# GNOME Programming Guidelines


## Federico Mena Quintero

**federico@nuclecu.unam.mx**

## Miguel de Icaza

**miguel@kernel.org**

## Morten Welinder



This article contains several guidelines and suggestions for GNOME programmers, as well as certain policies that should be followed when writing GNOME programs. This is intended for programmers to know about the GNOME development process and its philosophy. GNOME is a team effort, so it will be useful for programmers to know "the way things are done".

# Introduction

GTK+, the basic user interface library for GNOME, has taught us some important lessons in software design. The GTK+ code is clean, consistent, maintainable, and makes sense. Such code is not only a pleasure to work with, but also encourages good programming practices on those who extend and modify it.

In this article we try to present some suggestions and guidelines that you should take into account when writing code for the GNOME project. We present some of the policies that must be followed when modifying other people's code, using the CVS repository, and making sure your code is fit for inclusion in GNOME. We also present information that will be useful to package maintainers.

Please make sure you read the GNU Coding Standards in addition to this document. These are available as the `(Standards)` info node in the standard GNU documentation.

# The Importance of Writing Good Code

GNOME is a very ambitious free software project, and as such it is composed of many software packages that are more or less independent of each other. Most of the work in GNOME is done by volunteers; programmers may come and go at any time and they will be able to dedicate different amounts of time to the GNOME project. Many people work on free software on their spare time or as a hobby, so if their 'real world' responsibilities change, this will be reflected in the amount of work they can do for free software projects.

Software takes a long time to write and it involves large amounts of work. This is why most part-time volunteers cannot start big projects by themselves; it is much easier and more rewarding to contribute to existing projects, as this yields results that are immediately visible and usable.

Considering this, and also the fact that free software programmers are a scarce resource, we conclude that it is very important for existing projects to make it as easy as possible for people to contribute to them. One way of doing this is by making sure that programs are easy to read, understand, and modify.

Messy code is hard to read, and people may lose interest if they cannot decipher what the code tries to do. Also, it is important that programmers be able to understand the code quickly so that they can start contributing with bug fixes and enhancements in a short amount of time. Source code is a form of communication, and just as someone would not like to read a novel with spelling errors and sloppy punctuation, programmers should strive to write good code so that it is easy to understand and modify by others.

These are some important qualities of good code, and why each one of them is important for free software developers:

Cleanliness

Clean code is easy to read; this lets people read it with minimum effort so that they can understand it easily.

Consistency

Consistent code makes it easy for people to understand how a program works; when reading consistent code, one subconsciously forms a number of assumptions and expectations about how the code works, so it is easier and safer to make modifications to it.

Extensibility

General-purpose code is easier to reuse and modify than very specific code with lots of hardcoded assumptions. When someone wants to add a new feature to a program, it will obviously be easier to do so if the code was designed to be extensible from the beginning. Code that was not written this way may lead people into having to implement ugly hacks to add features.

Correctness

Finally, code that is designed to be correct lets people spend less time worrying about bugs and more time enhancing the features of a program. Users also appreciate correct code, since nobody likes software that crashes.

To summarize this, programmers often contribute to free sofware projects in their spare time, and even regular contributors may have to stop doing so at any time, so it is important that the code be good so that they can modify it easily. The end result will be better software that programmers will like to enhance.

# Coding Style

'Coding style' refers to the way source code is formatted. For C, this involves things like brace placement, indentation, and the way parentheses are used. GNOME has a mix of coding styles, and we do not enforce any one of them. The most important thing is for the code to be consistent within a program or library — code with sloppy formatting is not acceptable, since it is hard to read.

When writing a new program or library, please follow a consistent style of brace placement and indentation. If you do not have any personal preference for a style, we recommend the Linux kernel coding style, or the GNU coding style.

Read the `(Standards)Writing C` info node in the GNU documentation. Then, get the Linux kernel sources and read the file `linux/Documentation/CodingStyle`, and ignore Linus's jokes. These two documents will give you a good idea of what we recommend for GNOME code.

## Indentation Style

For core GNOME code we prefer the Linux kernel indentation style. Use 8-space tabs for indentation.

Using 8-space tabs for indentation provides a number of benefits. It makes the code easier to read, since the indentation is clearly marked. It also helps you keep your code honest by forcing you to split functions into more modular and well-defined chunks — if your indentation goes too far to the right, then it means your function is designed badly and you should split it to make it more modular or re-think it.

8-space tabs for indentation also helps you to design functions that fit nicely in a single screen, which means that people can understand the code without having to scroll back and forth in order to understand it.

If you use Emacs, then you can select the Linux kernel indentation style by including this in your `.emacs` file:

```
(add-hook 'c-mode-common-hook
          (lambda ()
            (c-set-style "k&r")
            (setq c-basic-offset 8)))
```

On newer Emacsen or with a newer cc-mode, you may be able to simply do this instead:

```
(add-hook 'c-mode-common-hook
          (lambda ()
            (c-set-style "linux")))
```

If you use vim, then you can select the GNOME kernel indentation style by including this fragment in your `~/.vimrc` file:

```
set ts=8
if !exists("autocommands_loaded")
  let autocommands_loaded = 1
  augroup C
      autocmd BufRead *.c set cindent
  augroup END
endif
```

The GNU indentation style is the default for Emacs, so you do not need to put anything in your `.emacs` to enable it. If you wish to select it explicitly, substitute "gnu" for "linux" in the example above.

If you know how to customize indentation styles in other popular editors, please tell us about it so that we can expand this document.

## Naming Conventions

It is important to follow a good naming convention for the symbols in your programs. This is especially important for libraries, since they should not pollute the global namespace — it is very annoying when a library has sloppily-named symbols that clash with names you may want to use in your programs.

Function names should be of the form `module_submodule_operation`, for example, `gnome_canvas_set_scroll_region` or `gnome_mime_get_keys`. This naming convention eliminates inter-module clashes of symbol names. This is very important for libraries.

Symbols should have descriptive names. As Linus says, do not use `cntusr()`, use `count_active_users()` instead. This makes code very easy to read and almost self-documenting.

Try to use the same naming conventions as in GTK+ and the GNOME libraries:

- Function names are lowercase, with underscores to separate words, like this: `gnome_canvas_set_scroll_region()`, `gnome_mime_get_keys()`.
- Macros and enumerations are uppercase, with underscores to separate words, like this: GNOMEUIINFO_SUBTREE() for a macro, and GNOME_INTERACT_NONE for an enumeration value.
- Types and structure names are mixed upper and lowercase, like this: GnomeCanvasItem, GnomeIconList.

Using underscores to separate words makes the code less cramped and easier to edit, since you can use your editor's word commands to navigate quickly.

If you are writing a library, then you may need to have exported symbols that are to be used only within the library. For example, two of the object files that compose the library `libfoo.so` may need to access symbols from each other, but this symbols are not meant to be used from user programs. In that case, put an underscore before the function name and make the first words follow the standard module/submodule convention. For example, you could have a function called `_foo_internal_frobnicate()`.

### Consistency in Naming

It is important that your variables be consistently named. For example, a module that does a list manipulation may choose to name the variables that hold a list pointer "l", for terseness and simplicity. However, it is important that a module that manipulates widgets and sizes does not use variables called "w" for both widgets and widths (as in width/height values); this would make the code inconsistent and harder to read.

Of course, these very short and terse names should only be used for the local variables of functions. Never call a global variable "x"; use a longer name that tells what it does.

## Cleanliness

GNOME code should be as clean as possible. This implies using a consistent indentation style and good naming conventions, as described above. It also implies the following.

Learn the correct use of the static keyword. Do *not* make all your symbols global. This has the advantage that you can use shorter names for internal functions within a single source file, since they are not globally visible and thus you do not need the module/submodule prefix.

Learn the correct use of the const keyword. Use it consistently, as it can make the compiler catch a lot of stupid bugs for you.

If you have a function that returns a pointer to internal data which the user is not supposed to free, you should use a const modifier. This will warn the user if he tries to do something incorrect, for example:

```
const char *gnome_mime_get_info (const char *info);
```

The compiler will warn the user if he tries to free the returned string. This can catch a lot of bugs.

If you have random 'magic values' in your program or library, use macros to define them instead of hardcoding them where they are used:

```
/* Amount of padding for GUI elements */
#define GNOME_PAD          8
#define GNOME_PAD_SMALL    4
#define GNOME_PAD_BIG      12
```

If you have a list of possible values for a variable, do not use macros for them; use an enum instead and give it a type name — this lets you have symbolic names for those values in a debugger. Also, do not use an 'int' to store an enumeration value; use the enum type instead. This lets the compiler catch errors for you, allows the debugger to show proper values for these values and makes it obvious what values a variable can take. An example follows:

```
/* Shadow types */
typedef enum {
GTK_SHADOW_NONE,
GTK_SHADOW_IN,
GTK_SHADOW_OUT,
GTK_SHADOW_ETCHED_IN,
GTK_SHADOW_ETCHED_OUT
} GtkShadowType;

void gtk_frame_set_shadow_type (GtkFrame *frame, GtkShadowType type);
```

If you define a set of values for a bit field, do it like this:

```
/* Update flags for items */
enum {
GNOME_CANVAS_UPDATE_REQUESTED  = 1 « 0,
GNOME_CANVAS_UPDATE_AFFINE     = 1 « 1,
GNOME_CANVAS_UPDATE_CLIP       = 1 « 2,
GNOME_CANVAS_UPDATE_VISIBILITY = 1 « 3,
GNOME_CANVAS_UPDATE_IS_VISIBLE = 1 « 4
};
```

This makes it easier to modify the list of values, and is less error-prone than specifying the values by hand. It also lets you use those values as symbols in a debugger.

Do not write obfuscated code, but also try to be spartan. Do not use more parentheses than are necessary to clarify an expression. Use spaces before parentheses and after commas, and also around binary operators.

Please do not put hacks in the code. Instead of writing an ugly hack, re-work the code so that it is clean, extensible and maintainable.

Make sure your code compiles with absolutely no warnings from the compiler. These help you catch stupid bugs. Use function prototypes in header files consistently.

Within GNOME you can use the GNOME_COMPILE_WARNINGS Autoconf macro in your `configure.in`. This will take care of turning on a good set of compiler warnings in a portable fashion.

Comment your code. Please put a comment before each function that says what it does. Do not say how it does it unless it is absolutely necessary; this should be obvious from reading the code. If it is not, you may want to rework it until the code is easy to understand.

While documenting API functions for a library, please follow the guidelines specified in the file `gnome-libs/devel-docs/api-comment-style.txt`. This allows your source code to provide inline documentation that is later extracted by the `gtk-doc` system to create a DocBook manual automatically.

## GTK+-related Issues

GTK+ lets you do a lot of magic and obfuscation with signal handlers, passed closures, and datasets. If you find yourself doing a lot of gtk_object_set_data() all over the place, or passing state around in bizarre ways via signal handlers, please rework the code. If you need to attach a lot of data to a particular object, then it is a good candidate for a new derived class, which will not only make the code cleaner, but more extensible as well.

A lot of heuristics in complicated event handlers can often be replaced by clean code in the form of a state machine. This is useful when you want to implement tricky things like selection and dragging behavior, and will make the code easier to debug and extend.

# Correctness and Robustness

It is extremely important that GNOME code be correct and robust. This means that the code should do what is expected of it, and it should handle exceptional conditions gracefully. While this may seem obvious, this section will give you some tips about how to ensure correctness in your GNOME code. This is very important, since users expect and deserve reliable software that does not crash and performs correctly.

## Ensuring Consistency

Use the Glib assertion macros to ensure that your program's state is consistent. These macros help locate bugs very quickly, and you'll spend much less time in the debugger if you use them liberally and consistently.

Insert sanity checks in your code at important spots like the beginning of public functions, at the end of code that does a search that must always succeed, and any place where the range of computed values is important.

## Assertions and Preconditions

Assertions and preconditions help you ensure that your program's state is consistent. Glib provides macros to put assertions and preconditions in your code. You should use them liberally; in return they will help you locate bugs very quickly, and you'll spend much less time in the debugger tracking bugs down.

These are the Glib macros for preconditions; they will emit a message when their conditions fail, and they will return the function from which they are called. They should be used at the beginning of functions.

g_return_if_fail (condition)

  Returns from the current function if condition is false.

g_return_val_if_fail (condition, value)

  Returns the specified value from the current function if condition is false.

These are the macros for assertions. They will emit a message when their conditions fail and will call `abort(3)` to terminate the program. They should be used to ensure consistency for the code's internals.

g_assert (condition)

  Aborts the program if condition is false.

g_assert_not_reached ()

  Aborts the program if the macro is ever called.

These functions should be used to impose preconditions on the code and check its correctness — think of them as sanity checks for your program. You should use them liberally to assist you in catching bugs quickly; once your program or is fully debugged, you can compile it with these macros disabled so that they will not add any runtime overhead.

The g_return_*() macros should be used at the beginning of public functions in libraries to ensure that the arguments passed to them are correct and within the valid range. If a function returns nothing (i.e. it is void), you should use g_return_if_fail(). Otherwise, you should use g_return_val_if_fail() to return a 'safe' value. When a library function that uses these macros is invoked with incorrect argument values, it will issue an error message, and continue running. The client program may just take a hiccup, do nothing, or just crash, but at least you will know *where* you have passed an incorrect value to a function.

The g_assert() macro should be use to ensure the internal consistency of a library or program. Instead of returning from the current function and continuing execution if the condition fails, g_assert() issues an error message and immediately aborts the program. This is to avoid the program from running in an inconsistent state. You should use this macro when you want to ensure that the program or library is using sane internal values.

The g_assert_not_reached() macro is used to mark a place in the code that should never be reached. For example, if you have a switch statement and you think you are handling all the possible values in the case labels, you should put a g_assert_not_reached() in the default label to ensure that the code never gets there (this would mean you missed handling a value, or your program is incorrect).

These macros help you find bugs faster by warning you as soon as the program reaches an inconsistent state. Use them frequently, and you will find a lot of bugs very easily.

## GTK+-related Issues

Be careful when writing event handlers — make sure they mark events as handled in the proper situations. Make sure your signal handlers have the correct prototypes. This is very important! Remember that not all signal handler prototypes look like this:

```
static void my_handler (GtkWidget *widget, gpointer data);
```

For example, event handlers take an extra event parameter and return a gint Check the GTK+ header files if you need to double-check this.

Make sure your program deals with all user-generated actions in the proper way. Remember that the user can close windows at any time with the window manager; take this into account and write the necessary code to handle this.

If you are testing for modifier keys in an event's state mask, do this:

```
if ((event->status & (GDK_CONTROL_MASK | GDK_SHIFT_MASK))
    == (GDK_CONTROL_MASK | GDK_SHIFT_MASK))
do_some_action ();
```

This is necessary; if you do this instead

```
if (event->state == (GDK_CONTROL_MASK | GDK_SHIFT_MASK))
do_some_action ();
```

then your program will not work correctly if the user has the NumLock key on, for example — NumLock is also a modifier, and if it is turned on, then the event state mask will not be what the second example expects.

### Visuals and Colormaps

Some users have high-end video cards (SGIs and Suns, for example) that support multiple simultaneous visuals. Roughly, a visual defines the memory representation that a piece of hardware uses to store the contents of an image. Most PC video cards support a single visual at a time, but high-end hardware can have different windows and pixmaps in different visuals at the same time.

It is important that you understand visuals and colormaps if you will be writing code that creates windows or pixmaps on its own, instead of using high-level wrappers like GnomeCanvas and GnomePixmap. Please read an Xlib programming manual for more information.

In general, you just need to remember that the visual and colormap for a drawable must match the ones of another drawable if you want to copy an area from the first drawable into the second one. If they are not the same, you will get a BadMatch error from X and you application will most likely abort.

If you create a graphics context (GC) and share it to paint onto several drawables, make sure all of them have the same visual and colormap for which the GC was defined. The same applies if you want to copy an area from a pixmap to a window; both must have the same visual and colormap

If you are not sure that your code is doing this correctly, please ask politely on one of the GNOME development mailing lists for someone to test it with a video card that supports this feature. That person will likely know how to fix the problem and will tell you about it.

## Unix-related Issues

Check the return values of *all* system calls that your program makes. Remember that a lot of system calls can be interrupted (i.e. the call will return -1 and errno will be set to EINTR) and must be restarted.

Do not assume, for example, that `write(2)` will write the whole buffer at a time; you have to check the return value, which indicates the number of bytes written, and try again until it is zero. If the return value is -1, remember to check errno and handle the error appropriately.

If your application calls `fork(2)` without calling `execve(2)`, remember that the child process cannot make any X calls. You can usually diagnose this problem because you get an obscure Xlib error message.

Please read a book like "Advanced programming in the Unix environment", by Stevens, to learn about all these issues and to make sure that your programs use the Unix API correctly. If you are not sure you are using an Unix call correctly, please ask on the mailing lists.

# Security Considerations

Security is a complex issue, and this section cannot nearly explain everything that there is to it. We will try to point out the most common situations where your programs need to be concerned about security.

It is very easy to create security holes by incorrectly creating temporary files under `/tmp`. You must guarantee that the files you will use do not exist at the time of creation. Using an "unpredictable" or "unique" filename is not enough; you must guarantee that a file with that name will not be created by someone else between the time you compute the name and you actually create the file (basically, the attack involves a second party creating a symlink to a file they want to overwrite).

Fortunately, this is easy to do. You can use the following piece of code:

```
char *filename;
int fd;

do {
filename = tempnam (NULL, "foo");
fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
} while (fd == -1);
```

Remember to `free()` the filename and `close()` and `unlink()` the file after you are done. If you want to use the Standard I/O library, you can use `fdopen()` to transform the file descriptor into a FILE *, or you can use `tmpfile()` to do it in one step.

Try not to use fixed-size buffers. Fixed-size buffers for strings are a typical source of exploitable holes that can lead to obscure bugs. If you absolutely must use fixed-size buffers for strings, please use the `g_snprintf()` function to specify the maximum size of your buffer.

Glib provides the very convenient `g_strdup_printf()` function, which works just like `sprintf()` but will automatically allocate a buffer of the correct size. The return value of this function should be released using `g_free()`. This is often more convenient to use than `g_snprintf()`, as it does not limit the size of strings that your program can handle.

If you want to concatenate a group of strings together, you can use the `g_strconcat()` function, which takes a variable-length list of strings and a NULL pointer as the last argument.

*Under no circumstance* create a GTK+ program that is setuid root. The GTK+ and GNOME libraries are big and complex, and they have not been audited for security. In any case, you would not want to have such a big piece of code be setuid root anyways. It you must absolutely use root privileges for something, write your program so that the user interface is in a normal, non-privileged process and create a helper program that is setuid and performs the ‘dangerous’ operations. Also, please notify the GNOME development mailing lists about this and request that someone perform a security audit on your helper program.

In general, if you are not sure that something may create a security risk, please ask on the GNOME development mailing lists about it.

You can read more about some security issues that you might find while programming Unix applications in Wietse Venema’s Murphy’s law and computer security (http://www.fish.com/security/murphy.html). There are other security papers in that site (http://www.fish.com/security) that you might find interesting.

# Performance

Do not read this section until you have made sure that your program is correct, i.e. when you are sure that it works correctly and has no bugs. It is more important for your code to be correct than to be fast. A slow, correct program is better than a fast, buggy one.

If you want to optimize your program, the first step is to profile the program running with real life data and collect profiling information. This will help you pinpoint the actual hot spots that need optimizing. It is important to never optimize ahead of time if you do not have a clear idea on what the problem is. You might end up losing time in speeding up a routine that is not causing the speed bottleneck and you might obfuscate the routine as a result. This could reduce code readability and maintainability with no visible gain in speed.

Simple code is good because it is easy to understand and easy to maintain. If you can write simple code that is also poweful and fast, all the better. If you have a smart piece of code that is not easy to follow, please document it so that people do not break it accidentally.

Do not write code that is hard to read and maintain if it is only to make the code faster. Look for a nice and clean algorithm instead, and implement it cleanly.

Doing a good job in the general case is often better than having many special cases as well. Only provide special cases when you have identified weak spots in the program.

## List Management in Glib

Avoid using constructs that will lead to slow algorithms. If you use `g_list_insert_sorted()` or `g_list_append()` carelessly, you can easily get an algorithm that runs in time proportional to $O(n^2)$. Usually you can create the list backwards, using `g_list_prepend()`, and reverse it when you are done by using `g_list_reverse()`. This is a $O(n)$ operation. And if you need a sorted list, you can create it by prepending elements and then run `g_list_sort()` when you are done.

If you need a list to be sorted at all times, you may be better served by a tree structure or a tree/list hybrid. If you need to construct a list by appending nodes to it, keep a pointer to the tail of the list and update it as appropriate; this will let you append or prepend a node in constant time.

# Localization

GNOME is intended to run on different locales and languages, and so your code should take this into account. You do not have to localize your programs; you just have to make them translatable and localizable.

You should rememeber that different human languages have different grammars, so you should not make assumptions on how sentences are formed. This is important when building strings from separate parts, for example.

Concatenation is usually not the correct way to build a string to be presented to the user This usually leads to messages that cannot be translated correctly to all languages, so try to use `g_strdup_printf()` instead of concatenation. For example,

```
/* Not very good way */
char *message = g_strconcat (_("Hello, "),
     name,
     _(", would you like fries with that?"),
```

```
     NULL);

/* Better way */
char *message = g_strdup_printf (_("Hello, %s, would you like fries with that?"),
 name);
```

This will allow the translator to move the %s around if the language's grammar he is working on requires it.

Remember that not all languages form plurals by appending an "s" to words. Also, sentence structures may change with plurals. Please look at this example:

```
/* Not very good way */
printf (_("%d happy monkey%s bouncing on the bed."),
num_monkeys,
(num_monkeys > 1) ? "s", "");

/* Better way */
printf ((num_monkeys > 1)
? _("%d happy monkeys bouncing on the bed."),
: _("%d happy monkey bouncing on the bed."));
```

This is required since the plural will be formed in different ways in different languages, and the whole sentence's structure may change.

If your program prints dates or times, please use the `strftime()` to format them into strings. This will take care of using the proper date and time representation for the user's preferred locale. Also, if your program must generate a visual representation of a calendar, remember that in some countries consider Monday to be the first day of a week, instead of Sunday. Your program should allow both forms of calendars.

If your program deals with measurements, please make sure it supports both Metric and U.S. systems.

## Binary Compatibility in Libraries

For stable and released libraries, it is very important to try to preserve binary compatibility across major revisions of the code. Library developers should try not to alienate users by making frequent binary incompatible changes on production libraries. Of course, libraries that are being developed or are labeled as not done can change as much as they need.

A library typically exports a number of interfaces. These include routine names, the signatures or prototypes of these routines, global variables, structures, structure fields (both the types and the meaning of these), the meaning of enumeration values, and the semantics of the files that are created by the library. To keep binary compatibility means that these interfaces will not be changed. You can add new interfaces without breaking binary compatibility, but you cannot change the existing ones, since old programs will no longer run correctly.

This section includes a number of hints on how you can make your library code binary compatible with old versions of itself.

Keeping binary compatibility means that programs and object files that were compiled against a previous version of the code will continue to work without recompiling if the library is replaced. A detailed description of this can be found in the `(libtool)Interfaces` info node in the GNU libtool documentation.

## Private Information in Structures

In the GNOME system, it is very common task to create a new GtkObject by creating a structure, whose first member is the class this object inherits from, and then a number of instance variables that are added after this first member.

This design scheme usually leads to code that is hard to update and change: if the internal state that the object needs to maintain is extended, programmers need to resort to various hacks to keep the size of the structures the same. That is, it is hard to add new fields to the public structures without breaking binary compatibility, since the structure fields may change and the structures may change size.

We therefore suggest a strategy that developers may adopt when creating new objects. This strategy will ensure binary compatibility and at the same time will improve the readability and maintainability of the code.

The idea is that for any given object, the programmer should create three files: the first one contains the public contract between the library and the user (this is the header file that gets installed in the system); the second one contains the implementation of the object; and the third file contains a structure definition for the internal or private fields that do not need to be in the public structure.

The public API structure would include a private pointer element that would point to the per-instance private data. The implementation routines would dereference this pointer to access private data; the pointer of course points to a structure that is allocated privately.

For example, imagine we are creating a GnomeFrob object. The widget implementation will be split in three files:

**Table 1. Files that make up the sample GnomeFrob widget**

| Filename | Contents |
|---|---|
| `gnome-frob.h` | GnomeFrob public API |
| `gnome-frob.c` | GnomeFrob object implementation |
| `gnome-frob-private.h` | GnomeFrob object private implementation data. You might want to use this if you plan on sharing the private data across various C files. If you limit the use of this private information to a single file, you do not need this, defining the structure in the implementation file will achieve the same effect. |

**Example 1. Sample `gnome-frob.h` public API.**

```
typedef struct _GnomeFrob GnomeFrob;
typedef struct _GnomeFrobPrivate GnomeFrobPrivate;

struct _GnomeFrob {
        GtkObject parent_object;

        int  public_value;
        GnomeFrobPrivate *priv;
} GnomeFrob;

GnomeFrob *gnome_frob_new (void);
```

```
void        gnome_frob_frobate (GnomeFrob *frob);
```

**Example 2. Sample `gnome-frob-private.h`**

```
typedef struct {
        gboolean frobbed;
};
```

**Example 3. Sample`gnome-frob.c` implementation.**

```
void gnome_frob_frobate (GnomeFrob *frob)
{
        g_return_if_fail (frob != NULL);
g_return_if_fail (GNOME_IS_FROB (frob));

frob->priv->frobbed = TRUE;
}
```

Notice the use of structure prototypes: this enables the compiler to do more checking for you at compile time about the types being used.

This scheme is useful in some situations, particularly the case in which you can afford to store an extra pointer per object instance. If this is not possible the programmer might need to resort to other workarounds for this problem.

The purpose of having an extra private header file for the private structures is to allow for derived classes to use this information. Of course, good programming practice would indicate that interfaces or access methods would be provided precisely to keep the programmer from having to poke into the private structures. You should try to achieve a balance between good practice and pragmatism when creating private structures and public access methods.

Some problems might arise, for example, when you have shipped a version of your code that is being widely used and you want to extend it. There are two possible solutions for this.

The first option is to find a pointer field in the public structure that you can make private, and replace it with a pointer to the private part of the structure. This preserves the size of the structure, since for the purposes of GNOME, pointers can be assumed to be all of the same size. This field can be made to point to the private part of the structure that will be allocated by the function that originally created the public structure. It is important that this field is not called *`private`*, as this is a C++ keyword and will create problems when your header file is being used from C++ sources — call it *`priv`* instead. Of course, this type of change will only work if the old pointer field was really being used for internal purposes only; if users of the library had had to access that field for any purpose, you will have to find another field or resort to a different solution.

If your original structure was a derivative of GtkObject and there are no pointer fields that you can replace, you can use the GTK+ object data facility instead. This allows you to associate arbitrary data pointers to objects. You can use the `gtk_object_set_data()` function to attach values or structures to your object and then you can use the `gtk_object_get_data()` to retrieve back those values. This can be used to attach a private structure to a GTK+ object. This is not as efficient as the previous approach, but it might not even matter for your particular application domain. If you will be accessing this data very frequently, you can set and get it using quarks instead of strings by using the `gtk_object_set_data_by_id()` and `gtk_object_get_data_by_id()` functions, respectively.

# Modifying Other People's Code

GNOME is a team project, so contributions of code to other people's programs are always appreciated. Please follow these guidelines when modifying other people's code.

## General Etiquette

Please follow the same indentation style that the original code uses. The original code will be around for a longer time than you may want to dedicate to it, so keeping your contributions consistent with respect to indentation is more important than forcing your indentation style on the code.

Although your patches may implement very cool functionality, it is very annoying for the author to have to re-indent your code before applying the patch to the code base. So if the original code looks like

```
int
sum_plus_square_of_indices (int *values, int nvalues)
{
int i, total;

total = 0;

for (i = 0; i < nvalues; i++)
total += values[i] + i * i;

return total;
}
```

then do not add a function that looks like

```
int sum_plus_cube_of_indices(int *values, int nvalues) {
  int i,total;

  total=0;

  for (i=0;i<nvalues;i++)
    total+=values[i]+i*i*i;

  return total;
}
```

In this second example, indentation and brace placement do not match the original code, there are no spaces around operators, and the code will just not look like the original one. Please follow the original author's coding style and your patches are more likely to be accepted.

Please do not fix bugs with quick hacks or work-arounds; do the correct fix instead. Also, do not add features as hacks or with code that is not extensible; it is better to rework the original code to be extensible, and then to add your new feature using that code as a framework.

Code cleanups are always welcome; if you find dirty pieces of code in GNOME, we will greatly appreciate it if you submit patches to make the code pretty and easy to maintain.

As usual, make sure your contributed code compiles without warnings, has the correct prototypes, and does things following the guidelines in this document.

## Documenting Your Changes

GNOME uses the standard GNU `ChangeLog` files to document changes to the code. Every change you make to a program should be accompanied by a ChangeLog entry. This lets people read the history of changes to the program in an easy way.

If you use Emacs, you can add ChangeLog entries by pressing "C-x 4 a". If you know the proper incantation for other popular editors, please tell us so that we can expand this document.

ChangeLog entries have the following general format:

```
1999-04-19  J. Random Hacker  <jrandom@foo.org>

* foo.c (some_function): Changed the way MetaThingies are
frobnicated.  We now use a hash table instead of a linked list to
look MetaThingies up.
(some_other_function): Support the MetaThingies hash table by
feeding it when necessary

* bar.c (another_function): Fixed bug where it would print "Take
me to your leader" instead of "Hello, World".

1999-04-18  Johnny Grep  <grep@foobar.com>

* baz.c (ugly_function): Beautified by using a helper function.
```

If you add a new function to a library, please write the necessary reference and programming documentation for it. You can write inline reference documentation by using the comment format described in `gnome-libs/devel-docs/api-comment-style.txt`. If you use Emacs, `gnome-libs/tools/gnome-doc/gnome-doc.el` provides a keyboard shortcut you can use to automatically add documentation templates to your code.

## Changing Code on CVS

If you have write access to the GNOME CVS repository, there are additional policies that must be followed. Since you are working on the master copy of the sources, you have to be especially careful.

If you fix something in a program that is on CVS or if you add functionality to it, and if you have not been working on that program for a long time, please ask the original author before committing your patches. Generally it is OK to ask these questions on the `gnome-devel-list` mailing list.

Once the author of the program has acknowledged you as a 'frequent contributor', you may begin to commit your patches without prior consent. If you want to do a major re-organization of the code, though, you should ask first.

Some modules on CVS have stable branches and development branches. Usually development should go on in the HEAD branch on CVS, and the stable branch should be kept separately. Generally no new features are to be put in stable branches; only bug fixes. Make the bug fix to the stable branch and ask the main author about the policy on merging patches to the development branch; some authors prefer to do it in batches, while others prefer to merge them immediately.

If you will be working on experimental features that could break a lot of code, please create a branch on CVS and do your changes there. Do not merge them onto the main development branch until you are reasonably confident that they work correctly and are nicely integrated with the rest of the code. Using a branch to work on experimental features lets you avoid disrupting the work of other developers. Please ask the main author before merging your branch back into the main part of the tree.

As usual, add a ChangeLog entry when you make a change. Some authors have the policy of rejecting changes that do not have such an entry, and rightly so.

Sometimes different commiting policies exist for the modules, check if the module contains a `README.CVS` file, and if so, please check that file before making changes.

## Branches and CVS

The new policies for branches in CVS are to use FOO-branchpoint for the a branch point tag, and FOO-branch for the branch itself. Although this is in not used everywere currently this is the new policy we are using.

# Maintaining a Package

A package maintainer is a person who takes care of making releases, integrating changes from other people, and in general being responsible for the well-being of a package. During the lifetime of a package, it may change maintainers, for example, if a person loses interest or can no longer dedicate enough time to being a maintainer.

In conformance with the GNU Coding Standards, GNOME uses GNU Autoconf to handle portability issues and GNU Automake to create makefiles. Automake makes it especially easy to build correct packages, so if you are a package maintainer, you should learn how to use it. Autoconf and Automake have a lot of tricky issues, so please feel free to ask for help in the GNOME development mailing lists if you have questions about how to build correct makefiles for your package.

A number of people regularly contribute with localized translations for the message catalog files in GNOME packages. In order to keep translations as up to date as possible, package maintainers should coordinate with translators so that the latter ones can update the translations in time for a release of the package. It is considered good form to notify translators some days in advance before making a release; please send a notification message to <`gnome-i18n@nuclecu.unam.mx`>. This will let them update their respective translations on time so that they will be included in the release.

# Why You Should Worry About Memory Leaks.

Take a look at the Mozilla bug lists and one thing is clear: new code that enters Mozilla is not (or has not been) sufficiently checked for memory leaks and access trouble.

Memory leaks are bad for several reasons:

- They'll eat your swap space slowly. Eventually your program or even your machine will be brought to its knees.

  You don't want to give your program or library a bloat-ware reputation just because you were lazy. Leave that to people in Redmont.

- They hide bad use of memory. If you forget to free memory, there really is no way to catch reads and writes to that chunk of memory. If you later fix the leak, a completely different part of the program could go *poof*.
- When using automatic memory checkers, you don't want to see 600 problems, 500 of which in support libraries. You really want to know that things were by and large clean before you started messing, and therefore that all the problems are yours to fix.

  This is one of Mozilla's problems right now: it leaks like crazy so how are you ever going to know if you just added to the problem?

## Some Advice in the Fight Against Memory Leaks.

- Use "const" whenever you can (for pointers that is).

  If an argument is "const", then undoubtedly the function you call will not free it for you.

  If a function result type is "const", then clearly you are not resposible for freeing it. You *probably* are for a non-"const" result (that isn't just an integer or some such).

  Note, that unfortunately this does not work very well with ref-counted objects. It works great with strings.

  Since C uses call-by-value, there is no point in making ints, doubles, and such types "const".

- *Document resposibilities.*
  - If a function takes ownership of an object/ref, be explicit about it.
  - Always say if the caller should free/unref results.
  - Document *how* to get rid of memory: unref, free, g_free, ...

- *Be careful when using cut-and-paste.*

  The process of cutting and pasting does not improve code, contrary to the belief of many programmers. Please look the code over first, and if you intend to produce many copies, then perhaps you need a helper function.

- *Free everything at exit.*

  This takes time, so perhaps you should only do this within a conditional.

  The reason for this piece of advice is that memory leak checkers have a hard time distinguishing between leaks you know and don't care about and other leaks.

- *Don't leave dangling pointers in your data structures.*

  Assign NULL or (void *)0xdeadbeef to freed members unless you are going to free the structure they are in right away.

  Dangling pointers have a bad tendency of shielding leaks.

- *Run your new code a million times in a loop.*

  If it leaks, you'll know – just follow the process with top in another window.

- *Fix it now, not later.*

  Do not write sloppy code; it will get back to you later.