Making C++ Loadable Modules Work

Frank Pilhofer
fp@informatik.uni-frankfurt.de

October 9, 1998

1 Introduction

A question frequently popping up on comp.lang.tcl is why loadable modules written in C++ "won't work". Sometimes, they refuse to link, others refuse to load, and some link and load, but then crash for no apparent reason.

Although you will find some recepies at the end of this document, most text is devoted to explain the problems and potential solutions. The author feels that a solid background is worth more than short, unexplained hacks. This way, you can continue experimenting on your own if the recepies don't apply for you or don't work out.

In the past, some people have suggested that simply recompiling Tcl with a C++ compiler helps. There are indeed configurations where this might help, but they are few, for more or less easy explainable reasons. In most cases, recompiling and replacing your current tclsh is not necessary.

But be forewarned – there is just no solution that works everywhere. On modern ELF-based systems (Linux, SunOS 5.x, Digital Unix, etc.), it is reasonably easy to build a C++ loadable module. If you can limit yourself to one of these systems, you probably won't need to read further. Other systems, however, have their own gotchas. This text should help you to make your modules work on just about any individual system, but expect to run into major difficulties when trying to make your module work portably.

Unfortunately, understanding what happens when you load a C++ module and where problems can arise requires some looks behind the scenes, stuff you ordinarily wouldn't want to know about. If you feel you aren't up to that, you can avoid all this by dropping loadable modules and building a custom tclsh that includes your package statically. Yes, that is unsatisfactory, but works almost everywhere, reliably.

This raises the question when loadable modules are useful, required or rather - due to the difficulties - a nuisance. For example, when you're writing a package that is only used internally in one or two very special applications, it's probably not worth spending the time to make your package load dynamically. However, when you have a middleware package that is widely distributed and supposed to coexist with other packages in non-related applications, having your package as a loadable module makes things much easier for the application programmer.

Things get easier when you're limited, for example by available third-party libraries or tools, to a small range of platforms where you can add special cases to your configuration for each of them.

2 Adam and Eve

Think of what happens when you write, compile, link and run a simple C program:

```
int
main (int argc, char *argv[])
{
  printf ("Hello World");
}
```

The compiler realizes that no implementation for `printf()` is available in this compile unit (read: this .c file), and therefore produces an entry in the object file's symbol table saying that `printf()` is an "unresolved reference."

The linker sees this unresolved reference and browses the available libraries for an implementation of `printf()`. This is found in the C library. Now, the linker has two options:

- The linker can take the `printf()` implementation from the library and copy it into the final program. The linker will then search the `printf()` implemenation for other unresolved references, and again consult the libraries for resolution. This process is performed iteratively until no unresolved references remain.

  This is known as *static linking.*

- If the C library is realized as a "shared library," the liker can simply put a reference to the C library into the final program. Still, the linker performs symbol resolution checking as above, to determine if the reference to the `printf()` function necessiates further references to other (shared or nonshared) libraries (like the math library).

  This is known as *dynamic linking.*

What happens when you run the program depends on whether it was linked statically or dynamically:

- A statically linked program is self-contained. It is loaded into memory. The entry point, whose designation is system-dependent (for example, the '`__main`' symbol) is found and called. This entry function, usually provided by the compiler or a library, performs some setup and initialization and then calls the user-defined *main()* function.

- In a dynamically linked program, after loading the program itself into memory, the *dynamic linker* springs into action first. It reads the library references to dynamic libraries produced by the linker, and loads them into memory, too. It then performs symbol resolution again and updates all references to symbols in the shared library to point to their actual location – which can only be determined at runtime, because the shared libraries might be loaded to different memory locations each time the program is run.

The dynamic linker also has the option to abort the program if the dynamic linking fails – for example if the shared library has been modified, and a reference known to the linker now isn't available any more.

You can check what dynamic libraries a program depends on by using

`dump -H <program>`	AIX
`chatr <program>`	HP-UX
`ldd <program>`	Linux, SunOS 5.x

At linking time, some systems encode the full path name to the required shared libraries in the program, others only use the file name without path information. The runtime linker locates shared libraries by checking some common directories and with the help of an environment variable,

`$LIBPATH`	AIX
`$SHLIB_PATH`	HP-UX
`$LD_LIBRARY_PATH`	Linux, SunOS 5.x ...

which contain a listing of directories containing shared libraries, separated by a colon ':' (precisely like `$PATH`).

3 Shared Libraries

Usually, shared libraries must be compiled in a special way, for *position-independence* (PIC). Because a shared library may be mapped into memory at different locations each time it is loaded, addresses within the library would be variable. Therefore, all addresses inside the library are stored zero-based, relative to the beginning of the library. A register is then set aside to contain that address, so that all references in the library can be easily computed at runtime.[1]

To compile a file for position-independence, you have to add custom switches to the compiler's command line. For example, gcc/egcs uses the '`-fPIC` switch.

This is also a frequent source of errors: Using non-PIC code in a shared library does not always cause an error. It even works sometimes, but may cause weird problems on occasion. However, it is never a problem to use PIC code outside a shared library.

When linking a shared library, symbol resolution is performed as with an executable program, and the linker again puts references to necessary libraries into a table within the shared library, for use by the dynamic linker.

Compilers also always implicitly add a number of libraries to the linker command line, hidden to the user. One candidate is, of course, the C library, which you would otherwise have to specify manually ('`-lc`') – which you don't. But most compilers also use custom libraries, which contain implementations of functions used in compiler-generated code. For example, gcc/egcs have the '`libgcc`' library that is always included in linking operations.

This is why it is a bad idea to link a shared library with ld. Always link a shared library with the compiler you used to compile the object files if possible, which then only acts as a frontend to the linker but adds the necessary libraries and linker flags.

[1] On RS/6000 and PowerPC machines running AIX, all code is PIC by default.

You can see which libraries are implicitly linked by adding '-v' to the compiler's command line when linking.

4 Tcl loadable modules

Modern systems allow you to load shared libraries into memory yourself. You can then read their symbol table, locate symbols, and access data and code. See the man pages for the system calls `dlopen()` (most systems) or `shl_load()` (HP-UX). This process is named *runtime linking*.

This is how Tcl loadable modules are realized. When executing the Tcl command

```
load ./library.so
```

the Tcl interpreter maps the library into memory, and searches its symbol table for the "Library_Init" code symbol. This function is then invoked, which in turn registers its functions with the Tcl interpreter.

So far, so good. Now let's try to do the same with C++ code.

5 Using C++ code

Some semantics of C++ code are different. C++ supports function "overloading", using multiple functions with the same name but different parameter lists. In C, this was not possible, so a function name could be used as a unique identifier in the symbol table. But in C++, the function name is not unique. To obtain a unique identifier usable for the linker, C++ function names are *mangled* with information about their parameter list. For example, using gcc, a function '`foobar`' with a single integer parameter becomes '`foobar__Fi`'.

Because the name mangling scheme is compiler-dependent, it is usually not possible to link together code produced by different C++ compilers. Sometimes, the name mangling even differs between compiler releases, for example from gcc 2.7 to 2.8 – old libraries would have to be recompiled to be linked against code compiled with the new compiler.

This is important if you have to work with legacy libraries that you don't have source-code to – then you're stuck with the old compiler.

Another difference in C++ code that you usually don't notice is argument passing in function calls. C pushes arguments onto the stack right-to-left (the leftmost argument is topmost on the stack) – C++ uses right-to-left, which supposedly saves a few machine-language instructions. This is termed as *linkage*, therefore, we can speak of a function having C or C++ linkage.[2]

When mixing C and C++ code, you have to be aware of these differences and adjust your code accodingly. For example, Tcl is written in C. Now, when it invokes functions in a loadable module, it expects them to use the C calling convention, too. This is why you have to put these functions in an `extern "C"` block:

[2] As a side note, variable argument lists require right-to-left argument passing, so whenever declaring a varargs C++ function, it implicitly uses C linkage.

```
extern "C" {
int My_Tcl_Function (ClientData cd, Tcl_Interp *interp,
                     int objc, Tcl_Obj *CONST objv[])
{
  ...
}
}
```

The `extern "C"` instructs the C++ compiler to use C linkage for `My_Tcl_Function`, meaning that it will expect its arguments right-to-left, and that the function name will not be mangled.

The entry function that Tcl searches for in a loadable module must also be of C linkage. Argument ordering doesn't matter, since the function only takes one parameter (a `Tcl_Interp` pointer), but its name must not be mangled, or Tcl won't find it.

Note that `extern "C"` must not only be used with the function body, but also with function declarations, if present. This is why header files for C modules are usually wrapped in

```
#if defined(__cplusplus)
extern "C" {
#endif

/* decls */

#if defined(__cplusplus)
}
#endif
```

Unfortunately, the linkage specification is usually handled incompletely by C++ compilers. For example, it should be possible to declare a static function with C linkage:

```
extern "C" {
static int My_Tcl_Function (...) { ... }
}
```

Yet, some compilers complain about this – according to the standard – perfectly legal construct and ignore the one or the other linkage specification. You will have to hope that the 'static', not the `extern "C"` is ignored.

6 Unresolved References

One of the functions you are probably using in C++ code is the '`new`' operator. Because it is not a trivial function, the compiler does not generate inline code, but a call to an external function. For g++/egcs, this function is named '`__builtin_new`', for HP-UX CC, it is '`__nw__FUi`', etc.

This function is then provided by an external library. For g++/egcs, this is the '`gcc`' library.

However, because the Tcl core does not use the new operator itself, the function is not available, and the dynamic linker will complain about the "unresolved reference" when loading the module. You will have to make the function available yourself, or provide an appropriate reference.

- Some C++ compilers, for example the HP-UX CC compiler, provide their internal functions as a shared library, in this case in the 'C' library. To provide the dynamic linker with the required information, all you have to do is to link your shared library with the '`-lC`' option.

- A bad example is gcc 2.7.x, where the internal functions in the '`libgcc.a`' library (located in the same directory as the '`specs`' file, see the output of '`g++ -v`)'. This is a static library!

  When linking your module, the linker will copy the implementation of '`__builtin_new`' from libgcc into the module, as with static linking. However, because the function has not been compiled for position-independence, it is illegal in a shared library, and the the linker will complain.

- gcc 2.8.x and egcs are smarter. They still use a static library to contain their internal functions, but now, the functions have been compiled for position-independence. Still, the '`__builtin_new`' implementation will be copied into your module, resulting in slightly larger code, but the resulting module can be loaded dynamically.

  Why is libgcc not a shared library? The authors give compatibility as a good reason – if you installed a new compiler with an incompatible '`libgcc.so`', all programs and shared libraries compiled to use the old library would break. So actually, this solution is valid and quite intelligent.

So, making a C++ loadable module is possible if your compiler either uses the first approach and provides its internal functions in a shared library, or the third approach of gcc 2.8.x and egcs.

There are hacks to make making C++ loadable modules possible with gcc 2.7, too, but they are ugly: You could force the static linker to add all necessary code of all potentially relevant internal functions when linking the Tcl core, you can sit down and re-implement the necessary functions yourself, or you can recompile libgcc. However, it is much cleaner to dump gcc 2.7 and switch to gcc 2.8 or egcs.

The '`new`' operator only served as a placeholder in this section. The same is true of various other internal functions that are not used by Tcl but by the module – for g++, this also includes exception handling and some math functions. In each case, you will have to determine where the function is supposed to come from – in more serious cases, this might mean examining all potentially relevant libraries with 'nm'.

It is also sometimes useful to link your module statically into a custom Tclsh. If that build succeeds, you can monitor the linker by adding the '`-v`' option to the compiler command line that ultimately invokes the linker. Seeing what objects and libraries are used there might help.

7 "All right, but I still can't use iostreams!"

That's because iostreams and other classes are provided by yet another library. If you use g++, these are provided by the '`stdc++`' library, which is again linked in implicitly without manual intervention.

If you are lucky, your system has a shared `libstdc++` library, and all you have to do is to specify the right options to the linker.

However, if your `libstdc++` is not shared, you're in bad shape. You have the options of recompiling the `libstdc++` library yourself into a shared library, or to force-link all of the `libstdc++` library into a custom tclsh.

This would be done by extracting all modules from `libstdc++.a` (or whatever name that library has on your system) and then adding all object files to the linker command – selective linking is only done with libraries, not with object files:

```
ar xv /where/ever/libstdc++.a
gcc -o mytclsh tclAppInit.c *.o
```

Then, all iostreams functions can be resolved by the Tcl core program, at the cost of expanding it by several hundred kilobytes.

By the way, when installing g++, egcs or the separate `libstdc++` from source, you must add the "enable-shared" option to `./configure`, or the installation will not even *try* to build a shared library.

When trying to use iostreams, you must also make sure that global constructors are handled properly, because for example '`cout`' is a global symbol that needs construction. If global constructors do not work, accessing '`cout`' is likely to crash.

8 Global Constructors

Another troublesome issue are constructors for global objects in shared libraries. Global objects must be initialized before any function in the library is entered. However, many systems don't do this properly. This means that the objects would not be initialized, and performing any operation on them may produce unexpected results.

The ELF standard[1] takes the special requirements of C++ code in shared libraries into account. For the interested, an ELF shared library contains the special symbol '`_DYNAMIC`', which points to a structure containing various information about the library. One of the entries in the structure is of type '`DT_INIT`', which points to a function to be called by the dynamic/runtime linker when the library is loaded.

Still, the initialization function must be properly constructed by the compiler or linker to invoke the necessary constructors.

This is why you must use your C++ compiler as a frontend to the linker, because only the C++ compiler knows that the code to be linked is actually C++ code, and that it might be necessary to construct an initialization function.

It has also been mentioned that static objects inside a function may be a problem, but this is false. Initializing local static objects has not been a problem on any system. According to the C++ standard, ANSI states that such objects are initialized when first entering the function. This means that the necessary call to the constructor is part of the function's code and not part of a global initialization.

It is interesting to learn how g++ handles global constructors on non-ELF systems. It uses one of the following two approaches.[3]

8.1 g++ and `_GLOBAL__DI`

This approach is used for example on HP-UX. For each global constructor, g++ produces a code fragment at compilation time whose name starts with '`_GLOBAL_.I.`' (use '`nm`' on an object file to see them). Then, when linking the shared library, g++ first invokes the '`collect2`' program, which searches all object files and libraries for such symbols. It then produces a C source file declaring a function '`_GLOBAL__DI`' which calls all required constructors.[4]

G++ then compiles the file produced by `collect2`. This is why you must add '`-fPIC`' to the command line when linking the library – otherwise, that file would end up with non-PIC code, causing the linker to complain!

This way, all initialization is "compacted" into the single function '`_GLOBAL__DI`', all that is left to do is to arrange for this function to be called when the library is loaded.

However, HP-UX does not provide a generic mechanism to do so. It's no problem in programs compiled and linked with g++ themselves, because when linking a main program, `collect2` again takes care to produce a global initialization function that is then called from g++'s custom setup code (for example in `__main`, before invoking your `main()`).

It becomes a problem with runtime linking, because `shl_load()` does not know about this initialization procedure. As a hack on HP-UX, you can pass the '`-Wl,+I -Wl,_GLOBAL__DI`' options to the linker that cause the `collect2`-produced function to be invoked as we want it to be.

The same problem exists on other systems than HP-UX, for example on AIX. Alternatively to the above linker option, which does not exist on other systems than HP-UX, you could either

- patch the Tcl core to look up the `_GLOBAL__DI` symbol in a loaded module and call it
- invoke `_GLOBAL__DI` in your own `Module_Init()` function.

Both options are specific to g++/egcs on non-ELF systems. The first one is harmless with other configurations, but it is understandable that the Tcl developers don't want to add this hack. The second option would break your code (a) on other compilers, and (b) on ELF systems, where such a symbol doesn't exist, causing a linker error.

[3] That's one advantage of g++: it's much more well-documented than native compilers.

[4] You can see this file by linking with g++ using the '`-Wl,-debug`' option.

8.2 g++ and __CTOR_LIST__

This mechanism is similar to the above, and is used for example on Ultrix. No initialization function is produced, but rather, a static array containing function pointers to constructors named '__CTOR_LIST__' is generated.[5] As above, this is performed at link time by the `collect2` program, which examines all code for constructors. A short C program is produced, which is then compiled and linked as before.

If the main application is linked with g++, the gcc-provided '`__main`' takes care to walk through the list at startup and thus performs all necessary initialization.

Again, if the application is not linked with g++, you will have to walk the list yourself.

8.3 Global Destructors

The same problem as with global constructors exists with global destructors. This is usually non-critical, because (a) shared libraries are usually not "detached" until the process exists, (b) at that point, cleaning up memory is good style but not really necessary, as the process's memory is claimed back anyway, and (c) even all open files are closed properly by the system.

Still, a good programmer should also make sure that destructors are called if possible. But this is not so easy as Tcl does not provide a mechanism for module unloading.

Again, this is not a problem on ELF systems, where destructors are called automatically.

On non-ELF systems, when using g++, there exist two mechanisms equivalent to the above:

- On HP-UX (and similar systems), you would have to arrange for the '`_GLOBAL__DD`' function (again produced by `collect2`) to be called. Both functions keep a counter, so that constructors and destructors are called at most once. '`_GLOBAL__DD`' must be called at least as often as '`_GLOBAL__DI`' in order to call destructors.
- On Ultrix (and similar systems), `collect2` produces a table of function pointers called '`__DTOR_LIST__`' that you can walk through.

9 Further problems

There are two more issues that can cause problems: templates and exceptions.

Templates are usually declared in a header file. Each time a template is used, code is generated to instantiate the template with the desired parameters. Most C++ compilers handle code generation individually for each compile unit, meaning that in every compiled file, the required code for templates is reproduced.

Now that is a waste of space, since you will use the same template over and over again in different files, often with the same template types. The generated code would be duplicated.

[5]The first element of this array is not a function pointer but a system-dependent value. It must be ignored.

Some compilers work with a "template repository." No template code is generated at compile time, the compiler just remembers where the template code came from. Then, at link time, as the compiler/linker puts all parts together, it notices which templates actually need to be generated. This code is then produced, compiled, and linked into the application.

This is a problem with shared libraries, where no actual linking takes place. If your compiler keeps a template repository, you must make sure that template code is generated when producing the shared library. For example, when using aCC on HP-UX, this can be done by "dummy-linking" your files. This will fail because of a missing `main()` function, but generates the necessary template instantiations.

As for exceptions – well, there are some configurations where exceptions don't work in shared libraries, like on HP-UX using g++ or egcs, where exceptions are realized using setjmp() and longjmp(), which don't work from inside a shared library. This is a known bug in g++/egcs.

10 System-specific Notes

Here are a some comments on how to make a C++ loadable module using various configurations. A "configuration" depends on the operating system version and the C/C++ compiler version. We imagine that we have a single source file '`module.cc`' that we want to build into a loadable module.

Note that "g++" refers to both g++ 2.8.x or egcs. As explained above, g++ 2.7.x is broken with respect to creating shared libraries. Don't use it, upgrade to g++ 2.8.x or to egcs.

See figure 1 for a small module that we want to load dynamically. It simply prints "Hello World" when loaded into Tcl. We also create a global object, which should print "Global constructor okay" upon construction.

The body of `Module_Init` contains two sections that can be compiled in conditionally. They perform initialization as described in section 8. The first one calls the initialization function produced by `collect2`, the second walks the generated table of constructors.

10.1 ELF systems (SunOS 5.x, Linux, Digital Unix), g++

This configuration is the easiest – C++ loadable modules are easy to build and work as expected, including global constructors. Just do

```
g++ -fPIC -c module.cc
g++ -fPIC -shared -o module.so module.o
```

10.2 HP-UX (generic)

When loading a loadable module fails because of unresolved externals, the runtime linker only reports that there is a problem, but does not list the missing references. During debugging, you might want to hack the Tcl core to change this behaviour. You can change

```
#include <tcl.h>
#include <stdio.h>

class A {
public:
  A (const char *);
};

A::A (const char * msg)
{
  printf ("%s\n", msg);
}

A a ("Global constructor okay.");

extern "C" {
int
Module_Init (Tcl_Interp * interp)
{
  printf ("Hello World\n");

#ifdef CONSTR
  extern void _GLOBAL__DI ();
  _GLOBAL__DI ();
#endif

#ifdef CTORLIST
  typedef void entry_pt();
  extern entry_pt * __CTOR_LIST__[];
  entry_pt ** iter = __CTOR_LIST__;
  for (++iter; *iter; iter++) {
   (*iter) ();
  }
#endif

  return TCL_OK;
}
}
```

Figure 1: A small sample Module

```
handle = shl_load (fileName, BIND_IMMEDIATE|BIND_VERBOSE, 0L);
```

(note the added BIND_VERBOSE flag) in unix/tclLoadShl.c to get a list of missing externals upon a failing 'load'.

10.3 HP-UX 10.x, g++

On HP-UX, the collect2-generated initialization function is not called by default. We therefore enable our hack to call the init function ourselves.

```
g++ -fPIC -DCONSTR -c module.cc
g++ -fPIC -shared -o module.sl module.o
```

As an alternative to calling _GLOBAL__DI() explicitely, you can also use the linker flag

```
g++ -fPIC -shared -Wl,+I -Wl,_GLOBAL__DI -o module.sl module.o
```

10.4 HP-UX 10.x, aCC

"aCC" refers to HP's Ansi C++ compiler, which is usually installed in /opt/aCC/bin if available.

Our module will only work if the main function, meaning the one from tclsh, is compiled and linked with aCC – otherwise, you will get unresolved symbols when loading a module. There is no need to rebuild Tcl, just use the file tclAppInit.c that is installed in the Tcl library directory, and do

```
/opt/aCC/bin/aCC -c tclAppInit.c
/opt/aCC/bin/aCC -o mytclsh tclAppInit.o -ltcl8.0 -lm
```

Note that you must edit tclAppInit.c (which still uses old K&R-style function declarations) so that it compiles as C++ code.[6]

Afterwards, you can build the module using

```
/opt/aCC/bin/aCC -c +Z module.cc
/opt/aCC/bin/aCC module.o || /bin/true
/opt/aCC/bin/aCC -b +Z -o module.sl module.o
```

Note: The '-Z' option is used to generate PIC code, and '-b' instructs the linker to produce a shared library.

Since aCC keeps a template repository, the second line is introduced to "dummy-link" the module, which generates required templates. Since this linking operation fails (because of a missing main function), the error is ignored.

[6]Unfortunately, it is not enough to *link* the application with aCC; if you compile tclAppInit.c with c89 or gcc and just do the linking with aCC, constructors won't work.

10.5 HP-UX 10.x, CC

"CC" is an older incarnation of "aCC". Consequently, it works similar, but it is not as capable as aCC. Again, we must recompile and relink Tcl with CC:

```
CC -c tclAppInit.c
CC -o mytclsh tclAppInit.o -ltcl8.0 -lm
```

Then, we compile the module as before:

```
CC -c +Z module.cc
CC -b +Z -o module.sl module.o
```

However, global constructors will *not* work!

10.6 AIX 4.2 and above, g++

This configuration gave me some headache. For some reason, the dynamic linker refuses to load `libstdc++` at runtime and aborts with "Exec format error." However, it does not have any problem loading *applications* that are dynamically linked against `libstdc++` and can then continue to load our module just fine.[7]

Therefore, we must again compile our own `tclsh` and make sure it is linked against `libstdc++`.

```
gcc -c tclAppInit.c
g++ -o mytclsh tclAppInit.o -ltcl8.0 -ldl -lm
```

Our module can then be compiled using

```
g++ -c -DCONSTR module.cc
g++ -shared -o module.so module.o
```

As before, we must call the initialization function ourselves, because AIX's `dlopen()` system call does not know about it. Linking the shared library will produce some warnings about duplicate symbols, which can be ignored.

Note that all code on AIX is PIC by default, so there's no need to use '`-fPIC`'.

10.7 AIX 4.2 and above, xlC

On this configuration, making the loadable module is straightforward:

```
xlC -c module.cc
/usr/lpp/xlC/bin/makeC++SharedLib -p 0 -o module.so module.o
```

The '`-p 0`' switch sets the shared library's "priority".[8]

There seems to be a bug with certain versions of xlC.[9] Version 3.1.4.0 has been seen to fail (loading the module fails with "exec format error"), while version 3.1.4.7 seems to work.

[7] The Author welcomes any explanations and solutions.
[8] What's that?
[9] You can check the compiler version using '`lslpp -L xlC.C`'.

10.8 AIX 3.x up to 4.1, xlC

AIX versions prior to 4.2 used a different mechanism for runtime linking. They used the `load()` system call before switching to the more widely accepted `dlopen()` mechanism. Tcl comes with a library that emulates `dlopen()` and friends on these systems. It appears that this library, contributed by Helios Software, is buggy. The following text was contributed by Paul Duffin:

There is a bug in `tclLoadAix.c` which does need fixing. The solution is to use `makeC++SharedLib` (in `/usr/lpp/xlC/bin`) to link the modules together into the shared object as opposed to the normal `ldAix`. This script uses various tools to collect all the information about the constructors and generate some special code to allow them to be initialised, this uses a priority mechanism to make sure that the constructors are called in the correct order. If you have a look in the `tclLoadAix.c` file you will see some references to `__cdtors`. The C definition of this symbol would be as follows.

```
typedef struct {
  void (*init)(void);     /* call static constructors */
  void (*term)(void);     /* call static destructors */
} Cdtor, *CdtorPtr;

static Cdtor __cdtors [] =
{
  { ..., ... },
  { ..., ... },
  { ..., ... },
  { 0, 0 }
};
```

To call the constructors you simply have to walk through the table calling the init functions. To call the destructors you simply have to walk backwards through the table calling the term functions.

However there is a problem in that the first element of the array is used to store the priority of the shared library so that shared libraries can be initialised in the correct order. This causes Tcl to crash as it does not take that into account.

The two functions `loadAndInit` and `terminateAndUnload` are provided to do this automatically for you, but unfortunately these are only available in the C++ library so you would have to build a `tclsh/wish` which linked to that.

10.9 Ultrix, g++

Ultrix represents the systems that don't know about shared libraries and dynamic linking. On the Tcl side, loadable modules are handled in a very ugly manner, by calling the linker on its own at runtime to relocate the code.

For starters, we can build the module as:

```
g++ -G 0 -c module.cc
echo tclLdAout gcc {-G 0} | tclsh8.0 -r -G 0 -o module.a module.o -lc
```

Since Ultrix does not know about shared libraries, the '-fPIC' option does not work here. Instead, you must use '-G 0'.

One drawback is that using iostreams, or anything else from libstdc++ will not work, because that library was compiled without '-G 0'. If you require libstdc++, it must be recompiled with this option.

Another problem is that we can't let g++ perform the linking (because g++ does not know how to generate shared libraries on Ultrix, because they don't exist). Therefore, collect2 will not be used to produce an initialization function. Without further efforts, global constructors will not work.

It is actually possible to make global constructors work, but it's not for the faint-hearted. What we need is the table of constructors that collect2 *would* produce, had it been called. What we do is to actually build an application containing our module. Here's our dummy 'main.c':

```
int main () { return 0; }
```

We then link the application and "steal" the file that collect2 produces, which is printed among other information on stderr when given the '-Wl,-debug' option. We use gawk[10] and grep to extract the source file:[11]

```
g++ -Wl,-debug -o foobar main.c module.o 2>&1 | \
  gawk '/===.*c_file/,/===.*end of c_file/' | \
  grep -v '^===' | grep -v __main > ctor.c
```

This produces the file 'constr.c' containing the definition of the '__CTOR_LIST__' and '__DTOR_LIST__' function pointer tables, which we can compile into our module and traverse at startup ourselves:

```
g++ -G 0 -c -DCTORLIST module.cc
gcc -G 0 -c ctor.c
echo tclLdAout gcc {-G 0} | \
  tclsh8.0 -r -G 0 -o module.a constr.o module.o -lc
```

Here, when compiling the module, we enable our code to walk the list of constructor functions at startup.

11 Epilogue

This author of this text does not consider himself an expert on the topic of shared libraries. I rather try not to give up on a problem until it is solved. What I know, I have learned from experience.

Much experience was gathered while tryint to get "Tclobj" to run on different systems. Although I now consider Tclobj, a package to integrate C++ classes and objects into Tcl, obsolete (SWIG is easier to use), I continue to have similar problems with my current

[10]Use GNU awk, Ultrix's standard awk doesn't work here.

[11]The redirection of stderr to stdout requires Bourne shell.

"Tclmico" package, a Tcl interface to the free CORBA implementation MICO (which is written in C++).

Besides reading this text, you should consult your system's developer's documentation. The '`ld`' manual page is also likely to contain valuable information.

The above configurations represent the systems the author has access to. As you can see, each system has its own individual quirks. If your configuration is not among them, you can try to see which approach comes closest, and then start experimenting from there on.

References

[1] Lu, Hongjiu, "ELF: From The Programmer's Perspective"
`ftp://sunsite.unc.edu/pub/Linux/GCC/elf.latex.tar.gz`
`http://www.debian.org/Documentation/elf/elf.html`