



According to Hoyle...

<http://www.jonhoyle.com> Copyright ©2006 Jonathan Hoyle

Cross-Platform Software Development from a Macintosh Perspective: C++ Application Programming with REALbasic

by Jonathan Hoyle, November 2006
jhoyle at macCompanion dot com

For over a year now, we have been touring some of the more popular cross-platform development choices from a Macintosh perspective. These are the topics we have covered thus far:

- **Intro:** http://www.maccompanion.com/archives/september2005/Columns/According_to_Hoyle_1.htm
- **Qt:** <http://www.maccompanion.com/archives/october2005/Columns/AccordingtoHoyle.htm>
- **wxWidgets:** <http://www.maccompanion.com/archives/november2005/Columns/AccordingtoHoyle.htm>
- **CPLAT:** <http://www.maccompanion.com/archives/december2005/Columns/AccordingtoHoyle.htm>
- **REALbasic:** <http://www.maccompanion.com/archives/january2006/Columns/AccordingtoHoyle.htm>
- **Runtime Revolution:** <http://www.maccompanion.com/archives/february2006/Columns/AccordingtoHoyle.htm>
- **AMPC:** <http://www.maccompanion.com/archives/march2006/Columns/AccordingtoHoyle.htm>
- **Java compilers:** <http://www.maccompanion.com/archives/april2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part I):** <http://www.maccompanion.com/archives/may2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part II):** <http://www.maccompanion.com/archives/june2006/Columns/AccordingtoHoyle.htm>
- **Converting Legacy Frameworks:**
<http://www.maccompanion.com/archives/july2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part III):**
<http://www.maccompanion.com/archives/october2006/Columns/AccordingtoHoyle.htm>

We began this series with a focus on C++ development, and we come back around to focus on it again. Despite the inroads made by languages like Java (for the web), C# (on Windows) and Objective-C (on the Mac), C++ still remains the predominant professional language in the market today. It is truly the *lingua franca* of software development, even if it is beginning to show signs of age due to its lack of modern language features, such as automatic garbage collecting. There is no more powerful language today than C++, yet it still suffers from a particularly annoying Achilles' heel; the API available with C/C++ is one for console applications only. Creating even the most elementary dialog-based "Hello, World!" app is outside C++'s native capabilities. Building a C++ GUI app requires platform-specific function calls, which are not in general, compatible on other operating systems.

Cross-Platform C++ Approaches

Although there exist a number of large cross-platform frameworks, such as *Qt* and *wxWidgets* (each having been reviewed last year), their APIs are large and daunting. *REALbasic*, on the other hand, is a very intuitive RAD (Rapid Application Development) compiler which can easily generate sophisticated GUI-based applications. Most C++ programmers think of *REALbasic* as simply an IDE for Basic programming; however, it is also a very powerful GUI generator which can be accessed by C++ code.

In last January's column on *REALbasic*, there was a small section entitled *Mixing REALbasic with C++*. In it, I spoke of the ability for C/C++ programmers to access *REALbasic's* cross-platform GUI capabilities. In this article, we shall give the specifics on how this is done. The remainder of this article assumes that you have a workable knowledge of C and C++, and some familiarity with *REALbasic* or *Visual Basic*.

Designing Using Model-View-Controller

Model-View-Controller (MVC) is a way to architect software, which separates an application's core data (*the Model*) from its user interface (*the View*). Typically, the *Model* is written in a platform-independent manner, so MVC allows the developer to create a separate *View* for each platform. The *Model* can be written with Xcode, CodeWarrior, Visual C++, or essentially any C compiler. Likewise, the *View* can also be written in any tool, although we will be focusing on *REALbasic* for this article. The code that interfaces between the *Model* and the *View*, is called the *Controller*. Since we wish to keep our C++ code platform-independent, we will place our *Controller* on the *REALbasic* side of the fence.

Note that a well-architected MVC application need not require a cross-platform framework or application at all. For example, by using available platform-dependent RAD tools, such as *Interface Builder* for Mac OS X and *Visual C# for .NET* on Windows, thin platform-specific GUI apps can be created very easily. These modern tools are dynamic and easy to learn and are often more efficient for learning than complicated cross-platform framework APIs. Using *REALbasic* for the *View* offers nearly the best of both worlds: a cross-platform GUI with an intuitive and powerful API.

From a Macintosh C++ programmer's perspective, *REALbasic* can be thought of as a cross-platform replacement for *Interface Builder*. Typical Mac OS X applications have their models built in C++ with *Xcode* and house the *View* as GUI modules inside *Interface Builder* .nib files, which live inside the application package bundle. Substituting IB with RB, the role is somewhat reversed: the *Model* is still built in C++ with *Xcode*, but it will be built as a dynamic library and this will live inside the *REALbasic* application package. The dynamic library will sometimes be referred to as the server and the *REALbasic* application as the client.

Building Dynamic Libraries

Creating a dynamic library is fairly straightforward, once you know how it's done. If your project already exists, simply add a DLL target. *Xcode* creates Mach-O dynamic libraries, which are called *dylib's*, and they typically have a "lib" prefix and a ".dylib" extension. REAL Software's Jonathan Johnson has an excellent web page outlining a step-by-step procedure of creating dylib's in Xcode usable with *REALbasic*: <http://www.nilobject.com/?p=184> .

For those wishing to be backward compatible with Mac OS 9, you will be creating PEF/CFM dynamic libraries, called *Shared Libraries*, and you will be using another compiler, such as *Metrowerks CodeWarrior*. For Windows compilation, dynamic libraries are called *DLL's*, and they can be written with *Visual C++* or *CodeWarrior* for Windows. I will use the term "DLL" to describe any of these dynamic library types.

Wrapping Your C Model Code

Once you have completed writing the model portion of your project, you are ready to make it accessible to a client application. Let's begin by taking a simple example. Suppose we have the following ANSI C functions we wish to expose to our GUI:

```
void foo(int parm1, double parm2);
int bar(const char *parm);
```

For Mach-O libraries, such those built by *Xcode*, these functions are automatically exported by default. However, if you are compiling for Windows (e.g. *Visual C++*) or PEF on the Macintosh (e.g. *CodeWarrior*), you will need to use the `__declspec(dllexport)` directive to tell the compiler that these functions are to be exported:

```
// Exported functions associated with the MyModel class
#ifdef __MACH__
    #define export
#else
    #define export __declspec(dllexport)
#endif

export void foo(int parm1, double parm2);
export int bar(const char *parm);
```

Finally, you must take care to note whether your C functions were compiled with a standard C compiler or with a C++ compiler. Typically, you can determine this from the source file extension: if it end with `.c`, it usually means that it was compiled as C; if its extension is `.cpp`, `.cp` or `.cc`, it usually means it was compiled with C++. In the event your code was compiled with C++, you will need to place your export definitions inside an extern "C" wrapper, as such:

```
extern "C"
{
    export void foo(int parm1, double parm2);
    export int bar(const char *parm);
};
```

And now your C functions are available to the client GUI app!

Wrapping Your C++ Model Code

C++ programmers will typically want to export class definitions, not just C functions. To do this, the class must be thunked down into C using wrapper functions. For example, suppose we wish to expose the following C++ class:

```
// A C++ model class used in the library's implementation
class MyModel
{
public:
    MyModel();
    virtual ~MyModel();
    void foo(int parm1, double parm2);
    int bar(const char *parm);

protected:
    // Remaining implementation
    ...
};
```

Our client application will be accessing the class only through simplified wrapper functions, which we can define as follows:

```
// Exported function definitions
extern "C"
{
    export int MyModel_CreateHandle();
    export void MyModel_DestroyHandle(int modelHdl);
    export void MyModel_Foo(int modelHdl, int parm1, double parm2);
    export int MyModel_Bar(int modelHdl, const char *parm);
}

// Exported function implementations
int MyModel_CreateHandle()
{
    // Return this pointer as an int handle
    return (int) new MyModel;
}

void MyModel_DestroyHandle(int modelHdl)
{
    // Convert the handle back to a pointer before deleting
    delete ((MyModel *) modelHdl);
}

void MyModel_Foo(int modelHdl, int parm1, double parm2)
{
    // Convert the handle back to a pointer before dereferencing
    ((MyModel *) modelHdl)->foo(parm1, parm2);
}

int MyModel_Bar(int modelHdl, const char *parm)
{
    // Convert the handle back to a pointer before dereferencing
    return ((MyModel *) modelHdl)->bar(parm);
}
```

Most C++ methods will be able to be thunked down to C in this fashion. You can see from the example above that we used the `int` type to hold our class pointer, so we are assuming a 32-bit library for this particular examples. Our REALbasic application will treat `modelHdl` as an opaque reference and not be concerned with the fact that it is actually a memory address. You'll note that memory allocation and deallocation is done in the server, not in the client.

Accessing Exported Functions from within REALbasic

It is important to remember that client application must be of the same compilation type as the server it uses. For example, if the server is built as an Intel-based Mach-O dylib, then only an Intel-based Mach-O client app will be able to access it. Likewise with PowerPC Mach-O, PowerPC PEF, Windows and Linux. To build an Intel-based or universal binary client, you must use *REALbasic 2006 Release 4* or later.

Once you have your functions exported from the DLL, they are now available to be called by your REALbasic client application. At the top of a REALbasic method in which you call a library routine, you must Declare the exported function before using it. If our exported C function were of the form:

```
extern "C" ReturnTyp FcnName(pType parm, ...);
```

then the REALbasic declaration must be of the form:

```
Declare Function FcnName lib LibName(parm as pType, ...) as ReturnTyp
```

If ReturnTyp is void, then the word Function is replaced with Sub and as ReturnTyp is dropped. For Mach-O, LibName is a complete pathname to the dylib, which can be relative to the Unix executable (note that the Unix executable live two levels beneath the app's bundle). For non-Mach-O targets, just the library name will do if the library is in the same path as the application. REALbasic source code for a lib may look like this:

```
#if TargetCarbon
    const ModelLib = "MyModel Shared Library"
#endif

#if TargetMachO
    const ModelLib = "@executable_path/../../libMyModel.dylib"
#endif

#if TargetWin32
    const ModelLib = "MyModel.dll"
#endif

#if TargetLinux
    const ModelLib = "libMyModel.so"
#endif

Declare Function MyModel_CreateHandle lib ModelLib() as integer
Declare Sub MyModel_DestroyHandle lib ModelLib(modelHdl as integer)
Declare Sub MyModel_Foo lib ModelLib(modelHdl as integer, parm1 as integer, parm2
as double)
Declare Function MyModel_Bar lib ModelLib(modelHdl as integer, parm as CString)
as integer

Dim modelHandle as integer
Dim barValue as integer

modelHandle = MyModel_CreateHandle()
MyModel_Foo(modelHandle, 12, 3.0)
barValue = MyModel_Bar(modelHandle, "Hello, World")
MyModel_DestroyHandle(modelHandle)

return barValue
```

Conclusion

By following some very simple design and implementation principles, Macintosh application projects can be easily transitioned into cross-platform projects with the use of REALbasic as a functional replacement for Interface Builder. For those interested, I created a sample C++/REALbasic project for the *MacHack 2005* conference, which you can download by visiting: <http://www.jonhoyle.com/machack/>.