Coding Standards for GNUstep Libraries

26 Jun 1996

Adam Fedor

Coding Standards

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 summarize, 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.

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.

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.

Tabstops should be 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 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-

```
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

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

For function implementations, the function names must begin on column zero (types on the preceeding 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.

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

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

Memory Management

In anticipation of the day when we can make the use of a Garbage Collector possible for all GNUstep apps (it's almost-usable/usable-with-care for non-gui apps now), the normal use of retain/release/autorelease is deprecated.

You should always use the macros RETAIN(), RELEASE() and AUTORELEASE() (defined in NSObject.h) instead.

There are also some extra macros that may be of use -

- ASSIGN(object,value) to assign an object variable, preforming 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.
- TEST_RETAIN(object) to retain an object if it is non-nil
- TEST_RELEASE(object) to release an object if it is non-nil
- TEST_AUTORELEASE(object) to autorelease an object if it is non-nil
- CREATE_AUTORELEASE_POOL(name) to create an autorelease pool with the specified name.
- IF_NO_GC(X) compile the code 'X' only if GarbageCollection is not in use.

Error Handling

Initialization methods (e.g. -init) should, upon failure to initialize 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 endeavor 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 [(NSMutableData *)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 reraise 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.

Variable Declaration

All variables should be decalred 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 behavior.

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.

Object Persistance

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 persistance 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.

Before You Commit

- Make sure you have a ChangeLog entry
- Make sure everything still compiles
- Make sure you've tested the change as much as is reasonable.
- If you have added a class, add the class to '`Foudation/Foundation.h`' or '`Appkit/Appkit.h`' if appropriate.
- If you have updated and configure checks, be sure to run both autoconf and autoheader.

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.