In either case, the lack of an ABI is obviated by consistent use of a set of tools to build both libraries and executables.

Table 3/1.1.2a Static linking compatibility (C code)

Compiler (across using libraries made by down)	Borland	CodeWarrior	Digital Mars	GCC	Intel	Visual C++
Borland		X	X*	X	X	X
CodeWarrior	X		X		X	X
Digital Mars	X	X		X	X	X
GCC	X		X			
Intel	X		X			
Visual C++	X		X			

* Note that DMC++ will actually build an executable with the Borland library, but it is not a valid image. The DMC++ linker gives the strong hint that FIXLIB is needed [NOTE TO SELF: FIND OUT WHAT FIXLIB IS]

Table 3/1.1.2b Static linking compatibility (C++ code)

Compiler (across using libraries made by down)	Borland	CodeWarrior	Digital Mars	GCC	Intel	Visual C++
Borland		X	X	X	X	X
CodeWarrior	X		X		X	X
Digital Mars	X	X		X	X	X
GCC	X	X	X	X	X	X
Intel	X		X	X		
Visual C++	X		X	X		

1.1.3 Dynamic Linking

Modern operating systems, and many modern applications, make use of a technique known as dynamic linking. In this case, you link against the library in a similar way, but the code is not copied into the finished executable. Rather, entry points are recorded, and when an executable is loaded by the operating system, the dynamic libraries on which it depends are also loaded, and the entry points altered to point into the actual code points within the dynamic library in the new process's address space. On Win32 systems, the creation of a dynamic library is usually accompanied by the generation of a small static library, known as an export library, which contains the code that the application will use to fix up addresses when the dynamic library is loaded. Executables are linked against such export libraries in the same way as they are with normal (static) libraries.

An advantage to using dynamic linking is that demands on disk space and operating system working sets are reduced, because there are not duplicated blocks of code spread throughout several executable files, or in several concurrently executing processes. They also mean that fixes can be effected without requiring any rebuilds. Indeed, where the libraries are part of the operating system, such updates can be done without the program vendors, or even the users, being aware of it.

Naturally, there are downsides to using shared libraries, the so-called "DLL Hell" [RICHT, XXX], but the advantages generally outweigh such concerns, and it is hard to conceive of a move back to pure static linking.

The impact of dynamic linking on C++'s absent ABI is significant. We've seen how we could avoid the problems of clash of different mangling schemes in static linking just by providing compiler-specific variants of our libraries against which clients can build their executables. But dynamic libraries are potentially shared at runtime between several processes, which may or may not be built with different compilers. If the symbol names in the dynamic library are mangled with a convention not understood by the compiler used to create another process, that process will not load.

So, one answer to this is to produce several dynamic libraries, each with the appropriately mangled named. We could envisage shipping `mylib_dmc_intel_vc.dll`, `mlib_gcc.dll`, `mlib_bland.dll`, etc. But would you want to do that?

There are several problems with this. First, it defeats both purposes of DLLs, in that more libraries will be resident on disk and in memory, and updates to the library must be consistently built and installed in all forms. Also, some dynamic libraries act as more than just code repositories. They are also allowed to contain static data (and we'll touch more on the ramifications of this in the next chapter), which can act to provide program logic, e.g. a custom-memory manager, or a socket-pool. If there can be several compiler-specific versions of the same logical library, things could get very hairy.

So although it is feasible to get around the dynamic-library C++ ABI problem by supplying multiple compiler-specific dynamic libraries, it is frought with problems, and I am not aware of any systems that actually do so.

So far, we seem to be agreeing with the portents of doom outlined by my friend.

Which compilers produce the same DLL mangling?
Talk about incompatible .lib formats.

There's another issue with dynamic linking. On Win32 platforms, the available symbols in a dynamic link library (DLL) are stipulated in an exports section, and are referenced by either ordinal, or by name.

Table 3/1.1.3a Dynamic linking compatibility (C code, StdCall)

Compiler (across using libraries made by down)	Borland	CodeWarrior	Digital Mars	GCC	Intel	Visual C++
Borland		X	X	X	X	X
CodeWarrior	X			X		
Digital Mars	X			X		
GCC	X	X	X		X	X
Intel	X			X		
Visual C++	X			X		

Table 3/1.1.3b Dynamic linking compatibility (C code, CDecl)

Compiler (across using libraries made by down)	Borland	CodeWarrior	Digital Mars	GCC	Intel	Visual C++
Borland		X	?	X	X	X
CodeWarrior	X		?			
Digital Mars	X		?			
GCC	X		?			
Intel	X		?			
Visual C++	X		?			

1.2 I can C clearly now

In this section we're going to see how we can get around these problems a little. First we're going to look closely at C and C++ compatibility.

1.2.1 extern "C"

Since C++ is a near superset of C, there naturally has to be a way of interacting with C functions. Since C++ mangles every function, in case it is overloaded, there needs to be a mechanism to tell it not to mangle functions that are implemented in C. This is done using `extern "C"`, as in: