Personal Coding Standards

Edition 1.2, 2001-09-11

Ben Pfaff

Table of Contents

1 Introduction

This document describes my personal coding standards. Most of my published code is part of the GNU Project, so the standards here are an extension to the GNU Coding Standards. If you are not familiar with those standards, then you should consider reading them before reading further, because points made there are not repeated here (see section "Top" in *GNU Coding Standards*).

Two different kinds of conventions are expressed here:

descriptive
: These are what I use in programming and have used for some time.

prescriptive
: These are what I have, recently or some time ago, decided would be a good idea. These conventions are not necessarily used in most or all of my code, but will probably start to appear in my future code. (If they don't, then I'll amend this document as necessary.)

Little distinction is made in this documentation between these two types of convention.

Imperative sentences are used throughout this document for ease of expression. In standard English, the subject of an imperative sentence is "you", but in this document it should be taken to be "I". This document is not meant to suggest that others adhere to this style, although they are welcome to it. Rather, it is a codification of my own practice.

On the other hand, if you plan to contribute code to one of my own projects, the closer your code comes to these conventions, the easier it will be for me to integrate it and the happier I'll be to accept it.

Some of the conventions expressed here are particularly arbitrary; for example, the preference for octal over hexadecimal and for `for (;;)` over `while (1)`. Don't read more into it than necessary: it's just what I prefer. You probably have different preferences.

Comments, positive and negative, are welcome. Send them to me at `pfaffben@msu.edu`.

2 General

General standards:

ANSI compliance
: Know the ANSI C89 and C99 standards. Comply to them where possible. Make sure that any violations are intentional. Add comments explaining assumptions in the machine or the compiler implementation that go beyond guarantees given in the standards. If it is possible to test for a required implementation property, do so and issue an `#error` if it is not present.

other standards
: Standards other than ANSI C exist. Understand the standards to which a programs complies.

line length
: Limit source code lines to 79 characters or less.

tab width Tabs are 8 characters in width. This doesn't mean that indentations are 8 characters in width: a good programmer's editor is smart enough to decide when to use tabs and when to use spaces to make up whatever quantity of indentation is needed.

compiler extensions
Avoid C language extensions except where fallback to a portable substitute is possible. In that case, use the extension only if a compiler supporting it is detected.

It is okay to use nonessential compiler extensions that can simply be disabled on other compilers. In particular, GCC's `__attribute__ ((format (...)))` is handy.

The same applies to new features in C99.

warnings Code should compile clean on GCC with options `-Wall -W -Wpointer-arith -Wstrict-prototypes`. Another option to consider using is `-Wwrite-strings`. Options `-ansi -pedantic` are good for finding standards violations, but compiling clean against them is not always realistic for real-world code, especially as sometimes system headers trigger warnings from these options. Additionally, `-fno-builtin` can find mistakes, but using it for release versions may not be desirable. Finally, `-Werror` is good for development but not release.

3 Modules

A module is a logically related collection of data and code grouped together for convenience. A module consists of a public header file and one or more C source files that implement the public interface and any needed private interfaces. If the module contains multiple C source files, then it may also have a private header file for use only by its own source files.

name prefixes
To avoid name clashes, each identifier that appears in a public header file must have a prefix of at least three letters followed by an underscore. This applies to all identifiers: variable names, macros, structure and union tags, structure members, enumerations, parameter names in function prototypes, and so on. The same prefix should be used throughout a module.

Names of non-static, private functions shared between source files in a multi-file module must also include the prefix.

one purpose
A module should have a single purpose. A module that defines an ADT should define only a single ADT. See Chapter 7 [ADTs], page 7, for more information.

logical division
Divide a module into multiple source files logically. Don't divide one source file into several just because it is becoming "too large", unless there is a way to logically and systematically divide it.

file names The file name for a module's public header file should be the name of the module, or an abbreviation for it: '*module*`.h`'. In a module with a single source file, this file should be named similarly: '*module*`.c`'. Otherwise, add a dash and another part to the name for each source file: '*module*`-`*part*`.c`'. The private header file for a module should be named '*module*`P.h`' (note the capital 'P').

Limit file names to 14 characters at most due to a limitation of old System V filesystems. Abbreviate aggressively to fit within this limit.

header files

Header files should `#include` a minimal set of needed header files. "A minimal set" does not mean one header file that in turn `#include`s every needed or unneeded header! In fact, don't depend on included header files to include your own dependencies, just their own.

In particular, if a header file uses `size_t`, be sure to `#include <stddef.h>`, or some other standard C library header that also defines it if it is needed for another reason. If a header file uses `FILE`, be sure to `#include <stdio.h>`.

Do not include a header file just for `struct` *tag*`;`, which creates unnecessary dependencies. Write the `struct` *tag*`;` line inline.

Header file inclusion should be idempotent; that is, including a header multiple times should have the same effect as including it once. One possible exception is headers for debugging, like '`debug-print.h`' (discussed below).

initialization

If a module, as opposed to any ADT it defines, requires initialization, name the routine to initialize it *module*`_module_init()`. The corresponding uninitialization routine is *module*`_module_uninit()`. Check with an assertion at every function entry point that the module has been initialized.

Alternatively, don't require explicit initialization: initialize at the first call to any function in the module. For an ADT usually the first call has to be to one particular function anyhow.

debug flags

Define `NDEBUG` to turn off assertions and similar code that only kicks in when something goes wrong. These should be nonintrusive tests that do not cause anything user-visible to happen when nothing untoward is detected, although they may legitimately slow the program down considerably and may add members to structures in public header files. Due to the last property, normally the entire program should be compiled with the same `NDEBUG` setting (either defined or undefined).

Define `DEBUG` to turn on intrusive debug code. This code may do things such as print extra messages to `stderr` or `stdout`, create log files, and so on. Whether `DEBUG` is defined should not affect definitions in public header files. This flag should be defined or undefined on a per-module basis. It is reasonable to include a `#define` or `#undef` for it at the top of a module's source file or private header file. `DEBUG` should not be defined when `NDEBUG` is defined.

Use the macros defined in '`debug-print.h`' for conditional output to `stderr` based on whether `DEBUG` is defined. Be sure to `#include` this file *after* the `#define` or `#undef` for `DEBUG`, if any.

unit tests It is a good idea to include a unit test for each module. These can either be implemented as part of the module's own code, turned on by defining `UNIT_TEST`, or by another file named '*module*`-test.c`'.

4 Names

name length
Names should be unique in their first 31 characters. Don't bother with 6-character uniqueness as required for strict C89 compliance.

macros Macros without parameters and function-like macros that evaluate their arguments multiple times or that have unsafe side effects should have names in all capitals (including the prefix). Well-behaved function-like macros should have their names in lowercase.

Macro parameters should have all-uppercase names.

number Prefer singular nouns to plural in names. In particular, arrays should have singular names.

qualifiers Qualifiers such as `ttl`, `sum`, `avg`, `min`, `max`, ... go at the end of names following an underscore. (See the table of abbreviations below.)

Do not use `num` as a qualifier, since it can mean "number of the current item" or "number of items". Use `cur` or `idx` for the former meaning, `cnt` for the latter.

keywords in other languages
When writing a parser, compiler, etc., for a language, write that language's reserved words in all capitals in names; e.g., `parse_DATA_LIST()` as a function to parse a PSPP `DATA LIST` construct. Do not use such a capitalized keyword as the only component of a name.

status variables
Name status variables and Boolean variables for their meaning: `done`, `error`, `found`, `success`, ... There is an implied, unwritten "is": `(is_)done`, etc.

negative names
Avoid negative names such as `not_found`.

numbers in names
Names that contain digits should usually be restricted to ranges. Use 0 and 1 for range endpoints, not 1 and 2: `x0` and `x1`.

size versus length
A *size* is a count of bytes, a *length* is a count of characters. A buffer has size, but a string has length. The length of a string does not include the null terminator, but the size of the buffer that contains the string does.

reserved names
Avoid names reserved by ANSI C89 and C99 and by POSIX.

p-convention
A `_p` suffix indicates a boolean test for the condition described by the rest of the name. (This comes from Lisp.)

abbreviations

Consistently use the following standard abbreviations for English words in C names:

amount	`amt`	
average	`avg`	
buffer	`buf`	
calculate	`calc`	
command	`cmd`	
compare	`cmp`	
count	`cnt`	
current	`cur`	
decrement	`dec`	
destination	`dst`	
enumeration	`enum`	
expression	`expr`	
function	`func`	
increment	`inc`	
index	`idx`	
length	`len`	
lexical	`lex`	
maximum	`max`	
minimum	`min`	
number	—	Don't use. Substitute `cur` or `idx` for an index, `cnt` for a count.
pointer	`ptr`	Don't use in names of actual C pointers, only in the names of array indexes, etc., that act as pointers.
previous	`prev`	
procedure	`proc`	
quantity	`qty`	
record	—	Don't use.
source	`src`	
statement	`stmt`	
structure	—	Don't use.
table	`tbl`	
target	—	Use "destination" (`dst`) instead.
temporary	`tmp`	
variable	`var`	

Otherwise, try to avoid abbreviations.

5 Types

Boolean types

Don't use an enumeration type or `#define`s for Boolean variables or values. Declare them as an appropriate built-in type such as `int`.

intervals To define a range or an interval, use exclusive upper bounds, not inclusive, so that *upper* `-` *lower* is the number of values inside the interval. This practice

also simplifies iteration over the interval with `for` by reducing the amount of necessary thought: '`<`' can be used instead of the less-common '`<=`'.

bit masks Use only unsigned types for bit manipulation. This includes constants: write `x & 7u`, not `x & 7`. Bitwise operations are less well-defined on signed types.

`typedef` In general, avoid using `typedef`. Code is clearer if the actual type is visible at the point of declaration.

Do not declare a `typedef` for a `struct`, `union`, or `enum`. Do not declare a `typedef` for pointer types, because they can be very confusing.

A function type is a good use for a `typedef` because it can clarify code. Give a function `typedef` a lowercase name with the ending `_func`. The type should be a function type, not a pointer-to-function type. That way, the `typedef` name can be used to declare function prototypes. (It cannot be used for function definitions, because that is explicitly prohibited by the ANSI standard.)

6 Structures and Unions

In this section, when 'structure' is written in a rule, consider the rule to apply equally to unions.

tags Each structure should be defined and used in terms of its *tag*; e.g., `struct node`. The tag of a structure declared within a public header file should include the module's name prefix.

An exception might be a structure defined and used within the scope of a function (e.g., for use as a lookup table) if such a structure does not need a name at all, but more often even such a structure needs a tag.

names Do not include `struct` or `record` or similar in a structure's tag. Do put a module's name prefix at the beginning of a member's name if the structure is declared within a public header file.

In a union that directly contains many structures as members, give these members the same names as their structure tags. If the structures' names have a common prefix or suffix (other than the module name prefix), omit it.

comments Precede a structure definition by a comment that describes its intended usage.

Within a structure definition, every member should have a comment to explain its purpose. These comments should be to the right of the member declaration on the same line. All the member comments for a structure should ideally be aligned flush to a single tab position, but this is not always possible. For a single too-long declaration or, within a large structure, a few of them, it's okay to put the comment on the next line (but still aligned with the rest of the comments).

If a structure has more than a few members, then the members should be divided into logical groups, each separated by a blank line. Each group should be headed by an explanatory comment on a line of its own.

If a member or group of members is relevant only if another member of the structure is set to a particular value, or if the program is in a particular mode, etc., indicate that as part of a comment.

(These are the same as the rules that apply to local variables.)

bit fields For bit fields whose values are enumerated (Boolean values, etc.), not numbers, consider using bit operations on an unsigned type instead.

Do not declare bit fields of type `int`: ANSI allows these to be either signed or unsigned according to the compiler's whim. Use `unsigned int` or `signed int`.

order Try to order structure members such that they pack well on a system with 2-byte `short`s and 4-byte `int`s and `long`s. Prefer clear organization to size optimization in the absence of profiling data that shows a clear benefit.

It is acceptable to make use of the ANSI guarantee that a pointer to a structure is a pointer to its first member, and vice versa, but the fact that it is should be clearly marked in a comment just before the first member of the structure.

magic Consider including a magic number in each structure and testing for it in each function that manipulates the structure. This magic number and tests should be omitted if `NDEBUG` is defined.

external data
Avoid using structures to represent external data structures such as binary file formats or network packets, because padding between structure members differs among environments and the way to turn it off varies among compilers, if it can be done at all.

7 ADTs

An important aspect of *abstract data types* or *ADTs* in C is how the storage for instances is allocated. There are two main ways, plus an uncommon third:

internal allocation
The ADT allocates all its own storage, typically with `malloc()` and `free()`. This means that the ADT's public header file does not have to define the structure behind it, for better information hiding.

external allocation
Code using the ADT is responsible for allocating and freeing storage for the ADT structure itself, although not for any storage allocated indirectly through the structure. This way, instances of the ADT can be kept in automatic memory, instead of having to be stored in dynamic memory.

hybrid Some ADTs can be allocated either way, but this is rarely seen.

An ADT with internal allocation is more "heavyweight" than an ADT with external allocation, because calls to `malloc()` and `free()` tend to be relatively expensive in time. Externally allocated ADTs are more general, because they can always be transformed into internal ADTs with wrapper functions, but the reverse is not true.

Suppose that the ADT is defined as `struct adt`. The standard functions to define to manipulate this ADT are listed below. Note that many of these functions should take additional arguments not shown:

`void adt_init (struct adt *this);` (externally allocated)
`void adt_destroy (struct adt *this);` (externally allocated)
: Initialize or destroy an instance of an externally allocated ADT. If `adt_init()` can fail, then it returns a Boolean value, nonzero for success.

`struct adt *adt_new (void);` (internally allocated)
`void adt_discard (struct adt *this);` (internally allocated)
: Create or destroy a new instance of an internally allocated ADT. If `adt_new()` fails, it returns a null pointer.

`struct adt *adt_open (`*resource_t*`);` (internally allocated)
`int adt_open (struct adt *this,` *resource_t*`);` (externally allocated)
: Initializes an ADT instance from a specified external resource, such as a file, of type *resource_t*. Combines `adt_init()` with a function to access a resource.

`struct adt *adt_copy (const struct adt *src);` (internally allocated)
`int adt_copy (struct adt *this, const struct adt *src);` (externally allocated)
: Makes a copy of `src`. The new instance of the ADT is initialized before the copy.

  For ADTs where shallow and deep copies are different operations, `adt_copy()` performs a shallow copy. The corresponding deep copy operation is `adt_clone()`.

item_t `adt_get (struct adt *this);`
`void adt_put (struct adt *this,` *item_t*`);`
: Functions for storage and retrieval of items of type *item_t* within the ADT.

Additional points:

opacity The structure for an ADT is *opaque*, that is, even if it is declared in a public header file, its members should not be accessed directly by user code. Rather, access them through accessor functions or well-behaved function-like macros, which should be provided wherever relevant.

`this` **pointer**
: The first parameter to an ADT function should be a pointer to an instance of the ADT. Its name in the function definition should be `this`.

allocation If an ADT allocates memory with `malloc()` and it is intended for use outside the program it is originally written for, then there should be a way to substitute another allocator.

(More specific recommendations are not given here because I have not figured out a really good way to do this yet.)

8 Enumerations

purpose Use enumerations for *ordinals*, that is, values that represent concepts, such as colors, types, shapes, modes, ...

Don't use enumerations for bit masks. Use a collection of `#defines` instead. Use `u` or `ul` suffixes to force the bit masks to be of type `unsigned` or `unsigned long`. (Enumeration members are always of type `int`.)

tags Each enumeration type should have a tag. A variable declared to be of type `enum` *tag* should be used wherever reasonable for storing a value from the enumeration, for documentary purposes, and especially as a function parameter type.

names When an enumeration is defined within a public header file, include the module's name prefix within each enumeration. If a module defines more than a few enumeration types, add another prefix to identify the particular enumeration type.

Enumeration member names should be in all capitals.

count Most enumerations should have a final item named *prefix*`_CNT`. The value of this item is the number of items in the enumeration, its count. Specify this value explicitly when declaring an array with one element per member of the enumeration, in order to facilitate compiler verification of the array's size.

syntax Don't follow the final enumeration member name by a comma. ANSI C89 doesn't allow it, though ANSI C99 and many compilers do.

9 Functions

assertions Begin every function with a set of assertions that verify its argument values. Consider calling a function to do a detailed sanity check of any ADTs passed as arguments. Check magic numbers in ADTs for sure. Break up a single long `assert()` invocation into multiple shorter ones, both for readability and for more-specific reports as to which assertion triggered.

initializers Be careful with initializers for local variables. They are evaluated before assertions so they offer another opportunity to screw up.

parameter order

In the absence of good reasons for another order, the following parameter order is preferred:

1. `this` pointer.
2. Input-only parameters.
3. Input/output parameters.
4. Output-only parameters.
5. Status parameter.

order Within a file, public functions should come first, in the order that they are declared in the header file, followed by private functions. Private functions should be in one or more separate pages (separated by formfeed characters) in logical groups. A commonly useful way to divide groups is by "level", with high-level functions first, followed by groups of progressively lower-level functions. This makes it easy for the program's reader to see the top-down structure by reading from top to bottom.

prototypes

Always use prototype style for function definitions. Prototype every public function in its respective header file.

Prototypes for static functions should either all go at the top of the file, separated into groups by blank lines, or they should appear at the top of each page of functions.

Don't comment individual prototypes, but a comment on each group of prototypes is often appropriate.

Don't use parameter names in prototypes, except when there is considerable danger of confusion. If you use parameter names in public header files, don't forget to use the proper name prefix. Proper use of types along with `const` can often make the purpose of parameters clear without any need for names. For example, each parameter's purpose should be clear in this prototype, even without names: `void copy_string (void *, const void *, size_t);`

`const` Use `const` aggressively on pointer parameters to functions. Whether to use `const` for double-pointer parameters (`const char **`, etc.) is debatable, because double pointers whose targets have different qualifiers are not compatible, forcing casts in many cases. (See Steve Summit's C FAQ, question 10.11, for more information.)

comments Short, straightforward functions do not need any comments besides the one preceding it that explains its interface. Longer functions should be divided into logical groups of statements by blank lines and each group should have a brief comment that explains its purpose.

Especially tricky functions should be rewritten until they are clear. If this is not possible for whatever reason, detailed explanatory comments are a second choice.

algorithms

If the algorithms used in a function are published in a journal or book or on a webpage, etc., give a full source citation. If the functions in a file all use algorithms from the same source, cite it in a comment near the beginning of the file.

10 Variables

local variables

Add an end-of-line comment for each local variable declaration. Loop counters, variables used habitually in the same way in every function in a module, etc., are exempt.

If there are several local variables in a function, put them into logical groups, with a blank line between groups. Each group should be preceded by a comment line explaining their purpose as a group.

If some variables are relevant only if another variable is set to a particular value, etc., indicate that as part of a comment.

(These are the same as the rules that apply to structures.)

no alignment
: Do not align variable names, structure member names, etc., in declarations. It takes too much maintenance effort and isn't always possible.

pointer declarators
: Pointer declarators bind to the variable name, not the type name. Write `int *x`, not `int* x`.

scope
: Limit variable scopes, but not obsessively. Don't be afraid to use nested blocks to do so.

  Avoid global variables, except read-only and write-once, read-many variables.

  Avoid static buffers and fixed-size buffers for data of indefinite length. Instead of returning a pointer to a function-local static buffer, use a parameter with a pointer to a buffer to stuff the result into.

11 Expressions

white space
: Put a space on both sides of all binary operators and the ternary operator.

  Exceptions: '`()`', '`[]`', '`->`', '`.`', and '`,`'. The '`()`' and '`,`' operators have their own rules explained elsewhere. The others should not have white space on either side.

casts
: Avoid casts. In particular, never cast the return value of `malloc()`.

`sizeof`
: Where practical, measure the size of objects, not of types: `int *x = malloc (sizeof *x);`.

  The value of `sizeof (char)` is guaranteed to be 1, so don't ever write that expression.

tests for zero
: Write tests for numeric zero as *number* `== 0`, for the string null terminator as *character* `== '\0'`, and for the null pointer as *pointer* `== NULL`, and similarly for nonzero. Don't write tests as plain `if (`*variable*`)`.

floating-point comparisons
: Avoid direct floating-point comparisons, especially tests for equality and inequality. In fact, avoid floating point as much as practical.

array size
: Use the syntax `sizeof` *array* `/ sizeof *`*array* to calculate the number of elements in an array.

relational operands
: When using a relational operator like '`<`' or '`==`', put an expression or variable argument on the left and a constant argument on the right. Exception: tests for mathematical $a<x<b$, and similar, should be written as `a < x && x < b`.

assignment
: Use assignments with '`=`' and related operators only as statements by themselves. Rare exceptions occur when this forces code to become excessively awkward.

long strings
: Use the ANSI C method to split long strings between lines: write a separate string on each line and let the C compiler concatenate juxtaposed strings. Split strings after, not before, white space.

i18n
: Write code looking forward to future internationalization with GNU `gettext`. Most importantly, surround English text strings with `_(...)` in order to avoid the need to go through all the code again later. It is acceptable to write `#define _(TEXT) (TEXT)` at the top of the source file if full i18n is not yet desired. See section "Top" in *GNU* `gettext` *Manual*, for more details.

bit masks
: Generally prefer octal to hexadecimal for bit masks, because it is usually easier to see bit patterns in octal. Use hexadecimal if it is clearly superior.

12 Statements

impossible conditions
: Use the construct `assert (0);` when an impossible condition is detected, i.e., for an impossible `default:` case in a `switch` statement.

`switch` statements
: Always include a `default` case in a `switch`, even if it is a failing assertion or an empty statement.

labels and statements
: A label, both `goto` targets and `case`s, must be followed by a statement, even if that is an empty statement. This is required by the ANSI standard.

infinite loops
: Write infinite loops with `for (;;)`.

`goto`
: There are three acceptable uses for `goto`, two of them related:

  - A jump to an error recovery and unsuccessful exit routine at the end of the function, labeled `error:`. The code so labeled is only executed to handle an error or exception.
  - A jump to a routine to generally finish and exit, at the end of a function and labeled as `done:`. The labeled code is executed whether the function is successful or not. Most functions of this type have a Boolean local variable named `success` used to indicate whether the function's work was successful.
  - A jump out of nested loops or out of a loop containing a `switch`.

  Many error recovery actions of the first two types can be avoided by use of pools. The third type is rarely necessary.

common case following `if`
: In an `if/else` construct, put the common case after the `if`, not the `else`.

multi-statement macros
: If a macro's definition contains multiple statements, enclose them with `do { ... } while (0)` to allow them to work properly in all syntactic circumstances.