parc
A Xerox Company

Networking and Distributed Systems
Computing Science Laboratory

PARC C Style Guide

Glenn Scott[1]

Abstract
Like a book, magazine article, or technical paper, software is typically written by fewer people than those who read it. The structure of the written material - its order and consistent presentation - has a direct effect on the reader's ability to comprehend the ideas conveyed and is one of the predictors of the impact of the paper or book in its domain. Similarly, the structure of source code has an impact on the ability of other researchers, developers and maintainers to comprehend the ideas it implements.
This style guide is for the C programming language - one of the most widely used programming languages of all time. It is a prescriptive aid for software developers to improve the quality of their software by using a consistent writing style.

January 16, 2015

Keywords
Style Guide — C Language — Software Quality

[1] *Computing Science Laboratory, Palo Alto Research Center*
*__Corresponding author__: glenn.scott@parc.com

Contents

Introduction 1

1 General Principles 3

1.1 Standards . . . 3

1.2 Naming . . . 3

Naming as an Alternative to Commenting • Do Not Grunt and Point • Avoid Too Much Information • Use CamelCase Naming • Do Not Be Vague

1.3 Comments . . . 4

Documentation Comments • Single-Line Comments • Block Comments • Trailing Comments

2 Files 5

2.1 File Names . . . 5

2.2 Files names are case sensitive . . . 6

2.3 File Suffixes . . . 6

2.4 Copyright Notices . . . 6

3 C Header Files **6**
3.1 C Modules and Types 7
3.2 C Header File Names 7
3.3 C Header File Structure 7
Copyright Notice • Include Guards • Module Documentation • Preprocessor Include Directives • Global Constant Definitions • Global Macro Definitions • Global Function Declarations
3.4 Type Definitions 11
4 C Source Files **12**
4.1 C Source File Names 12
4.2 C Source File Structure 12
Copyright Notice • Preprocessor Include Directives • Private Preprocessor Constant Definitions • Private Preprocessor Macro Definitions • Static Function Definitions • Static Function Documentation • Global Function Definitions
5 Functions **14**
5.1 Function Formatting 14
5.2 Braces 15
5.3 White Space 15
Horizontal White Space • Vertical White Space
6 Statements **19**
6.1 Statement Blocks 20
6.2 Conditionals 20
6.3 Goto Statements 21
6.4 If Statements 21
6.5 For Loops 21
For Loop Iterator Variable
6.6 While Loops 22
6.7 Do-While Loops 23
6.8 Switch Statements 23
6.9 Return Statements 23
6.10 Variables 24
Declare Variables At First Use • Set Array Size At Runtime
Acknowledgments **25**

Introduction

Whether for coding style or in design, why have any kind of programming language conventions?

Consider this:

- 40%-80% of the lifetime cost of a piece of software is spent in maintenance.
- Software is rarely maintained by the original author throughout its lifetime.
- Conventions for programming language usage improve the readability of the software, allowing engineers to understand it more quickly and thoroughly.

- If computer program (or programs) source code is the product, it should be as well documented, packaged and clean as any other product.

Underlying all of these points is the fact that having a predictable style and a set of conventions reduces the amount of time it takes for a developer, maintainer or user to understand the software.

Like a book, magazine article, or technical paper, software is typically written by fewer people than those who will read it. The quality of writing that goes into a book, article or paper has a direct effect on the reader's ability to comprehend the ideas conveyed and is one of the predictors of the impact of the paper or book in its domain. Likewise the way a program is written has an impact on other developer's ability to comprehend the ideas it implements.

The fact that a piece of software can cause a computer to perform some function is an additional responsibility of the programmer not necessarily shared by the writer of a book or paper. A book or paper can be vague or suggest a type of solution without being specific. A programmer cannot. A programmer must write precise, complete, readable, and instructive software.

Perspective

This style guide is for the C programming language, one of the most widely used programming languages of all time. During the past 30 years, the language has been modified and updated in a variety of ways, and has directly inspired many other popular languages.

Over this same course of time, the Software Engineering discipline has also changed. Other programming models have been developed. Programming techniques and processes have been refined. Some techniques that were commonplace or even necessary when programming in 1978 are now considered bad style, if not simply bad design. Techniques that are commonplace in other languages nowadays, are oftentimes not readily expressible in C, however, style and design conventions help us take advantage of these techniques and improve the quality of software written in C.

This is a collection of guidelines and recommendations that have been derived over time as the result of experience, best practices, strategic choices and tradition. Often organized as a set of "dos and don'ts", each recommendation is accompanied by a description or rationale. If you think conforming to a style guide is tedious, try writing one.

A guide like this is only a tool. Source conforming to this or any other style guide is not necessarily guaranteed to be good software - writing software that conforms to a style guide does not automatically make one a good programmer. However, the reverse is likely true: Source not conforming to a consistent style is not good software, and a programmer that does not adhere to a consistent style is not a good programmer.

All aspects of software development - from architecture to design to organization to implementation - is about people. Software does not write itself, maintain itself, or reuse itself. Hardware running software cares nothing about separation of concerns, the Liskov substitution principle, or dependency inversion.

The software professional has the ability to design and implement efficient software that balances engineering productivity, machine performance, and cost. A coding style is one tool to aid in that process, though it is not a substitute for sound professional judgement.

This set of articles is a work-in-progress and undergoes updates and reorganization as needed.

Note: Throughout this document, good coding style is shown as:

```
1
    argv++;
3   argc--;
```

While bad coding style is shown with a red background as in:

```
2   argv++; argc--; // No
```

1. General Principles

1.1 Standards

A ll code should use follow the C99 standard and use C99 (ISO/IEC 9899:1999) features.

1.2 Naming

Use clear, direct, and descriptive names. It's worth the time to choose a good name as it is part of vocabulary necessary to express what the software is doing. For example, `PARCBuffer` is a good, descriptive name. Shortening this to `PBuf` loses all descriptive value.

Pro-tip! If you have to explain the name, it's a bad name.

1.2.1 Naming as an Alternative to Commenting

Often the choice of a good name makes code clearer and obviates the need to write ancillary comments.

1.2.2 Do Not Grunt and Point

Never invent spellings of normal words by dropping letters or making abbreviations unless those names are part of the extant vocabulary. For example, if you must have a variable name that points to a C-string, use a name that describes what it contains, or what it represents: `userName` is much more descriptive than `str` or `strng`.

1.2.3 Avoid Too Much Information

Do not include unnecessary or over-specific information in a name about its implementation or type, unless these are inexorable defining characteristics of the name. For example, focus on naming a function based on what it does, not how it does it. Similarly, name a variable based on what it represents, not how it is represented.

For example, `uint32_t` is expressing an unsigned integer of 32 bits which is the defining characteristic of the name. But `uint32_t PARCBuffer_GetUint32Position()` includes unnecessary information about the return type and should be renamed to indicate the utility of the return value rather than its type.

1.2.4 Use CamelCase Naming

Use "CamelCase" naming for composite names. The Naming Conventions in C Programs document offers a detailed description of the use of CamelCase in naming namespaces, files, functions, and constants. Generally, CamelCase should be used in lieu of underscores as shown below:

Good	Bad
`bufferLength`	`buffer_length`
`PARCBuffer`	`PARC_Buffer`

1.2.5 Do Not Be Vague

Avoid vague names like `handle` or `manage`. Instead try to use words that signal some action with a result or resolution.

1.3 Comments

Well written source code has judicious and well written comments. Consistency in the format, type and placement of comments leads to better readability.

We've all heard that writing comments is important, but poorly written, unmaintained comments add complexity rather than reduce it. For example, if you're writing comments to explain how your code works, then the code itself ought to written in a better, clearer, explanatory way. You may find that you won't need to write the comment in the first place.

Comments should be be grammatically correct English sentences.

Comments must be indented to indicate scope: All comments are a preamble to the code they are related to and should immediately precede that code, indented identically. See 'Trailing Comments' for the exception to this policy.

1.3.1 Documentation Comments

Use JavaDoc style comments for documentation.

```
1 /**
  ␣*
3 ␣*/
```

Document in the C Header File Since the header files are typically distributed with binary libraries and applications, the right place for the documentation is in the header file.

1.3.2 Single-Line Comments

```
1 //␣This is a single line comment.
```

1.3.3 Block Comments

Use `/* */` style comments for multi line comments that are not meant for automatic documentation generators.

```
/*
 * This is a
 * multiline comment.
 */
```

1.3.4 Trailing Comments

A trailing comment describes an important note on the current line and must use the // form, with enough space to visually separate it from the statement.

For example:

```
i = 3;      // Initialise at the 4th byte.
```

Comment Out Code With Single line Comments Sometimes you need to comment out a section of code when collaborating, testing, or adding new features. Using single line comments makes it easier to tell at a glance which lines are commented out.

```
// size_t
// parcBuffer_Position(const PARCBuffer *buffer)
// {
//     return buffer->position;
// }
```

Do not use block comments to 'comment-out' code.

```
/*
size_t
parcBuffer_Position(const PARCBuffer *buffer)
{
    return buffer->position;
}
*/
```

Delete Commented Out Code Do not comment out code and then leave it in the source file as it tends to remain there forever. Other developers normally won't have the courage to just delete commented-out code so there it will remain like some mystical monument.

Do Not Use Preprocessor Macros To Remove Code Almost as confusing as using a block comment to comment-out code, is to surround code with C preprocessor macros to remove the code from the compilation unit. Unless that code is explicitly meant to be optional and is accompanied by a clear preprocessor conditional.

2. Files

2.1 File Names

All files must have meaningful names that indicate to the reader the context in which the file exists (library, package, etc), what the content is expected to implement, and the type of file it is (C Source, Header, etc.)

For example, the file named `parc_LinkedList.c` indicates (by convention) that it's part of the PARC library, it implements a linked-list and is a C source file.

The *Naming Conventions in C Programs* document offers a detailed description of the naming conventions for files.

2.2 Files names are case sensitive

Files with the name `file.c` and `File.c` are different files without any relationship to each other (apart from the fact they only differ in the semantic meaning of their names).

The Mac OSX filesystem can cause trouble with file names because the file system itself is case-*insensitive*.

2.3 File Suffixes

By convention the type or kind of a file is indicated by a name suffix. Don't invent new conventions for file suffixes.

C source files end with the suffix `.c`.

C header files end with the suffix `.h`.

2.4 Copyright Notices

Every source and header file must contain a positive assertion of Copyright.

A positive assertion of Copyright removes ambiguity and informs the reader of the ownership of the content. Use the notice to provide directions on where to find additional legal information or instructions.

The Copyright notice is not the place to practice being a lawyer. Keep the notice simple and very direct.

Use Prescribed Ways to Declare Copyright Do not use the character construct `(C)` as a substitute for the word `Copyright`.

Use A Concise Copyright Notice Do not be explicit about the licensing details in the notice text. Instead, refer to another file or location where the licensing details can be found. This lets you modify those details for different releases, or if the requirements change, without having to modify all of your source files.

For example:

```
/*
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 * Copyright 2014 Palo Alto Research Center (PARC), a Xerox company.
 * All Rights Reserved.
 * The content of this file, whole or in part, is subject to licensing terms
   .
 * If distributing this software, include this License Header Notice in each
 * file and provide the accompanying LICENSE file.
```

```
 */
```

Listing 1. A Canonical Copyright Notice

3. C Header Files

Every C source file has a single, corresponding C header file that defines all of the necessary information to use the functionality implemented in the C source file.

3.1 C Modules and Types

A module implementing a type, consisting of a C Header file declaring the functions manipulating the type, and the C Source file implementing those functions has the following naming convention for its file name.

namespace_typeName.c

3.2 C Header File Names

The name of the C header file is exactly the same name as the corresponding C source file, with the suffix `.h` instead of `.c`.

File Name and Content Coupling It must be obvious when looking at a function name, where the declaration and definition of that function name are to be found. Use prefixes to function names to group them into meaningful collections.

For example the PARC Library uses the prefix `parc` to distinguish the functions and their source and header files.

The file named `parc_LinkedList.c` is a file in the PARC Library that implements a linked list. The names of the global functions in this C source file are prefixed by `parcLinkedList` (note the absence of the underscore character in the function name). This enables a reader of any other C source to know that the function `parcLinkedList_Append` is implemented in `parc_LinkedList.h` and `parc_LinkedList.c`.

The *Naming Conventions in C Programs*document offers a detailed description of the naming conventions for C header and source files.

3.3 C Header File Structure

The general layout of a C header file is:

1. Copyright Notice
2. Include Guard Beginning
3. Module Documentation
4. Preprocessor Include Directives
5. Preprocessor Constant Definitions
6. Global Macro Definitions
7. Global Function Declarations

8. Blank Line
9. Include Guard Ending

Do Not Use Per-File Editor Configurations Editors like 'emacs' and 'vim' support the parsing of the first line (or lines) of a file being edited to set attributes in the editor. This should not be used as it may have confusing consequences for later editors of the code. Configure your editor for the way you need it through its standard configuration files, not in the source files that you edit.

3.3.1 Copyright Notice

The Copyright notice for PARC C header file is:

```
/*
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 * Copyright 2014 Palo Alto Research Center (PARC), a Xerox company.
 * All Rights Reserved.
 * The content of this file, whole or in part, is subject to licensing terms
   .
 * If distributing this software, include this License Header Notice in each
 * file and provide the accompanying LICENSE file.
 */
```

Listing 2. A Canonical Copyright Notice

3.3.2 Include Guards

Use include guards in every C header file to prevent multiple inclusion which often results in compile or runtime errors. The name of the include guard is composed from the name of the library or application and the file name of the C header/source file.

For example, the library PARC Library (`libparc.a`) contains the module `parc_Buffer.o` produced from compiling `parc_Buffer.c` and `parc_Buffer.h`. The C header file `parc_Buffer.h` should use the include guard `libparc_parc_Buffer_h`

3.3.3 Module Documentation

A complete description of the functionality provided by the corresponding C source file is written in the C header file directly after the Copyright notice and before any other element in the C header file.

Because C header files typically accompany the distribution of a library without the source files, the right place for documentation is in the C Header file.

3.3.4 Preprocessor Include Directives

A C header file must never burden client code with unnecessary includes. The C header file should include only those external header files necessary for its own successful compilation.

Include everything that is needed A header file should include all of the external include files that provide definitions and values it uses.

Include only what is needed. A header file should not include files that are used only by the associated C Source file. For example, an implementation of a Properties module may permit the properties to be saved to a file. Furthermore, imagine that the C Source file uses the `fstat()` system call to determine some capability. This would not necessarily be a concern for the client

code of the `Properties.h` file, and as such the `Properties.h` file must not include the `<sys/stat.h>` file.

3.3.5 Global Constant Definitions

Manifest constants defined by a C preprocessor `#define` or as global C variables must be prefixed with the name of the project or module.

Constants should be defined with their types cast to add clarity and precision:

```
    #define PARCLibrary_VALUE 10

    #define PARCLibrary_VALUE ((short) 10)
```

Similarly, compiler command line switches and configuration scripts should be defined with a type cast if possible. For example:

```
    % cc '-DPARCLibrary_VALUE=((short) 10)' foo.c
```

Prefer `const` and `enum` to `#define` If a constant is a simple integer value, consider using an `enum` definition instead of a preprocessor define. This permits the value to be displayed by the debugger.

If the `enum` namespace is already cluttered with many other definitions, Consider using a `const` variable definition which allows more control over the scope of the definition. For example, `const int specialValue = 10;` or `static const specialValue = 10;`

Preprocessor defines are a source of a lot of dirty programming. Improve this by adding precision in type casting and using `enum` instead of `#define` where possible. Avoid using a `#define` to create a constant complex value, over and over. Instead create a `const` value as a singleton and reference it.

Do Not Use "Magic Numbers" Do not use unexplained numeric values. Numbers like 0, 1 or 2 might be obvious to you at the moment, but might not be obvious to others, or even to you in the future. For example, a generally obvious use of a constant would be:

```
{
    ...
    return array[index + 1];
}
```

But an expression like this is not at all obvious:

```
{
    ...
    x = ((int) (s & 0xf) << 12) + ((ns / 5 * 8 + 195312) / 390625);
    ....
}
```

In short, if a numeric value requires reading some other code in order understand what the value means, it should be defined with a meaningful name and documented and referenced by that name.

3.3.6 Global Macro Definitions

Use inline or static functions instead of macros Whenever possible, use inline or static functions instead of macros. This reduces or eliminates unexpected or hidden side effects of a macro expansion.

Macros are prefixed with the module name-space Macros defined by a C header file must be prefixed with the name of the project or the module.

Do Not Use Macros to Hide Program Return Do not use macros to hide `return` as in:

```
#define SUCCESS return true;
bool
function()
{
  SUCCESS
}
```

Instead, use an explicit return:

```
#define SUCCESS true;
bool
function()
{
  return SUCCESS;
}
```

Do Not Define Types With Macros Use `typedef` to define types, not a preprocessor macro. A `typedef` definition has scope, debugging information, and makes use of the type-checking of the compiler.

Do the following:

```
typedef char * String;
bool
function()
{
  String foo = "Hello World";
}
```

Not:

```
#define String char *
bool
function()
```

```
{
  String foo = "Hello World";
}
```

3.3.7 Global Function Declarations

All global functions defined in a C source file should be declared and documented in the corresponding C header file.

The global function declaration should be on one line:

Type_functionName (*parameters...*) ;

For example:

```
/**
 * Assert that an instance of PARCBuffer is valid.
 *
 * If the instance is not valid, terminate via 'trapIllegalValue()'
 *
 * Valid means the internal state of the type is consistent with its
 * required current or future behaviour.
 * This may include the validation of internal instances of types.
 *
 * param [in] instance A pointer to a 'PARCBuffer' instance.
 */
void parcBuffer_AssertValid(const PARCBuffer *instance);
```

If the single line exceeds the line length, break the function definition across its parameter list.

3.4 Type Definitions

Raw structures and plain intrinsic types should be avoided. Use `typedef` to name structures and other types to give meaning and establish context and vocabulary.

Type names are prefaced with the module namespace starting with a capital letter. If the namespace is an acronym, use the exact capitalization for the acronym.

Good Type Name	Bad Type Name	
`PARCArrayList`	`ParcArrayList`	PARC is an acronym with specific spelling of all capitals.
`CCNxInterest`	`CcnxInterest`	CCNx is an acronym with specific spelling.
`LongBowEvent`	`LongBow_Event`	Do not use _ in a type name.

Consider Returning a Defined Type Get the compiler to help you avoid mismatched function calls by returning a defined type, rather than a simple intrinsic type.

Avoid Plain Intrinsic Types as Parameters For example, use types like `int` when you are prepared to accept or provide values in the entire domain of `int`, otherwise use an explicit type name to give meaning to the expected domain of values.

For example, `double coefficient;` is less meaningful than `Coefficient coefficient;` where `Coefficient` is explicitly defined to be value in the range of $[0, 1.0)$

4. C Source Files

4.1 C Source File Names

Every C source file has a single corresponding C header file that defines all of the necessary features necessary to use the functionality implemented in the C source file.

Names for C Source and Header files must be prefixed with the name of the associated library or other name space concept with a single _ character separating the prefix from the rest of the file name. This is to visually separate the name-space of a file in a directory listing or similar.

For example:

`parc_LinkedList.c`	A module in the PARC library.
`parc_Buffer.c`	”
`parc_BufferComposer.c`	”
`parc_BufferDictionary.c`	”

The *Naming Conventions in C Programs* document offers a detailed description of the naming conventions for C header and source files.

4.2 C Source File Structure

The general layout of a C source file is:

1. Copyright Notice
2. Preprocessor Include Directives
3. Private Preprocessor Constant Definitions
4. Private Preprocessor Macro Definitions
5. Static Function Definitions
6. Global Function Definitions

4.2.1 Copyright Notice

The Copyright notice for PARC C source file is:

```
/*
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 * Copyright 2014 Palo Alto Research Center (PARC), a Xerox company.
 * All Rights Reserved.
 * The content of this file, whole or in part, is subject to licensing terms
   .
 * If distributing this software, include this License Header Notice in each
 * file and provide the accompanying LICENSE file.
 */
```

Listing 3. The PARC Copyright Notice

4.2.2 Preprocessor Include Directives

The following outlines the types of preprocessor include directives:

Include File Kind	Description
Configuration	`config.h` and other configuration files.
System	Files originating in `/usr/include`
Third party	Third party libraries and support
LongBow	LongBow runtime include files
Project	Project include files
Private	Private include files

Each grouping of include files should be separated by a blank line.

4.2.3 Private Preprocessor Constant Definitions

This is the same as the description for the C Header File.

4.2.4 Private Preprocessor Macro Definitions

4.2.5 Static Function Definitions

Avoid forward declarations of static functions Less is better: Forward declarations are simply more code to maintain. For example, defining the function once means you only have to modify the file in one place if you need to change the function's signature.

```
static int
_myFunction(void)
{
    return 1;
}
```

As a comparision, a forward declaration may easily get lost if you change the function's signature.

```

static int _myFunction(void);

 possibly many intervening lines of code and comments  ;

static int
_myFunction(void)
{
    return 1;
}
```

4.2.6 Static Function Documentation

Document static functions in place Static functions need documentation as much as any globally defined function. Keep in mind that the audience is different. The global function is intended to be used by arbitrary users of the function, while a static function is intended to be used and understood by the maintainer of the module using it internally.

4.2.7 Global Function Definitions

All global functions declared in the corresponding C Header file are defined in the C Source file, and vice versa.

The definition of the function always places the function's storage class and type on one line and the function name on the following line, starting in the first column as in:

```
modifier type
prefix_Name()
{
    ...
}
```

For example:

```
size_t
parcBuffer_Limit(const PARCBuffer *buffer)
{
  parcBuffer_AssertValid(buffer);

  return buffer->limit;
}
```

Not:

```
size_t parcBuffer_Limit(const PARCBuffer *buffer)
{
  parcBuffer_AssertValid(buffer);

  return buffer->limit;
}
```

5. Functions

Deprecate old functions Do not keep old functions forever. If they are replaced by better functions, mark them deprecated and plan for their eventual removal.

Do not use deprecated functions Similarly, do not write code that you know will need to be fixed later. Do not use deprecated functions. If you use a deprecated function, you are guaranteeing future work.

5.1 Function Formatting

A C function is defined according to this pattern:

```
Type
functionName (parameters...)
{
```

```
    ....
}
```

Note that the style for braces is different than the prescribed style of braces everywhere else.

5.2 Braces

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for nonempty blocks and block-like constructs:

- No line break before the opening brace, except for function definitions which must have a line break before the opening brace.
- One line break after an opening brace.
- One line break before a closing brace.
- One line break after the closing brace if that brace terminates a statement or the body of a function. For example, there is no line break after the brace if it is followed by else or a comma.

For example:

```
void
function()
{
    if (condition()) {
        something();
    } else if (condition()) {
        something();
    }
}
```

Exceptions to this are explained in the following sections.

Braces are Never Optional Braces are always used with `if`, `else`, `for`, `do` and `while` statements, even when the body is empty or contains only a single statement.

For example:

```

if (condition()) {
    something();
}
```

5.3 White Space

Horizontal and Vertical White Space are used as visual clues to help group related information.

The uniform use of white space, helps the reader to quickly identify the information needed, just as with spaces between printed words, an extra space after a period, blank lines between paragraphs, or indentation to signal the start of a new paragraph, a uniform use of white space.

5.3.1 Horizontal White Space

Tap Stop is 4 Spaces Indentation is a multiple of 4 spaces.

Do Not Use The Tab Character Raw tab characters in the source files make the source code difficult to read because different programs treat the tabs differently. As a result the same code can look very different depending on whether it's printed or displayed.

Line Length Lines are no longer than 132 characters.

Line lengths of 80 characters date back to the era of punched cards. Today we have high resolution graphical displays and Integrated Development Environments that annotate and color code, support selectable fonts, and pop-up images to describe a function call's signature or documentation. Line lengths of 132 characters is just as arbitrary, but this is the new norm.

Wrapping Lines When an expression will not fit on a single line, break it at the highest syntactic level possible:

1. Break after a comma.
2. Break before an operator.
3. Align the new line with the beginning of the expression at the same level on the previous line.

Indenting Function Calls Here are some examples of breaking lines function calls:

```
    function(longExpression1, longExpression2, longExpression3,
             longExpression4, longExpression5);

    var = function(longExpression1,
                   function2(longExpression2, longExpression3));
```

Indenting Arithmetic Expressions Break outside parenthesized expressions, keeping terms and sub-terms together.

```
    longName1 = longName2 * (longName3 + longName4 - longName5)
                + 4 * longname6;
```

Breaking within a syntactic group requires the reader to stop, mentally regroup, and proceed with visual parsing.

```
    longName1 = longName2 * (longName3 + longName4
                             - longName5) + 4 * longname6; // AVOID
```

Indenting Function Declarations A function definition that cannot fit on one line can break a line following a comma:

```
    int
    function(int anArg, void *anotherArg, char *yetAnotherArg,
    ␣␣␣␣void * andStillAnother)
    {
    ...
    }
```

For functions that require many arguments (which begs design questions and doubts) another technique is to put each parameter on a single line, indenting to the level of the opening parenthesis.

```
    static const long
    veryLongFunctionName(
                         int anArg,
                         void *anotherArg,
                         char *yetAnotherArg,
                         void *andStillAnother)
    {
      ...
  }
```

Formatting If Statements Avoid writing `if` statements that are long. If you must write a long `if` statement, organize the line wrapping to illustrate groups of like terms. Indent the lines two levels of indentation to avoid making them look like the body of the `if` statement.

For example, the indentation below inappropriately requires the reader to properly parse the grouping visually:

```

    if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
            doSomethingAboutIt();
    }
```

Instead use a format like this:

```

    if ((condition1 && condition2)
            || (condition3 && condition4)
            || !(condition5 && condition6)) {
        doSomethingAboutIt();
    }

```

```
    //OR USE THIS
    if ((condition1 && condition2) || (condition3 && condition4)
            || !(condition5 && condition6)) {
        doSomethingAboutIt();
    }
```

Formatting Ternary Expressions For expressions that cannot fit on one line, break before the `?` and `:` operators.

```

    alpha = (aLongBooleanExpression) ? beta : gamma;

    alpha = (aLongBooleanExpression) ? beta
                                     : gamma;

    alpha = (aLongBooleanExpression)
            ? beta
            : gamma;
```

Use Columns in initialization Blocks Lines that are columnar in nature (encouraged for array initializer, for example) and are aligned vertically, should be indented or aligned on columns that are multiples of 4.

Single Spaces A single space is used in the following circumstances:

1. A keyword followed by a parenthesis is separated by a space (See the description of `return` for exceptions) For example:

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

   Note that a blank space is not used between a function name and its opening parenthesis. This distinguishes keywords from function calls.

2. After commas in argument lists.

3. Between binary operators All binary operators are separated from their operands by spaces. Blank spaces never separate unary operators such as unary minus, increment (‘++‘), and decrement (‘–‘) from their operands.

   For example:

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

   ```

```
    while (d++ = s++) {
        n++;
    }
    printSize("size is " + foo + "\n");
```

4. The expressions in a `for` statement are separated by single blank spaces.

   For example:

```
    for (expr1; expr2; expr3) {
        statements...
    }
```

5. Casts are followed by a blank space.

   For example:

```
    functionA((byte) aNum, (PARCByteBuffer *) buffer);
    functionB((int) (cp + 5), ((int) (i + 3)) + 1);
```

5.3.2 Vertical White Space

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

1. Between function definitions.
2. Before a block or single-line comment.
3. Between logical sections inside a function to improve readability.

6. Statements

Make your statements clear and distinct. Do not become a contender for the Obfuscated C Code Contest.

Avoid the comma operator The comma operator often camouflages clear intent from the human eye.

One Statement Per Line Each line should only contain one statement.

For example:

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

Not:

```
    argv++; argc--; // No
```

6.1 Statement Blocks

A statement block is a list of statements enclosed in braces.

The enclosed statements must be indented one more level than the enclosing compound statement. The opening brace is at the end of the line that begins the statement block; the closing brace begins a line and is indented to same level as the brace beginning the statement block.

For example:

```
{
    for (int i = 0; i < 10; i++) {
        statements...
    }

    {
        // A statement block.
        statements...
    }
}
```

6.2 Conditionals

Do not substitute logical values for integer values Many times functions will be defined as returning `int` and yet overload a return value with NULL or zero to indicate a failure. Do not conflate the value zero or NULL with false in a conditional. Be explicit about the value.

For example, this:

```
    if (strcmp(stringA, stringB) == 0) {
        printf("Equal!\n");
    }
```

Not this:

```
    if (!strcmp(stringA, stringB)) {
        printf("Equal!\n");
    }
```

6.3 Goto Statements

Don't use `goto` statements.

Typically a `goto` statement is used to impose structure on code at runtime, rather than addressing structure through design.

Instead, invest the time in designing the code to avoid jumping around.

6.4 If Statements

The if-else class of statements has the following form:

```
if␣(condition)␣{
      statements...
}

if␣(condition)␣{
      statements...
}␣else␣{
      statements...
}

if␣(condition)␣{
      statements...
}␣else␣if␣(condition)␣{
      statements...
}␣else␣{
      statements...
}
```

If statements should never be of the error-prone form:

```
if (condition)
      statements ;
```

Never Use Side-effects in Logical Operators It is best to not rely on side-effects as the result of a logical operation. In addition to detracting from clarity, the right-hand side of the `&&` and `||` operators are not evaluated if the left-hand side is sufficient to determine the logical value, therefore, the side-effects may not even be what you expect.

6.5 For Loops

A `for` statement has 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) has the following form:

```
for␣(initialization;␣condition;␣update)␣{
␣␣␣␣// empty
}
```

Do not use the comma operator within the update clause of a `for` statement.

```
for (int i = 0; i < 10; i++, p++) {
    // empty
}
```

Be clear and make it the body of the `for` statement:

```

for (int i = 0; i < 10; i++) {
    p++;
}
```

6.5.1 For Loop Iterator Variable

If a `for` loop interator has no scope outside of the `for` loop, the variable is declared in the `for` loop initializer.

```

for (int i = 10; i > 0; i--) {
   statements...
}
```

6.6 While Loops

A while statement has the following form:

```
while␣(condition)␣{
␣␣␣␣statements...
}
```

An empty while statement has the following form:

```

while␣(condition)␣{
␣␣␣␣// empty
}
```

6.7 Do-While Loops

A do-while statement has the following form:

```
do␣{
␣␣␣␣ statements...
}␣while␣(condition);
```

6.8 Switch Statements

A switch statement has the following form:

```
    switch␣(condition)␣{
        case␣ABC:
            statements...
            // falls through

        case␣DEF:
            statements...
            break;

        case␣XYZ:␣{
              statements...
            break;
        }

        default:
            statements...
            break;
    }
```

Every time a case falls through (i.e. It doesn't include a break statement), add a comment where the break statement would normally be to show the intent. This is shown in the preceding code example with the `// falls through` comment.

Every switch statement must include a default case. The break in the example `default:` case is redundant as it is the last case in the `switch` statement, but it prevents a fall-through error if another case is added later.

6.9 Return Statements

A return statement with a value should not use parentheses unless they are necessary for proper evaluation, or make the return value more obvious and distinct.

For example:

```
    return;

    return␣result;
```

```
    return (size + 2) / 3;
```

6.10 Variables

Variable Names Are Nouns Variable name should be nouns.

One Declaration Per Line There is one constant or variable declaration per line.

Always Initialize Variables Variables and constants should be initialized in their declarations as in:

```
    char *p = "Hello World";
```

Avoid Floating Point Arithmetic If at all possible, avoid using floating point arithmetic, particularly in loops or where precision is necessary.

Do Not Use Float Type Use the `double` type instead.

6.10.1 Declare Variables At First Use

Avoid C89 variable declarations, and simply declare variables as close as possible to where they are used. This saves the reader from having to remember the declaration of a variable prior to its use. For example, avoid this style of declaring variables:

```
{
  int i;
  char *p;
  bool done;

  for (i = 10; i > 0; i--) {
     statements...
  }

  p = NULL;

  done = true;
}
```

Instead, reformat to declare the variables where they are used:

```
{
  for (int i = 10; i > 0; i--) {
     statements...
  }

  char *p = NULL;
```

```
7
    bool done = true;
9 }
```

6.10.2 Set Array Size At Runtime

Avoid allocating memory and use the C99 dynamic array allocation where possible.

For example:

```
1
    int size = atoi("10");
3   char array[size];
```

Instead of the archaic method:

```
2   int size = atoi("10");
    char *array = calloc(sizeof(char), size);
4   statements...
    free(array);
```

Acknowledgments

Thanks to:

- Laura Hill
- Marc Mosko
- Ignacio Solis
- Alan Walendowski
- Christopher Wood

For their helpful comments and insight.