Mikro-introdukcja:

*Celem tego dokumentu jest przedstawienie kilku możliwych metod tworzenia i wykorzystywania DLL'ek które są tworzone w paradygmacie „run-time dynamic linking" - czyli gdy nasze biblioteki dll są ładowne do programu w czasie działania tego programu, a nie na jego stracie („load-time dynamic linking").

*Zawartość została podzielona na trzy małe rozdziały - **„Podstawy"** - jest krótkim wprowadzeniem do koncepcji i technik które będą wykorzystywane w dalszych rozdziałach. **„DLL + Header"** dotyczy DLL'ek w których mamy wgląd do pliku nagłówkowego na podstawie którego zostały stworzone ów DLL'e i możemy go swobodnie załączyć. **„Standalone DLL"** opisuje sytuacje gdy mamy tylko samotny plik DLL.

*Guide powstał w oparciu o wykorzystanie Visual Studio 2015

*Jest to wersja alfa która zapewne ani się nie ustrzegła od drobnych rozminięć merytorycznych jak i błędów językowych.

Podstawy - od strony tworzenia dll

Tworzenie projektu dll'owego

Tworzenie dll'a wygląda pod wieloma względami identycznie jak tworzenie pomocniczego pliku (.cpp + .h) do swojego projektu zawierającego deklaracje i definicje funkcji, klas oraz metod. Główną różnicą jest fakt, że w wyniku kompilacji dll'owej uzyskuje się szereg plików pomocniczych (typu .lib) a zamiast pliku .cpp uzyskujemy .dll. W visualu aby stworzyć dll'ke wystarczy wybrać podczas określania natury projektu w finalnym oknie tworzenia projektu w opcji „aplication type": „DLL".

Eksport

Plik .dll jest w dużym uproszczeniu zakodowanym plikiem .cpp z którego zostały wybrane wszystkie oflagowane do eksportu klasy, funkcje oraz metody. Jest kilka metod na oflagowanie wybranych definicji, jedna z nich - metoda tworzenia pliku definicji (.def) która jest bardzo dobrze opisana w wszelkich źródłach (zarówno internetowych jak i papierowych) nie zostanie tutaj przedstawiona - osobiście jestem jej niechętny ponieważ dodaje kolejny plik do zarządzania który w większości wypadków jest po prostu redundantną informacją. Dalej zostaną opisane metody oflagowania definicji, w tym miejscu jedynie zaznaczę, że dzięki temu oflagowaniu powstaną w pliku .dll specjalne „odkodowane" sekcje dzięki którym program „klienta" będzie mógł z pewną pomocą twórcy DLL'a korzystać z zawartości udostępnionej biblioteki.

__declspec(dllexport)

Pierwszą metodą oflagowania funkcji/klas do eksportu jest wykorzystanie słowa kluczowego „`__declspec`" wraz z deklaracją „`(dllexport)`" , przy tworzeniu projektu DLL możemy zaznaczyć by IDE stworzyło nam domyślne deklaracje eksportu, które wyglądają następująco:

```
#ifdef NAZWA_PROJEKTU_DLL_EXPORTS
#define NAZWA_PROJEKTU_DLL_API __declspec(dllexport)
#else
#define NAZWA_PROJEKTU_DLL_API __declspec(dllimport)
#endif
```

Co oznacza w skrócie, że dla projektu DLL zdefiniowana jest deklaracja `NAZWA_PROJEKTU_DLL_API` jako `__declspec(dllexport)` a co za tym idzie wszystkie definicje/deklaracje z dopiskiem `NAZWA_PROJEKTU_DLL_API` zostaną wyeksportowane do pliku .dll.

Jednocześnie ta sama kombinacja ifdef'ów spowoduje, że w projekcie który chce zaimportować wybrane funkcje/klasy z dopiskiem `NAZWA_PROJEKTU_DLL_API` zostaną potraktowane jako `__declspec(dllimport)` i zostaną zaimportowane (co jest dla nas tak naprawdę nieistotne bo dotyczy zabaw z plikiem .lib oraz ładowania dll'i w konwencji load-time dynamic linking).

Przykład eksportu klasy oraz funckji:

W pliku nagłówkowym (.h) naszego dll'a gdy projekt DLL nazywa się „biblio":

```
#ifdef BIBLIO_EXPORTS
#define BIBLIO_API __declspec(dllexport)
#else
#define BIBLIO_API __declspec(dllimport)
#endif

// deklaracja klasy do eksportu - wraz ze wszystkimi metodami publicznymi
class BIBLIO_API Cbiblio {
public:

	int cow;

	Cbiblio(void);
	int GetCow();
};

BIBLIO_API int funBiblio(void); //deklaracja funkcji do eksportu
```

W pliku źródłowym (.cpp) naszego dll'a gdy projekt DLL nazywa się „biblio":

```
#include "stdafx.h"
#include "biblio.h"
#include <iostream>

using namespace std;


int funBiblio()
{
	return 42;
}

Cbiblio::Cbiblio()
{
	cow = 13;
}

int Cbiblio::GetCow()
{
	return cow;
}
```

Jak widać plik źródłowy niczym nie różni się od „klasycznego" pliku źródłowego nie będącego biblioteką DLL.

#pragma comment(linker, "/EXPORT …)

Drugą metodą jest wykorzystanie specjalnej dyrektywy preprocesora wdrożonej przez Microsoft służącej do wysłania bezpośredniego żądania do linkera o oflagowanie danych funkcji/metod i wyeksportowanie do dll'a. Takie rozwiązanie wiąże się ze sporymi zaletami. Po pierwsze jest o wiele czytelniejsze i wymagające mniej uwagi od twórcy biblioteki, po drugie mamy o wiele większą kontrolę pod jakimi nazwami zostaną wyeksportowane funkcje/metody do .dll'a (co będzie miało w dalszej części duże znaczenie).

Składnia dyrektywy wersja A:

```
#pragma comment(linker, "/EXPORT:" __FUNCTION__"=" __FUNCDNAME__)
```

Jedyne co warto w tym miejscu wytłumaczyć to dwa słowa kluczowe: „__FUNCTION__" pobiera aktualną nazwę naszej funkcji lub metody wraz z jej pochodzeniem klasowym i w umieszcza ją w swoje miejsce. „__FUNCDNAME__" pobiera nazwę którą by opisał daną funckje/metodę kompilator i umieszcza ją w swoje miejsce. Następnie całość jest eksportowana do pliku .dll gdzie będzie miała przykładową postać:

```
ordinal hint RVA      name

      1    0 000112B7 Cbiblio::Cbiblio = @ILT+690(??0Cbiblio@@QAE@XZ)
      2    1 0001124E Cbiblio::GetCow = @ILT+585(?GetCow@Cbiblio@@QAEHXZ)
      3    2 0001123F funBiblio = @ILT+570(?funBiblio@@YAHXZ)
```

Składnia dyrektywy wersja B:

```
#pragma comment(linker, "/EXPORT:" "WybranaPrzezNasNazwa=" __ FUNCDNAME__)
```

Jedyną różnicą w tym wariancie jest fakt ustawiania własnej nazwy dla wybranych funkcji/metod. Przykładowe wyeksportowane funkcje/metody do .dll'a będą wyglądały następująco:

```
ordinal hint RVA      name

      1    0 000112B7 CbiblioConstructor = @ILT+690(??0Cbiblio@@QAE@XZ)
      2    1 0001124E GetCow = @ILT+585(?GetCow@Cbiblio@@QAEHXZ)
      3    2 0001123F funBiblio = @ILT+570(?funBiblio@@YAHXZ)
```

Uwaga: Visual może nam podkreślić część polecenia „#pragma…" jakby był to błędny zapis, jest to swego rodzaju nadwrażliwość ze strony syntax check'era, należy ten warning po prostu zlekceważyć.

Przykład eksportu klasy oraz funckji:

W pliku nagłówkowym (.h) naszego dll'a gdy projekt DLL nazywa się „biblio":

```
// deklaracja klasy do eksportu - wraz ze wszystkimi metodami publicznymi
class Cbiblio {
public:

	int cow;

	Cbiblio(void);
	int GetCow();
};

int funBiblio(void); //deklaracja funkcji do eksportu
```

Jak widać tym razem plik nagłówkowy niczym nie różni się od „klasycznego" pliku nagłówkowego nie będącego biblioteką DLL.

W pliku źródłowym (.cpp) naszego dll'a gdy projekt DLL nazywa się „biblio":

```
#include "stdafx.h"
#include "biblio.h"
#include <iostream>

using namespace std;


int funBiblio()
{
#pragma comment(linker, "/EXPORT:" __FUNCTION__"=" __FUNCDNAME__)

	return 42;
}

Cbiblio::Cbiblio()
{
#pragma comment(linker, "/EXPORT:" __FUNCTION__"=" __FUNCDNAME__)

	cow = 13;
}

int Cbiblio::GetCow()
{
#pragma comment(linker, "/EXPORT:" __FUNCTION__"=" __FUNCDNAME__)

	return cow;
}
```

dllmain.cpp

Zazwyczaj podczas tworzenia nowego projektu DLL, nasze IDE automatycznie utworzy nam plik o nazwie „dllmain.cpp”. W pliku tym będzie tak zwany punkt wejścia do biblioteki. Plik ten nie jest niezbędny do działania/tworzenia biblioteki. Umożliwia on zaprogramowanie pewnych specjalnych wydarzeń podczas gdy np. biblioteka jest ładowana do programu lub odłączana od niego.

Podstawy - od strony wykorzystania dll

Wykorzystanie w swoim projekcie dll'a

Do korzystania z biblioteki DLL należy podchodzić tak jakby się korzystało ze stworzonego przez nas (lub kogoś innego) zestawu plików pomocniczych do naszej aplikacji, który zawiera definicje klas, metod lub funkcji, które będą wykorzystywane w naszym programie tak jakby były standardowo określone w plikach (.h + .cpp). Odpowiedź na pytanie dlaczego w takim wypadku warto w ogóle korzystać (i kiedy) z bibliotek DLL znajduje się na dziesiątkach stron internetowych i jest bardzo klarownie opisane, dlatego też nie będę tutaj generował dodatkowej i niepotrzebnej zawartości.

Dumpbin.exe

W wielu miejscach bardzo przydatną aplikacją dostarczoną przez Visuala okaże się dumpbin.exe znajdujący się w katalogu Visuala, przykładowa ścieżka w której można odnaleźć dumpbin'a:

„C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin"

Aby z niego skorzystać w zakresie który nas interesuje należy w wierszu poleceń przejść do katalogu w którym się znajduje, a następnie wpisać poniższe polecenie:

dumpbin.exe /exports "sciezka_do_naszego_dlla\nasz_DLL.dll"

Tip: dla wygody można też stworzyć skrót do dumpbin.exe w dowolnym miejscu i korzystać z niego w konsoli, jedyną różnicą będzie wtedy wywołanie dumpbina za pomocą komendy: „dumpbin.lnk" zamiast „dumpbin.exe"

Dumpbin umożliwia nam zajrzenie do danego pliku .dll i podejrzenie nazw funkcji/metod oraz ich aliasów dzięki czemu będziemy w przyszłości mogli importować je po nazwie.

Przykładowy wykaz nazw zrobiony przez oflagowanie typu „__declspec(dllexport)":

```
ordinal hint RVA      name

      1    0 000112C1 ??0Cbiblio@@QAE@XZ = @ILT+700(??0Cbiblio@@QAE@XZ)
      2    1 00011172 ??4Cbiblio@@QAEAAV0@$$QAV0@@Z = @ILT+365(??4Cbiblio@@QAEAAV0@$$QAV0@@Z)
      3    2 00011163 ??4Cbiblio@@QAEAAV0@ABV0@@Z = @ILT+350(??4Cbiblio@@QAEAAV0@ABV0@@Z)
      4    3 00011258 ?GetCow@Cbiblio@@QAEHXZ = @ILT+595(?GetCow@Cbiblio@@QAEHXZ)
      5    4 00011249 ?funBiblio@@YAHXZ = @ILT+580(?funBiblio@@YAHXZ)
```

W powyższym przykładzie pierwsza metoda to konstruktor klasy Cbiblio, druga to operator przypisania, a na przykład ostatnia, to funkcja niezależna od klasy Cbiblio.

Jest to też bardzo dobry przykład dlaczego „#pragma comment” wydaje się o wiele wygodniejszy, niż standardowy eksport poprzez deklaracje „__declspec(dllexport)”: po prostu unika się zagmatwanych nazw z którymi nie trzeba potem pracować.

Ładowanie Biblioteki DLL

Aby załadować bibliotekę .dll do naszej aplikacji należy po pierwsze załączyć statyczną bibliotekę <windows.h>, a następnie wywołać funkcję ładującą:

```
HINSTANCE LoadMe = LoadLibrary(L"Nazwa_Naszego_DLL'a.dll");

if (LoadMe != NULL)
    cout << "dll zostal zaladowany!\n";
else
    cout << "nie udalo sie zaladowac dll'a!\n";
```

Pointery na metody i funkcje (typedef vs using)

Najprostszą drogą do wykorzystywani metod oraz funkcji zawartych w załadowanym dll’u jest odniesienie się do nich poprzez ich wskaźniki. Generalnie rzecz biorąc wskaźniki na funkcje i metody są bardzo brzydkimi tworami pod kątem zarówno składniowym jak i objętościowym, dlatego bardzo wysoce wskazanym jest jednorazowe zdefiniowanie ich i przypisanie do wybranych słów kluczowych aby w dalszych częściach programu nie trzeba było ponownie wypisywać ciągów gwiazdek, typów i nazw. Zdefiniować wskaźniki można na dwa sposoby albo przy pomocy słowa kluczowego „typedef” albo (od standardu C++11) z wykorzystaniem słowa kluczowego „using”. Pod tym kątem obie metody są sobie równorzędne, aczkolwiek „using”, w mojej ocenie, wydaje się czytelniejszy w zapisie.

Uwaga (na niedaleką przyszłość): Metody statyczne z punktu widzenia kompilatora (oraz importu) są funkcjami a nie metodami.

TYPEDEF:

Składnia wskaźnika na funckję dla „typedef”:

```
typedef typ_zwracany (*nazwa_wybrana_przez_nas_wskaznika)(typ_argumentu);
```

Oczywiście „typ_argumentu” nie musi być jednym argumentem funkcji, po przecinku możemy wypisywać kolejne argumenty funkcji do której robimy wskaźnik.

Przykład:

Chcemy zrzutować funkcję o postaci:

```
void FunkcjaDodajaca(int a, long long b);
```

Definicja wskaźnika na tę funkcję może wyglądać następująco:

```
typedef void(*LibFunkcjaDodajaca)(int, long long);
```

Składnia wskaźnika na metodę danej klasy dla „typedef”:

```
typedef typ_zwracany (nazwa_klasy::* nazwa_wskaznika)(typ_argumentu);
```

Przykład:

Chcemy zrzutować metodę „MetodaDodajaca” klasy „Kalkulator” o postaci:

```
double MetodaDodajaca(int a, long long b);
```

Definicja wskaźnika na tę metodę może wyglądać następująco:

```
typedef double (Kalkulator:: *LibMetodaDodajaca)(int, long long);
```

USING:

Składnia wskaźnika na funckję dla „using”:

```
using nazwa_wybrana_przez_nas_wskaznika = typ_zwracany (*)(typ_argumentu);
```

Oczywiście „typ_argumentu” nie musi być jednym argumentem funkcji, po przecinku możemy wypisywać kolejne argumenty funkcji do której robimy wskaźnik.

Przykład:

Chcemy zrzutować funkcję o postaci:

```
void FunkcjaDodajaca(int a, long long b);
```

Definicja wskaźnika na tę funkcję może wyglądać następująco:

```
using LibFunkcjaDodajaca = void(*)(int, long long);
```

Składnia wskaźnika na metodę danej klasy dla „using”:

```
using nazwa_wskaznika = typ_zwracany (nazwa_klasy:: *)(typ_argumentu);
```

Przykład:

Chcemy zrzutować metodę „MetodaDodajaca” klasy „Kalkulator” o postaci:

```
double MetodaDodajaca(int a, long long b);
```

Definicja wskaźnika na tę metodę może wyglądać następująco:

```
using LibMetodaDodajaca = double (Kalkulator:: *)(int, long long);
```

Importowanie oflagowanych funkcji GetProc'em

Importowanie funkcji z DLL'a odbywa się poprzez funkcję znajdującą się również w bibliotece <windows.h> o nazwie „GetProcAddress" która zwraca wskaźnik o egzotycznie brzmiącym typie FARPROC. FARPROC zwracany jest przez różne funkcje skojarzone z WinAPI (takie jak właśnie GetProcAddress). FARPROC jest to generyczny wskaźnik na niewyspecyfikowane funkcje co oznacza, że jest w uproszczeniu uniwersalnym wskaźnikiem na funkcje (coś jak void*, ale o wiele bezpieczniejszy). Przykładowy kod wykorzystujący GetProc'a do załadowania funkcji z DLL'a może wyglądać następująco:

Przykład:

Załóżmy że chcemy zaimportować funkcję: która została określona w następujący sposób w pliku .dll:

?funBiblio@@YAHXZ = @ILT+580(?funBiblio@@YAHXZ)

A w pliku nagłówkowym (.h) biblioteki wygląda tak:

```
int FunBiblio(void); //deklaracja funkcji do eksportu
```

W przykładzie założono, że plik biblioteki DLL nazywa się „biblio.dll":

```
#include <Windows.h>

using namespace std;

// definiujemy nazwę dla specjalnego typu wskaźnika pokazującego na naszą funkcję
// jeżeli wolimy można tę definicje również podać wewnątrz funkcji main
using LibFunBiblio = int (*)(void);

// wersja w wykorzystująca „typedef" zamiast „using":
// typedef int (*LibFunBiblio)(void);


int main()
{
// ładujemy bibliotekę dll do naszej aplikacji
HINSTANCE LoadMe = LoadLibrary(L"biblio.dll");

// tworzymy wskaźnik o typie pokazującym na naszą funkcję
LibFunBiblio FunBiblio;

// za pomocą funkcji GetProc ustawiamy nasz wskaźnik na funkcję z biblioteki
FunBiblio = (LibFunBiblio)GetProcAddress(LoadMe, "?funBiblio@@YAHXZ");

// następnie wykorzystujemy nasz wskaźnik jak normalną funkcję
cout << FunkcjaDodajaca () << endl;

// zwalniamy bibliotekę DLL związaną z naszą aplikacją
FreeLibrary(LoadMe);
}
```

Kluczowym przy załadowaniu funkcji jest linijka:

```
FunBiblio = (LibFunBiblio)GetProcAddress(LoadMe, "?funBiblio@@YAHXZ ");
```

Którą należy interpretować następująco:

Do wskaźnika `FunBiblio` przypisz zrzutowany wskaźnik (typu `FARPROC`) na wskaźnik typu `LibFunBiblio` uzyskany z funkcji `GetProcAddress` która pobrała jako argumenty uchwyt do instancji biblioteki DLL oraz nazwę funkcji pobieranej z pliku dll.

Uwaga: Jak widać na powyższym przykładzie plik nagłówkowy pochodzący z biblioteki w wypadku gdy importuje się tylko i wyłącznie funkcje jest zbędny (o ile tylko znamy postać deklaracji importowanych funkcji).

Importowanie oflagowanych metod GetProc'em

…….

DLL + Header

Wykorzystanie wirtualizacji

Najprostszą i wymagająca najmniejszego wysiłku metodą zarówno tworzenia jak i używania dll’a jest wykorzystanie pewnego wytrychu opierającego się na eksportowanie wirtualnych

Pełne rzutowanie FARPROCA na metody

……

Standalone DLL

Undname.exe

……

Tworzenie „przyjaznego” DLL’a Standalone

……

Wykorzystanie „przyjaznego” DLL’a Standalone

……

Hackowanie DLL’a

……