Good Practices in Library Design, Implementation, and Maintenance

Ulrich Drepper, Red Hat Inc.
drepper@redhat.com

Version 0.1
March 7, 2002

Common to all programming projects is the requirement to write code in a way which allows continued development without raising the costs disproportional. Many techniques can be used and it is up the developer, the group, or the organization they work in to decide. This whole complex will not be discussed here.

The development of software libraries, as opposed to applications, has additional requirements. Libraries are meant to be used in one or more projects and to do this an interface for the user has to be provided.

Since a software product rarely is finished changes in the library implementation happen all the time. The problem now becomes to maintain the exposed interface programmers use in other projects. Maintaining means that the behavior of the functions must not change for any legal input. The only exception must be a change to fix a difference from the documented and/or intended behavior.

The remainder of this article will introduce techniques which can be used to achieve this goal. Most of the techniques are specific to C and C++ but maybe users of other languages can also use a thing or two. Not all of the proposed techniques are applicable everywhere but it is highly advised to implement whatever is possible for a given platform even if this means having slightly inconsistent implementations across different platforms. Making higher requirements on the tools which can be used to build the project should also not get into the way of producing a stable interface.

1 Designing the Interface

There are not too many rules which can generally be applied to the design of the interface of a library. If a standard governs the library interface there is not much choice. Otherwise the way the library is used might impose a certain interface on the library. If there is a choice following some or all of the following rules will help to reduce maintenance headaches.

Most problems are introduced by the use of structures. Using structures is necessary and the correct way to deal with the complexity of programs but some rules should be followed. Normal types of variables and parameters are not causing that often problems. If a type really has to be enlarged (no need to reduce the size) this normally requires drastic changes in the entire interface. There are techniques to handle this; the Large File Summit (LFS) specification to introduce 64 bit file offsets on 32 bit systems is one example. See the Reference section 4 for a link to the specification. One design rule is true for variables of any type:

Avoid under all costs variables as part of the API. Implement get- and set-functions to handle an internal variable instead.

The reason for this wide-ranging rule is found in the ELF binary format and how it is implemented on some architectures. Most architectures allow the application code itself be compiled in a way which is not position independent. This makes sense since an application is always loaded at the same address and the code is often slightly more efficient. But it creates big problems when using variables in dynamic shared objects (DSOs) the application is linked with.

The application code itself requires to know the address of the variable which is used. Since it is not known where the DSO with the variable is loaded (or even which DSO will provide the variable definition) the variable must be placed in the executable itself. The DSOs will have to live with that and use the variable in the executable. Initialized variables will have to be initialized by the dynamic linker with a copy relocation.

All this has the consequence that the size of the variable is hard-coded in the application. It cannot change without completely breaking the entire program. If the variable would only be referenced by DSOs the definition can stay where it is and could be extended by creating a structure with an element of the type of the original variable as the first element. But even this is creating problems since the access to the variables is not controlled. If the user expects a newer version than the definition provides the variable access can still go astray.

By using functions to access variables these problems can be eliminated. The functions can make sure the versions of user and definition match.

How version matches happens is another point to consider. Some libraries require the user to signal which version is expected by a special function call which has to be made before any other function of the library can be used. See `elf_version` in the `libelf` library for an example. This works quite well but only if there is only one user in the program. Many libraries will be used more widely and independently. So it could be that the main executable and a DSO it uses both independently use the services another library provide. How should the used library react if it gets requests for two different version numbers?

An alternative approach is to use versioning in each data structure. I.e., one field of each data structure describes the version of the interface this data structure corresponds to. This is the best method to ensure compatibility. But it comes at a cost. The additional field is present in each instance of the structure which might add up when many instances must be created. Also, setting the field, even for short-lived temporary objects, requires operations at runtime.

A compromise is perhaps to specify the interface in the function calls as an additional parameter. This eliminates the storage requirements but passing the additional parameter might include a minor performance penalty. Passing the parameter could even be hidden by using macros which implicitly add the additional parameter.

There is no clear advantage for any of the methods. The library designer should use what is best for the situation at hand.

Another important aspect of the design is consideration for the namespace. Namespace rules, conventions about naming the interfaces of a library in this case, help to avoid conflicts which might in some instances go undetected for some time.

For every library prepend a chosen prefix to every interface, variable, functions, data structures, a library provides. The prefix need not necessarily be the same for all objects, a library can use more than one prefix.

The already mentioned `libelf` library prepends `elf_` to every function and objects. Data type names have `Elf_` prepended. The prefix does not add any significant burden on the programmer. The avoided headaches caused by name conflicts certainly outweigh the extra characters which have to be added. Other library designers and implementors will probably not collide with the names since they will at least investigate what other libraries doing a similar job are available and therefore notice the choice of the prefix.

While this mechanism is usable for C++ libraries as well ISO C++ provides a better mechanism. The language provides native support for namespaces. This should be used instead. The only drawback is that a more recent compiler might be required. This should not be a deterrent. In the header all definitions should be enclosed by

```
namespace ident {

}
```

where `ident` is the user-chosen namespace identifier. The files implementing the library then must use the prefix in the appropriate places as well. The main advantage of using the C++ namespaces instead of name prefixes is that it happens automatically and the user can pull the names of the interface in the global namespace with the `using` keyword. For more information consult a book on programming with C++.

Headers were just mentioned being part of the interface. In fact, they constitute a large part of the interface. It is important to keep them clean.

Installed header files for C and C++ libraries should only contain the definitions and declarations which are necessary to define the interface.

The headers are a main mean for users to learn about the interface. Adding definitions or declarations for internal functions in the installed headers will at the very least irritate users. If no export control in the library possible (it's not available or static linking is used) this can lead to compatibility problems in the future. The library developer rightly assumes that s/he is allowed to change or remove internal interfaces.

The headers for the projects should consist of the self-contained installed headers and the internal headers which contain all the rest. Type definitions can and should also be deferred to internal headers.

Using forward declarations to introduce incomplete types is the correct way if the user never has to allocate an object of that type herself.

To allow the user to allocate objects of these types the library should export functions which do just that. A good example for this kind of handling of a type is the `FILE` type from `<stdio.h>`. Most implementation over time changed the internal representation of the type but the interfaces did not have to change. The `FILE` objects are created in the C library by calls to functions like `fopen`.

The functions to handle `pthread_attr_t` objects in the POSIX thread library are another example. There is a set of functions to set and get the value for each attribute stored in this structure. Unfortunately can attribute objects we allocated by the user since the type is defined in the `<pthread.h>` header so this is no perfect example.

One more aspect of header file design is of extreme important and often misunderstood or not followed.

The header file for a library should not change for instances of the library which are compiled with different configurations.

Often libraries have configurations which allow to opt-in and -out of supporting various kinds of additional services. For instance, a library could support various compression or encryption methods. If these configuration options require change in data structures of function interfaces which are exported these changes must *not* depend on the configuration options. Instead the public interface should always reflect the most general configuration. Only this can ensure that a program using such a library can run regardless of how the library was configured. To signal that certain functionality is not available in the library runtime tests or some macros used as flags should be used.

One method to ensure this is possible without requiring, for instance, data types of disabled configurations to be available is to use oversized unions. That means all the data which is needed by the different configurations is put into a union. Also in this union is another type, often an array, which requires more space than the largest data structure used by any configuration.

```
struct exported_struct
{
  int field1;
  union
  {
#ifdef CONFIGURATION1
```

```
    struct
    {
      int foo;
    } data1;
#endif

#ifdef CONFIGURATION2
    struct
    {
      some_type_t bar;
      int baz;
    } data2;
#endif

    long int filler[64];
  } u;
  struct exported_struct *next;
};
```

In this example the element `u.filler` is the unused union element which is only added to keep the size of `exported_struct` constant regardless of the configuration option. If this would not be the case the absence of the configuration options would change the offset of the `next` field in the structure.

And speaking of filling structures. Over time some exported data structures might have to contain additional information. Without breaking compatibility this is only possible if the memory was already reserved ahead of time.

If it is not possible to provide an incomplete definition of a data type the type definition should always create at least a minimal amount of padding to allow future growth.

This means first that all exported types probably should be structure unless it is clear that forever only integer, float, or pointer values have to be represented. This might require a certain form of function interfaces (structures should not be return values) but this is normally an acceptable limitation.
Second, a structure should contain at the end a certain number of fill bytes.

```
struct the_struct
{
  int foo;
  // ...and more fields
  uintptr_t filler[8];
};
```

The `filler` field in the structure adds eight more words to the structure which all can contain pointer values or similarly sized integer or even float values. Using `uintptr_t` is a much better choice than `int` or `long int`.

If at a later time a field has to be added to the structure the type definition can be changed to this:

```
struct the_struct
{
  int foo;
  // ...and more fields
  union
  {
    some_type_t new_field;

    uintptr_t filler[8];
  } u;
};
```

Alternatively the number of elements of the `filler` array can be reduced and the new field be added somewhere in the structure. This is also OK as long as the position of the original fields is not changed and the total size is not changed either.

2 Implementing the Library

While creating the actual library many things can be done to improve to increase compatibility. It starts with general rules which can and should always be followed and ends with special steps only library implementors really have to know.

Define as many functions and variables as local to the object file by adding `static` to the definition.

When not exporting functions and variables which are not part of the API these cannot possibly create problems. Besides, doing has other advantage like enabling the compiler to perform more and better optimizations. Some people feel comfortable with creating larger source files which implement a larger number of interfaces to increase the number of functions and variables which can be defined with file scope.

But often functions and variables cannot be defined local to a single object file since they are used in other object files of the application. If such a situation occurs in static library (archive) there is not much one can do. There is no way to restrict the use of internal symbols. The only possibility is to combine all object files which need certain internal resources into one using `ld -r` and then restrict the symbols which are exported by this combined object file. The GNU linker has options to do just this. The user wouldn't have to know that this happened if this object file is then put in an archive. The drawback is that the advantages of an archive, namely that only the object files which are really needed in the final application are linked in. No recommendation here, it is up to the library developer to decide what is best in each situation. When the library is provided as a DSO the recommendation is clear:

Always reduce the symbols which are exported to the bare minimum. Ideally only the documented interfaces should be exported.

Doing this eliminates or at least significantly reduces the risk that application are using a library in a way other than it was designed for. The library maintainer has the freedom s/he needs since internal functions can be changed and removed without fearing compatibility problems.

To restrict the exported symbols there exist several methods. One possibility is to linker maps to name symbols which are to be exported. The GNU linker supports this for a long time and it is also possible to use similar techniques on other platforms (e.g., Solaris).

Another possibility was introduced more recently. The ELF ABI now defines visibility rules for symbols. Two of the rules, hidden and internal, instruct the linker not to export the symbol the symbol from the DSO. The symbol is available to resolve undefined references in other object files which are linked into the DSO. Defining one of these two visibility rules for a symbol is therefore more or less equivalent to having a linker map (without version information) which instructs the linker not to export the symbol.

Before gcc 3.1 the programmer would have to use `asm` to add the information:

```
int foo;
asm (".hidden foo");
```

Starting with gcc 3.1 attributes can be used:

```
int foo __attribute__ ((visibility ("hidden")));
```

The internal visibility rule is architecture dependent and not defined for many architectures. It should only be used when it is understood what it does.

For C++ libraries both methods are available as well. The linker scripts can contain demangled symbol names with a special syntax. Mangled names would be specified just like C symbol names. When using the attribute method the hassle of determining mangled or demangled names fall away. In addition to this will the compiler get more information about the symbol which can result in better code to be generated. Therefore the use of attributes is preferred but it is not available with older compilers.

A final point about creating DSOs is the existing of SONAMEs (Shared Object Names). This is a name given to the DSO which will be recorded in the application which require it at runtime. The SONAME should consist of the library file name plus an interface number. E.g., the SONAME for the Linux C library is `libc.so.6`. `libc.so` is the filename of the DSO which is searched for by the linker, the interface number is 6.

Defining SONAMEs this way allows easy and intuitive identification and the interface number allows multiple versions of the same library with different interface numbers to be used on the system without collision. This is only the last resort solution but it should be possible.

3 Maintaining the Library

Despite best efforts made every non-trivial library will need maintenance and eventually changes and extensions. Making changes in a way which is not causing problems can be really tricky.

One sort of changes which should be made without hesitation are bug fixes. They should be made even if the API is changed. Some people argue that some programs are depending on the old, broken behavior. But this argumentation is flawed. Equally well there are programs which depend on the correct behavior, written by people who read the specification. No program which depends on broken behavior deserves protection. If a programmer has found such a discrepancy and changed the application to use it instead of reporting it to the library maintainer s/he gets what s/he deserves.

New interfaces which were not present in previous versions should be marked so that an application expecting the new interface does not even start running.

By default will the dynamic linker defer the lookup of all function symbols required in the application until the time the function is actually needed (lazy relocation). This can lead to problems if the application uses a symbol which is used in a new version of a library which is not available at runtime. The application would start running and only when the new function is actually used fail. By then some “damage” might already have been done.

One sledge hammer method to deal with this is to bump the interface number recorded in the SONAME every time the library is extended or otherwise changed. This will lead to an inflation in the number of DSOs since old versions normally cannot be removed for some time. It also can introduce problems if different versions of the same library are dependents of the same application. This can easily happen if, for instance, the application and one of its DSO dependencies depend on these two different version. For the dynamic linker the two versions of the library are entirely unrelated and what happens if both versions are loaded and used at the same time in the same process depends on the library. It might just work but more often it will cause problems severe enough for immediate termination of the process.

A more suitable method is to use internal versioning. Each exported interface gets a version number assigned. The user of a DSO record the symbol requirements for each symbol and the maximum version required from the DSO. The maximum version number is checked when the DSO is loaded and if the DSO does not at least provide this version number the process can stop right away.

If an interface changes for reasons other than correcting mistakes the old interface should be available.

One way to achieve this was mentioned above when the design of the library interface was discussed. By adding version information to data structures or passing version information as additional parameters to functions the library functions can implement a different behavior depending on the version information provided by the caller.

If this kind of interface is not possible or desirable changes must be handled differently. On systems with the GNU C library the symbol versioning mechanism can used. While the use of versioning to handle new interfaces just described is available also on other platforms such as Solaris the mechanisms we describe now are only available with the GNU C library.

The key is that a DSO can export the same interface with different version numbers. I.e., a function `foo` can be contained in a DSO in two versions with completely different implementations. For details on how to use this mechanism consult the description of symbol versioning. Section 4 provides a reference to the document.

Symbol versioning allows to cope with most kind of interface changes without jeopardizing compatibility. There are limitations, though. Different objects (the application, the DSOs) might use different versions of the interfaces which are incompatible. This can create problems is references to objects are passed between these components of the application. It is not easy or not possible to deal with these kind of situations automatically. These situations occur almost never and most can be resolved by relinking the application of DSO dependency in question.

Because changes in the interfaces can have such a dramatic effect it should not be necessary to stress the importance of testing. Testing is always important but especially when writing libraries since the effects are not local to the application one defaults and has control over. Instead a change will affect all the users of the library.

Ideally all aspects of the documented interface of a library should be documented. At least new tests should be added if an interface must be changed to ensure that the old behavior is still available and the new differs in just the right way.

Building a regression test suite is essential for a stable library API. The interdependencies of a non-trivial library are generally too complicated for any programmer to determine the ramification of change at the same time it is made. Good practice is that before the change is made a test case for the new behavior is made and that it fails in the correct way for the old implementation. In addition there should be a test for the current behavior and it should of course succeed with the old implementation. This will allow to verify whether the change was successful right away and will help to detect side effects of other changes.

4 References

The GNU C Compiler documentation contains detailed descriptions of the use of `asm` statements. The manual of version 3.1 and up include a description of the visibility attributes and how gcc implements them.

The GNU linker manual describes the format of linker maps and the effect of the linker. For a detailed description of the symbol versioning technique read my documentation of symbol versioning
(`http://people.redhat.com/drepper/symbol-versioning`).

The GNU C library build process relies heavily on symbol versioning at provides a complete example on how to use this technology.

As one example for a technique to implement dramatic interface changes read the official LFS document
(`http://www.unix-systems.org/version2/whatsnew/lfs20mar.html`) describing the large file support implemented on most modern Unix systems.