C++/CLI Language Extensions

C++/CLI is a set of C++ language extensions, which enables creating hybrid managed and native DLLs. In C++/CLI, you can have both managed and unmanaged classes or functions even within the same .cpp file. You can use both managed types as well as native C and C++ types, just as you would in ordinary C++, i.e. by including a header and linking against the library. These powerfulcapabilities can be used to construct managed wrapper types callable from any .NET language as well as native wrapper classes and functions (exposed as .dll, .lib and .h files) which are callable by native C/C++ code.

Marshaling in C++/CLI is done manually, and the developer is in bettercontrol and more aware of marshaling performance penalties. C++/CLI can be successfully used in scenarios with which P/Invoke cannot cope, such as marshaling of variable-length structures. Another advantage to C++/CLI is that you can simulate a chunky interface approach even if you do not control the callee's code, by calling native methods repeatedly without crossing the managed-to-native boundary each time.

In the code listing below, we implement a native `NativeEmployee` class and a managed `Employee` class, wrapping the former. Only the latter is accessible from managed code.

If you look at the listing, you'll see that `Employee`'s constructor showcases two techniques of managed to native string conversion: one that allocates `GlobalAlloc` memory that needs to be released explicitly and one that temporarily pins the managed string in memory and returns a direct pointer. The latter method is faster, but it only works if the native code expects a UTF-16 null-terminated string, and you can guarantee that no writes occur to the memory pointed to by the pointer. Furthermore, pinning managed objects for a long time can lead to memory fragmentation (see Chapter 4), so if said requirements are not satisfied, you will have to resort to copying.

`Employee`'s `GetName` method showcases three techniques of native to managed string conversion: one that uses the `System.Runtime.InteropServices.Marshal` class, one that uses the `marshal_as` template function (that we will discuss later) which is defined in the msclr/marshal.h header file and finally one that uses `System.String`'s constructor, which is the fastest.

`Employee`'s `DoWork` method takes a managed array or managed strings, and converts that into an array of `wchar_t` pointers, each pointing to a string; in essence it's an array of C-style strings. Managed to native string conversions are done via the `marshal_context`'s `marshal_as` method. In contrast to `marshal_as` global function, `marshal_context` is used for conversions which require cleanup. Usually these are managed to unmanaged conversions that allocate unmanaged memory during the call to `marshal_as` that needs to be released once it's not longer required. The `marshal_context` object contains a linked list of cleanup operations that are executed when it is destroyed.

```
#include <msclr/marshal.h>
#include <string>
#include <wchar.h>
#include <time.h>

using namespace System;
using namespace System::Runtime::InteropServices;

class NativeEmployee {
public:
  NativeEmployee(const wchar_t *employeeName, int age)
   : _employeeName(employeeName), _employeeAge(age) { }

  void DoWork(const wchar_t **tasks, int numTasks) {
   for (int i = 0; i < numTasks; i++) {
   wprintf(L"Employee %s is working on task %s\n",
       _employeeName.c_str(), tasks[i]);
   }
  }

  int GetAge() const {
   return _employeeAge;
  }

  const wchar_t *GetName() const {
   return _employeeName.c_str();
  }

private:
  std::wstring _employeeName;
  int _employeeAge;
};

#pragma managed

namespace EmployeeLib {
  public ref class Employee {
```

```
public:
 Employee(String ^employeeName, int age) {
 //OPTION 1:
 //IntPtr pEmployeeName = Marshal::StringToHGlobalUni(employeeName);
 //m_pEmployee = new NativeEmployee(
 // reinterpret_cast<wchar_t *>(pEmployeeName.ToPointer()), age);
 //Marshal::FreeHGlobal(pEmployeeName);

   //OPTION 2 (direct pointer to pinned managed string, faster):
    pin_ptr<const wchar_t> ppEmployeeName = PtrToStringChars(employeeName);
    _employee = new NativeEmployee(ppEmployeeName, age);
    }

   ~Employee() {
   delete _employee;
   _employee = nullptr;
   }
   int GetAge() {
   return _employee->GetAge();
   }

   String ^GetName() {
   //OPTION 1:
   //return Marshal::PtrToStringUni(
   // (IntPtr)(void *) _employee->GetName());

   //OPTION 2:
   return msclr::interop::marshal_as<String ^>(_employee->GetName());
   //OPTION 3 (faster):
   return gcnew String(_employee->GetName());
 }

   void DoWork(array<String^>^ tasks) {
   //marshal_context is a managed class allocated (on the GC heap)
   //using stack-like semantics. Its IDisposable::Dispose()/d'tor will
   //run when exiting scope of this function.
   msclr::interop::marshal_context ctx;
   const wchar_t **pTasks = new const wchar_t*[tasks->Length];
   for (int i = 0; i < tasks->Length; i++) {
   String ^t = tasks[i];
   pTasks[i] = ctx.marshal_as<const wchar_t *>(t);
 }
 m_pEmployee->DoWork(pTasks, tasks->Length);
   //context d'tor will release native memory allocated by marshal_as
   delete[] pTasks;
 }

  private:
   NativeEmployee *_employee;
};
}
```

In summary, C++/CLI offers fine control over marshaling and does not require duplicating function declarations, which is error prone, especially when you often change the native function signatures.

The `marshal_as` Helper Library

In this section, we will elaborate on the `marshal_as` helper library provided as part of Visual C++ 2008 and later.

`marshal_as` is a template library for simplified and convenient marshaling of managed to native types and vice versa. It can marshal many native string types, such as `char *`, `wchar_t *`, `std::string`, `std::wstring`, `CStringT<char>`, `CStringT<wchar_t>`, `BSTR`, `bstr_t` and `CComBSTR` to managed types and vice versa. It handles Unicode/ANSI conversions and handles memory allocations/release automatically.

The library is declared and implemented inline in marshal.h (for base types), in marshal_windows.h (for Windows types), in marshal_cppstd.h (for STL data types) and in marshal_atl.h (for ATL data types).

`marshal_as` can be extended to handle conversion of user-defined types. This helps avoid code duplication when marshaling the same type in many places

and allows having a uniform syntax for marshaling of different types.

The following code is an example of extending `marshal_as` to handle conversion of a managed array of strings to an equivalent native array of strings.

```
namespace msclr {
 namespace interop {
   template<>
   ref class context_node<const wchar_t**, array<String^>^> : public context_node_base {
   private:
     const wchar_t** _tasks;
     marshal_context _context;
   public:
     context_node(const wchar_t**& toObject, array<String^>^ fromObject) {
     //Conversion logic starts here
     _tasks = NULL;
   const wchar_t **pTasks = new const wchar_t*[fromObject->Length];
   for (int i = 0; i < fromObject->Length; i++) {
   String ^t = fromObject[i];
   pTasks[i] = _context.marshal_as<const wchar_t *>(t);
   }
   toObject = _tasks = pTasks;
   }

   ~context_node() {
   this->!context_node();
   }

   protected:
   !context_node() {
     //When the context is deleted, it will free the memory
   //allocated for the strings (belongs to marshal_context),
     //so the array is the only memory that needs to be freed.
     if (_tasks != nullptr) {
   delete[] _tasks;
     _tasks = nullptr;
   }
     }
   };
 }
}
//You can now rewrite Employee::DoWork like this:
  void DoWork(array<String^>^ tasks) {
    //All unmanaged memory is freed automatically once marshal_context
    //gets out of scope.
  msclr::interop::marshal_context ctx;
    _employee->DoWork(ctx.marshal_as<const wchar_t **>(tasks), tasks->Length);
}
```

IL Code vs. Native Code

An unmanaged class will by default be compiled to IL code in C++/CLI rather than to machine code. This can degrade performance relative to optimized native code, because Visual C++ compiler can optimize code better than the JIT can.

You can use `#pragma unmanaged` and `#pragma managed` before a section of code to override complication behavior. Additionally, in a VC++ project you can also enable C++/CLI support for individual compilation units (.cpp files).