



Exposing native to managed - C++/CLI vs. P/Invoke

Shmuel Zang, 10 Sep 2013

CPOL



4.93 (37 votes)

This article shows a basic introduction for exposing native C++ content to managed .NET projects, using C++/CLI or P/Invoke.

[Download source - 40.8 KB](#)

Table of contents

- [Introduction](#)
- [Background](#)
- [Creating the native library](#)
 - [The Worker class](#)
 - [Implement Worker tasks](#)
 - [Run Worker threads](#)
 - [Protect critical sections](#)
 - [Gather execution time statistics](#)
 - [Provide three ways for executing the worker tasks](#)
- [Exposing our native library as a managed DLL](#)
 - [First approach - Call native DLL exports using P/Invoke](#)
 - [Native DLL](#)
 - [Create a native DLL project](#)
 - [Link our native static library](#)
 - [Add functions for exposing the logic](#)
 - [Export our DLL's functions](#)
 - [Calling convention](#)
 - [Keep the function entry point name as the original function name](#)
 - [Managed DLL](#)
 - [Second approach - Wrap the native library with a managed C++/CLI project](#)
 - [Why C++/CLI](#)
 - [Create a C++/CLI project](#)
 - [Wrap the native Worker class with an appropriate managed class](#)
 - [Third approach - Write the full native and managed code in one project](#)
- [Test and compare performance](#)
 - [Create C# console project](#)
 - [Add wrappers' DLLs](#)
 - [Implement the tests](#)
 - [The Tests' results](#)

- [Conclusion](#)

Introduction

As native C++ developers, we sometimes have to expose our content to other platforms. One of the common needs for that is when we develop a native C++ content (like a driver that deals with hardware or, some other low-level resources) that is intended to be used by .NET developers. For that purpose, we may want to provide our content as a .NET (MSIL) DLL that can be added as a reference for .NET projects.

In my case, I had to develop a communication driver (for an internal communication protocol of the company) that was intended to be used for .NET systems. At that point, I was already familiar with the [P/Invoke](#) approach, which let us create a native [DLL](#) and use its exports in a managed .NET DLL. I'd heard about the C++/CLI approach which let us create a DLL that contains both the native and the managed code and, wanted to give it a try. Since performance was an important factor of that driver, I decided to do some homework and do some performance tests on these approaches.

This article presents the example native library and the managed wrappers used for testing the performance of the approaches. In this article I go over the content of these projects and discuss some issues that, in my opinion, should be mentioned. Some of the issues in this article may be trivial for experienced developers but for beginners, this article can be a basic introduction.

In this article we create an example native "worker" that can execute given tasks. For that worker, we expose some methods for asking tasks' execution (call to our native "worker" from a managed platform):

- Execute asynchronous tasks:
 - A call for each task.
 - One call for all of the tasks.
- Execute synchronous tasks.

For wrapping our native "worker" with a managed class, we present 3 approaches (there are more approaches we can take for achieving that goal (like [COM](#)) but, those approaches are beyond the scope of this article):

1. Calling a native DLL extensions, from a managed DLL using P/Invoke
2. Wrapping a native static library, with a managed C++/CLI DLL.
3. Create one project, that contains the native C++ implementation and, the managed C++/CLI implementation.

This article shows a basic introduction to these approaches and, shows some performance tests on them.

Background

This article is intended for native C++ developers who want to expose their native content for .NET developers and, assumes a familiarity with the C++ language.

Since the managed code of this article is provided using the C# language, a basic familiarity with C# is recommended too.

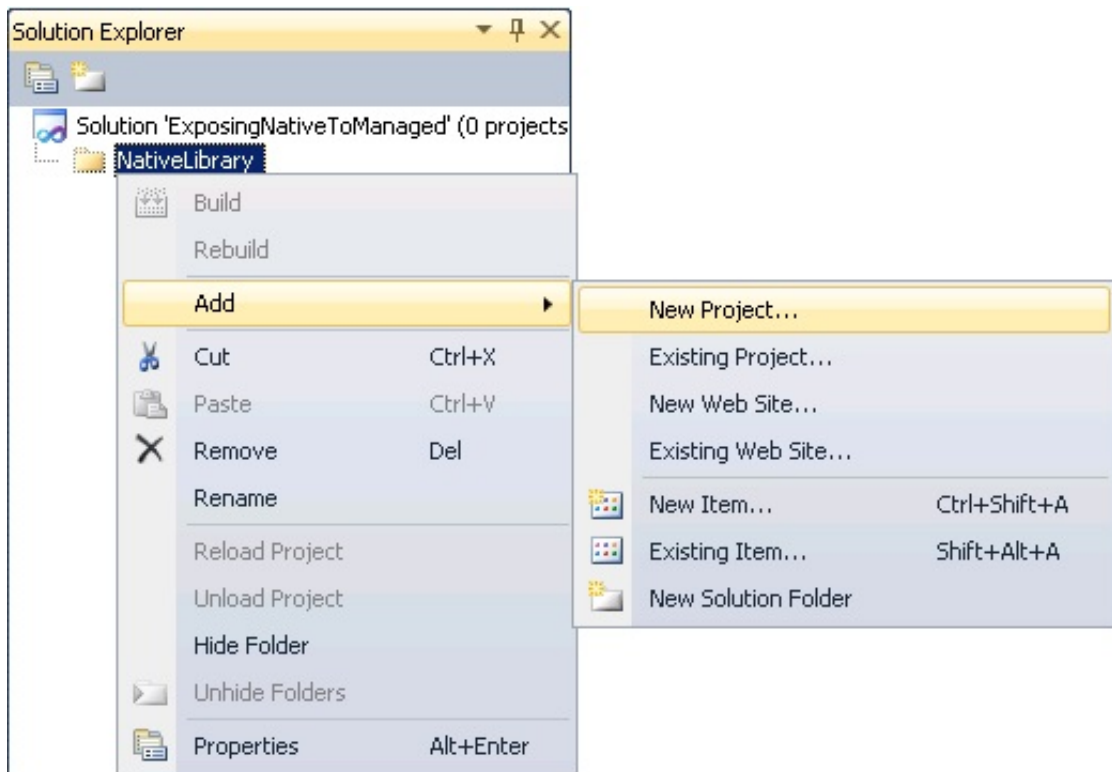
Creating the native library

The Worker class

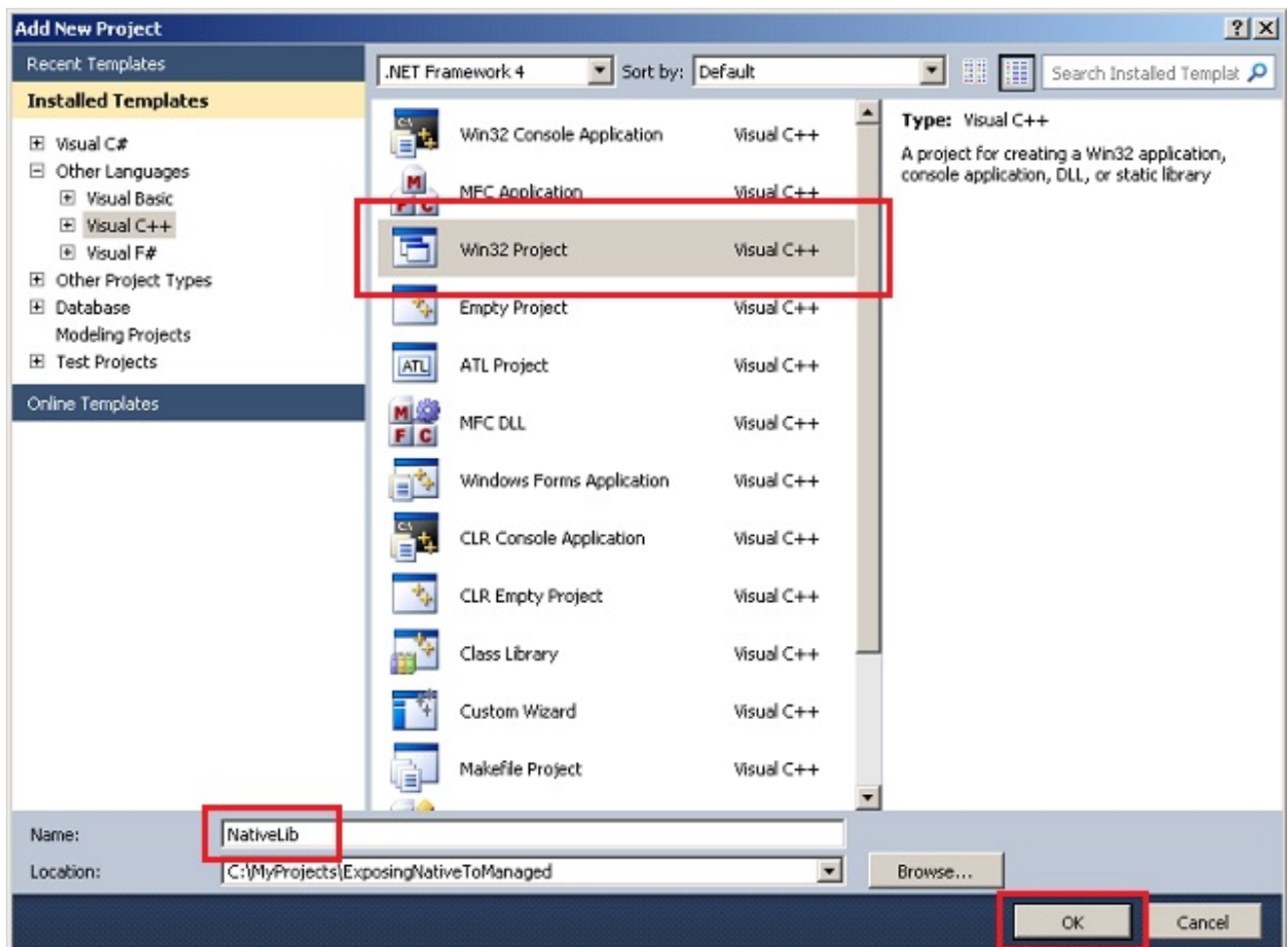
Before starting the discussion about wrapping a native implementation, we need a native implementation to wrap. In this section, we describe the content of our example native library. In the [next section](#), we'll see how we can wrap this native library as a managed DLL, that can be used in .NET projects.

The first step for creating our native library is creating an appropriate project. That can be done by the following steps:

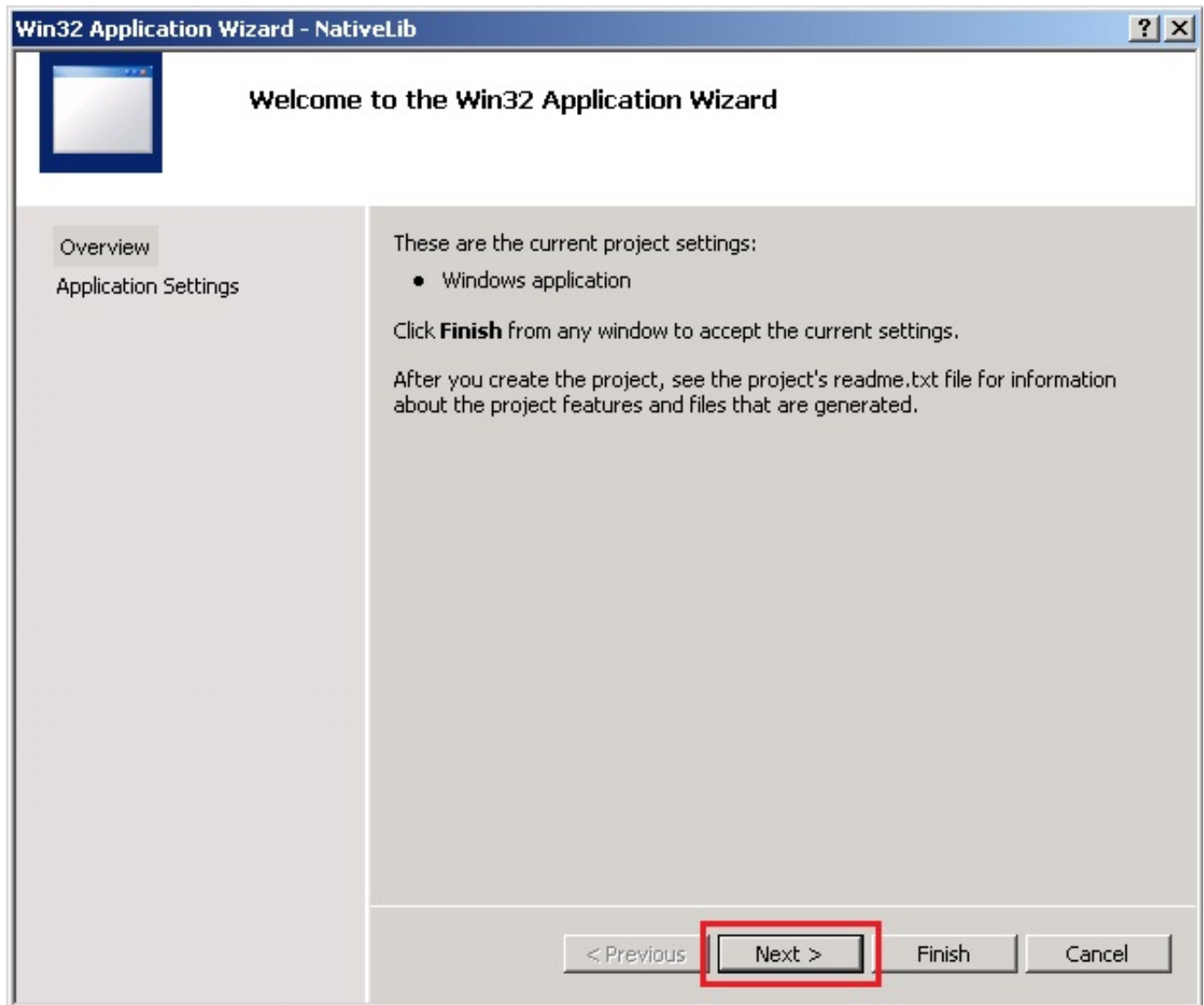
1. Add new project to our solution:



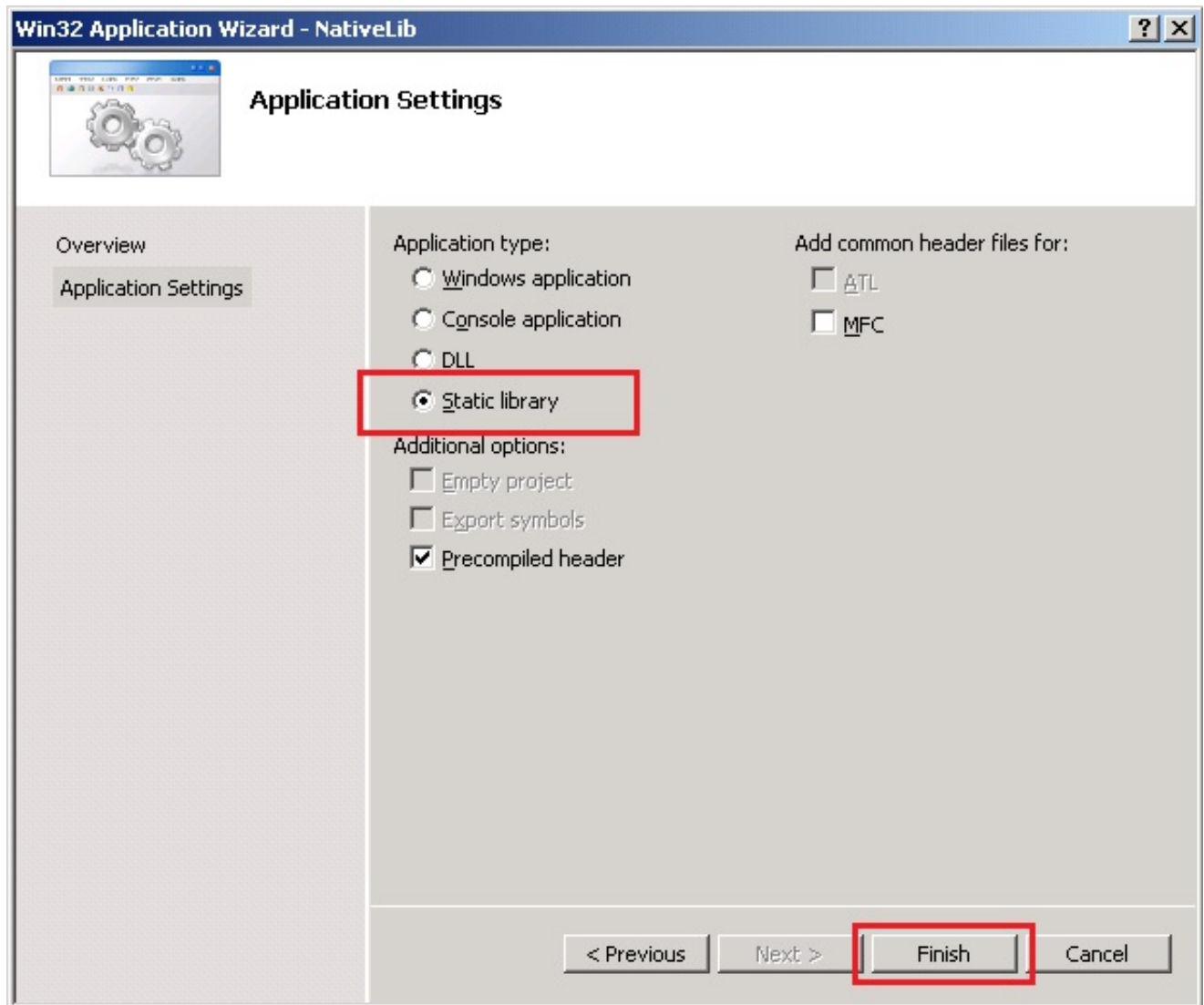
2. In the opened dialog, choose "Win32 Project", Enter a name and, click "OK":



3. In the opened dialog, click "Next":



4. In the opened dialog, choose "Static library" for the application type and, click "Finish":



Now, after we created our project, we can write our implementation. In our case, we create a "worker" that executes given tasks:

```
class Worker
{
public:
    Worker();
    virtual ~Worker();
}
```

Implement Worker tasks

For executing tasks with our "worker", we:

1. Create a class for presenting a task:

```
class WorkerTask
{
public:
    WorkerTask(int outerLoopCount, int innerLoopCount, bool isEndingTask = false);
    WorkerTask(const WorkerTask& src);
    ~WorkerTask();

    void Execute();

    bool IsEndingTask() const { return m_isEndingTask; }

private:
```

```

    unsigned int m_outerLoopCount;
    unsigned int m_innerLoopCount;
    bool m_isEndingTask;
};

```

2. Add an implementation for the task's execution:

```

void WorkerTask::Execute()
{
    // Do some work.

    for (unsigned int outerCount = 0; outerCount < m_outerLoopCount; outerCount++)
    {
        int* pi = new int[m_innerLoopCount];

        for (unsigned int innerCount = 0; innerCount < m_innerLoopCount; innerCount++)
        {
            pi[innerCount] = innerCount * outerCount;
        }

        delete[] pi;
    }
}

```

3. Add a **queue** for holding the waiting tasks:

```
std::queue<WorkerTask> m_waitingTasks;
```

4. Add a function for queuing tasks:

```

void Worker::QueueWorkerTask(const WorkerTask& task)
{
    m_waitingTasks.push(task);
}

```

Run Worker threads

For running our "worker" algorithm (executing queued tasks) using some threads we:

1. Add a function for waiting on the tasks queue and, get the queued tasks:

```

WorkerTask Worker::GetQueuedWorkerTask()
{
    bool isTaskFound = false;

    WorkerTask res(0,0);

    while (!isTaskFound)
    {
        if (!m_waitingTasks.empty())
        {
            res = m_waitingTasks.front();
            m_waitingTasks.pop();
            isTaskFound = true;
        }
    }

    return res;
}

```

2. Add a function for running the "worker" thread's algorithm:

```
unsigned int __stdcall Worker::WorkerThreadProc(void* pParam)
```

```

{
    Worker* theWorker = reinterpret_cast<Worker*>(pParam);

    while (true)
    {
        WorkerTask task = theWorker->GetQueuedWorkerTask();

        if (task.IsEndingTask())
        {
            break;
        }

        task.Execute();
    }

    return 0;
}

```

3. Add a **vector** for holding the "worker" threads' handles:

```
std::vector<HANDLE> m_workerThreadsHandles;
```

4. Add a function for starting the "worker" threads:

```

void Worker::Start()
{
    unsigned int threadID = 0;

    //Start the worker's threads.
    for(int threadInx = 0; threadInx < WORKER_THREADS_NUM; threadInx++)
    {
        m_workerThreadsHandles[threadInx] =
            (HANDLE)::_beginthreadex(0, 0, WorkerThreadProc, (void*)this, 0, &threadID);
    }
}

```

5. Add a function for stopping the "worker" threads:

```

void Worker::Stop()
{
    // Queue ending tasks.
    for(int endingTaskInx = 0; endingTaskInx < WORKER_THREADS_NUM; endingTaskInx++)
    {
        QueueWorkerTask(WorkerTask(0,0,true));
    }

    //Stop the worker's threads.
    for(int threadInx = 0; threadInx < WORKER_THREADS_NUM; threadInx++)
    {
        HANDLE currHandle = m_workerThreadsHandles[threadInx];

        if (0 == currHandle)
        {
            continue;
        }

        // Let the thread to be ended.
        ::WaitForSingleObject(currHandle, INFINITE);

        // Close the thread.
        ::CloseHandle(currHandle);
        m_workerThreadsHandles[threadInx] = 0;
    }
}

```

Protect critical sections

When developing multi-threaded applications, we usually want to protect some code sections from simultaneously multi-threaded access. In our case, we want to protect the access to the task's queue. For that purpose we:

1. Add a class for holding a critical-section:

```
// .h

class CriticalSectionHolder
{
public:
    CriticalSectionHolder(void);
    virtual ~CriticalSectionHolder(void);

    LPCRITICAL_SECTION GetCriticalSection() { return &m_criticalSection; }

private:
    CRITICAL_SECTION m_criticalSection;
};

// .cpp

CriticalSectionHolder::CriticalSectionHolder(void)
{
    ::InitializeCriticalSection(&m_criticalSection);
}

CriticalSectionHolder::~CriticalSectionHolder(void)
{
    ::DeleteCriticalSection(&m_criticalSection);
}
```

2. Add a class for encapsulating the locking behavior (entering and leaving) of a critical-section:

```
// .h

class CriticalSectionLocker
{
public:
    CriticalSectionLocker(LPCRITICAL_SECTION pCriticalSection);
    virtual ~CriticalSectionLocker();

private:
    LPCRITICAL_SECTION m_pCriticalSection;
};

// .cpp

CriticalSectionLocker::CriticalSectionLocker(LPCRITICAL_SECTION pCriticalSection)
    : m_pCriticalSection(pCriticalSection)
{
    ::EnterCriticalSection(m_pCriticalSection);
}

CriticalSectionLocker::~CriticalSectionLocker()
{
    ::LeaveCriticalSection(m_pCriticalSection);
}
```

3. Add a **CriticalSectionHolder** data-member to our **Worker**:

```
CriticalSectionHolder m_criticalSection;
```


4. Lock the appropriate code sections:

```
void Worker::QueueWorkerTask(const WorkerTask& task)
{
    CriticalSectionLocker locker(m_criticalSection.GetCriticalSection());

    m_waitingTasks.push(task);
}

WorkerTask Worker::GetQueuedWorkerTask()
{
    // ...

    while (!isTaskFound)
    {
        CriticalSectionLocker locker(m_criticalSection.GetCriticalSection());

        if (!m_waitingTasks.empty())
        {
            // ...
        }
    }

    // ...
}
```

Gather execution time statistics

For testing the working time of the **Worker** we:

1. Add data-members for holding the start and stop clocks:

```
clock_t m_startClocks;
clock_t m_stopClocks;
```

2. Set the appropriate clock when starting and stopping:

```
void Worker::Start()
{
    m_startClocks = clock();

    // ...
}

void Worker::Stop()
{
    // ...

    m_stopClocks = clock();
}
```

3. Add a function for getting the **Worker**'s working time:

```
double Worker::GetWorkingSeconds() const
{
    return (double)((m_isWorking ? clock() : m_stopClocks) - m_startClocks) /
    CLOCKS_PER_SEC;
}
```

In addition to that, we can test also the execution time of each task. For that purpose we:

1. Add a data-member for holding the number of the execution seconds:

```
double m_executionSeconds;
```

2. Calculate the number of the execution seconds for the task's execution:

```
void WorkerTask::Execute()
{
    clock_t beginClocks = clock();

    // ...

    clock_t endClocks = clock();

    m_executionSeconds = (double)(endClocks - beginClocks) / CLOCKS_PER_SEC;
}
```

For presenting the gathered statistics, we:

1. Add a **list** for holding the executed tasks:

```
std::list<WorkerTask> m_executedTasks;
```

2. Store the executed tasks:

```
void Worker::AddExecutedTask(const WorkerTask& task)
{
    CriticalSectionLocker locker(m_criticalSection.GetCriticalSection());

    m_executedTasks.push_back(task);
}

unsigned int __stdcall Worker::WorkerThreadProc(void* pParam)
{
    Worker* theWorker = reinterpret_cast<Worker*>(pParam);

    while (true)
    {
        // ...

        task.Execute();

        theWorker->AddExecutedTask(task);
    }

    return 0;
}
```

3. Add a function for getting the task's description:

```
string WorkerTask::GetTaskDescription() const
{
    stringstream ss;
    ss << "(" << m_outerLoopCount << ",
        " << m_innerLoopCount << ") Execution seconds: "
        << m_executionSeconds << ".";

    return ss.str();
}
```

4. Add a function for printing the **Worker**'s statistics:

```
void Worker::PrintStatistics()
{
    CriticalSectionLocker locker(m_criticalSection.GetCriticalSection());

    cout << "Worker working seconds: " << GetWorkingSeconds() << endl;
    cout << "Tasks executions seconds:" << endl;
```

```

for_each(m_executedTasks.begin(), m_executedTasks.end(), [](WorkerTask task) -> void
{
    cout << "\t" << task.GetTaskDescription() << endl;
});
}

```

Provide three ways for executing the worker tasks

In order to enable some ways for testing the managed and unmanaged interoperability, we add three functions for executing the **Worker**'s tasks:

1. Queue one task for asynchronous execution:

```

void Worker::QueueWorkerTask(unsigned int outerLoopCount, unsigned int innerLoopCount)
{
    QueueWorkerTask(WorkerTask(outerLoopCount, innerLoopCount));
}

```

2. Queue some demo tasks for asynchronous execution:

```

void Worker::QueueDemoWorkerTasks()
{
    for (unsigned int taskLoopCount = 10000; taskLoopCount < 50000; taskLoopCount += 1000)
    {
        QueueWorkerTask(taskLoopCount, taskLoopCount);
    }
}

```

3. Execute a task synchronously:

```

void Worker::ExecuteWorkerTask(unsigned int outerLoopCount, unsigned int innerLoopCount)
{
    WorkerTask task(outerLoopCount, innerLoopCount);
    task.Execute();

    AddExecutedTask(task);
}

```

Exposing our native library as a managed DLL

First approach - Call native DLL exports using P/Invoke

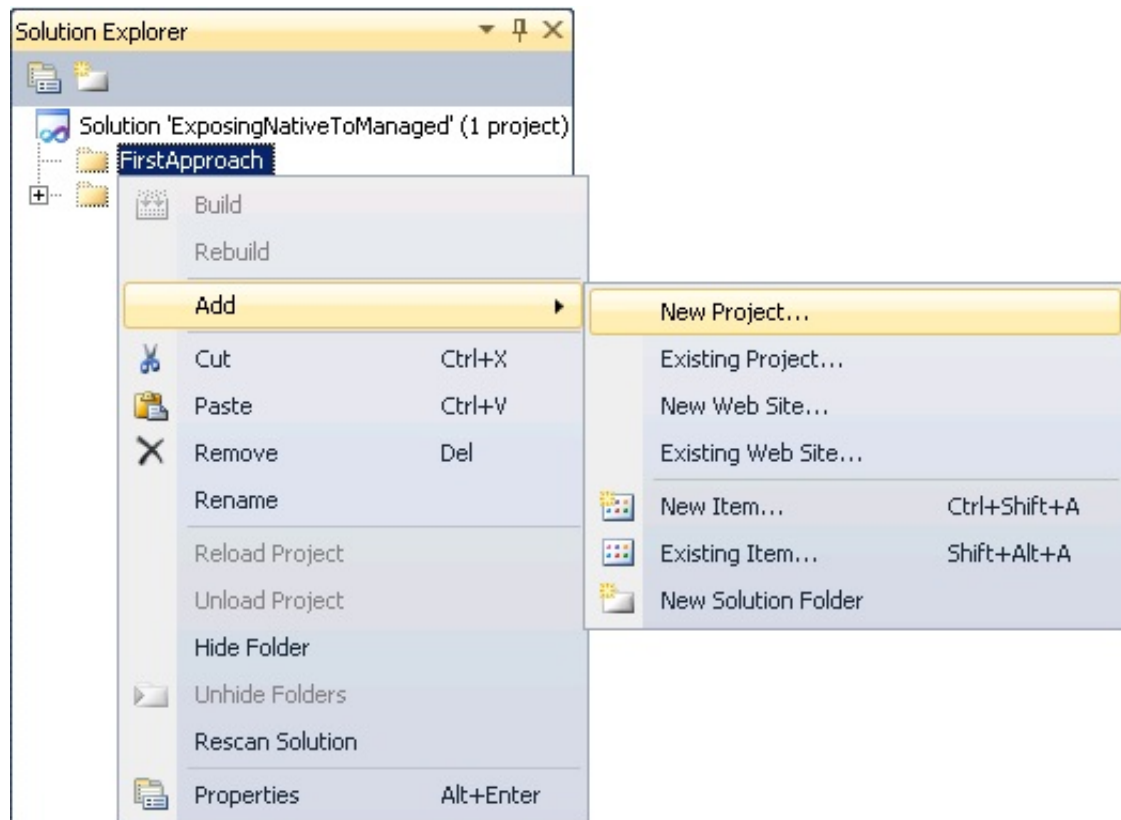
Native DLL

Create a native DLL project

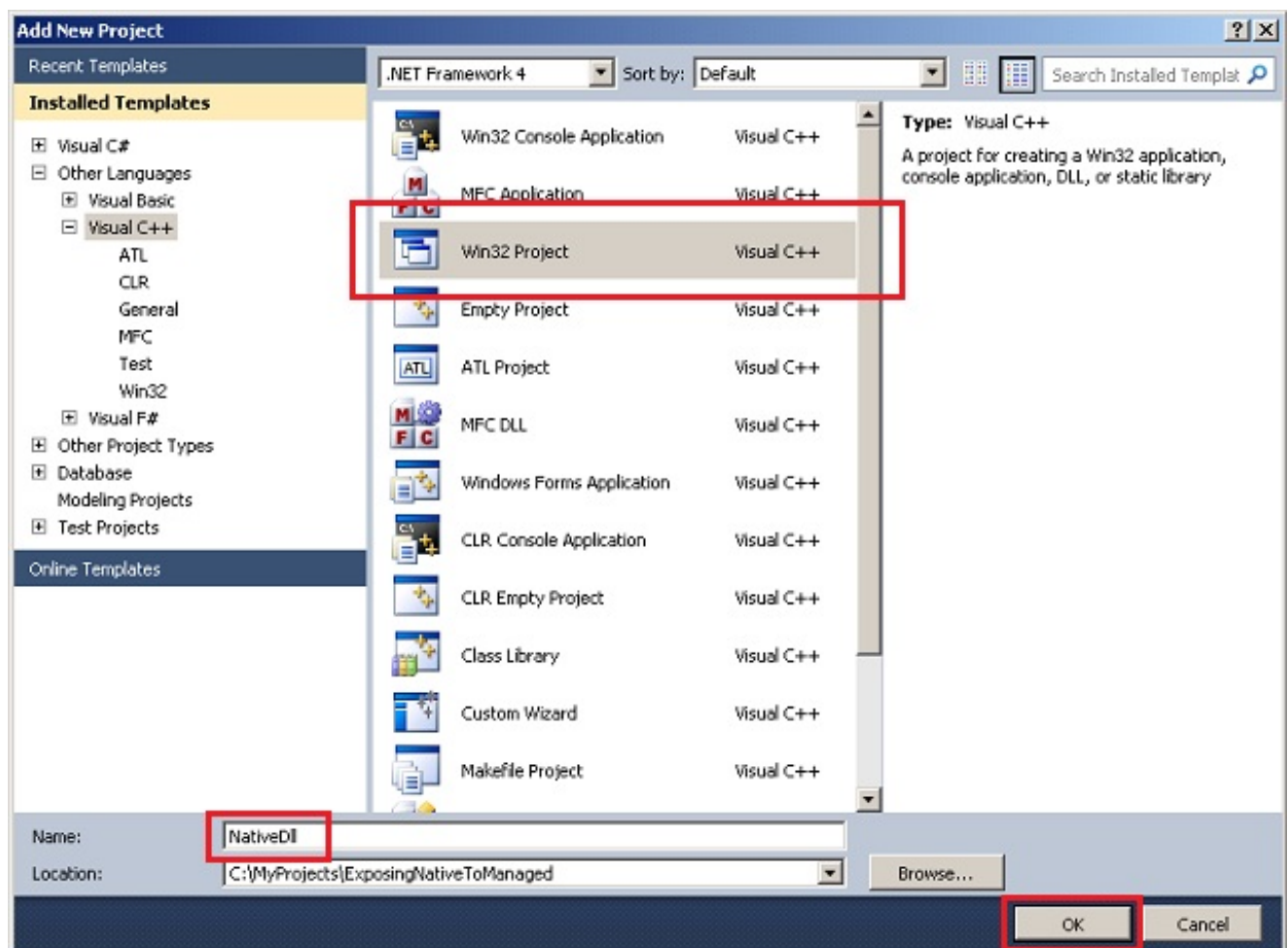
Now, after we have our native **Worker**, we can wrap it with a managed class. The first approach we take for doing that is: wrapping our native static library (.lib) with a native dynamic-link library (.dll) and, create another DLL with a managed wrapper class that calls the native DLL exports.

The first step is to create a project for our native DLL. That can be done by the following steps:

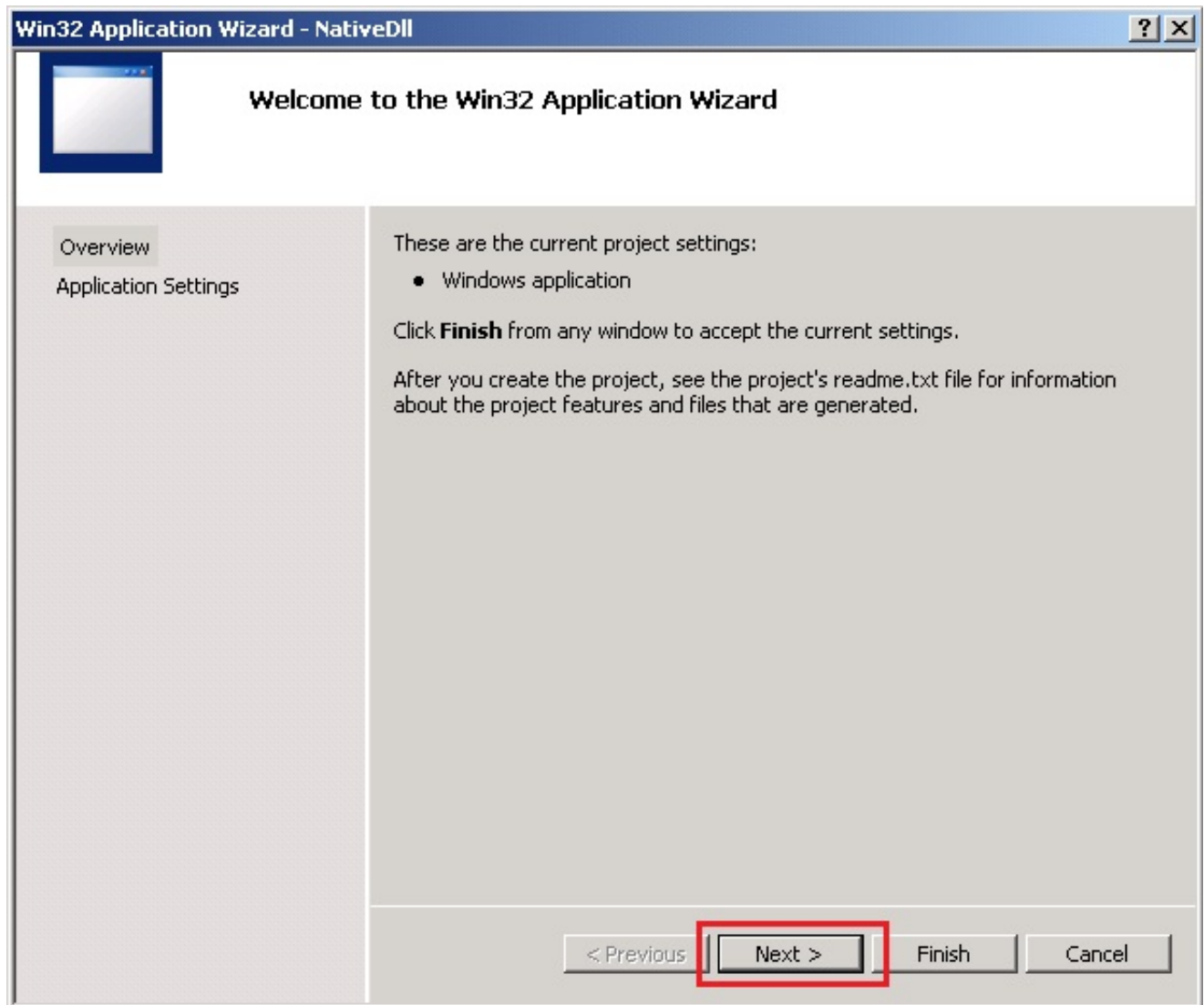
1. Add new project to our solution:



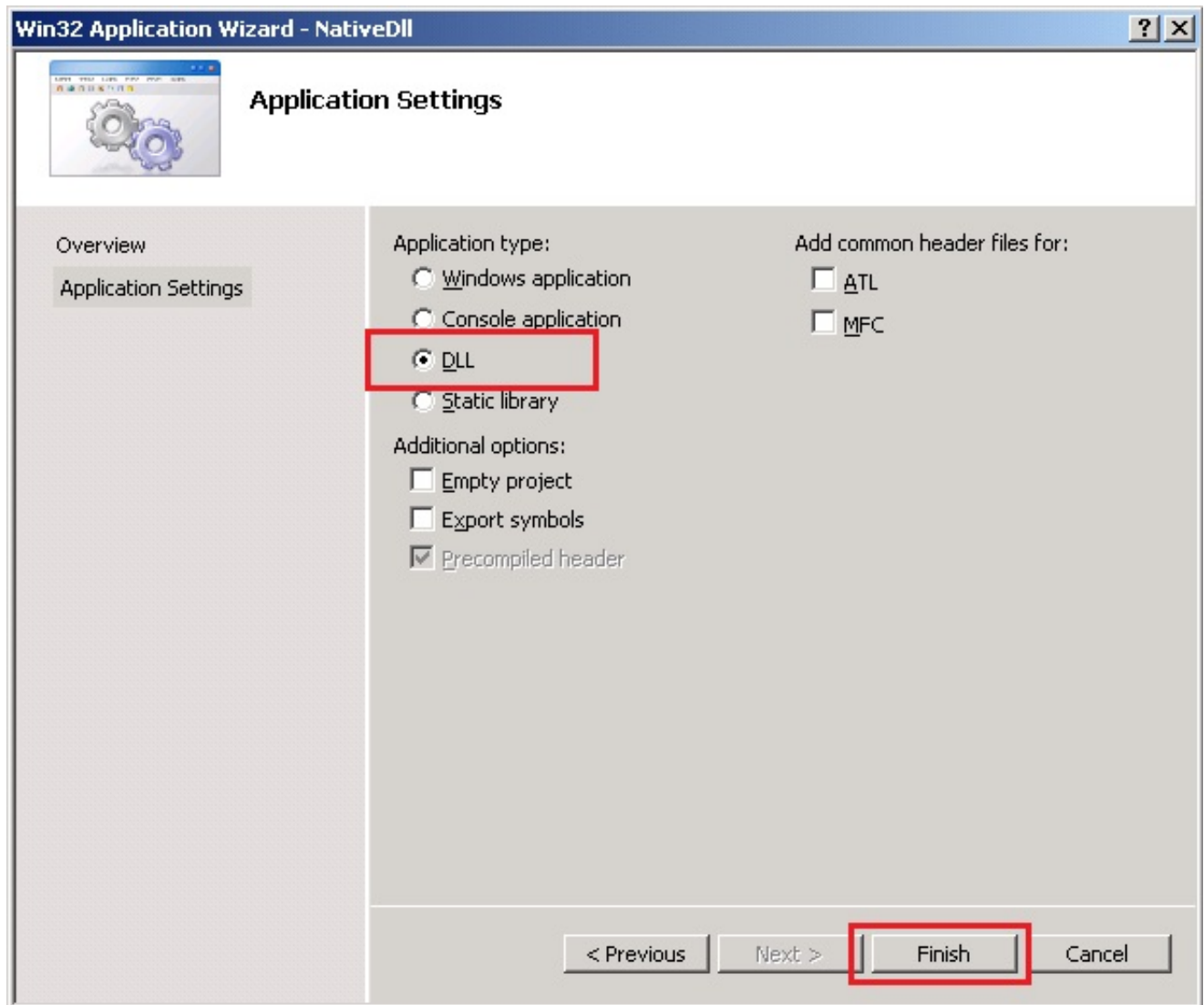
2. In the opened dialog, choose "Win32 Project", Enter a name and, click "OK":



3. In the opened dialog, click "Next":



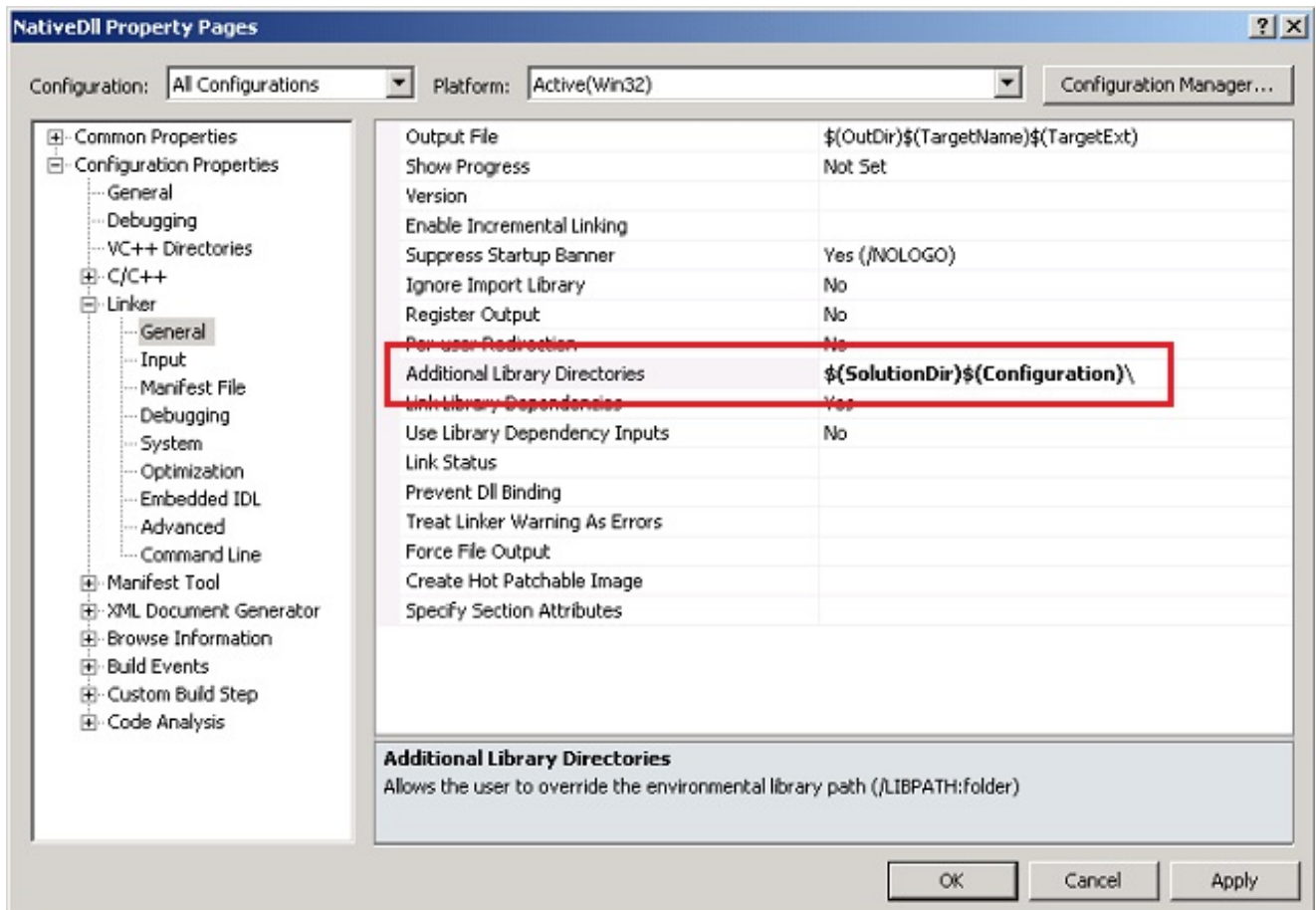
4. In the opened dialog, choose "DLL" for the application type and, click "Finish":



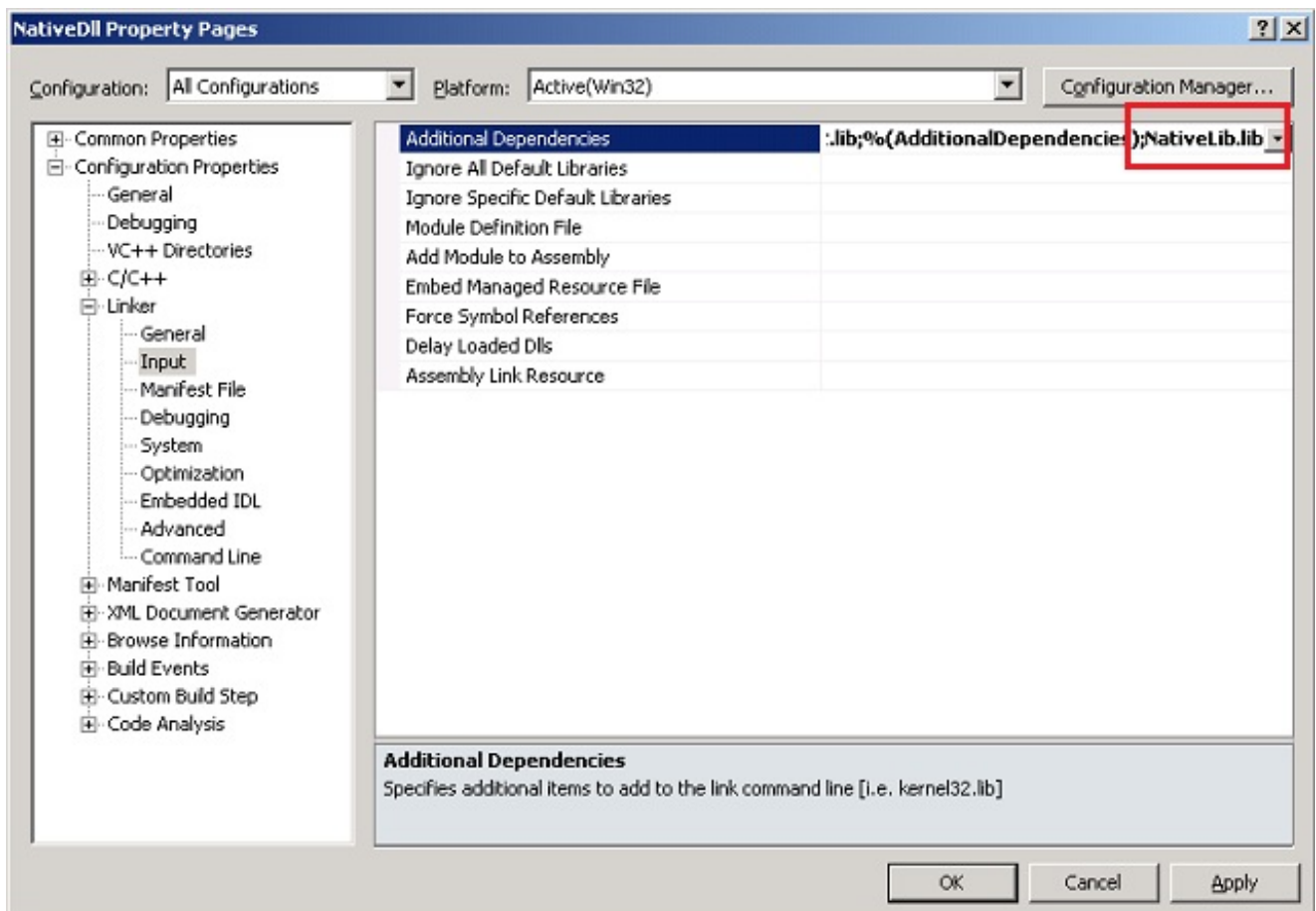
Link our native static library

Ok, we have a project for our native DLL. The next step is to link our native library with our DLL. That can be done by the following steps:

1. Add the output folder of our native static library, to the "Additional Library Directories" of our project's properties:



2. Add our native static library, to the "Additional Dependencies" of our project's properties:



Add functions for exposing the logic

So, we have our static library linked to our project. The next step is adding functions for exposing the wanted logic.

In our static library, we created a **Worker** that can execute given tasks. In our DLL, we want to expose functions for creating workers and, execute tasks using them.

For enabling creation of workers, we:

1. Add a **map** for holding the active workers:

```
map<unsigned int, Worker*> g_theWorkers;
```

2. Add a function for creating a new worker and, return an identifier for the new created worker:

```
// Note: In a real DLL, we should lock the creation of the workers,
//       the deletion of the workers and, the workers' methods calls,
//       in order to prevent some multi-threaded issues (e.g.
//       creating 2 workers but holding only one - since the same id
//       is assigned to the both of the workers by different threads,
//       using a worker that has been deleted by another thread, etc...).
//       We can simply achieve that goal, by using a CriticalSectionHolder
//       global variable for holding a critical-section and, using a
//       CriticalSectionLocker variable for locking the critical-section
//       for each function.
//       But, in order to enable testing the third way (execute worker
//       tasks synchronously) by calling it parallely (using .NET TPL)
//       we don't use a critical-section here (Since, if the function
//       will be locked by a critical-section, the tasks won't run
//       parallely...).

unsigned int CreateWorker()
{
    static unsigned int s_nextWorkerId = 1;

    unsigned int currWorkerId = s_nextWorkerId;
    g_theWorkers[currWorkerId] = new Worker();

    s_nextWorkerId++;

    return currWorkerId;
}
```

3. Add a function for deleting a worker:

```
void DeleteWorker(unsigned int workerId)
{
    auto workerItr = g_theWorkers.find(workerId);
    if (workerItr != g_theWorkers.end())
    {
        delete workerItr->second;
        g_theWorkers.erase(workerItr);
    }
}
```

For enabling performing operations on a worker, we add functions that find a worker according to a given identifier and, call the appropriate function on it:

```
void Start(unsigned int workerId)
{
    auto workerItr = g_theWorkers.find(workerId);
    if (workerItr != g_theWorkers.end())
    {
        workerItr->second->Start();
    }
}

void Stop(unsigned int workerId)
{
    auto workerItr = g_theWorkers.find(workerId);
```



```

    if (workerItr != g_theWorkers.end())
    {
        workerItr->second->Stop();
    }
}

void QueueDemoWorkerTasks(unsigned int workerId)
{
    auto workerItr = g_theWorkers.find(workerId);
    if (workerItr != g_theWorkers.end())
    {
        workerItr->second->QueueDemoWorkerTasks();
    }
}

void QueueWorkerTask(unsigned int workerId,
    unsigned int outerLoopCount, unsigned int innerLoopCount)
{
    auto workerItr = g_theWorkers.find(workerId);
    if (workerItr != g_theWorkers.end())
    {
        workerItr->second->QueueWorkerTask(outerLoopCount, innerLoopCount);
    }
}

void ExecuteWorkerTask(unsigned int workerId,
    unsigned int outerLoopCount, unsigned int innerLoopCount)
{
    auto workerItr = g_theWorkers.find(workerId);
    if (workerItr != g_theWorkers.end())
    {
        workerItr->second->ExecuteWorkerTask(outerLoopCount, innerLoopCount);
    }
}

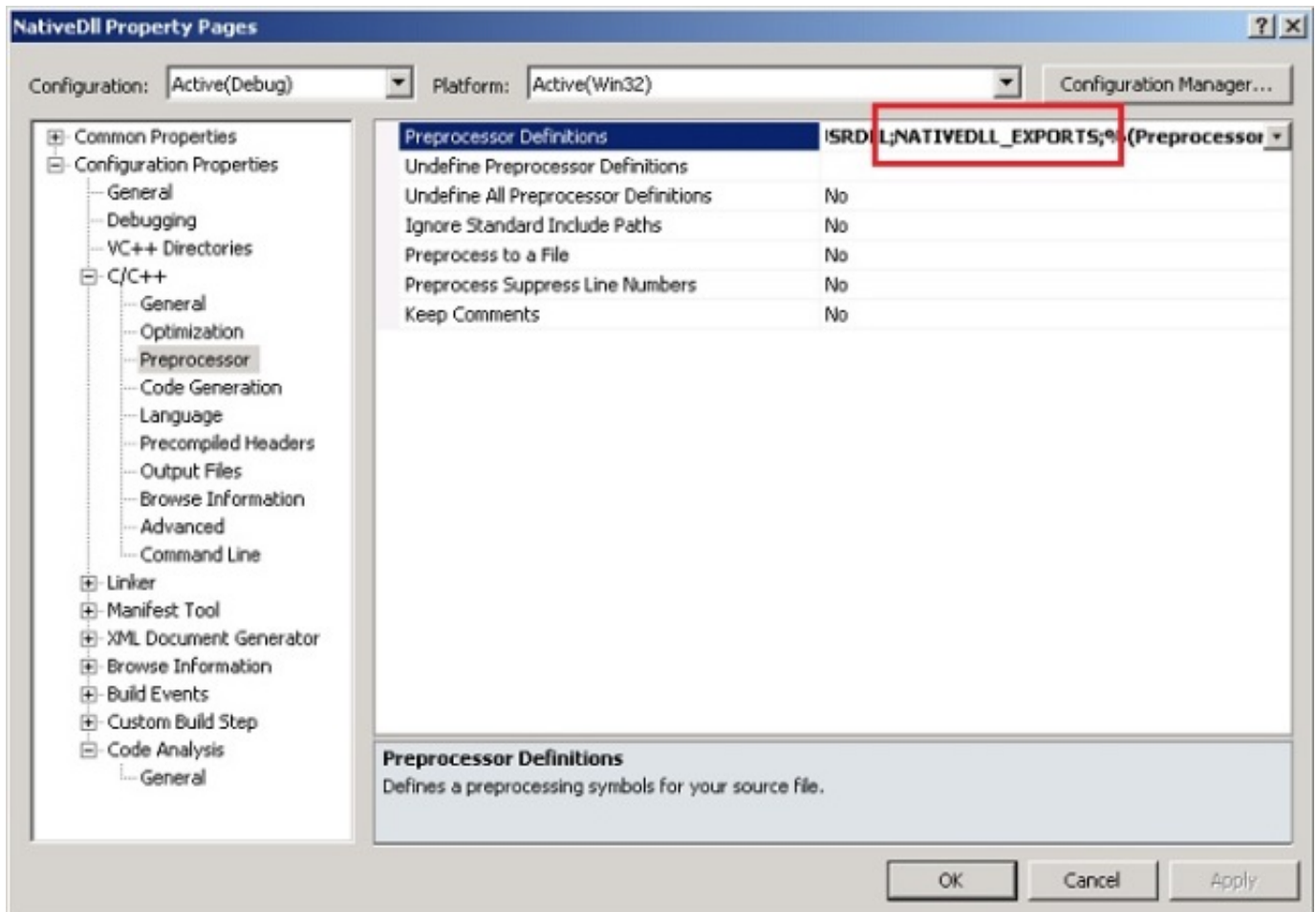
void PrintStatistics(unsigned int workerId)
{
    auto workerItr = g_theWorkers.find(workerId);
    if (workerItr != g_theWorkers.end())
    {
        workerItr->second->PrintStatistics();
    }
}

double GetWorkingSeconds(unsigned int workerId)
{
    auto workerItr = g_theWorkers.find(workerId);
    if (workerItr != g_theWorkers.end())
    {
        return workerItr->second->GetWorkingSeconds();
    }
}

```

Export our DLL's functions

When the DLL's project has been created, an additional definition has been added to the project's properties (**NATIVEDLL_EXPORTS**):



That definition helps us for compiling our functions as exports when compiling our DLL and, compile our functions as imports when compiling other projects (that uses our DLL's functions).

In order to apply that behavior, we:

1. Add a definition for treating our functions as DLL-exports or DLL-imports, according to the **NATIVEDLL_EXPORTS** definition:

```
#ifdef NATIVEDLL_EXPORTS
#define NATIVEDLL_API __declspec(dllexport)
#else
#define NATIVEDLL_API __declspec(dllimport)
#endif
```

2. Use that definition for all of our DLL's functions:

```
NATIVEDLL_API unsigned int CreateWorker();
NATIVEDLL_API void DeleteWorker(unsigned int workerId);
NATIVEDLL_API void Start(unsigned int workerId);
NATIVEDLL_API void Stop(unsigned int workerId);
NATIVEDLL_API void QueueDemoWorkerTasks(unsigned int workerId);
NATIVEDLL_API void QueueWorkerTask(unsigned int workerId,
    unsigned int outerLoopCount, unsigned int innerLoopCount);
NATIVEDLL_API void ExecuteWorkerTask(unsigned int workerId,
    unsigned int outerLoopCount, unsigned int innerLoopCount);
NATIVEDLL_API void PrintStatistics(unsigned int workerId);
NATIVEDLL_API double GetWorkingSeconds(unsigned int workerId);
```

Calling convention

One of the things that should be taken in consideration when exporting functions that takes some parameters is: [Calling Conventions](#). Generally, it's the way for telling who (the Caller or the Callee) is responsible for cleaning the stack after the function has been finished. The default calling convention for a C++ program is **__cdecl**. Anyway, we can declare it

explicitly (using the `__cdecl` keyword):

```
NATIVEDLL_API unsigned int __cdecl CreateWorker();
NATIVEDLL_API void __cdecl DeleteWorker(unsigned int workerId);
NATIVEDLL_API void __cdecl Start(unsigned int workerId);
NATIVEDLL_API void __cdecl Stop(unsigned int workerId);
NATIVEDLL_API void __cdecl QueueDemoWorkerTasks(unsigned int workerId);
NATIVEDLL_API void __cdecl QueueWorkerTask(unsigned int workerId,
    unsigned int outerLoopCount, unsigned int innerLoopCount);
NATIVEDLL_API void __cdecl ExecuteWorkerTask(unsigned int workerId,
    unsigned int outerLoopCount, unsigned int innerLoopCount);
NATIVEDLL_API void __cdecl PrintStatistics(unsigned int workerId);
NATIVEDLL_API double __cdecl GetWorkingSeconds(unsigned int workerId);
```

Keep the function entry point name as the original function name

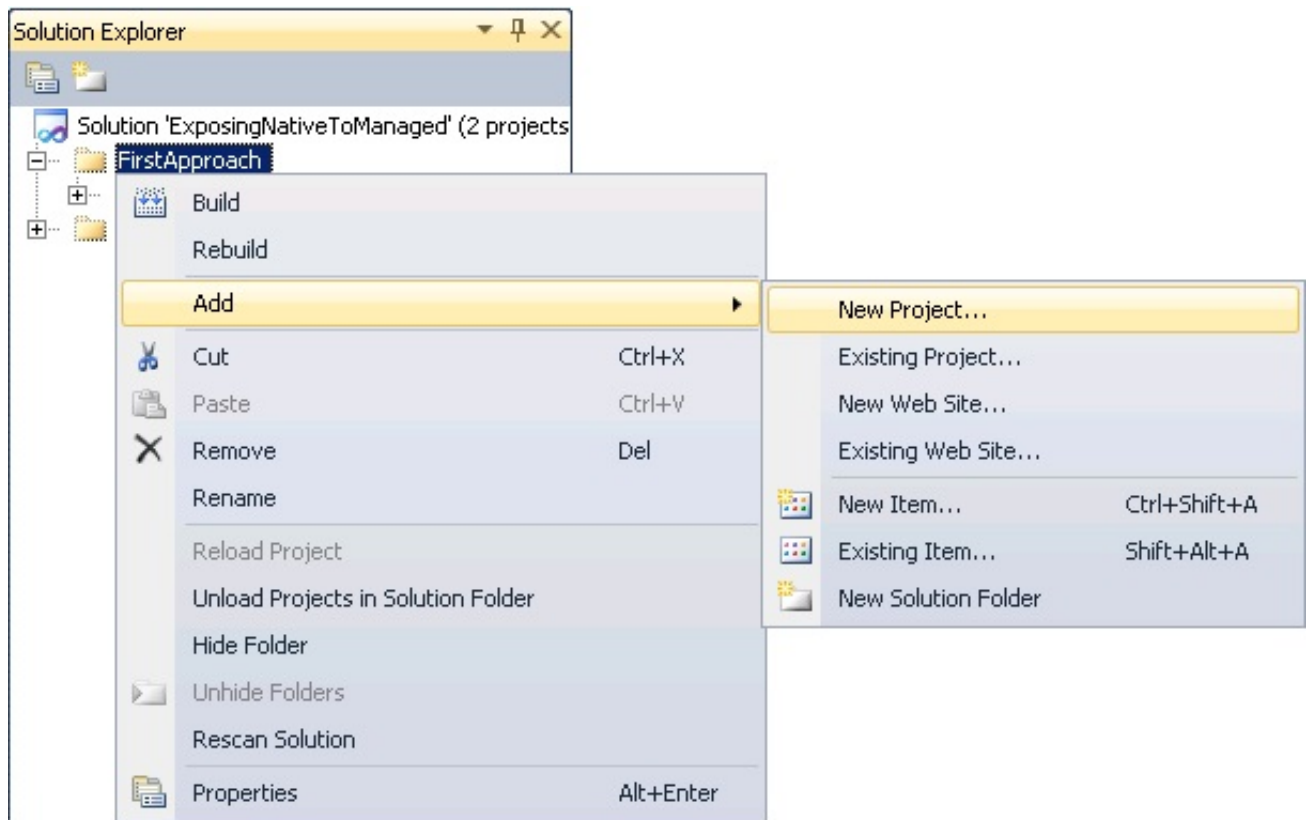
Another thing that should be taken in consideration when exporting functions, is the functions' entry-points' names. In order to enable [Function Overloading](#), the C++ compiler performs [Name mangling](#) on the functions' names. For preventing the C++ name mangling (and keep the functions entry-points as the original functions' names), we can declare the functions with the **extern "C"** keyword:

```
extern "C" NATIVEDLL_API unsigned int __cdecl CreateWorker();
extern "C" NATIVEDLL_API void __cdecl DeleteWorker(unsigned int workerId);
extern "C" NATIVEDLL_API void __cdecl Start(unsigned int workerId);
extern "C" NATIVEDLL_API void __cdecl Stop(unsigned int workerId);
extern "C" NATIVEDLL_API void __cdecl QueueDemoWorkerTasks(unsigned int workerId);
extern "C" NATIVEDLL_API void __cdecl QueueWorkerTask(unsigned int workerId,
    unsigned int outerLoopCount, unsigned int innerLoopCount);
extern "C" NATIVEDLL_API void __cdecl ExecuteWorkerTask(unsigned int workerId,
    unsigned int outerLoopCount, unsigned int innerLoopCount);
extern "C" NATIVEDLL_API void __cdecl PrintStatistics(unsigned int workerId);
extern "C" NATIVEDLL_API double __cdecl GetWorkingSeconds(unsigned int workerId);
```

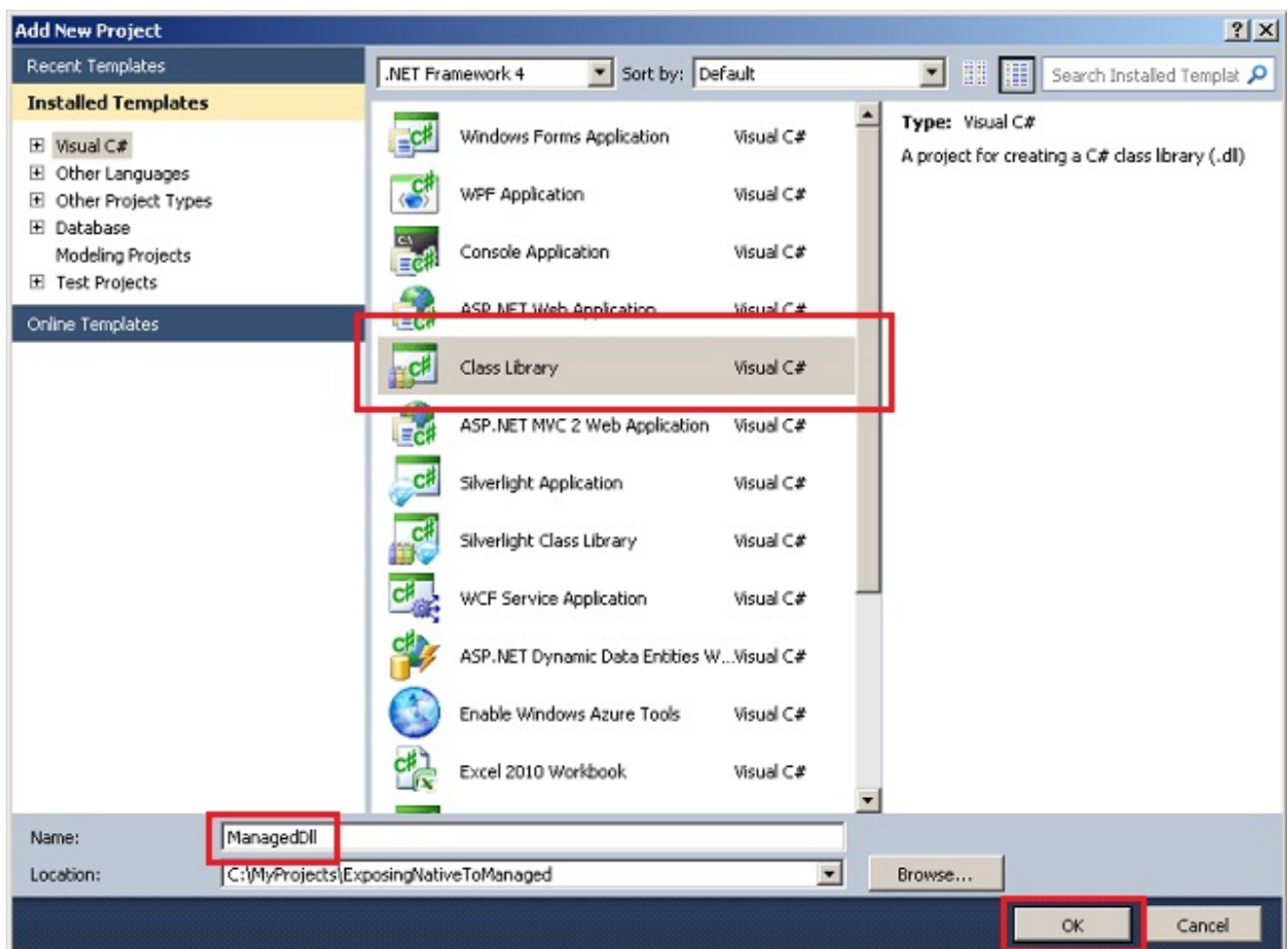
Managed DLL

Now, after we have our native DLL, we can wrap it with a managed DLL that can be added as a reference for .NET projects. For creating a project for our managed DLL, we:

1. Add new project to our solution:



2. In the opened dialog, choose a C# "Class Library", Enter a name and, click "OK":



After we've created the project for our managed DLL, we can add a class that wraps the native DLL extensions:

```
public class ManagedWorker : IDisposable
{
}
```

In that class we:

1. Add methods for calling the native DLL's exports:

```
[DllImport("NativeDll.dll", EntryPoint = "CreateWorker")]
protected static extern uint _CreateWorker();

[DllImport("NativeDll.dll", EntryPoint = "DeleteWorker", CallingConvention =
CallingConvention.Cdecl)]
protected static extern void _DeleteWorker(uint workerId);

[DllImport("NativeDll.dll", EntryPoint = "Start", CallingConvention =
CallingConvention.Cdecl)]
protected static extern void _Start(uint workerId);

[DllImport("NativeDll.dll", EntryPoint = "Stop", CallingConvention =
CallingConvention.Cdecl)]
protected static extern void _Stop(uint workerId);

[DllImport("NativeDll.dll", EntryPoint = "QueueDemoWorkerTasks", CallingConvention =
CallingConvention.Cdecl)]
protected static extern void _QueueDemoWorkerTasks(uint workerId);

[DllImport("NativeDll.dll", EntryPoint = "QueueWorkerTask", CallingConvention =
CallingConvention.Cdecl)]
protected static extern void _QueueWorkerTask(uint workerId, uint outerLoopCount, uint
innerLoopCount);

[DllImport("NativeDll.dll", EntryPoint = "ExecuteWorkerTask", CallingConvention =
CallingConvention.Cdecl)]
protected static extern void _ExecuteWorkerTask(uint workerId, uint outerLoopCount, uint
innerLoopCount);

[DllImport("NativeDll.dll", EntryPoint = "PrintStatistics", CallingConvention =
CallingConvention.Cdecl)]
protected static extern void _PrintStatistics(uint workerId);

[DllImport("NativeDll.dll", EntryPoint = "GetWorkingSeconds", CallingConvention =
CallingConvention.Cdecl)]
protected static extern double _GetWorkingSeconds(uint workerId);
```

2. Store a native **Worker** instance, for each instance of the wrapper class:

```
protected uint _workerId;

public ManagedWorker()
{
    _workerId = _CreateWorker();
}

~ManagedWorker() // Finalize
{
    Dispose();
}

public void Dispose()
{
    _DeleteWorker(_workerId);
}
```

3. Add methods for calling the appropriate native DLL's exports:

```
public void Start()
{
    _Start(_workerId);
}
```

```

public void Stop()
{
    _Stop(_workerId);
}

public void QueueDemoWorkerTasks()
{
    _QueueDemoWorkerTasks(_workerId);
}

public void QueueWorkerTask(uint outerLoopCount, uint innerLoopCount)
{
    _QueueWorkerTask(_workerId, outerLoopCount, innerLoopCount);
}

public void ExecuteWorkerTask(uint outerLoopCount, uint innerLoopCount)
{
    _ExecuteWorkerTask(_workerId, outerLoopCount, innerLoopCount);
}

public void PrintStatistics()
{
    _PrintStatistics(_workerId);
}

public double GetWorkingSeconds()
{
    return _GetWorkingSeconds(_workerId);
}

```

As we can see, the calling-convention of the exported functions is **Cdecl** and, the entry-points' names of the exported functions are as same as the function's names (as discussed before). In addition to that, in order to make the things simple, all of the types of the exported functions are simple types. So that, we don't have to deal too much with marshaling. For more information about marshaling, you can read the MSDN topic about: [Marshaling Data with Platform Invoke](#).

Second approach - Wrap the native library with a managed C++/CLI project

Why C++/CLI

As we saw, we can expose a C++ native implementation, as a C++ native DLL that can be consumed within a C# managed DLL. In that manner, C++ native developers can create a C++ native DLL and, .NET developers can consume that DLL using their familiar programming language (e.g., C#, etc...). If we want to save the consumers of our library from dealing with the native DLL, we can create a managed DLL that does this work for them (as we did in the previous section). Using that approach, the consumers of our library have to consume two DLLs:

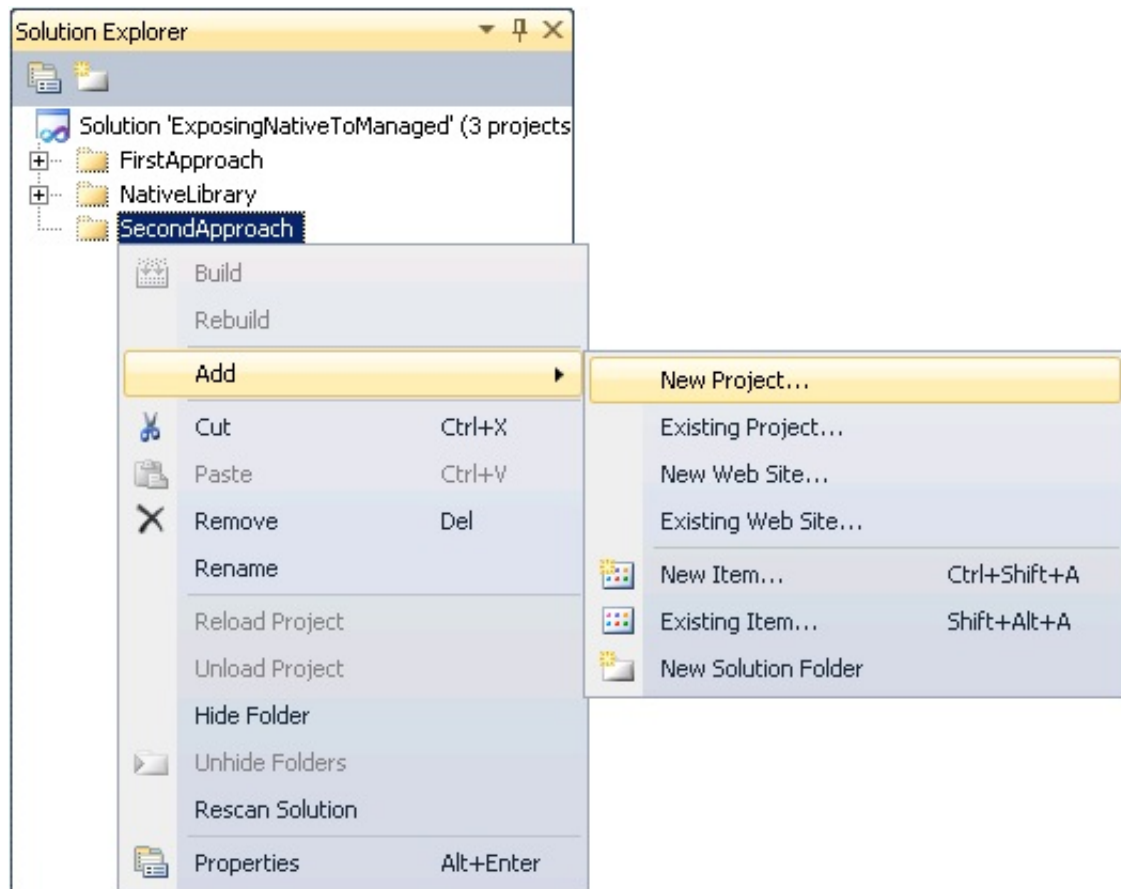
1. A managed DLL that can be added as a reference for .NET projects.
2. A native DLL for the native implementation.

Another approach for exposing our library is: creating one DLL that contains the managed and the native implementation. In order achieve that goal, we need a technology that enables interoperability between managed code and C++ native code, in one project. For that purpose, we have the C++/CLI language. The C++/CLI language, gives us an easy way for interoperability between managed and native C++ code, as we'll see in the following sections.

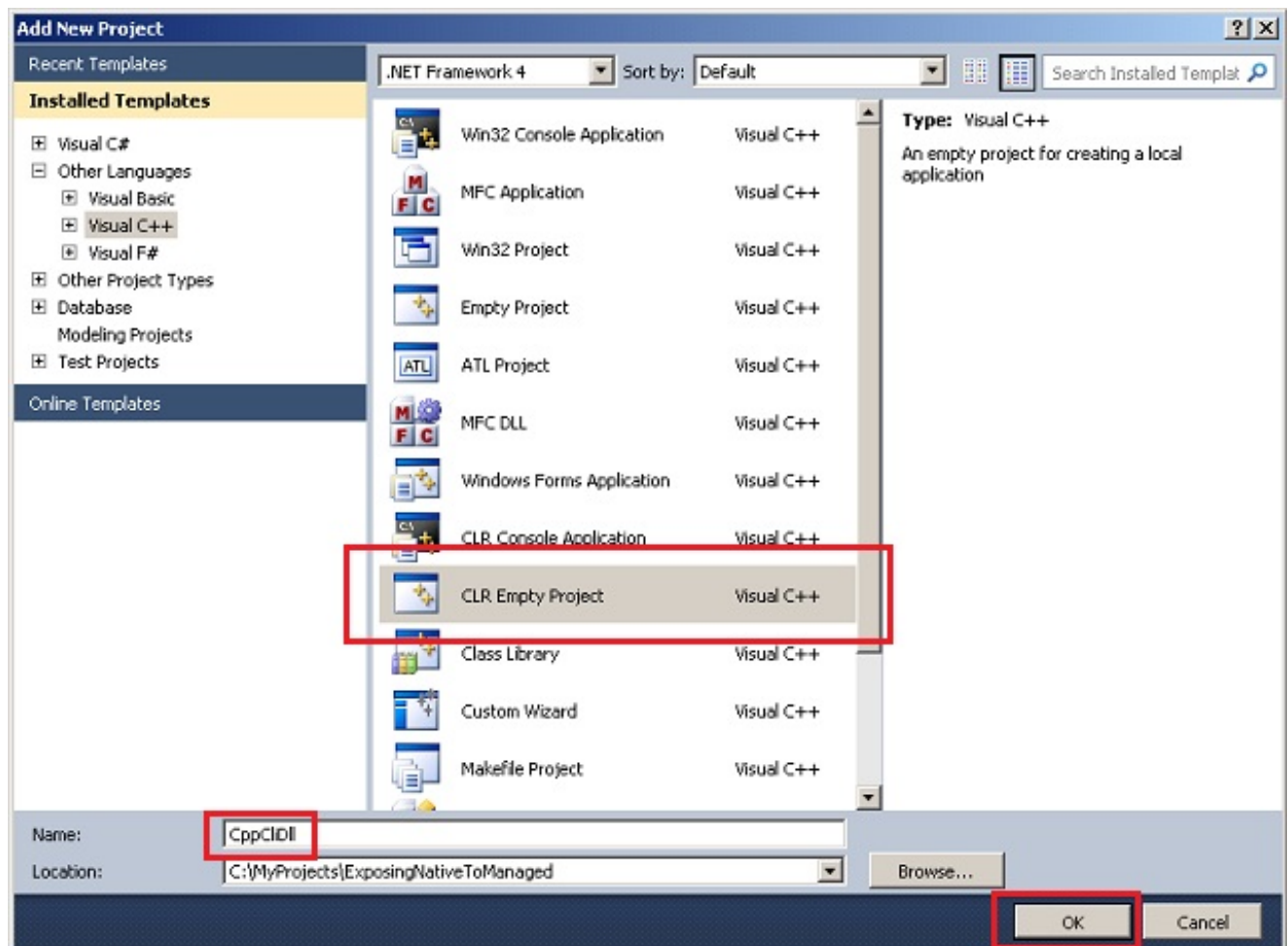
Create a C++/CLI project

The first step for creating our mixed (managed and native) DLL, is to create a C++/CLI project. That can be done by the following steps:

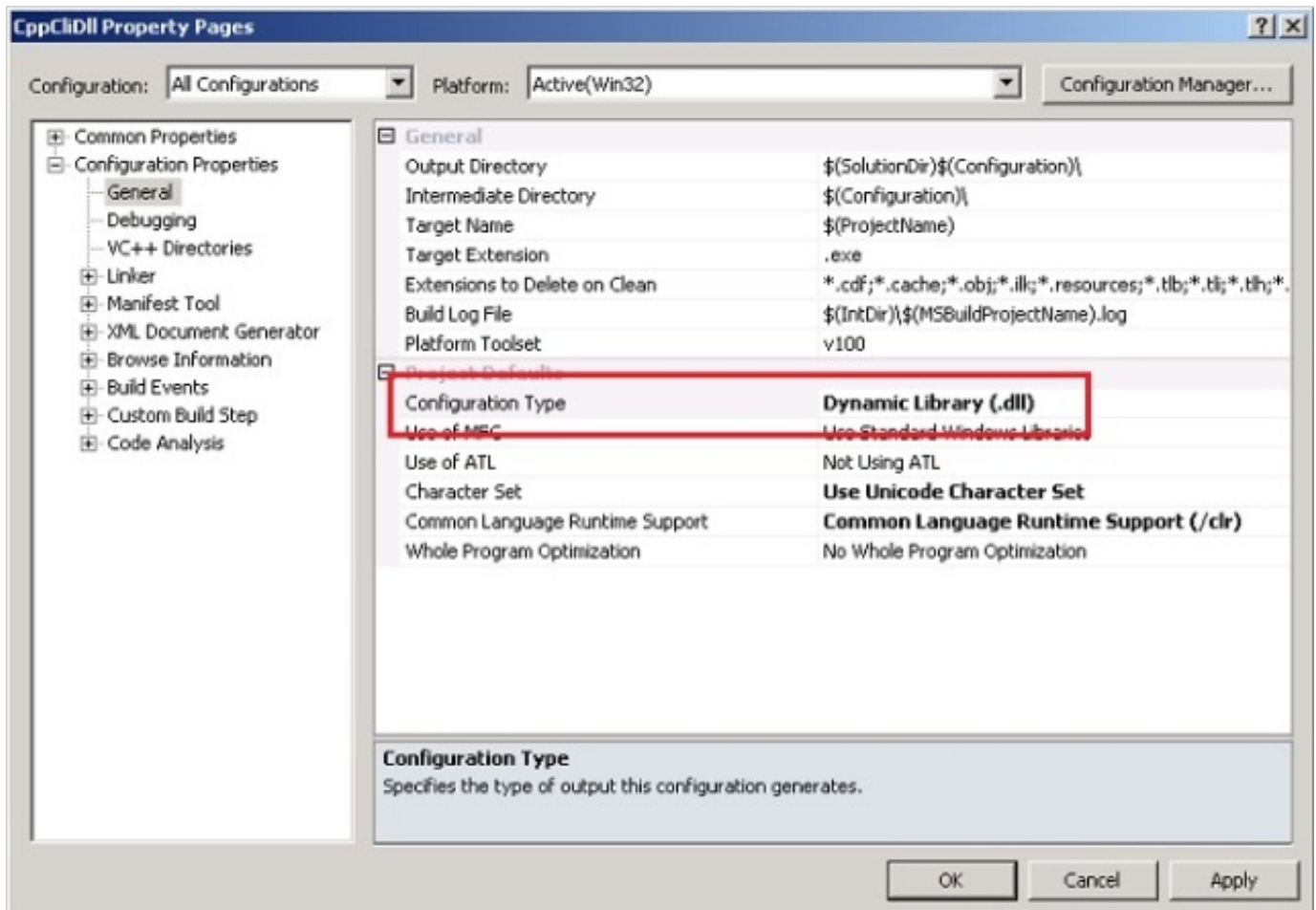
1. Add new project to our solution:



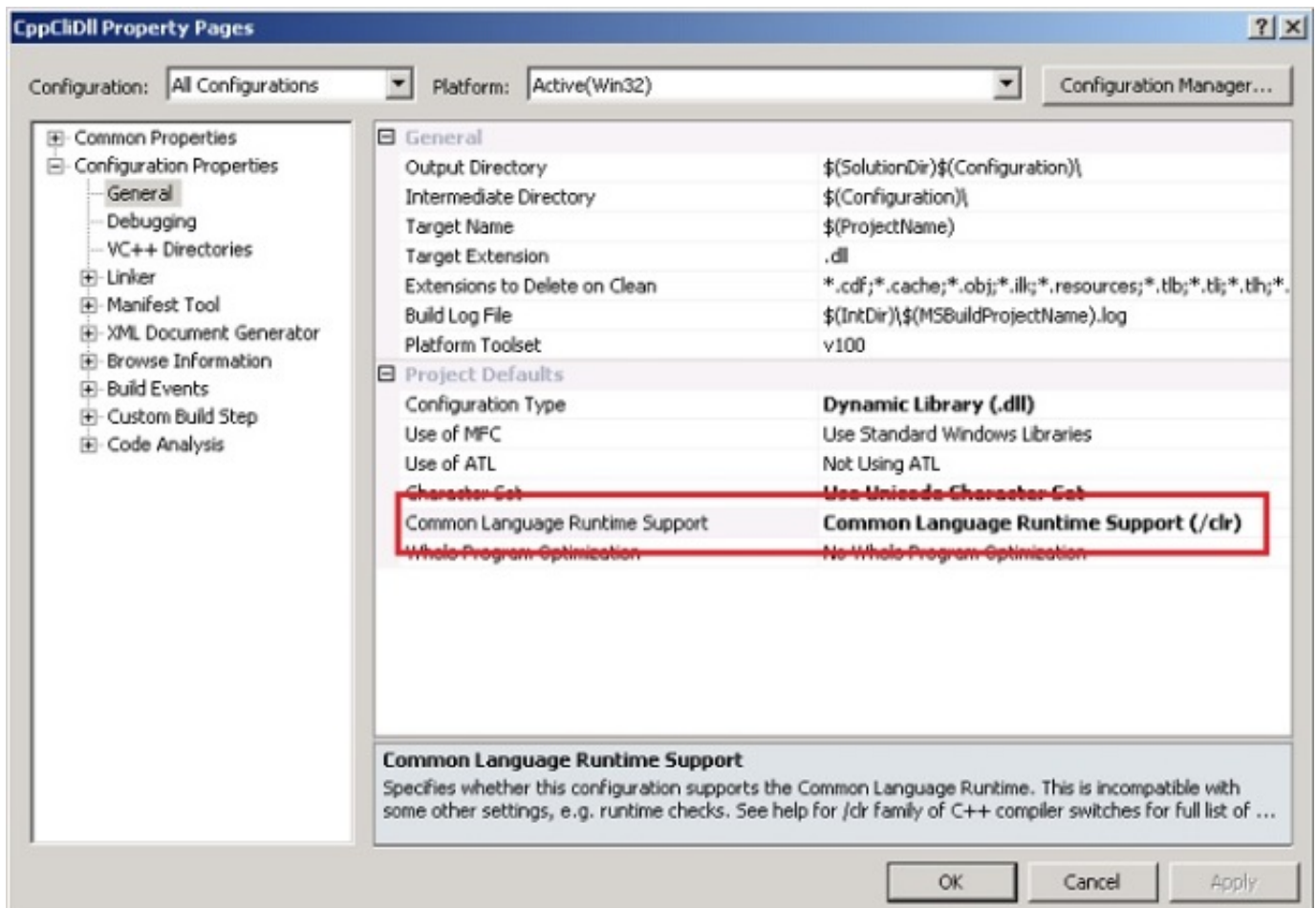
2. In the opened dialog, choose C++ "CLR Empty Project", Enter a name and, click "OK":



After we've created our project, we set its type to a Dynamic Library:



When looking in the project's properties, we can notice the `/clr` definition:



That definition is needed, for enabling using C++/CLI managed code, in our project.

For using our native **Worker**, we can link our native static library with our project. That can be done in the same manner as we did for our native DLL ([Link our native static library](#)).

Wrap the native Worker class with an appropriate managed class

So, we've created our C++/CLI project. The next step is adding a managed class into it. As mentioned before, the C++/CLR language, enables writing managed and unmanaged (native) C++ code. For declaring a managed class we should use the **ref** keyword:

```
ref class CppCliWorker
{
}
```

For exporting our managed class for other projects (that have our DLL as a reference), we should mark our class as **public**:

```
public ref class CppCliWorker
{
}
```

Now, after we added our managed class we can wrap our native **Worker** with it. The first step is holding a pointer for our native **Worker** class. For that purpose we:

1. Add a data-member for holding a pointer to the native **Worker**:

```
Worker* m_actualWorker;
```

2. Implement the constructor to create a new native **Worker**:

```
CppCliWorker::CppCliWorker()
{
    m_actualWorker = new Worker;
}
```

3. Implement the **Dispose** method to delete the native **Worker**:

```
CppCliWorker::~CppCliWorker()
{
    if (nullptr != m_actualWorker)
    {
        delete m_actualWorker;
        m_actualWorker = nullptr;
    }
}
```

4. Implement the **Finalize** method to delete the native **Worker** (For cases that the **Dispose** method hasn't been called):

```
CppCliWorker::~!CppCliWorker()
{
    if (nullptr != m_actualWorker)
    {
        delete m_actualWorker;
        m_actualWorker = nullptr;
    }
}
```

In the [previous approach](#), where we created a DLL that exposes global functions, we maintained a **map** for connecting between worker-handles and worker instances. In this approach, since we have a managed class for each native worker, we just hold the native worker instance as a data-member of our managed class.

So, we have a managed class that wraps a native **Worker**. The next step is to implement functions for calling the native **Worker**'s functions:

```
void CppCliWorker::Start()
{
    if (nullptr != m_actualWorker)
    {
        m_actualWorker->Start();
    }
}

void CppCliWorker::Stop()
{
    if (nullptr != m_actualWorker)
    {
        m_actualWorker->Stop();
    }
}

void CppCliWorker::QueueDemoWorkerTasks()
{
    if (nullptr != m_actualWorker)
    {
        m_actualWorker->QueueDemoWorkerTasks();
    }
}

void CppCliWorker::QueueWorkerTask(unsigned int outerLoopCount, unsigned int innerLoopCount)
{
    if (nullptr != m_actualWorker)
    {
        m_actualWorker->QueueWorkerTask(outerLoopCount, innerLoopCount);
    }
}

void CppCliWorker::ExecuteWorkerTask(unsigned int outerLoopCount, unsigned int innerLoopCount)
{
    if (nullptr != m_actualWorker)
    {
        m_actualWorker->ExecuteWorkerTask(outerLoopCount, innerLoopCount);
    }
}

void CppCliWorker::PrintStatistics()
{
    if (nullptr != m_actualWorker)
    {
        m_actualWorker->PrintStatistics();
    }
}

double CppCliWorker::GetWorkingSeconds()
{
    if (nullptr != m_actualWorker)
    {
        return m_actualWorker->GetWorkingSeconds();
    }
}
```

That's it, now we have a .NET library with a managed class, that can be used with other .NET languages.

Third approach - Write the full native and managed code in one project

So far, we saw two approaches for exposing native implementation as a managed class:

1. Create a C++ native DLL (that is linked with a C++ native static library) and, call its exports from a C# .NET DLL,

using P/Invoke.

2. Create a C++/CLI .NET DLL (that is linked with a C++ native static library).

In the both of the approaches above, we created a native static library and, linked it to another project. Another approach can be: including all of the code (the native **Worker** implementation and, the managed wrapper class), in one project.

Using that approach, we create a C++/CLI project that includes a managed class (like we did in the [second approach](#)) but, instead of linking to a separated static library, we include the native implementation in the same C++/CLI project.

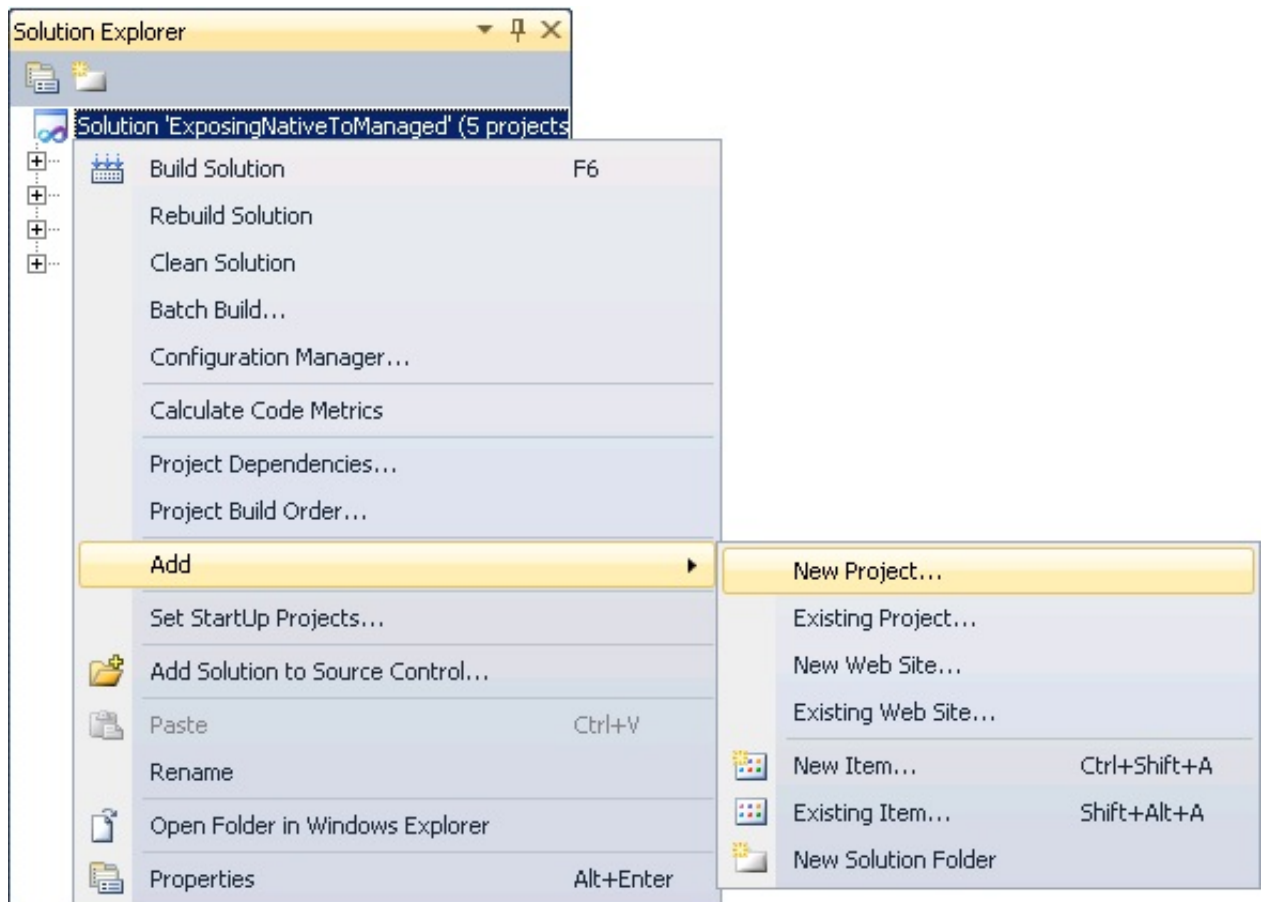
In the next section, we'll run a tester that compares the performance of the different approaches.

Test and compare performance

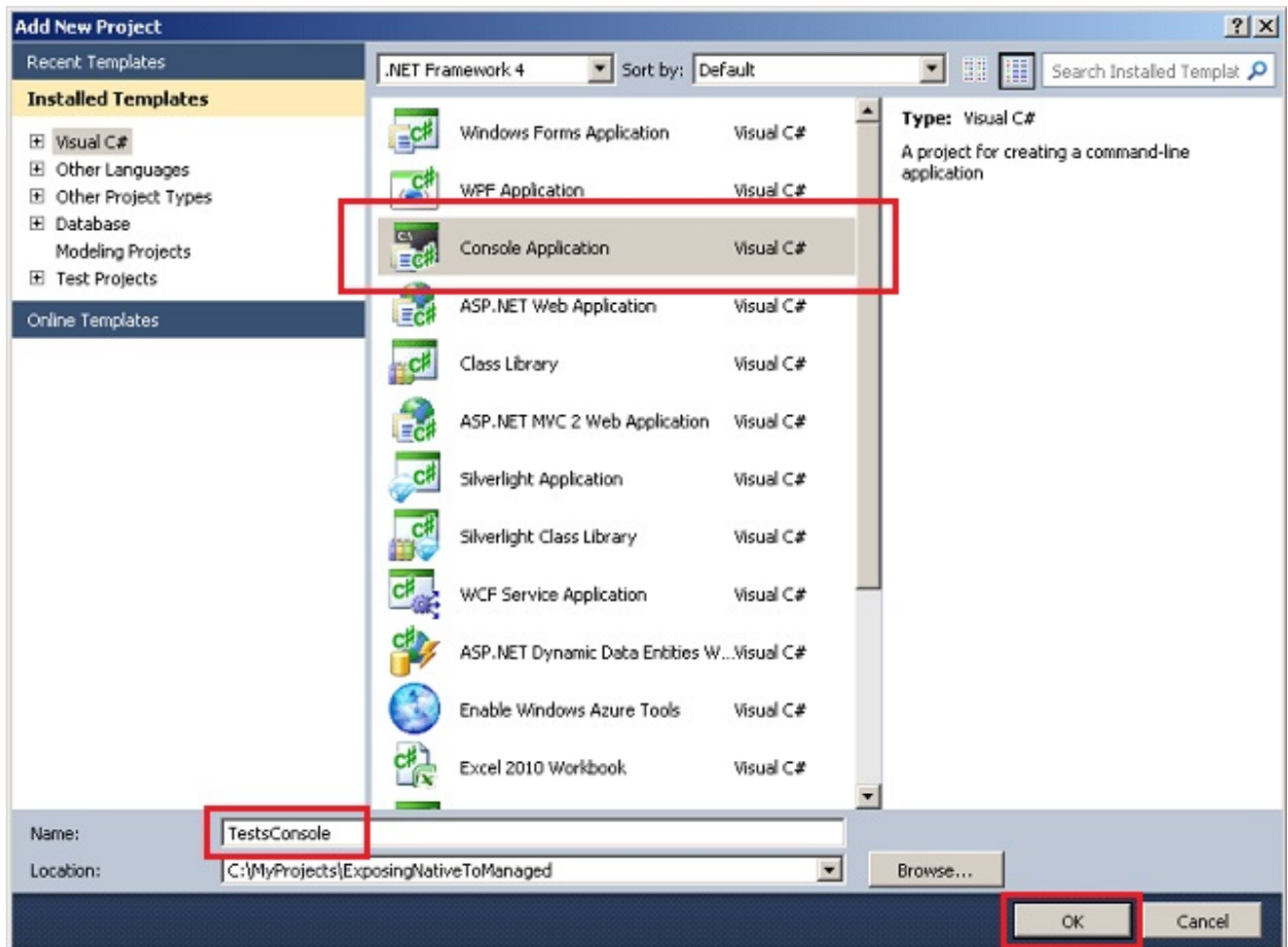
Create C# console project

After we've created the our wrappers (using the three approaches), we can perform some performance tests on them. For demonstrating the .NET interoperability and testing our wrappers, we create a C# console application that uses our wrappers. The first step of creating our tests console is: creating a C# console application project. That can be done by the following steps:

1. Add new project to our solution:



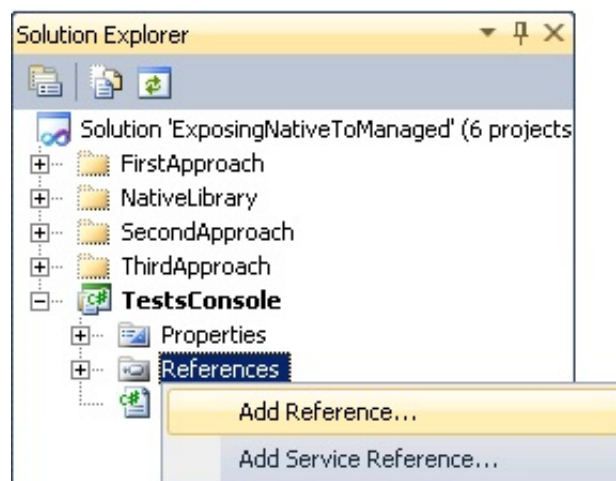
2. In the opened dialog, choose C# "Console Application", enter a name and, click "OK":



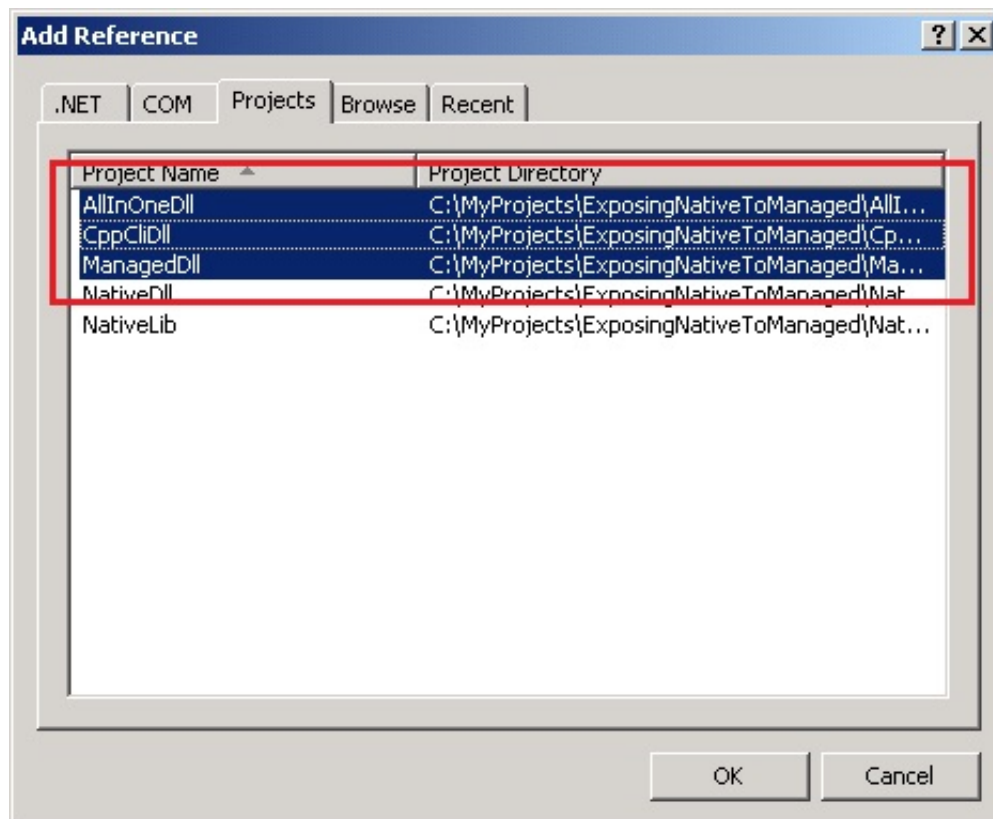
Add wrappers' DLLs

For using our wrappers, we have to add reference to the relevant DLLs. That can be done by the following steps:

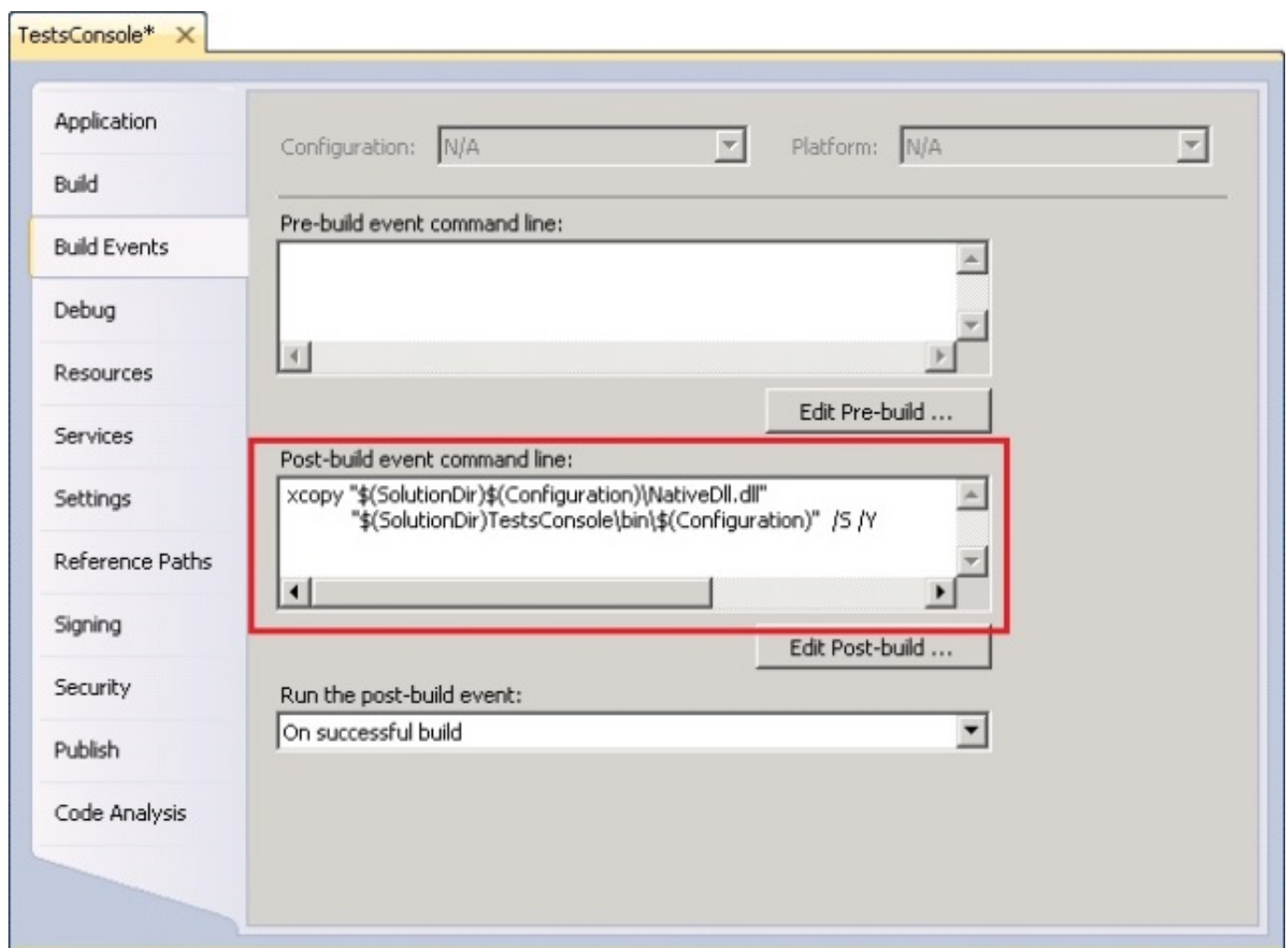
1. On the context-menu of the test project, choose "Add reference":



2. On the opened dialog, select the "Projects" tab, choose the wrappers' DLLs and, press "OK":



Since our managed DLL (from the first approach) uses the exports of the native DLL, for running our tests (or other applications that use the DLL), we need the native DLL to be available too. We can achieve that goal, by copying the native DLL, to the destination folder of our tests console application. That can be done by adding a post-build event that performs that task:



Implement the tests

In the previous sections, we created a native library, that contains a **Worker** class, that provides three ways for executing tasks. After creating the native library, we wrapped it with a managed class, using three different approaches. In this section, we'll implement C# code, that executes worker tasks using the three ways of the three wrappers and, prints the time that takes for each execution.

For our tests, we:

1. Add lists for holding the execution time of the tests:

```
private static List<double> _managedWorkerWorkingSeconds;
private static List<double> _cppCliWorkerWorkingSeconds;
private static List<double> _allInOneWorkerWorkingSeconds;
```

2. Implement methods for each test:

- First approach (Call native DLL exports using P/Invoke):
 - First way (Queue asynchronous tasks: Queue tasks, one after one):

```
static void MultipleCallsTest_TestManagedWorker()
{
    Console.WriteLine("Multiple calls - Managed worker (P/Invoke) test:");

    using (ManagedWorker worker = new ManagedWorker())
    {
        Console.WriteLine("Starting...");

        worker.Start();

        foreach (uint taskLoopCount in _tasksLoopCounts)
        {
            worker.QueueWorkerTask(taskLoopCount, taskLoopCount);
        }

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();

        _managedWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}
```

- Second way (Queue asynchronous tasks: Queue some tasks once):

```
static void OneCallTest_TestManagedWorker()
{
    Console.WriteLine("One call - Managed worker (P/Invoke) test:");

    using (ManagedWorker worker = new ManagedWorker())
    {
        Console.WriteLine("Starting...");

        worker.Start();

        worker.QueueDemoWorkerTasks();

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();
    }
}
```

```

        }
        _managedWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}

```

- Third way (Execute synchronous tasks, using TPL):

```

static void MultipleSyncCallsTest_TestManagedWorker()
{
    Console.WriteLine("Synchronous calls - Managed worker (P/Invoke) test:");

    using (ManagedWorker worker = new ManagedWorker())
    {
        Console.WriteLine("Starting...");

        worker.Start();

        Parallel.ForEach(_tasksLoopCounts, t => { worker.ExecuteWorkerTask(t, t); });

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();

        _managedWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}

```

- Second approach (Wrap the native library with a managed C++/CLI project):

- First way (Queue asynchronous tasks: Queue tasks, one after one):

```

static void MultipleCallsTest_TestCppCliWorker()
{
    Console.WriteLine("Multiple calls - C++/CLI worker test:");

    using (CppCliWorker worker = new CppCliWorker())
    {
        Console.WriteLine("Starting...");

        worker.Start();

        foreach (uint taskLoopCount in _tasksLoopCounts)
        {
            worker.QueueWorkerTask(taskLoopCount, taskLoopCount);
        }

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();

        _cppCliWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}

```

- Second way (Queue asynchronous tasks: Queue some tasks once):

```

static void OneCallTest_TestCppCliWorker()
{
    Console.WriteLine("One call - C++/CLI worker test:");

    using (CppCliWorker worker = new CppCliWorker())
    {
        Console.WriteLine("Starting...");
    }
}

```

```

        worker.Start();

        worker.QueueDemoWorkerTasks();

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();

        _cppCliWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}

```

- Third way (Execute synchronous tasks, using TPL):

```

static void MultipleSyncCallsTest_TestCppCliWorker()
{
    Console.WriteLine("Synchronous calls - C++/CLI worker test:");

    using (CppCliWorker worker = new CppCliWorker())
    {
        Console.WriteLine("Starting...");

        worker.Start();

        Parallel.ForEach(_tasksLoopCounts, t => { worker.ExecuteWorkerTask(t,
t); });

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();

        _cppCliWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}

```

- Third approach (Write the full native and managed code in one project):

- First way (Queue asynchronous tasks: Queue tasks, one after one):

```

static void MultipleCallsTest_TestAllInOneWorker()
{
    Console.WriteLine("Multiple calls - All-in-one (C++/CLI) worker test:");

    using (AllInOneWorker worker = new AllInOneWorker())
    {
        Console.WriteLine("Starting...");

        worker.Start();

        foreach (uint taskLoopCount in _tasksLoopCounts)
        {
            worker.QueueWorkerTask(taskLoopCount, taskLoopCount);
        }

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();

        _allInOneWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}

```


- Second way (Queue asynchronous tasks: Queue some tasks once):

```
static void OneCallTest_TestAllInOneWorker()
{
    Console.WriteLine("One call - All-in-one (C++/CLI) worker test:");

    using (AllInOneWorker worker = new AllInOneWorker())
    {
        Console.WriteLine("Starting...");

        worker.Start();

        worker.QueueDemoWorkerTasks();

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();

        _allInOneWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}
```

- Third way (Execute synchronous tasks, using TPL):

```
static void MultipleSyncCallsTest_TestAllInOneWorker()
{
    Console.WriteLine("Synchronous calls - All-in-one (C++/CLI) worker test:");

    using (AllInOneWorker worker = new AllInOneWorker())
    {
        Console.WriteLine("Starting...");

        worker.Start();

        Parallel.ForEach(_tasksLoopCounts, t => { worker.ExecuteWorkerTask(t, t); });

        worker.Stop();

        Console.WriteLine(" - Finished.");

        worker.PrintStatistics();

        _allInOneWorkerWorkingSeconds.Add(worker.GetWorkingSeconds());
    }
}
```

3. Run the tests and, write the results:

```
static void Main(string[] args)
{
    Console.Title = "Exposing native to managed - performance tests";

    MultipleCallsTest();
    OneCallTest();
    MultipleSyncCallsTest();

    WriteWorkersWorkingSeconds();

    Console.WriteLine("Press <Enter> to quit.");
    Console.ReadLine();
}
```

```
private static void MultipleCallsTest()
{
    MultipleCallsTest_TestAllInOneWorker();
    Console.WriteLine();
    MultipleCallsTest_TestCppCliWorker();
    Console.WriteLine();
    MultipleCallsTest_TestManagedWorker();
    Console.WriteLine();

    MultipleCallsTest_TestManagedWorker();
    Console.WriteLine();
    MultipleCallsTest_TestCppCliWorker();
    Console.WriteLine();
    MultipleCallsTest_TestAllInOneWorker();
    Console.WriteLine();
}

private static void OneCallTest()
{
    OneCallTest_TestAllInOneWorker();
    Console.WriteLine();
    OneCallTest_TestCppCliWorker();
    Console.WriteLine();
    OneCallTest_TestManagedWorker();
    Console.WriteLine();

    OneCallTest_TestManagedWorker();
    Console.WriteLine();
    OneCallTest_TestCppCliWorker();
    Console.WriteLine();
    OneCallTest_TestAllInOneWorker();
    Console.WriteLine();
}

private static void MultipleSyncCallsTest()
{
    MultipleSyncCallsTest_TestAllInOneWorker();
    Console.WriteLine();
    MultipleSyncCallsTest_TestCppCliWorker();
    Console.WriteLine();
    MultipleSyncCallsTest_TestManagedWorker();
    Console.WriteLine();

    MultipleSyncCallsTest_TestManagedWorker();
    Console.WriteLine();
    MultipleSyncCallsTest_TestCppCliWorker();
    Console.WriteLine();
    MultipleSyncCallsTest_TestAllInOneWorker();
    Console.WriteLine();
}

private static void WriteWorkersWorkingSeconds()
{
    Console.WriteLine("Managed worker (P/Invoke) - working seconds:");
    foreach (double managedWorkerSeconds in _managedWorkerWorkingSeconds)
    {
        Console.WriteLine("\t{0}", managedWorkerSeconds);
    }
    Console.WriteLine();

    Console.WriteLine("C++/CLI worker - working seconds:");
    foreach (double cppCliWorkerSeconds in _cppCliWorkerWorkingSeconds)
    {
        Console.WriteLine("\t{0}", cppCliWorkerSeconds);
    }
    Console.WriteLine();

    Console.WriteLine("All-in-one (C++/CLI) worker - working seconds:");
    foreach (double allInOneWorkerSeconds in _allInOneWorkerWorkingSeconds)
    {
```

```

        Console.WriteLine("\t{0}", allInOneWorkerSeconds);
    }
    Console.WriteLine();
}

```

For testing our wrappers (using the three approaches), we run the same algorithms (one algorithm for each way) twice (the second time, we call the testing methods in the opposite order of the first time), for each wrapper. The only differences between the testing methods (of the different wrappers) are: the wrapper object that is created and, the list that gets the execution time result.

The Tests' results

After running our tests console, we get the following result:

```

Exposing native to managed - performance tests

Managed worker <P/Invoke> - working seconds:
13.7
13.925
13.674
13.532
18.681
17.583

C++/CLI worker - working seconds:
13.483
13.415
13.816
13.553
17.832
18.063

All-in-one <C++/CLI> worker - working seconds:
13.254
13.31
13.463
13.368
17.506
18.849

```

Since the test results may be affected by environment issues, we run the same tests 10 times. The following table shows a summary of the performed tests' execution time (in seconds) results:

First approach			Second approach			Third approach		
First way	Second way	Third way	First way	Second way	Third way	First way	Second way	Third way
13.7	13.674	18.681	13.483	13.816	17.832	13.254	13.463	17.506
13.925	13.532	17.583	13.415	13.553	18.063	13.31	13.368	18.849
13.449	13.388	18.083	13.402	13.703	18.301	13.35	13.832	17.179
13.552	13.487	21.37	13.638	13.167	21.123	13.559	13.312	20.905
13.706	13.45	22.671	13.406	15.008	17.816	13.47	13.455	17.664
13.651	13.371	18.353	13.418	13.155	23.2	13.332	13.427	19.17
13.869	13.317	17.847	13.321	13.799	18.197	13.331	13.369	21.46
13.318	13.45	21.5	13.425	13.168	17.344	13.367	13.418	17.35
13.401	13.374	21.582	13.488	13.246	18.604	13.466	13.363	21.983
13.185	13.481	18.408	13.215	13.354	21.571	13.4	13.167	21.326

In the table above, we can see the result working seconds of our native worker, using three (different) managed wrappers, using three ways for tasks' execution. We can see that the performances of the three approaches are nearly the same. In the three approaches, using TPL (the third way) is less efficient than using our native threads implementation (the first and second ways).

Conclusion

So, we saw three approaches of how we can wrap native content with a managed .NET DLL. All of the approaches have nearly the same performance.

In the [first approach](#), we created a native DLL and, a managed DLL that uses its exports. This approach can be a good choice if we want to use our native content also in native projects. In that manner, we have our native content, in the native DLL and, we use the same native DLL in native and managed projects.

In the [second approach](#), we wrapped a native static library with a managed DLL. This approach can be a good choice if we don't have to use our native content in native projects and, we want a more comfortable way for consuming our content. In that manner, we have our native content, in a managed DLL, that can be added as a reference for .NET projects.

In the [third approach](#), we wrote all of the code (native and managed), in one project. The result of this approach, is as same as the result of the second approach (managed DLL that contains the native content). But, in the second approach, we have our native content, separated in a different native static library. So that, in cases that we want to use our native content in native projects too, we can simply apply the first approach (create a native DLL) by wrapping the separated static library. That reusability is missing in the third approach. Therefore, in my opinion, the third approach isn't recommended.

For my driver, since we didn't need it for native projects, I chose the second approach. For other projects, where we have a native application and, we want to run it also as a managed Windows service, I choose the first approach for separating the core native implementation. Sometimes it's difficult to decide which approach to choose - it depends on the needs of the specific project.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

Share

About the Author



Shmuel Zang

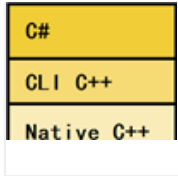


Software Developer

Israel 

No Biography provided

You may also be interested in...



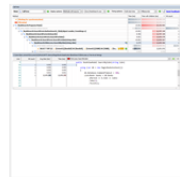
Native under Managed



Speed Up Your Git Repository: Introducing Git-Over-FASP



Using lambdas - C++ vs. C# vs. C++/CX vs. C++/CLI



Is SQL Server killing your application's performance?




Using generics in C++/CLI



SAPrefs - Netscape-like Preferences Dialog

Comments and Discussions

 **23 messages** have been posted for this article Visit <http://www.codeproject.com/Articles/651516/Exposing-native-to-managed-Cplusplus-CLI-vs-P-Invo> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Terms of Use](#) | Mobile
Web03 | 2.8.151108.1 | Last Updated 10 Sep 2013

Select Language ▼

Article Copyright 2013 by Shmuel Zang
Everything else Copyright © [CodeProject](#), 1999-2015