



Using OEChem and Ogham with Microsoft Visual Studio .NET

Roger Sayle and Andrew Grant

April 20, 2007

3600 Cerrillos Road, Suite 1107
Santa Fe, NM 87507
www.eyesopen.com
support@eyesopen.com

Introduction

Microsoft's Visual Studio development environment is a popular development environment, commonly used by the corporate IT groups of large pharmaceutical companies. Microsoft Visual Studio provides an integrated development environment (IDE) for several programming languages including C, C++, C#, J# and Microsoft Visual Basic that can be used to target graphical and console applications to Intel (and AMD) compatible systems running Microsoft Windows.

This document describes how to integrate OpenEye Scientific Software's toolkit libraries into Visual Studio applications, for example using Microsoft Windows Forms designer, to deliver the power of OEChem's chemistry functionality to utilities on a chemist's desktop.

Microsoft Versions

One potentially problematic aspect of using Microsoft's development tools to build, debug and deploy is their rapid and continual rate of change. As Microsoft's Windows operating systems evolve, so do their developer tools to support new functionality, simplified development and ever changing programming paradigms. This means that with each release the user interface of their IDE, the APIs of their libraries and components and even the syntax and names of their programming languages typically change, often in backwardly incompatible ways.

The majority of this document describes the use of Microsoft Visual C++ .Net 2003 to develop "managed" Windows Forms applications on Windows 2000 and higher. Although, the steps and code examples given below are known not to work with earlier versions and probably won't work without modification on later versions, the explanations given in this document should be sufficient for someone skilled/familiar with Microsoft's tools to adapt to their particular development environment.

OpenEye distribution

The Microsoft Windows versions of OpenEye Scientific Software's toolkit libraries are available for download from OpenEye's public anonymous FTP site at <ftp://ftp.eyesopen.com/pub>, or from OpenEye's WWW site at <http://www.eyesopen.com/download>. Each recent product is available to download from its own subdirectory, and typically multiple versions of the product are available from further subdirectories. As OpenEye supports a wide range of hardware, operating systems and development platforms, it's necessary to download the files suitable for a particular machine. For all platforms, OpenEye currently distributes products a `.tar.gz` "tarball", which may be unpacked with many standard tools.

For working with Microsoft Windows, the required files will typically have filenames containing the string "microsoft-win32-msvc-i686" or something similar. For the examples below, the files that currently need to be downloaded are:

```
oechem-1.3.2-microsoft-win32-msvc-i686.tar.gz  
ogham-1.0b3-microsoft-win32-msvc-i686.tar.gz
```

These files should be unpacked into a directory on your machine, in the examples below we assume something like `C:\openeye`. The toolkit tarballs assume that any dependent products been unpacked into the same subdirectory, and each product distribution's installed directory tree can be "overlaid" with OpenEye's other products.

In this example, the Ogham libraries are dependent upon and require the OEChem libraries. Typically, a product's release notes will list the version requirements of its dependencies. Currently, Ogham version 1.0 beta 3 requires OEChem version 1.3.2.

For C++ toolkits, such as OEChem and Ogham, the necessary header files are placed in `C:\openeye\include` and the binary object libraries will be placed in `C:\openeye\lib`. Additionally, most toolkits will come with pedagogical example source code, which will be placed in `C:\openeye\examples`.

Compiling with OEChem in Visual Studio

We assume that the reader has some familiarity with Microsoft Visual Studio IDE, and can create a new project and a template/stub application for themselves. For some, of the example below we assume that the user has selected "New...", then "Project" from the "File" menu, and selected "Windows Forms Application (.NET)" from the ".NET" folder inside the "Visual C++" projects folder. Though many other types of project behave similarly.

To use OEChem (or any other OpenEye toolkit) functionality within a Visual C++ source file, either a `.cpp` or a `.h` file, you first need to include the appropriate header files at the top of the source file, much like you'd include the standard C library or STL headers in C++.

For a typical OEChem application, the user will typically add the three lines to the top of the appropriate source file.

```
#include "oeplatform.h"  
#include "oesystem.h"  
#include "oechem.h"
```

There's some benefit to including these header files first before any other system header files. The reason being that the OpenEye headers will specify compiler pragmas and other flags to disable annoying warnings in system header files, allowing the user's application to be compiled with a high warning level but without overwhelming the output with the potential problems in the operating systems own system headers or SDK.

To include the "Depiction" and "Lexichem" functionality from the Ogham toolkit, a typically application would also include one or both of the following headers (respectively).

```
#include "oedepict.h"  
#include "oeiupac.h"
```

In order to successfully compile or recompile the source file, the Visual Studio project now needs to be told where on the file system to find these header files. This is done by adding the appropriate include path to the project properties, by selecting "properties..." at the bottom of the "Project" menu, and then pulling up the "General" dialog in the "C/C++" folder of the "Configuration Properties" folder. The "Additional Include Directories" entry should include the appropriate path, for example, `C:\openeye\include`.

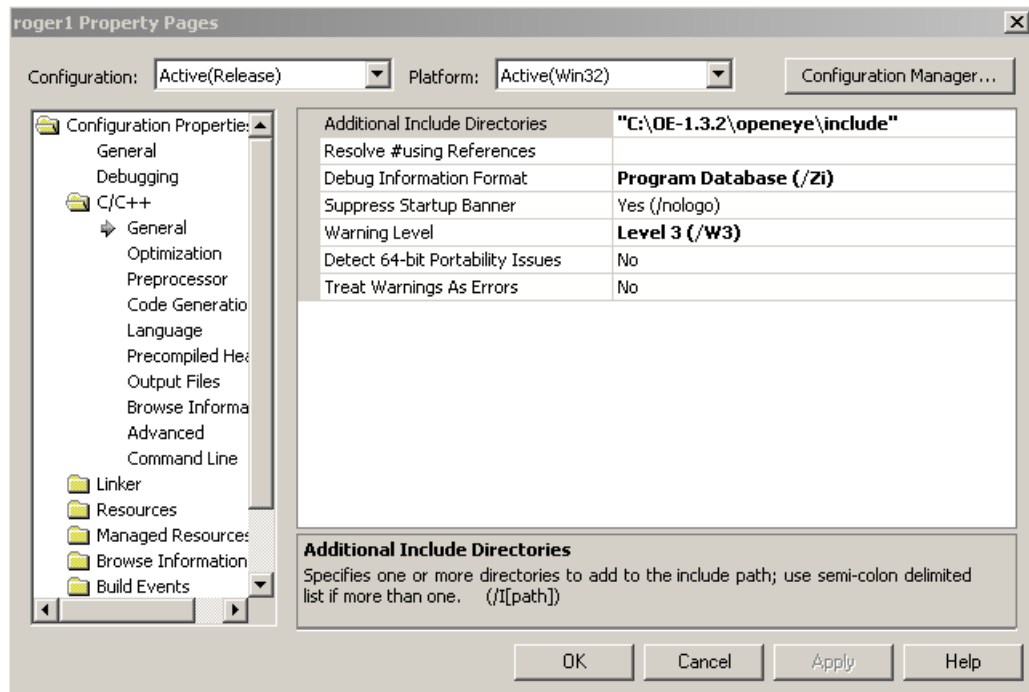


Figure 1: Additional Include Directories

If the correct include patch hasn't been set, or has been set incorrectly attempting to include the compile the source file or build the project will result in an error message similar to the following.

```
fatal error C1083: Cannot open include file 'oechem.h':
No such file or directory
```

An alternative approach to setting paths on a per project basis is to set them globally in Visual Studio, such that all projects pick up the OpenEye include and library paths automatically. This is done via the "Options..." menu item on the "Tools" menu. Selecting the "VC++ Directories" options dialog from "Projects" folder, shows the default paths used by the compiler. A drop-down list at the top allows the user to specify paths for "Include files" and/or "Library files". Adding `C:\openeye\include` to the first and `C:\openeye\lib` to the other will be picked up by all projects built on the machine. The downside of this global approach, however, is that it complicates the task of keeping more than one version of OpenEye's toolkits on a machine.

Unfortunately, even without introducing any calls to OpenEye code to you application, including the above headers may often expose a conflict in Microsoft's SDK header files, that causes an existing working application to fail to compile. The issue is that OpenEye's headers include the usual Windows SDK header files for C. Overtime, as the Win32 API has evolved, the corresponding SDK headers have adapted by defining macros that rename obsolete function calls to alternate names. Unfortunately, these `#defines` can conflict with modern Visual C++ .Net function names.

Examples, of these conflicts know to cause problems include the macros `MessageBox` and `GetObject` which are defined to the values `MessageBoxA` and `GetObjectA` respectively. The simplest solution to the problem, should such a conflict arise, is to `#undef` the problematic macro, immediately after including OpenEye's headers.

```
#undef GetObject
#undef MessageBox
```

With luck, the above steps should allow you to compile your source files, exposing OEChem and Ogham functionality.

Linking with OEChem in Visual Studio

The next hurdle is linking your application.

The first step is to list the necessary library paths and binary library files in your project's configuration. These are configured on the project "Properties..." dialog on the "Project" menu, like the include paths described earlier. The library paths are set using the "Additional Library Directories" item on the "General" dialog which is in the "Linker" folder, which in turn is in the "Configuration Properties" folder.

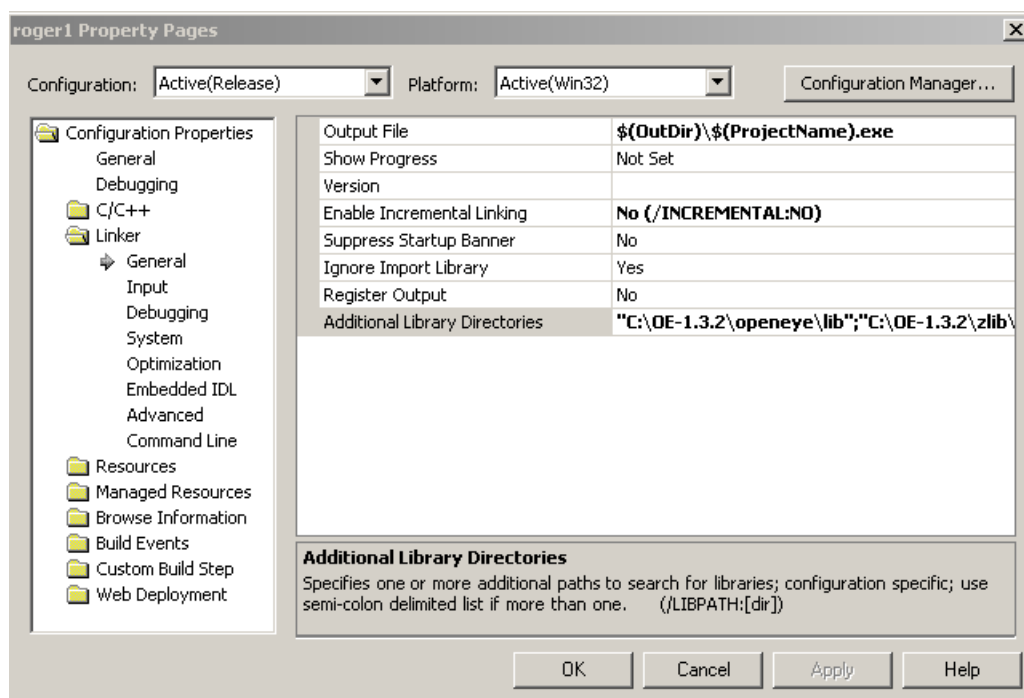


Figure 2: Additional Library Directories

This needs to specify the paths to the OpenEye binary library files which is something like `C:\openeye\lib`, and additionally to the location of the GNU zlib library, `libz.lib`. Future versions of OEChem may redistribute precompiled versions of zlib in the same library directory, to save downloading a precompiled version from the Internet, or rebuilding one yourself from source code locally.

Likewise, the list of library files that need to be linked are specified in the "Additional Dependencies" item on the "Input" dialog which is in the "Linker" folder, which in turn is in the "Configuration Properties" folder.

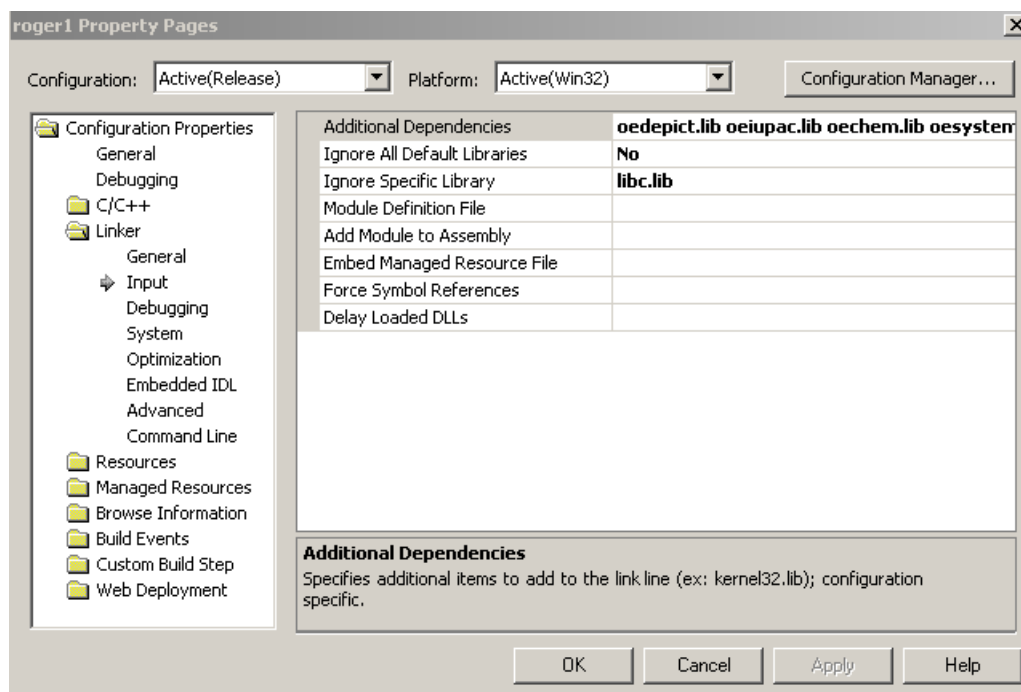


Figure 3: Additional Dependencies

To make use of OEChem functionality this needs to list the following libraries.

oechem.lib oesystem.lib oeplatform.lib zlib.lib netapi32.lib wsock32.lib

To make use of the Lexichem (name-to-structure and structure-to-name) functionality, you also need to specify `oeiupac.lib` in addition to the OEChem list. And to make use of the 2D co-ordinate generation and rendering (depiction) functionality of Ogham, you also need to specify `oedepict.lib`.

Currently, OpenEye's toolkit libraries impose the constraint on Visual Studio applications that they must be built to use Microsoft's "Multithreaded DLL" runtime libraries, as specified by the `/MD` command line option to their compiler. Support for multithreading is compulsory in newer versions of Visual C++ and allows code to operate in Microsoft's new .Net framework.

To configure your project to use the "Multithreaded DLL" run-time libraries, set the "Runtime Libraries" option in the "Code Generation" dialog of the "C/C++" folder of the "Configuration Properties" dialog. As described for setting the project include paths, this can be done by selecting the "properties..." menu option at the bottom of the "Project" menu.

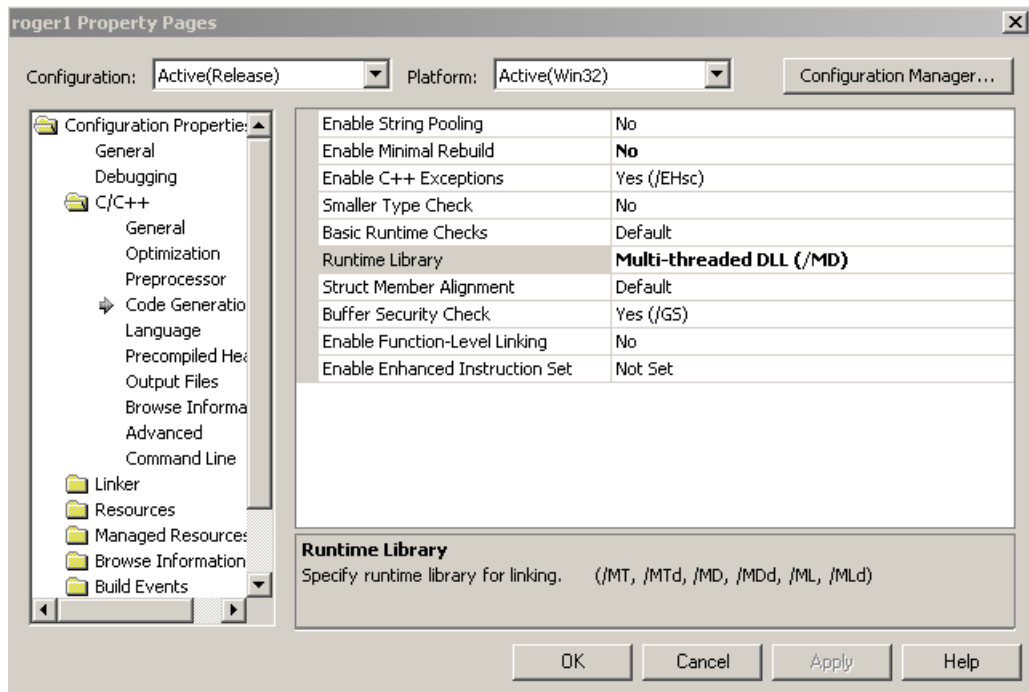


Figure 4: Runtime Library

Microsoft Visual C/C++ currently supports generating Windows applications that use one of three sets of runtime libraries. These runtime libraries each provide implementations of commonly used functions like `strcmp`, `memcpy`, `fprintf`, etc... and for C++ the STL libraries containing `std::string` and `std::vector`. The three versions are “statically linked single threaded” that links against the binary library `libc.lib`, “statically linked multithreaded” that links against `libcmtd.lib` and finally “dynamically linked multithreaded” that links against `msvcrt.lib`. The last of these is the most recent, and the one preferred by Microsoft and required by OpenEye.

Run-time Licensing

OpenEye Scientific Software’s toolkit libraries typically require a run-time license that is checked the first time the functionality of the toolkit is called. This run-time license checking looks for a valid license file, usually called `oe_license.txt` that should be visible on the machine running the OpenEye code.

License files are available by contacting support@eyesopen.com.

On most platforms, including Microsoft Windows, the location of the license file is specified to the executable by setting the `OE_LICENSE` environment variable to point to the location of the valid license file. A typical setting might be

```
OE_LICENSE=C:\openeye\oe_license.txt
```

On Microsoft Windows, this environment variable may be set or modified by “System” control panel in the

“Control Panel” option on the “Settings...” option of the “Start” menu. In the “System” control panel, select the “Advanced” tab, and activate the environment variables dialog by selecting the “Environment Variables...” button.

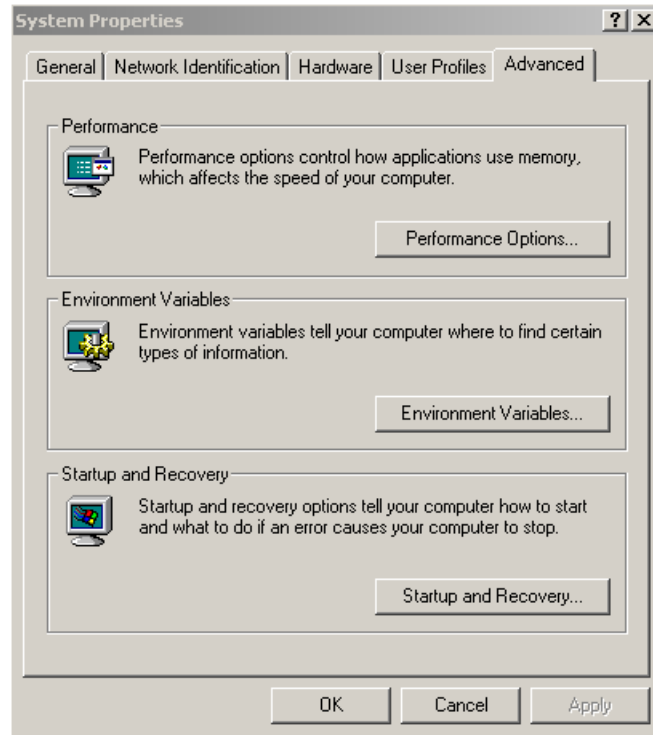


Figure 5: System Properties

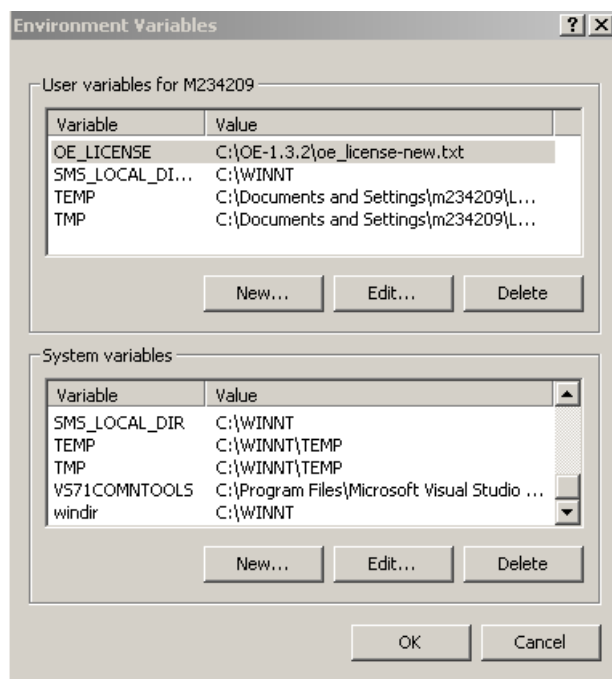


Figure 6: Environment Variables

Alternatively, the `OE_LICENSE` environment variable may be set by typing the command `set OE_LICENSE=c:\openeye\oe_license.txt` into a Microsoft “Command Prompt” window before launching a program manually.

The contents of OpenEye license files may be concatenated together, to place multiple product licenses in a single file. Hence, an application that calls multiple toolkit libraries, will require licenses for each of them in the file specified by `OE_LICENSE`.

Hooking SMILES into Microsoft Windows Forms .Net

Consider a simple Microsoft Windows form that allows you to canonicalize a SMILES string that’s entered into a `TextBox` and places the result in a `Label`. The process of conversion may be triggered off of the “Click” event of a “Convert” Button, or the “Enter” event of the `TextBox` or even it’s “Text Changed” Event.

A simple model of this that doesn’t use `OEChem` might be to simply copy the string, which is a `System::String` in the managed C++ environment, from the `Text` field of the `TextBox`, to the `Text` field of the label. If we assume that the `TextBox` is called “*textbox1*” and the `Label` is called “*label1*”, the code might look like.

```
private:
System::Void textbox1_Enter(System::Object *sender,
                           System::EventArgs *e)
{
```

```

    this->label1->Text = this->textBox1->Text;
}

```

Unfortunately, the OEChem toolkits and the C++ Standard Template Library (STL) are not directly compatible with Visual C++ .Net's managed strings so we need to convert between one form and another. This is a process known as *marshaling*.

To convert a `System::String` to a C-style NUL-terminated `char*`, we need to use the function `StringToHGlobalAnsi` in the `InteropServices::Marshal` namespace to provide a pointer handle. This handle may be safely converted to an address using its `ToPointer` method. Once we've finished using it, we must remember to call `FreeHGlobal` (also in the `InteropServices::Marshal` namespace) to deallocate it afterward.

The easiest way to do the reverse and convert a C-style NUL-terminated `char*` to a managed `System::String` is to simply construct using the pointer as the constructor argument. This is identical to how you'd typically construct an STL `std::string` from a NUL-terminated buffer, such as a string literal.

The following example demonstrates how to convert the input managed string into an STL `std::string`, and then back to a managed .Net string for output.

```

private:
System::Void textBox1_Enter(System::Object *sender,
                           System::EventArgs *e)
{
    // Convert System::String to C-style string
    char *src = (char*) InteropServices::Marshal::StringToHGlobalAnsi \
                (this->textBox1->Text).ToPointer();

    // Convert C-style string to STL std::string
    std::string tmp(src);

    // Free the allocated C-style string
    InteropServices::Marshal::FreeHGlobal(src);

    // Do something useful with tmp here

    // Convert STL string to System::String via C-style string
    System::String *dst(tmp.c_str());

    this->label1->Text = dst;
}

```

Now that all the nasty work of converting strings from one format to another has been explained, the following code example shows how you can canonicalize SMILES strings.

```

private:
System::Void textBox1_Enter(System::Object *sender,
                           System::EventArgs *e)
{
    // Convert System::String to C-style string
    char *src = (char*) InteropServices::Marshal::StringToHGlobalAnsi \
                (this->textBox1->Text).ToPointer();

    // Attempt to parse this into a OEChem molecule

```

```

OEChem::OEGraphMol mol;
if (OEChem::OEParseSMILES(mol,src))
{
    std::string tmp;

    // We managed to parse the SMILES, now regenerate a SMILES.
    OEChem::OECreatCanSmiString(tmp,mol);

    // Place the result into the form label
    System::String *dst(tmp.c_str());
    this->label1->Test = dst;
}
else
    this->label1->Text = S"Error: Invalid SMILES";

// Free the allocated input string
InteropServices::Marshal::FreeHGlobal(src);
}

```

Hooking Names into Microsoft Windows Forms .Net

Now that we've integrated some molecular chemistry into this example, in the form of SMILES strings using OEChem, we can go one step further and use Ogham's Lexichem functionality, to convert names to structures and structures to names.

In fact, a simple chemical name normalization/correction application can be constructed from the form described above.

```

private:
System::Void textBox1_Enter(System::Object *sender,
                           System::EventArgs *e)
{
    // Convert System::String to C-style string
    char *src = (char*) InteropServices::Marshal::StringToHGlobalAnsi \
                (this->textBox1->Text).ToPointer();

    // Attempt to parse this into a OEChem molecule
    OEChem::OEGraphMol mol;
    if (OEIUPAC::OEParseIUPACName(mol,src))
    {
        // We managed to parse the name, now regenerate a name.
        std::string tmp = OEIUPAC::OECreatIUPACName(mol);

        // Place the result into the form label
        System::String *dst(tmp.c_str());
        this->label1->Test = dst;
    }
    else
        this->label1->Text = S"Error: Compound name not understood";

    // Free the allocated input string

```

```
InteropServices::Marshal::FreeHGlobal(src);  
}
```

With this latest mini-application, a chemist can now type in the compound name “hydroxybenzene”, press the return or enter key, and see the name corrected to be “phenol”. Likewise, typing the string “durene” and pressing return generates the output “1,2,4,5-tetramethylbenzene”.

Clearly, with little effort a programmer can extend the above applications to output both the SMILES and the name. Indeed, development versions of Ogham can be used to generate names in different styles, such as strict IUPAC, CAS and traditional allowing variations on the name to be reported. Likewise, it doesn’t take too much though to extend this application to recognize either a common name or valid SMILES string by testing the return values of the various parsing functions.

An exercise left to the corporate reader is to integrate this system with ODBC or similar client-server connectivity to allow the chemist to type in a registry number, and if recognized as such, to query the corporate database to retrieve the SMILES (or name) directly back into the application.

Hooking Depictions into Microsoft Windows Forms .Net

Whilst one dimensional descriptions of molecules such as SMILES and IUPAC names are useful, by far the most preferred way of showing a chemical structure in graphical user interfaces is as a 2D depiction. Fortunately, this functionality is provided by Ogham’s depiction code, which may be relatively easily be integrated with Windows Forms .Net.

Let’s imagine that in addition to the TextBox and Label used in the examples above, the Form designer has also placed a PictureBox control on the form, which we’ll assume is called “*pictureBox1*”.

Then the code to display the 2D depiction in this control, for example from the same event as above can be by the following functions.

The first of which is I’ve called “*PrepareMolecule*” which takes an OEMolBase as created by either the OEParseSMILES function or the OEParseIUPACName described above.

```
void PrepareMolecule(OEChem::OEMolBase &mol)  
{  
    OEChem::OESuppressHydrogens(mol);  
    OEChem::OEAddDepictionHydrogens(mol);  
    OEChem::OEDepictCoordinates(mol);  
    OEChem::OEMDLPerceiveBondStereo(mol);  
}
```

The *boiler plate* function above is described in the Ogham manuals, and assumes that the specified molecule doesn’t have any co-ordinates. A slightly more complex variant of this code is shown below, which will use 2D co-ordinates if already specified, for example, when the molecule has been read from a 2D .mol file, and will extract stereochemistry from a 3D molecule if read from a Tripos .mol2 file or a PDB file.

```
void PrepareMolecule(OEChem::OEMolBase &mol)  
{
```

```

if (mol.GetDimension() != 2)
{
    if (mol.GetDimension() == 3)
    {
        OEChem::OEPerceiveChiral(mol);
        OEChem::OE3DToBondStereo(mol);
        OEChem::OE3DToAtomStereo(mol);
    }
    OEChem::OESuppressHydrogens(mol);
    OEDepict::OEAddDepictionHydrogens(mol);
    OEDepict::OEDepictCoordinates(mol);
    OEChem::OEMDLPerceiveBondStereo(mol);
}
}

```

Then the code to render the prepared molecule into an Ogham OE8BitImage buffer (via an OEDepictView) and from there transfer it into a Microsoft System::Drawing::Bitmap which can then be assigned directly to the “Image” property of a PictureBox control.

```

// Get the depiction width and height from the control
int wide = this->pictureBox1->get_Width();
int high = this->pictureBox1->get_Height();

// Create and set-up the view
OEDepict::OEDepictView view;
view.SetMolecule(mol);
view.AdjustView(wide,high);

// Render the view of the molecule into an 8-bit buffer
OEDepict::OE8BitImage img(wide,high);
view.RenderImage(&img);

// Create a Managed System::Drawing::Bitmap
Imaging::PixelFormat fmt = Imaging::Format8bppIndexed;
Bitmap *bitmap = new Bitmap(wide,high,fmt);

// Lock down the Bitmap's contents, so we can write into it directly
Drawing::Rectangle rect(0,0,wide,high);
Imaging::ImageLockMode mode = Imaging::ReadWrite;
Imaging::BitmapData *bmd = bmp->LockBits(rect,mode,fmt);

// Copy the contents of the OpenEye bitmap to the Microsoft Bitmap
unsigned char *src = img.ptr;
unsigned char *dst = (unsigned char*)bmd->get_Scan0().ToPointer();
int src_stride = img.GetXSize();
int dst_stride = bmd->GetStride();
for (int y=0; y<high; y++)
    for (int x=0; x<wide; x++)
        dst[y*dst_stride+x] = src[y*src_stride+x];

// Unlock the Bitmap's contents
bmp->UnlockBits(bmd);

// Copy the color (colour!) palette from OpenEye to Microsoft
Imaging::ColorPalette *pal = bmp->get_Palette();

```

```
for (int i=0; i<img.colcount; i++)
    pal->Entries[i] = Drawing::Color::FromArgb(img.col[i].r,
                                              img.col[i].g,
                                              img.col[i].b);

bmap->set_Palette(pal);

// Update the Image property of the text box
this->pictureBox1->Image = bmap;
```

One nice feature of the above code fragment is that it determines the size of the bitmap to draw from the current dimensions of the PictureBox control. This means that the depiction will automatically be rescaled, if this function is called from the “Resized” event of the PictureBox. So if the PictureBox control has its anchoring properties set appropriately, such that the picture canvas is nicely resized as the user resizes the window, the image in that window will take care of itself (without having to be stretched or cropped).

The above code uses the default OEDepictView properties for displaying a black on white depiction, with aromatic bonds displayed as alternating single and double Kekulé bonds, and with a small OpenEye logo in the bottom right hand corner. All of these aspects and more, such as highlighting of substructures and contraction of superatoms can be controlled by modifying the “view” object prior to calling its SetMolecule method in the example above.

Bibliography

- [1] Simon Robinson, K. Scott Allen, Ollie Cornes, Jay Glynn, Zach Greenvoss, Burton Harvey, Christian Nagel, Morgan Skinner and Karli Watson, “**Professional C#**”, 2nd Edition, Wrox Press, Wiley Publishing Inc., 2003.