# 1.4. Building a Dynamic Library from the Command Line

## Problem

You wish to use your command-line tools to build a dynamic library from a collection of C++ source files, such as those listed in Example 1-2.

## Solution

Follow these steps:

1. Use your compiler to compile the source files into object files. If you're using Windows, use the *-D* option to define any macros necessary to ensure that your dynamic library's symbols will be exported. For example, to build the dynamic library in Example 1-2, you need to define the macro `GEORGERINGO_DLL`. If you're building a third-party library, the installation instructions should tell you what macros to define.

2. Use your linker to create a dynamic library from the object files created in step 1.

ers, and to tell the linker the names of the other libraries and where to find them. This is discussed in detail in Recipe 1.5.

The basic commands for performing the first step are given Table 1-8; you'll need to modify the names of the input and output files appropriately. The commands for performing the second step are given in Table 1-11. If you're using a toolset that comes with static and dynamic variants of its runtime libraries, direct the compiler and linker to use a dynamically linked runtime, as described in Recipe 1.23.

| Toolset | Command line |
|---|---|
| GCC | *g++ -shared -fPIC -o libgeorgeringo.so george.o ringo.o georgeringo.o* |
| GCC (Mac OS X) | *g++ -dynamiclib -fPIC -o libgeorgeringo.dylib george.o ringo.o georgeringo.o* |
| GCC (Cygwin) | *g++ -shared -o libgeorgeringo.dll -Wl,—out-implib,libgeorgeringo.dll.a-W1,—export-all-symbols -Wl,—enable-auto-image-base george.o ringo.o georgeringo.o* |
| GCC (MinGW) | *g++ -shared -o libgeorgeringo.dll -Wl,—out-implib,libgeorgeringo.a -W1,—export-all-symbols-Wl,—enable-auto-image-base george.o ringo.o georgeringo.o* |
| Visual C++ | *link -nologo -dll -out:libgeorgeringo.dll -implib:libgeorgeringo.lib george.obj ringo.obj georgeringo.obj* |
| Intel (Windows) | *xilink -nologo -dll -out:libgeorgeringo.dll -implib:libgeorgeringo.lib george.obj ringo.obj georgeringo.obj* |
| Intel (Linux) | *g++ -shared -fPIC -lrt -o libgeorgeringo.so george.o ringo.o georgeringo.o georgeringo.obj* |
| Metrowerks (Windows) | *mwld -shared -export dllexport -runtime dm -o libgeorgeringo.dll -implib libgeorgeringo.lib george.obj ringo.obj georgeringo.obj* |
| Metrowerks (Mac OS X) | mwld -shared -export pragma -o libgeorgeringo.dylib george.o ringo.o georgeringo.o |

| | |
|---|---|
| CodeWarrior 10.0 (Mac OS X)[4] | Consult the Metrowerks documentation. |
| Borland | *bcc32 -q -WD -WR -elibgeorgeringo.dll george.obj ringo.obj georgeringo.objimplib -c libgeorgeringo.lib libgeorgeringo.dll* |
| Digital Mars | *dmc -WD -L/implib:libgeorgeringo.lib -o libgeorgeringo.dll george.obj ringo.obj georgeringo.obj user32.lib kernel32.lib* |

[4] CodeWarrior 10.0 for Mac OS X will provide dynamic variants of its runtime support libraries; these should be used when building libgeorgeringo.dylib. (See Recipe 1.23.)

## TIP

As of September 2005, the Comeau toolset does not support building dynamic libraries on Unix or Windows. Comeau Computing is currently working on dynamic library support, however, and expects it to be implemented for some Unix platforms — including Linux — by the end of 2005.

For example, to compile the source files from Example 1-2 into object files with the Borland compiler, assuming that the directory containing the Borland tools is in your PATH, change to the directory *georgeringo* and enter the following commands:

```
> bcc32 -c -q -WR -o george.obj george.cpp
george.cpp:
> bcc32 -c -q -WR -o ringo.obj ringo.cpp
ringo.cpp:
> bcc32 -c -q -WR -DGERORGERINGO_DLL -o georgeringo.obj georgeringo.cpp
georgeringo.cpp:
```

The compiler option *-WR* is used here to specify a dynamic variant of the runtime library. These three commands will generate the object files *george.obj*, *ringo.obj,* and *georgeringo.obj*. Next, enter the command:

This will generate the dynamic library *libgeorgeringo.dll*. Finally, enter the command:

```
> implib -c libgeorgeringo.lib libgeorgeringo.dll
```

This will generate the import library *libgeorgeringo.lib*.

## Discussion

How dynamic libraries are handled varies greatly depending on the operating system and toolset. From the programmer's point of view, the two most important differences are as follows:

### Symbol visibility

Dynamic libraries can contain the definitions of classes, functions, and data. On some platforms, all such symbols are automatically accessible to code which uses a dynamic library; other systems offer programmers fine-grained control over which symbols are accessible. Being able to determine which symbols should be visible on a case-by-case basis is generally advantageous; it gives a programmer more explicit control of his library's public interface, and it often provides superior performance. It also makes building and using dynamic libraries more complex, however.

With most Windows toolsets, in order for a symbol defined in a dynamic library to be available to code which uses the dynamically library, it must be explicitly *exported* when the dynamic library is built and *imported* when an executable or dynamic library that uses the dynamic library is built. Some Unix toolsets also offer this flexibility; this is true for recent versions of GCC on several platforms, for Metrowerks on Mac OS X, and for Intel on Linux. In some cases, however, there is no alternative but to make all symbols visible.

### Passing libraries to the linker

On Unix, a dynamic library can be specified as input to the linker when code using the dynamic library is linked. On Windows, except when using GCC, dynamic libraries are not

## Import libraries and module definition files

Import libraries, roughly speaking, are static libraries containing the information needed to invoke functions in a DLL at runtime. It's not necessary to know how they work, only how to create and use them. Most linkers create import libraries automatically when you build a DLL, but in some cases it may be necessary to use a separate tool called an *import librarian*. In Table 1-11, I used the Borland import librarian *implib.exe* to avoid the peculiar command-line syntax required by the Borland linker *ilink32.exe*.

A module definition file, or *.def* file, is a text file that describes the functions and data exported by a DLL. A *.def* file can be written by hand or automatically generated by a tool. An example *.def* file for the library *libgeorgeringo.dll* is shown in Example 1-5.

*Example 1-5. A module definition file for libgeorgeringo.dll*

```
LIBRARY            LIBGEORGERINGO.DLL

EXPORTS
    Georgeringo    @1
```

## Exporting symbols from a DLL

There are two standard methods for exporting symbols from a Windows DLL:

- Use the `_ _declspec(dllexport)` attribute in the DLL's headers, and build an import library for use when linking code that uses your DLL.

  The `_ _declspec(dllexport)` attribute should be inserted at the beginning of the declarations of exported functions and data, following any linkage specifiers, and immediately following the `class` or `struct` keyword for exported classes. This is illustrated in Example 1-6. Note that `_ _declspec(dllexport)` is not part of the C++ language; it is a language extension implemented by most Windows compilers.

*Example 1-6. Using the _ _declspec(dllexport) attribute*

```
_  _declpec(dllexport) int m = 3;      // Exported data definition
extern _  _declpec(dllexport) int n;   // Exported data declaration
_  _declpec(dllexport) void f();          // Exported function declaration
class _  _declpec(dllexport) c {       // Exported class definition
   /* ... */
};
```

Using a *.def* file has certain advantages; for instance, it can allow functions in a DLL to be accessed by number rather than name, decreasing the size of a DLL. It also eliminates the need for the messy preprocessor directives such as those in the header *georgeringo.hpp* from Example 1-2. It has some serious drawbacks, however. For example, a *.def* file cannot be used to export classes. Furthermore, it can be difficult to remember to update your *.def* file when you add, remove, or modify functions in your DLL. I therefore recommend that you always use `_ _declspec(dllexport)`. To learn the full syntax of *.def* files as well as how to use them, consult your toolset's documentation.

## Importing symbols from a DLL

Just as there are two ways to export symbols from a DLL, there are two ways to import symbols:

- In the headers included by source code that uses your DLL, use the attribute `_ _declspec(dllimport)` and pass an import library to the linker when linking code that uses your DLL.
- Specify a *.def* file when linking code which depends on you DLL.

Just as with exporting symbols, I recommend that you use the attribute `_ _declspec(dllimport)` in your source code instead of using *.def* files. The attribute `_ _declspec(dllimport)` is used exactly like the attribute `_ _declspec(dllexport)`, discussed earlier. Like `_ _declspec(dllexport)`, `_ _declspec(dllimport)` is not part of the C++ language, but an extension implemented by most Windows compilers.

be to use two sets of headers: one for building your DLL and the other for compiling code that uses your DLL. This is not satisfactory, however, since it is difficult to maintain two separate versions of the same headers.

Instead, the usual approach is to define a macro that expands to `_ _declspec(dllexport)` when building your DLL and to `_ _declspec(dllimport)` otherwise. In Example 1-2, I used the macro `GEORGERINGO_DECL` for this purpose. On Windows, `GEORGERINGO_DECL` expands to `_ _declspec(dllexport)` if the macro `GEORGERING_SOURCE` is defined and to `_ _declspec(dllimport)` otherwise. By defining `GEORGERING_SOURCE` when building the DLL *libgeorgeringo.dll* but not when compiling code that uses *libgeorgeringo.dll*, you obtain the desired result.

## Building DLLs with GCC

The Cygwin and MinGW ports of GCC, discussed in Recipe 1.1, handle DLLs differently than other Windows toolsets. When you build a DLL with GCC, all functions, classes, and data are exported by default. This behavior can be modified by passing the option *—no-export-all-symbols* to the linker, by using the attribute `_ _declspec-(dllexport)` in your source files, or by using a *.def* file. In each of these three cases, unless you use the option *—export-all-symbols* to force the linker to export all symbols, the only exported functions, classes, and data will be those marked `_ _decl-spec(dllexport)` or listed in the *.def* file.

It's therefore possible to use the GCC toolset to build DLLs in two ways: like an ordinary Windows toolset, exporting symbols explicitly using `_ _declspec`, or like a Unix toolset, exporting all symbols automatically.[5] I used the latter method in Example 1-2 and Table 1-11. If you choose this method, you should consider using the option *—export-all-symbols* as a safety measure, in case you happen to include headers containing `_ _declspec(dllexport)`.

GCC differs from other Windows toolsets in a second way: rather than passing the linker an import library associated with a DLL, you can pass the DLL itself. This is usually faster than using an import library. It can also create problems, however, since several versions of a DLL may exist on your system, and you must ensure that the linker selects the correct ver-

**TIP**

With Cygwin, an import library for the DLL *xxx.dll* is typically named *xxx.dll.a*, while with MinGW it is typically named *xxx.a*. This is just a matter of convention.

## GCC 4.0's -fvisibility option

Recent versions of GCC on several platforms, including Linux and Mac OS X, give programmers fine-grained control over which symbols in a dynamic library are exported: the command-line option *-fvisibility* can be used to set the default visibility of symbols in a dynamic library, and a special attribute syntax, similar to `_ _declspec(dllexport)` on Windows, can be used within source code to modify the visibility of symbols on a case-by-case basis. The *-fvisibility* option has several possible values, but the two interesting cases are *default* and *hidden*. Roughly speaking, *default* visibility means that a symbol is accessible to code in other modules; *hidden* visibility means that it is not. To enable selective exporting of symbols, specify *-fvisibility=hidden* on the command line and use the *visibility attribute* to mark selected symbols as visible, as shown in Example 1-7.

*Example 1-7. Using the visibility attribute with the command-line option -fvisibility=hidden*

```
extern _  _attribute_  _((visibility("default"))) int m;      // exported
extern int n;                                              // not exported

_  _attribute_  _((visibility("default"))) void f();        // exported
void g();                                                     // not expo

struct _  _attribute_  _((visibility("default"))) S {  };      // exporte
struct T {  };                                                // not expo
```

In Example 1-7, the attribute `_ _attribute_ _((visibility("default")))` plays the same role as `_ _declspec(dllexport)` in Windows code.

Using the `visibility` attribute presents some of the same challenges as using `_ _declspec(dllexport)` and `_ _declspec(dllimport)`, since you want the attribute to be

with the preprocessor. For example, you can modify the header *georgeringo.hpp* from Example 1-2 to take advantage of the visibility attribute as follows:

```
georgeringo/georgeringo.hpp

#ifndef GEORGERINGO_HPP_INCLUDED
#define GEORGERINGO_HPP_INCLUDED

// define GEORGERINGO_DLL when building libgerogreringo
# if defined(_WIN32) && !defined(__GNUC__)
#  ifdef GEORGERINGO_DLL
#   define GEORGERINGO_DECL _  _declspec(dllexport)
#  else
#   define GEORGERINGO_DECL _  _declspec(dllimport)
#  endif
# else // Unix
#  if defined(GEORGERINGO_DLL) && defined(HAS_GCC_VISIBILITY)
#   define GEORGERINGO_DECL _  _attribute_  _((visibility("default")))
#  else
#   define GEORGERINGO_DECL
#  endif
# endif

// Prints "George, and Ringo\n"
GEORGERINGO_DECL void georgeringo();

#endif // GEORGERINGO_HPP_INCLUDED
```

To make this work, you must define the macro `HAS_GCC_VISIBILITY` when building on systems that support the *-fvisibility* option.

> **TIP**
>
> Recent versions of the Intel compiler for Linux also support the *-fvisibility* option.

## Symbol Visibility with Metrowerks for Mac OS X

option *-export all*, which is the default when building from the command-line. The method I recommend is to use `#pragma export` in your source code to mark the functions you wish to export, and to specify *-export pragma* on the command-line when building your dynamic library. The use of `#export pragma` is illustrated in Example 1-2: just invoke `#pragma export on` in your header files immediately before a group of functions you want to export, and `#export pragma off` immediately afterwards. If you want your code to work on toolsets other than Metrowerks, you should place the invocations of `#pragma export` between `#ifdef`/`#endif` directives, as illustrated in Example 1-2.

## Command-line options

Let's take a quick look at the options used in Table 1-11. Each command line specifies:

- The name of the input files: *george.obj*, *ringo.obj*, and *georgeringo.obj*
- The name of the dynamic library to be created
- On Windows, the name of the import library

In addition, the linker requires an option to tell it to build a dynamic library rather than an executable. Most linkers use the option *-shared*, but Visual C++ and Intel for Windows use *-dll*, Borland and Digital Mars use *-WD*, and GGC for Mac OS X uses *-dynamiclib*.

Several of the options in Table 1-11 help dynamic libraries to be used more effectively at runtime. For example, some Unix linkers should be told to generate *position-independent code* using the option *-fPIC* (GCC and Intel for Linux). This option makes it more likely that multiple processes will be able to share a single copy of the dynamic library's code; on some systems, failing to specify this option can cause a linker error. Similarly, on Windows the GCC linker the option *—enable-auto-image-base* makes it less likely that the operating system will attempt to load two dynamic libraries at the same location; using this option helps to speed DLL loading.

lowing *W* is a lowercase *l*.)

Most of the remaining options are used to specify runtime library variants, as described in Recipe 1.23.

## See Also

Recipe 1.9, Recipe 1.12, Recipe 1.16, Recipe 1.19, and Recipe 1.23

[5] Confusingly, exporting symbols using `_ _declspec(dllexport)` is sometimes called *implicit* exporting.

SUPPORT

Contact us

Newsletters

Privacy policy

INTERNATIONAL

Australia & New Zealand

Hong Kong & Taiwan

India

Indonesia

Japan

DOWNLOAD THE O'REILLY APP

Take O'Reilly with you and learn anywhere, anytime on your phone and tablet.

WATCH ON YOUR BIG SCREEN

View all O'Reilly videos, Superstream events, and Meet the Expert sessions on your home TV.

DO NOT SELL MY PERSONAL INFORMATION