Building a cross-platform C library

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

This tutorial shows you how to convert an existing C program or module into a shared library. It covers the mechanics of creating a shared library using GNU tools or MSVC, but the major focus is on design issues you will encounter when making a cross-platform shared library. While the mechanics are specific to the C language and the platforms mentioned, the design issues are of general interest.

Unlike most tutorials, we will not provide sample code to generate a toy shared library. Instead, we encourage you to apply the steps of this tutorial to a module or program of your own that you would like to release as a library. Alternatively, you may want to practice on the *spline program* we use in our examples.

In order to profit from this tutorial, you will need a reading knowledge of C and some sympathy for cross-platform problems. You do not need a C compiler or code to convert into a library. However, if you have them, we suggest using them. Also, you may want to download and install the GNU Autotools: Autoconf, Automake and Libtool.

Acknowledgements

We'd like to thank Robert Maier, the author of the original spline program, for allowing us to convert the spline code into a shared library to be released under the GNU LGPL (Lesser General Public License).

Getting help

For technical questions about the content of this tutorial, contact the authors, Danielle Michaels and Samuel Mikes, at *articles@cubane.com* .

Sam and Danielle founded Cubane Software in early 1999. They enjoy pair programming and free software. They live in Santa Monica, California with three cats, six computers and a hefty electricity bill.

Section 2. Platform-independent library design

Introduction

You use them all the time, but have you ever thought about what goes into making a good library? Have you ever thought about how much work the maintainer had to put in for you to be able to "seamlessly" compile your favorite free software library on your particular marginal platform -- or your client's platform du jour?

This tutorial discusses design issues and how to avoid some common cross-platform pitfalls. (We're sure we haven't covered all of them, and very much welcome more tips from you.) Then we'll go through how to actually build a shared library, using GNU tools and MSVC.

Recently, we converted spline.exe, a spline generation Unix utility distributed with GNU plotutils, to be a cross-platform shared library. We'll use examples from this conversion throughout the tutorial. In the design phase, we encourage you to apply the steps in this tutorial to some of your own code that you'd like to turn into a library. When we discuss the mechanics of shared library creation, however, you will probably find it useful to follow along, both with the *original code* and with our gspline library, which you can download as a *.zip file* or as *.tar.gz file* .

What's a spline?

Merriam-Webster's defines a *spline* as: a function that is defined on an interval, is used to approximate a given function, and is composed of pieces of simple functions defined on subintervals and joined at their endpoints with a suitable degree of smoothness

For our purposes, we can define a spline as a fancy smooth curve based on three or more control points and a tension. Splines are used to draw smooth curves through specific points. The tension parameter controls how sharply the spline is allowed to curve. For those who care deeply, 0 tension splines are simple cubic splines and nonzero tension splines obey the equation: *y'''' = t^2 * y''*.

The Jargon File has yet another take on the definition of spline: [XEROX PARC] This phrase expands to: "You have just used a term that I've heard for a year and a half, and I feel I should know, but don't. My curiosity has finally overcome my guilt." The PARC lexicon adds "Moral: don't hesitate to ask questions, even if they seem obvious."

Basic library structure

A library can be divided into two parts: interface and implementation. The interface consists of functions the user wants to call, and structures you want the user to have access to. The implementation consists of things the user doesn't want to have to care about. The trick, when designing a library, is to make the interface rich enough to be useful without exposing so much of the guts that the user is overwhelmed.

Spline is a Unix utility. Input data can be read in a variety of formats from files or standard input, processed in a variety of ways (controlled by command-line arguments), and the results are output on standard output. Errors are written to standard error, or, if serious, terminate the process.

While the command-line interface to spline is flexible, there is only one internal function that generates interpolated spline points (`do_spline`). So one of our challenges in converting the spline utility into a library is to preserve this flexibility when designing the library interface functions.

Choosing a library prefix

One important thing to do before beginning the work of specifying the interface functions is to choose a prefix for your library. This provides good unity among your library interface functions and prevents your function names from colliding with those specified by the user. For example, `do_spline` is an excellent user function name and, precisely for that reason, a terrible library interface function name.

Your prefix should be short and meaningful. You may be inspired by such to-the-point classic prefixes as:

* "str" (string functions of the standard library)
* "std" (header files for standard library)
* "f" (buffered file I/O)
* "X" (X window system calls)

Another consideration is that some C compilers and many FORTRAN compilers only recognize the first six or eight characters of an exported symbol. You should choose the names of your library functions and structures so that they are uniquely distinguished by the first six characters.

We chose "spl" as a prefix to make it all but impossible to pronounce our library functions.

Choosing a library prefix for spline

When you're starting from existing code, your functions might not follow a prefix convention. They might look something like this:

```
/* BAD: names that do not use a library prefix */
 int generate_simple_spline();
 int generate_spline( struct SPLINE * );
 void free_spline( struct SPLINE * );
```

The function names below use a consistent prefix. They will work well on modern systems, and many library authors can stop with a prefix like this:

```
/* BETTER: names that use a prefix, but are not unique in the first 6 characters */
 int spline_generate_simple();
 int spline_generate( struct SPLINE * );
 void spline_free( struct SPLINE * );
```

If your library needs to be callable from old FORTRAN or C compilers, you will need to go with a terser prefix, something like this:

```
/* BEST: names that use a prefix and are unique in 6 characters */
 int splgen();
 int splspl(struct SPLINE * );
 void splfree(struct SPLINE * );      /* names can be longer than 6 characters */
```

Factor your code

Now that we have a prefix, we're ready to separate the interface from the implementation. If you're converting a module into a library, this factoring may already be mostly done. Usually, one already has module interfaces or entry points that are in the public header file, and various supporting functions that are `static` (private) to the module.

If, as we are with spline, you're converting a stand-alone program to a program and a shared library, you may find it useful to think of factoring your code into three parts: interface, implementation, and shell. The shell becomes the program that calls the library and handles the user interface and I/O processing. People who use your library will write their own shell code or they'll be grafting your library to an existing program (and therefore will already have a user interface and other parts).

Factoring spline: before

Here is a list of functions defined in the body of spline.c, the old stand-alone program. (Function arguments have been removed for clarity.) As is common for small C programs, the functions are loosely grouped by purpose and location in the file. The next panel shows how we separated them into interface, implementation and shell.

```
bool do_bessel();
 bool is_monotonic();
 bool read_data();
 bool read_float();
 bool skip_whitespace();
 bool write_point();
 double interpolate();
 double quotient_sin_func();
 double quotient_sinh_func();
 double sin_func();
 double sinh_func();
 double tan_func();
 double tanh_func();
 int read_point();
 void do_bessel_range();
 void do_spline();
 void fit();
 void maybe_emit_oob_warning();
 void non_monotonic_error();
 void output_dataset_separator();
 void set_format_type();
```

Factoring spline: after

These functions are part of the shell, so we left them in spline.c. They are concerned with reading data, writing data, error reporting, and bessel function interpolation (a feature we decided not to move into the library).

```
/* reading data */
 bool read_data();
 bool read_float();
 bool skip_whitespace();
 int read_point();

 /* writing data */
 void output_dataset_separator();
 bool write_point();
 void set_format_type();

 /* errors and warnings */
 void maybe_emit_oob_warning();
 void non_monotonic_error();

 /* bessel functions */
 bool do_bessel();
 void do_bessel_range();
```

These are the interface functions we created in gspline.h. They're what the user will call to generate a spline and interpolate points on the spline. They are declared publicly in gspline.h.

```
int splgen();
 int splspl();
 void spldef();
```

These are the implementation functions we moved to gspline.c. They are necessary to generate spline functions and interpolate points on the spline, but they do not provide useful functionality to the user.

```
bool is_monotonic();
 void fit();
 double interpolate();
 double quotient_sin_func();
 double quotient_sinh_func();
 double sin_func();
 double sinh_func();
 double tan_func();
 double tanh_func();
```

Design failure out of the library

Try to leave most of the dirty work in the hands of the developer who will use your library. They'll be happy to have a library that is very tolerant of error conditions -- this will result in a more stable application overall. Check all function return codes and recover from errors if possible. If you can't recover, return an error code to the function that called you instead of terminating. There's nothing more frustrating than a picky library with its finger on the exit() button.

The top two causes of failure are the file system and the memory system. Your library will be more robust if it never reads or writes a file and never allocates any memory. This doesn't mean that your library can't use data from files or dynamically allocated memory. You can require that the user pass in allocated memory and read from files for you. If you *do* have to call file or memory functions from the library, remember that: `malloc()` can return NULL, file reading and writing functions can fail, and that even `fclose()` can fail. So make sure you check the return codes of these functions.

In spline, we have chosen to eliminate all file-related tasks and minimize memory-related tasks. Since it's possible for input data to be ill-conditioned (too few points, non-monotonic abscissa, and so on), we added an error code to implement passing errors out to the calling program.

Avoiding failure in spline

Here are the error codes we defined for the spline functions:

```
#define SPLINE_SUCCESS                  0
 #define SPLINE_OUT_OF_MEMORY           -1
 #define SPLINE_SINGULAR_TENSION        -2
 #define SPLINE_SINGULARITY             -3
 #define SPLINE_INVALID_ARGS            -4
 #define SPLINE_NOT_MONOTONIC           -5
 #define SPLINE_NO_POINTS               -6
```

As an example of a possible error in spline, when we ask for a spline under negative tension, the library must solve an equation involving `1 / ( sin ( tension * h ) )` . The sine of `tension * h` may be zero, in which case the problem cannot be solved due to a singularity caused by the tension. In the old code, this was handled as follows:

```
/* in the function fit() */
 if (sin (tension * h[i]) == 0.0)
 {
     fprintf (stderr, "%s: error: specified negative tension value is singular\n", progname);
     exit (EXIT_FAILURE);
 }
```

Avoiding failure in spline (continued)

The previous code fragment does two things that we don't want in the library: It writes to a file (standard error), and it terminates the program (exit). However, this code is in the function `fit()`, which (in the library) is called by the interface functions `splgen` and `splspl`. So we have to modify `fit()` to return an error or success code to its caller, and the calling function to handle an error return from `fit()`. Here's how we wound up doing it:

```
/* in the function fit() */
 if (sin (tension * h[i]) == 0.0)
 {
     return SPLINE_SINGULAR_TENSION;
 }

 /* in splgen */
 int nResult = fit (pspl, nDimension);
 if( nResult != SPLINE_SUCCESS )
 {
     spline_cleanup (pspl);
     return nResult;
 }
```

Notice that we save the error result code and pass it out to the caller of `splgen` in case there was an error.

Facilitate simple tasks

When designing your interface, think like a user of your library. Users are lazy. Would you use `strchr` if you had to specify n flags (direction of search, character width, cursor position after match...)? No! You use `strchr` for forwards, `strrchr` for backwards, `wcschr` for unicode and so on. The worst part of using any library function is finding out what all the flags are for. So avoid, or at least minimize, the use of flags in your interface functions.

Think carefully about the most common desire of your users, and give the function that accomplishes that desire the simplest name. In spline, we think the most common desire is to pass in some data points, and get out evenly spaced points that are close enough together that, when connected by line segments, they look like a smooth curve.

The function prototype is:

```
int splgen( int nPointsIn, double * pdfXIn, double * pdfYIn,
             int nPointsOut, double * pdfXOut, double * pdfYOut );
```

The output arrays can be plotted on the same graph as the input arrays and will provide a curve fit that goes through those points.

Use structures to your advantage

`splgen` is fine if all you want is one-dimensional splines with no tension and you want the user to supply all data. But what if the user wants to create a three-dimensional spline with periodic boundary conditions and nonzero tension? This was possible with the stand-alone program, and should be possible in our library as well.

We suggest using structures to encapsulate unwieldy options. Internally, all of the library implementation functions can use this structure. Simple interface functions (such as `splgen`) will build a structure and pass it to the implementation.

In spline, we'll define a SPLINE structure that contains more complicated information regarding how a spline should be generated. The user may fill it manually or by calling `spldef`, which will fill it with default values -- those used by `splgen`. The user can modify options in the structure as desired, copy input and output arrays into the structure, and call the interface function that takes a SPLINE structure as its only argument. We'll call that function `splspl`.

Using structures: motivation

In the stand-alone program, all the spline generation was done by a function called `do_spline`, which takes 20 arguments. Here is its declaration:

```
do_spline (int used, int len, double **t, int ydimension, double **y, double **z,
            double tension, bool periodic, bool spec_boundary_condition,
            double k, int precision, double first_t, double last_t,
            double spacing_t, int no_of_intervals, bool spec_first_t,
            bool spec_last_t, bool spec_spacing_t,
            bool spec_no_of_intervals, bool suppress_abscissa)
```

The reason for all the arguments is that many of the input parameters are optional, so there is a flag argument to tell that a given parameter was specified. For example, there is `spacing_t`, which is the output spacing, and `spec_spacing_t`, which is true if the user requested a specific output spacing.

Notice that there are no arguments for the output data. This is because the stand-alone spline program wrote its output data directly to the specified output file from `do_spline`. Our library will need to have output data arguments as well.

Using structures in spline

In the library, we've defined a structure to contain our spline data. We call it `struct SPLINE`:

```
struct GSPLINE_API SPLINE
 {
        unsigned int nMask;

        /* user-settable parameters */
        int nPointsIn;
        double * pdfXIn;
        double ** ppdfYIn;
        int nPointsOut;
        double * pdfXOut;
        double ** ppdfYOut;
        int nDimensions;
        double dfTension;
        double dfBoundary;

        /* "private" parameters */
        unsigned int nSplineAllocated;
        double ** ppdfY2Deriv;
 };
```

This structure contains variables to store all the numerical data necessary to compute a spline function and interpolate values on it. Notice that we have both input and output point arrays, and that the variables for "Y" data (`ppdfYIn` and `ppdfYOut`) are declared as pointers-to-pointers so we can handle multidimensional "Y" data. Also, instead of generating points for the "X" values from user-specified variables (`first_t`, `last_t`, and `spacing_t` in the old code), we expect the user to generate appropriate values and pass them in.

`nMask` is a bit mask used to specify which of the structure members have been set by the user. For example, if the user supplies a tension, she must set the bit corresponding to the tension, as follows:

```
pspl->dfTension = 1.0;
 pspl->nMask |= SPLINE_TENSION;
```

Thread safety

A thread-safe library either uses no static data or it locks access to static data while it is being used by one thread so that inconsistent accesses cannot occur.

For example, one way to implement the SPLINE structure discussed in the previous panel would be to have a single SPLINE structure declared as part of the library interface. This would simplify the interface because the structure could be initialized when the library is loaded. Several functions would need no arguments at all: They would just use the implicitly allocated spline structure. However, this arrangement would not be thread safe. If two threads wanted to draw different splines at the same time, the data from one might end up in the SPLINE structure while the second was executing. The results would be undefined.

A less contrived example involves a file stream. File streams are not thread safe. A mutual exclusion device (a mutex or semaphore) must be employed to prevent two threads from modifying a file at the same time. If your library allows the user to set a file stream (for example, for error reporting) you must use a mutex to guard that file stream against concurrent access.

Since the spline library will have no static data and will not access any file streams or other process-wide resources, it is intrinsically thread safe.

Expect the unexpected

This panel is for those of you who looked at the title of the last panel, said "I never use threads!", and skipped to here. *You* may never use threads, but you can bet that if you don't make your library thread safe, you will get e-mail asking about thread safety. Or in a year or two, when you've decided to use threads for some project, you'll find that your library needs to be restructured.

Try to make your library as safe as possible for a variety of situations. You don't need to actively support features that you don't use, but don't actively inhibit their use. Unless there's a reason: don't use static data, don't rely on getting fast response back from system calls, and don't permanently lock system resources.

Limit your dependencies

Your library should not require nine other packages to be installed before it is functional. Ideally it should require no other packages, for two reasons. First, the fewer dependencies you have, the fewer dependency-related platform headaches you can expect. The second is a human factor: No matter how straightforward it is, people don't like to have to install other libraries in order to use yours.

Of course you can use the ANSI C standard library functions, but some of those have (usually documented) platform issues. Notably, some systems distinguish between binary and text files in `fopen()`, and some still lack `memcpy()` and `memset()`. If you need functionality that's not in the C standard, try POSIX. If it's not in POSIX, find a third-party, cross-platform library with the functionality you need.

Section 3. Practical library tips

Introduction

In this section, we'll give some practical tips on making a cross-platform C library. These tips are less concerned with factoring and design and more with problems that crop up on particular platforms.

Use ANSI C: avoid vendor extensions

There has been an ANSI standard for C since 1989. To make your library work across the widest variety of platforms, you should use ANSI C. The first pitfall to avoid is the use of vendor extensions. For example, some vendors extend the ANSI C standard and add an `inline` keyword, but many compilers don't support the `inline` keyword and will treat it as a syntax error. The GNU C compiler defines several extensions, including `inline`, nested functions, and empty `default:` blocks. In the interests of portability, you should avoid these as well.

You can verify that your code uses only ANSI constructs by compiling it with the GNU C compiler in ANSI mode: `gcc -ansi file.c`.

The previous implementation of spline did not use any vendor extensions, so we didn't have to change anything.

Use ANSI C: support for older compilers

Most modern C compilers conform to the ANSI C89 standard. However, some platforms still have old compilers that only understand the pre-ANSI style of C, also known as "K&R C" -- after Brian Kernighan and Dennis Ritchie, the authors of The C Programming Language.

The difference that will affect everyone is how function parameters are specified. The traditional way to handle this is to use the macro `PARAMS`, which expands to its arguments in ANSI and to an empty argument list in K&R. This way you can have one line of function declaration that expands to the correct thing on the target system, as illustrated below:

```
                                    /* Suppose we declare a function as */
int foo PARAMS((int a, double b));

/* On an ANSI compiler it will expand to */
int foo (int a, double b);

/* while on a K&R compiler, we will get */
int foo ();
```

There are other differences that require more involved workarounds. See Resources on page 25for a link on more of the differences between ANSI and K&R C.

Memory allocation

As we mentioned earlier, we minimized the amount of memory allocation going on in spline to minimize the chance of failure. This also has an added benefit because memory must usually be freed in the same memory space in which it was allocated. On some systems (notably Windows) shared libraries allocate memory from a different arena than the user programs that invoke them. So if one of your library interface functions returns memory that was allocated inside the library, you must provide a library-specific *free* function to free that memory. This includes implicit memory allocations, such as when you call `fopen()` and the C library allocates a buffer.

The simplest solution is to never allocate memory in the library. If you can't do that, don't pass out references to allocated memory to be used by the calling program. Instead, let the calling program allocate the memory and pass it to you.

Create a README file

Every published software component should have at least a README file. That way, those people who read them will actually know how to install your library, and they'll know what it does. If you have a small library, like spline, with only a few interface functions, you can go ahead and document the interface functions and their arguments right in the README. For a larger distribution, you'll want to split up the information along traditional lines into a "README", "INSTALL", "BUGS", "NEWS" and so on. Definitely include contact information in the README and describe what you expect in a bug report.

A very detailed discussion of how to release an open-source project can be found in the Software Release HOWTO by Eric Raymond (see Resources on page 25for a link). Even if you don't plan to release your library as an open-source project, this HOWTO gives useful information on naming, coding, releasing, and promoting your library (see especially sections 2,5,6, and 8).

Even if your library is just for internal use, a minimal README with installation notes and contact information means you will spend less time explaining how to install it. You'll also have a better chance of hearing about bugs when they happen.

Documentation

Documentation is a must if you're releasing the code externally. While text in the README is fine for a small project, it's worth it to invest some time in generating modern documentation (HTML and man pages, at the least).

At a minimum, you should document every interface function in your library. You should discuss the type and purpose of each argument. When very complex things can be done with your library, provide several code samples. If some of your functions work on some platforms and not on others, you must document that. Also, if you provide two versions (for example, trial and full), you must document which functions, if any, are disabled in the trial version.

If your library is cross platform, you should also document which platforms it has been tested on. A list in your README of tested platforms will inspire contributions from users, as long as your contact info is up to date.

Section 4. Using Microsoft tools to construct the library

Introduction

In this section, we'll walk you through the steps to create a shared library on Windows using Microsoft tools (Microsoft Visual Studio and the Microsoft Visual C compiler).

If you intend to support Windows at all, you should read this section. First, even if you intend to distribute your library for users of the GNU tools on Windows, it is easy to make your library build with the native tools, and some people don't have the GNU tools. Also, in the examples, we show many of the steps necessary to ensure portability across platforms. These are not explicitly repeated in the following section (on GNU tools).

Create a DLL project

In Visual Studio, create a new project:

* Select File | New.... on the Projects tab.
* Select Win32 Dynamic-Link Library; name your project and select the project location.
* Click OK. Select "A DLL that exports some symbols." and click "Finish". This will generate a new C++ project which exports a function, a variable and a class. Go ahead and delete the function, variable and class, but save the code which defines DLLNAME_API in the dllname.h file.

Alternatively, you can create an empty DLL project and define the DLLNAME_API yourself. The following code listing shows what this should look like. If you've used the sample project method, you'll have to add the `#ifdef WIN32` line yourself, because `__declspec` is a Microsoft extension necessary to build DLLs on Windows. On other systems, you need to make DLLNAME_API expand to empty so the compiler is not confused.

```
#ifdef WIN32
 #ifdef GSPLINE_EXPORTS
 #define GSPLINE_API __declspec(dllexport)
 #else
 #define GSPLINE_API __declspec(dllimport)
 #endif
 #else /* not MSVC */
 #define GSPLINE_API
 #endif
```

Calling your library from C++

If you would like your library to be callable from C++, you need to make sure that your API is defined `extern "C"`. These preprocessor directives work for all C++ compilers.

```
#ifdef _cplusplus
#define BEGIN_DECLS extern "C" {
#define END_DECLS }
#else
#define BEGIN_DECLS
#define END_DECLS
#endif
```

Supporting K&R: function declarations

If you want to provide compatibility with compilers that don't support ANSI C, you will need to define and use a PARAMS macro as discussed earlier. The public header where your interface functions are declared will be installed in the system include directory. User programs will include it and it's not good practice to require that the user define a particular macro to let you know whether their C compiler supports ANSI prototypes. So, we explicitly list platforms that are known to support ANSI C, and fall back on K&R style C for all others.

```
/* PARAMS is a macro used to wrap function prototypes, so that compilers that
    don't understand ANSI C prototypes still work, and ANSI C compilers can
    issue warnings about type mismatches. */
#ifdef PARAMS
#undef PARAMS
#endif
#if defined (__STDC__) || defined (_AIX) \
        || (defined (__mips) && defined (_SYSTYPE_SVR4)) \
        || defined(WIN32) || defined(__cplusplus)
#define PARAMS(protos) protos
#else
#define PARAMS(protos) ()
#endif
```

Supporting K&R: function definitions

When you define your functions, you need to provide both an ANSI and K&R style definition. In this case, the PROTOTYPES macro can be used as a shortcut because you are controlling the way the file is compiled. Here's how we defined one of our interface functions; the ANSI style definition comes first.

```
GSPLINE_API
 int
 #ifdef PROTOTYPES
 splgen (int nPointsIn, double * pdfXIn, double * pdfYIn,
         int nPointsOut, double * pdfXOut, double * pdfYOut)
 #else
 splgen (nPointsIn, pdfXIn, pdfYIn, nPointsOut, pdfXOut, pdfYOut)
   int nPointsIn;
   double * pdfXIn, * pdfYIn;
   int nPointsOut;
   double * pdfXOut, * pdfYOut;
 #endif
```

If you enjoy using Visual Studio's autocomplete feature for referencing arguments inside a function, wait until development is finished to conditionalize your definitions. Visual Studio is confused by the conditional definition and may not recognize that you are inside a function.

Interface header: structures

Now we're ready to declare the interface structures and functions. Microsoft requires that the DLLNAME_API specification occur between the `struct` keyword and the structure name. For non-Windows builds, we defined DLLNAME_API to be blank, so it won't affect us.

```
/* the SPLINE structure */

 struct GSPLINE_API SPLINE
 {
         unsigned int nMask;

         /* user-settable parameters */
         int nPointsIn;
         double * pdfXIn;
         double ** ppdfYIn;
         int nPointsOut;
         double * pdfXOut;
         double ** ppdfYOut;
         int nDimensions;
         double dfTension;
         double dfBoundary;

         /* "private" parameters */
         unsigned int nSplineAllocated;
         double ** pdfY2Deriv;
 };
```

Interface header: functions

For functions, it's simplest to put the DLLNAME_API specification before their return argument. Only functions declared in this header with the DLLNAME_API can be called from the .lib stub that will be generated. Notice that we're using the PARAMS macro we defined above so we can support both styles of C.

```
/* the spline functions */

 GSPLINE_API
 int
 splgen PARAMS(( int nPointsIn, double * pdfXIn, double * pdfYIn, int nPointsOut, double * pdfXOut, double *

 GSPLINE_API
 void
 spldef PARAMS(( struct SPLINE * pspl ));

 GSPLINE_API
 int
 splspl PARAMS(( struct SPLINE * pspl ));
```

Configuration

Include the interface header in at least one of the sources which are compiled to make your library, even if your library never calls any of its own interface functions. The Microsoft linker will not generate a stub library (.lib) if there are no exported symbols. So the simplest way to ensure that a stub library is generated is to include your interface header. Verify that the macro DLLNAME_EXPORTS is defined. (You can check this in Project | Settings... | C/C++ tab, General drop down. DLLNAME_EXPORTS should appear at the end of the Preprocessor Definitions.)

Finally, you need to change your base address for the library. You can do this on the Link tab, Output drop down. If you don't change the base address, there will be an additional layer of indirection between your library functions and the user, which will cause some slowdown. You may select a base address between 0x1000000 and 0x8000000 -- it must be a multiple of 64K (0x10000). Microsoft's site has more information on base addresses (see Resources on page 25for a link).

Section 5. Using GNU tools to construct the library

Introduction

The GNU Autotools (Autoconf, Automake, and Libtool) can be used to generate dynamic libraries on Linux, Unix, and Windows using either the native compiler or the GNU compiler. GNU tools are intended primarily for Linux and Unix, so if you want to use them to generate a dynamic shared library on Windows, you will need to do some extra work. For starters, if you're on a Windows box, you need to have the Cygwin environment installed in order to use the GNU tools as described here (see Resources on page 25for a link).

Autoconf: scanning for dependencies

Autoconf is a utility that will help you detect platform-dependent libraries and function calls. It can be downloaded for free from the GNU site (see Resources on page 25for a link). To start using Autoconf, scan your source files for dependencies by running `autoscan` in your source directory.

This will generate a file called `configure.scan` which contains all of the platform-dependent "gotchas" it found, but in a very unreadable form. Rename `configure.scan` to `configure.in` and run `autoheader` in your source directory. This will produce a file called `config.h.in`. We'll discuss `config.h.in`, which is more readable than `configure.scan`, in greater detail in the next panel.

Autoconf: a closer look at config.h.in

config.h.in contains a macro for every dependency detected by autoscan. Here is an excerpt of what we obtained for the spline code.

```
/* config.h.in.  Generated automatically from configure.in by autoheader.  */

/* Define to empty if the keyword does not work.  */
#undef const

/* Define to `unsigned' if <sys/types.h> doesn't define.  */
#undef size_t

/* Define if you have the ANSI C header files.  */
#undef STDC_HEADERS

/* Define if you have the strerror function.  */
#undef HAVE_STRERROR

/* Define if you have the <string.h> header file.  */
#undef HAVE_STRING_H

/* Define if you have the m library (-lm).  */
#undef HAVE_LIBM

/* Define if compiler has function prototypes */
#undef PROTOTYPES
```

Here we see that some platforms do not support the keyword const, the standard type size_t, or the library function strerror! In the first two cases, config.h will contain a definition that produces a valid result. However, we do need to protect the code that uses strerror by using the macro HAVE_STRERROR. The code might look something like this:

```
#ifdef HAVE_STRERROR
  fprintf (stderr, "Cannot open file %s: %s.\n", psFileName, strerror(errno) );
#else
  fprintf (stderr, "Cannot open file %s (Error code %d).\n", psFileName, errno );
#endif
```

You should go through each of the macros in config.h.in and protect your code if necessary. If you need to check for something that is not defined by default, you can define your own Autoconf macros. A discussion of that is beyond the scope of this tutorial -- check the Autoconf manual for details.

Automake: modifications to configure.in

Add the following lines to `configure.in`, right after `AC_INIT`. Naturally you'll want to replace "spline" with your own library name and 1.0 with your own library version. The version may be any string you like.

```
AM_INIT_AUTOMAKE(spline,1.0)
 AM_CONFIG_HEADER(config.h)

 dnl Support Win32 dynamic libraries
 AC_LIBTOOL_WIN32_DLL
 AM_PROG_LIBTOOL
```

You may also add any other Automake configure macros (see the Automake manual for details) to the `configure.in` file. Now, regenerate your `configure` script by running the following commands.

```
$ aclocal
 $ autoheader
 $ autoconf
```

Automake: creating Makefile.am

Create the file `Makefile.am` with the following contents, customized for your library.

```
lib_LTLIBRARIES = libgspline.la

 libgspline_la_SOURCES = gspline.c
 libgspline_la_CFLAGS = -DGSPLINE_EXPORTS
 libgspline_la_LDFLAGS = -version-info 1:0:0 -no-undefined
```

On the first line, we tell Automake that we have one Libtool library to build and that it's called *libgspline*.

The next line specifies the source files; this may be a space-separated list of files. To support building with the GNU tools on Windows, you must specify that you wish to export functions by setting the `-DGSPLINE_EXPORTS` flag. Finally, specify the version, and, for Windows, that there will be no undefined symbols at link time. These Windows suppport options will not adversely affect compilation on other platforms.

Automake: generating the configuration scripts

Now is the time to run `automake -a` and hope that it can generate all of your dependencies. It will require the presence of the following files: README, AUTHORS, ChangeLog, and NEWS. It will complain about missing INSTALL and COPYING but will give you a default INSTALL and the GPL for COPYING. These are just defaults and can easily be overridden with any license you like.

Now run the following sequence of commands:

```
$ aclocal
 $ autoconf
 $ ./configure
 $ make
```

This should build your shared library. To distribute your library, run a `make distclean` and package your files for distribution. This directive tells `make` to delete object files and intermediates to the build process, so you can cleanly package only the sources.

Section 6. Summary

Platform independent libraries

Now you should be able to turn any module into a killer cross-platform library. You should now be aware of some common cross-platform pitfalls and ways to work around them. You should be able to build a shared library using GNU or Microsoft tools. The library thus created may be distributed on a variety of platforms.

This tutorial deals with only the simplest possible kind of shared library -- one that is of a high enough level to be agnostic of its platform. When you have to deal with hardware (for example, if you are creating a windowing or networking library), you will not be able to dodge these issues. However, thinking cross-platform from the beginning, and becoming familiar with `autoscan` should also make it easier to identify and solve platform problems.

Finally, we'd like to point out that just about any code is called on to run on multiple platforms. Even exclusively Microsoft- or Linux-targeted code must run on all flavors of the targeted operating system. As any developer knows, there are nontrivial differences between any two members of the same OS family.

Resources

Here are some resources you may find helpful:

* *Software Release HOWTO* .
* *Autobook* has a good discussion of the differences between ANSI and K&R C.
* *Automake homepage* .
* *Autoconf homepage* .
* *Libtool homepage* .
* Find out more about the *Cygwin* environment.
* Check out *www.linuxdoc.org* for more information on documentation.
* *Microsoft information* on base addresses.
* GNU *LGPL* (Lesser General Public License).
* *Merriam-Webster online*
* *Jargon File resources*

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

For questions about the content of this tutorial, contact the authors, *Danielle Michaels* and *Sam Mikes* .

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.