Coding Standards for GNUstep Libraries

26 Jun 1996

Adam Fedor

Copyright © 1997-2005 Free Software Foundation

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

1 Introduction

This document explains the official coding standards which developers for GNUstep should follow. Note that these standards are in addition to GNU coding standards, not a replacement of them.

To summarise, always add a ChangeLog message whenever your commit a change. Make sure your patch, if possible, improves the operation of the library, not just fixes things - i.e. there are many places where things are just hacked together from long ago and really aren't correct. It's better to rewrite the whole thing correctly, then just make some temporary fix.

Some particular pieces of code which may seem odd or wrong may in fact be there for particular and obscure, but necessary reasons. If you have questions, ask on `bug-gnustep@gnu.org` or `gnustep-dev@gnu.org`.

1.1 ChangeLog Entries

Always include a ChangeLog entry for work that you do. Look for the ChangeLog file in the current directory or look up to any number of parent directories. Typically there is one for each library.

Emacs currently formats the header like this:

```
2000-03-11  Adam Fedor  <fedor@gnu.org>
```

and formats changes to functions/methods like this:

```
* Source/NSSlider.m ([NSSlider -initWithFrame:]):
```

to which you add your own comments on the same line (with word wrapping). Although if you're making similar changes to multiple methods, it's ok to leave out the function/method name.

Important: Changelog entries should state what was changed, not why it was changed. It's more appropriate to put that in the source code, where someone can find it, or in the documentation.

1.2 Coding Style

The point is not what style is 'better' in the abstract – it's what style is standard and readily usable by all the people wanting to use/work on GNUstep. A reasonably good consistent style is better for collaborative work than a collection of styles irrespective of their individual merits. If you commit changes that don't conform to the project standards, that just means that someone else will have a tedious time making the necessary corrections (or removing your changes).

The GNUstep coding standards are essentially the same as the GNU coding standards (`http://www.gnu.org/prep/standards_toc.html`), but here is a summary of the essentials.

White space should be used for clarity throughout. In particular, variable declarations should be separated from code by a blank line and function/method implementations should be separated by a blank line.

Tabs should not be used (use spaces instead). If you do use them (please don't) they really, really, must be for tab-stops at the standard intervals of 8 spaces.

All binary operators should be surrounded by white space with the exception of the comma (only a trailing white space), and the `.` and `->` structure member references (no space).

```
x = y + z;
x += 2;
x = ptr->field;
x = record.member;
x++, y++;
```

Brackets should have space only before the leading bracket and after the trailing bracket (as in this example), though there are odd occasions where those spaces might be omitted ((eg. when brackets are doubled)). This applies to square brackets too.

Where round brackets are used to enclose function or macro paramters, there is no space between the function or macro name and the opening bracket, and where round brackets are used for type-casts or at the end of a statement, there is normally no space between the closing bracket and the following expression or semicolon (however there is a space between the round bracket and the start of a method name in a method declaration or definition) -

```
a = (int)b;
- (void) methodWithArg1: (int)arg1 andArg2: (float)arg2;
a = foo(ax, y, z);
```

The placement of curly brackets is part of the indentation rules. the correct GNU style is to indent by two spaces

```
if (...)
  {
    ...
  }
```

For function implementations, the function names must begin on column zero (types on the preceding line). For function predeclaration, the types and the name should appear on the same line if possible.

```
static int myFunction(int a, int b);

static int
myFunction(int a, int b)
{
  return a + b;
}
```

The curly brackets enclosing function and method implementations should be based in column 0. Indentation is in steps of two spaces.

```
int
myMax(int a, int b)
{
  if (a < b)
    {
      return b;
    }
  return a;
```

```
}
```

Lines longer than 80 columns must be split up, if possible with the line wrap occurring immediately before an operator. The wrapped lines are indented by two spaces form the original.

```
if ((conditionalTestVariable1 > conditionaltestVariable2)
  && (conditionalTestvariable3 > conditionalTestvariable4))
  {
    // Do something here.
  }
```

Some things the standards seem to think are 'should' rather than 'must':

Multiline comments should use `/* ... */` while single line comments may use `//`.

In a C/ObjC variable declaration, the '`*`' refers to the variable, not to the type, so you write

```
char *foo;
```

not

```
char* foo;
```

Using the latter approach encourages newbie programmers to thing they can declare two pointer variables by writing

```
char* foo,bar;
```

when of course they need

```
char *foo, *bar;
```

or (in my opinion better)

```
char *foo;
char *bar;
```

An exception to the indentation rules for Objective-C: We normally don't break long methods by indenting subsequent lines by two spaces, but make the parts of the method line up instead. The way to do this is indent so the colons line up.

```
[receiver doSomethingWith: firstArg
                      and: secondArg
                     also: thirdArg];
```

That's the style used mostly in the GNUstep code - and therefore the one I try to keep to, however, the standard two space indentation is also acceptable (and sometimes necessary to prevent the text exceeding the 80 character line length limit).

```
[receiver doSomethingWith: firstArg
  and: secondArg
  also: thirdArg];
```

My own preference (not part of the standard in any way) is to generally use curly brackets for control constructs, even where only one line of code is involved

```
if (a)
  {
    x = y;
  }
```

Where using conditional compilation you should comment the #else and #endif with the condition expression used in the #if line, to make it easy to find the matching lines.

```
#if condition
// some code here
#else /* not condition */
#endif /* condition */
```

1.3 ObjectiveC

Since GNUstep is primarily written in ObjectiveC the C language coding standards largely apply with modifications as specified in the previous section.

Most code is expect to be written in traditional ObjectiveC, but classes implementing newer APIs designed by Apple will sometimes need to be written using ObjectiveC-2.0, though compatibility with old compilers should be maintained wherever possible, and preprocessor macros must be used to at least conditionally build new code without breaking old code.

In particular, blocks are completely non-portable and must never be used internally (though methods with block arguments are provided for compatibilty with the Apple APIs). As well as being similar to the 'goto' operation in making code hard to maintain, bllocks have a number of issues which mean they are never likely to become standard across compilers (eg https://thephd.dev/lambdas-nested-functions-block-expressions-oh-my).

Another ObjectiveC-2.0 feature (the dot ('.') operator) is also forbidden. One problem is that, while apparently simple, the actual operation of this feature in unusual cases is actually undefined and varies between compiler versions. The more serious problem is that the feature is simply very bad style because it looks like a simple structure field access and yet the code is really doing something very different and much more expensive, so use of the feature tends to lead to performance problems, bugs, and less explicit/readable code.

1.4 Memory Management

We encourage the use of the following macros to ease retain and release and as a convenience for managing code which should work in both a conventional retain counting environment and one with automatic reference counting (ARC)

- ASSIGN(object,value) to assign an object variable, performing the appropriate retain/release as necessary.
- ASSIGNCOPY(object,value) to copy the value and assign it to the object.
- DESTROY(object) to release an object variable and set it to nil.
- ENTER_POOL and LEAVE_POOL to bracket statements which should be performed inside their own auutorlease context.

1.5 Error Handling

Initialisation methods (e.g. -init) should, upon failure to initialise the class, release itself and return nil. This may mean in certain cases, that it should catch exceptions, since the calling method will be expecting a nil object rather than an exception on failure. However, init methods should endeavour to provide some information, via NSLog, on the failure.

All other methods should cause an exception on failure*, unless returning nil is a valid response (e.g. [dictionary objectForKey: nil]) or if documented otherwise.

Failure here is a relative term. I'd interpret failure to occur when either system resources have been exceeded, an operation was performed on invalid data, or a required precondition was not met. On the other hand, passing a nil object as a parameter (as in [(NSMutable-Data *)data appendData: nil]), or other "unusual" requests should succeed in a reasonable manner (or return nil, if appropriate) and/or reasonable default values could be used.

If an error is recoverable or it does not damage the internal state of an object, it's ok not to raise an error. At the very least, though, a message should be printed through NSLog.

Special care should be taken in methods that create resources like allocate memory or open files or obtain general system resources (locks, shared memory etc.) from the kernel. If an exception is generated between the allocation of the resource and its disposal, the resource will be simply lost without any possibility to release. The code should check for exceptions and if something bad occurs it should release all the allocated resources and re-raise the exception.

Unfortunately there is no nice way to do this automatically in OpenStep. Java has the "finally" block which is specifically designed for this task. A similar mechanism exists in libFoundation with the CLEANUP and FINALLY blocks.

1.6 Variable Declaration

All variables should be declared at the beginning of a block. The new C99 standard (and gcc 3.X) allow variables to be declared anywhere in a block, including after executable code. However, in order to be compatible with older compilers, all GNUstep programs should keep the old behaviour.

Certainly we would consider it a bug to introduce code into the GNUstep libraries which stopped them compiling with one of the commonly used compilers.

Instance variables in public APIs should generally be limited to those which are explicitly declared to be public and which will never change (we want to avoid breaking ABI between releases by changing instance variable layouts). Eventually compilers supporting a non-fragile ABI will be available and this will no longer be an issue, but until then we need to deal with the fragile API instance variable problem.

The standard mechanism to support this is to provide a single private pointer variable (void *_internal;) which will be used to point to an area of memory containing the actual instance variables used internally. The internal implementation is then free to change without any change to the size of instances of the class.

The GNUstep-base library has a standardised set of macros for writing code which deals with use of an _internal pointer to instance variables at the same time as allowing the instance variables to be used directly in the class if the code is built using the non-fragile ABI.

1.7 Naming Conventions

The convention for naming items in GNUstep differs from the GNU standard as it needs to be compatible with OpenStep/MacOS-X.

Public classes, variables, functions and constants begin with the NS prefix if they are part of the OpenStep or MacOS-X APIs, and begin with GS if they are GNUstep extensions. GNUstep extensions must not use the NS prefix.

Class, public function, and global variable names have the first letter of each word in the name capitalised (underscores are not used).

```
@class NSRunLoop;
GSSetUserName();
NSGenericException;
```

Method and instance variable names are similarly capitalised, except that the first letter of the first word is usually not capitalised (there are a few exceptions to this where the first word is an acronym and all the letters in it are capitals). Underscores are not used in these names except to indicate that the method/variable is private, in which case the name begins with an underscore.

```
{
  int publicInstanceVariable;
  int _privateInstanceVariable;
}
- (void) publicMethod;
- (void) _privateMethod;
```

The names of accessor methods (methods used to set or get the value of an instance variable) must mirror the names of the instance variables. The name of a setter method is of the form 'setVar' where 'Var' is the instance variable name with any leading underscore removed and with the first letter converted to uppercase. The name of the getter method is the same as the instance variable name (with any leading underscore removed).

```
{
  int _amplitude;
  int frequency;
}
- (int) amplitude;
- (int) frequency;
- (void) setAmplitude: (int)anAmplitude;
- (void) setFrequencey: (int)aFrequency;
```

1.8 Object Persistence

The standard method of saving and restoring object information in GNUstep is through the use of the -encodeWithCoder: and -initWithCoder: methods. Any object which requires persistence implements these methods. They are used, for instance by Gorm, to save GUI interface elements. It is important that all changes to these methods be backward compatible with previously stored archives (for instance, those created by Gorm). The easiest way to do this is to use class version numbers to indicate which archive configuration should be read. Modern implementations are expected to suppoort keyed archiving and should use the same keys that are used in OSX.

1.9 Documentation

Document every method you change or add! This makes it easier to fix our lack of documentation and keep up to date with changes. Make sure you do not copy either the OpenStep or Cocoa documentation. Some methods are so simple you might have to intentionally reword the documentation so it is different.

Public documentation should be in the header files, formatted so that the autogsdoc tool can extract it.

1.10 Before You Commit

- Make sure you have a ChangeLog entry
- Make sure any new method/class is documented in the header file. or `Appkit/Appkit.h` if appropriate.
- If you have added a class, add the class to `Foundation/Foundation.h`
- If you have updated and configure checks, be sure to run both autoconf and autoheader.
- Make sure everything still compiles at least on the most common platform (ie Intel processor, GNU/Linux operating system, with the GCC compiler and ObjC runtime), and ideally on ms-windows too.
- Make sure you've tested the change and contributed testcase code to the testsuite. Run the testsuite on the systems where you compiled.
- Make sure that documentation generation still works by running 'make' in the Documentation directory.

1.11 Contributing

Contributing code is not difficult. Here are some general guidelines:

- We maintain the right to accept or reject potential contributions. Generally, the only reasons for rejecting contributions are cases where they duplicate existing or nearly-released code, contain unremovable specific machine dependencies, or are somehow incompatible with the rest of the library.
- Acceptance of contributions means that the code is accepted for adaptation into GNUstep. We reserve the right to make various editorial changes in code. Very often, this merely entails formatting, maintenance of various conventions, etc. Contributors are always given authorship credit and shown the final version for approval.
- Contributors must assign their copyright to FSF via a form sent out upon acceptance. Assigning copyright to FSF ensures that the code may be freely distributed.
- Assistance in providing documentation, test files, and debugging support is strongly encouraged.

Extensions, comments, and suggested modifications of existing GNUstep features are also very welcome.