose of a coding standard is to improve coding efficiency, not to hinder it. Since people are inherently falible, 100% adherence to the standard is an impossible goal for human-generated code. Furthermore, laziness and slopiness are human nature. To minimize the amount of effort spent in keeping software in line with a coding standard is to use automatic re-formatting tools to enforce the standard. Various software tools exist to perform these tasks. One of these is astyle, an open-source tool which takes a variety of options which allow the use to tailor the coding standard to suit their preferences. We have determined the following collection of astyle options which achieve many of the coding standard guidelines specified above. Note that any of the listed options (always in the ’long’ form) may be specified in the user’s .astylerc file by dropping the leading dash.
**--mode=c** - This option tells `astyle` to recognize the source code as C code.

**--indent-switches** - This option tells `astyle` to indent the case statements in a switch. Do not also specify **--indent-cases**.

**--indent-labels** - This option tells `astyle` to add indentation to labels so they are indented one level less than the current level.

**--min-conditional-indent=0** - This option tells `astyle` not to add indentation to successive lines of multiple-line conditional statements.

**--max-instatement-indent=20** - This option tells `astyle` to limit the total amount of indentation to 20 spaces.

**--pad=oper** - This option tells `astyle` to add padding about binary operators.

**--brackets=break** - This option tells `astyle` to break brackets from their pre-block statements, dropping them to the next line.

**--convert-tabs** - This option tells `astyle` to convert tabs into the equivalent number of space characters.

**--indent=spaces=4** - This option tells `astyle` to use 4 spaces per indent level.