Wprowadzenie

Na wstępnie chciałbym zaznaczyć, że dokument traktować należy jako dodatek do wykładu oraz dokumentu „guide_DLL”. Mocno zalecane jest zatem zapoznanie się z nim zanim zacznie się czytanie dalszej części, ponieważ nie zamierzam ponownie tłumaczyć podstawowych kwestii związanych z plikami DLL i ich strukturą.

Niniejsza praca poświęcona będzie sposobowi na eksportowanie symbolu klasy, dzięki któremu dostępne będzie użycie całej jej zawartości w pliku klienckim, czyli pliku, z którego budowany będzie plik exe.

Opis metody

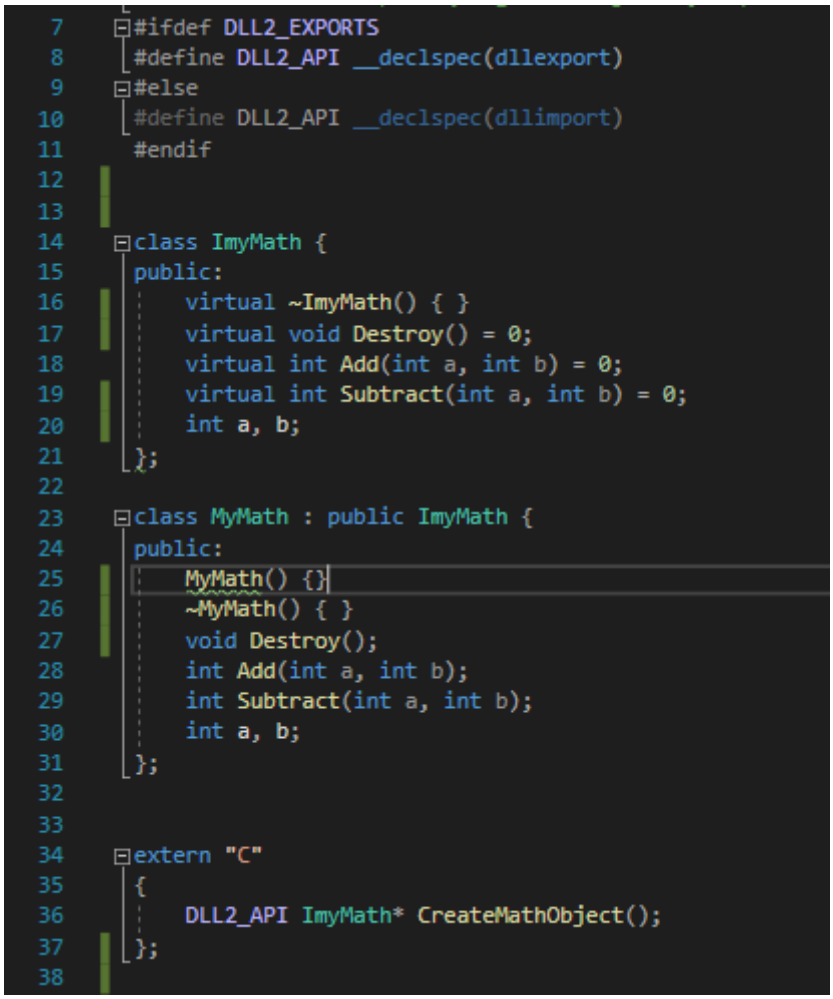
Cały trick polega na tym, że nie eksportujemy bezpośrednio symbolu klasy, a funkcję, która zwraca wskaźnik na nowo utworzoną instancję owej klasy. Dokonujemy tego przy użyciu wzorca projektowego „Factory”.

Dlaczego warto?

Przede wszystkim wyokrzystujemy OOP. Podejście obiektowe ma swoje zalety jak i wady, zależnie od projektu i celów, do których program będzie używany może być to łatwiejsze i lepsze rozwiązanie. Na pewno bardziej intuicyjne i przyjazne osobom wdrażającym się w tematykę DLL.

Tworzenie interfejsu klasy

Na poniższym obrazku przedstawię przykład prostego interfejsu, który przedstawi schemat działania opisywanej metody.



```
#ifdef DLL2_EXPORTS
#define DLL2_API __declspec(dllexport)
#else
#define DLL2_API __declspec(dllimport)
#endif


class ImyMath {
public:
    virtual ~ImyMath() { }
    virtual void Destroy() = 0;
    virtual int Add(int a, int b) = 0;
    virtual int Subtract(int a, int b) = 0;
    int a, b;
};

class MyMath : public ImyMath {
public:
    MyMath() {}
    ~MyMath() { }
    void Destroy();
    int Add(int a, int b);
    int Subtract(int a, int b);
    int a, b;
};


extern "C"
{
    DLL2_API ImyMath* CreateMathObject();
};

```

Destroy()

Na powyższym obrazku rzuca się w oczy metoda Destroy(). Uzasadnienie jej egzystencji jest bardzo proste, chociaż niekoniecznie zrozumiałe na pierwszy rzut oka. Służy ona jako destruktor klasy i zapobiega w oczywisty sposób wyciekowi danych. Może budzić ona wątpliwości, szczególnie w zestawieniu z destruktorami, ale z przyczyn nie do końca mi znanych i zrozumiałych, alokowana pamięć nie zawsze zostaje automatycznie zwolniona. Aby temu zapobiec zrzucamy ten obowiązek na programistę i jesteśmy pewni, że nie powstanie żaden wyciek danych. Pragnę również zwrócić uwagę na fakt, iż

posiadając metodę Destroy() nie potrzebny będzie virtualny destruktor. Aczkolwiek musimy pamiętać o tym aby na 100% wywołać metodę Destroy() jako ostanią oraz być pewnym, że obiekt został zbudowany za pomocą ‘new’. Takie podejście przy konstrukcji ‘new[]’ itp. będzie nieodpowiednie.

Komentarz prowadzącego zajęcia: Powyższe rozwiązanie jest OK. Może jednak lepiej dla spójności interfejsu klasy zastosować funkcję DestroyMathObject – bliźniaczą do funkcji tworzącej. Takiego rozwiązania używa się np. w WinAPI.

Metody i zmienne

W skrócie – w interfejsie deklarujemy to do czego chcemy aby klient miał dostęp. Jest to standardowe podejście, nie wprowadza się tu żadnych innowacji. Obowiązują te same zasady tworzenia klas i jej składowych.

Eksport

```
34  extern "C"
35  {
36      DLL2_API ImyMath* CreateMathObject();
37  };
```

Linijki 34-37 prezentują eksportowaną funkcję. Pakujemy deklarację w extern „C” aby zapewnić wyeksportowanie nazwy funkcji pod podstacią „CreateMathObject” bez dodawania do niej różnych symboli.

Poniżej przedstawiam zrzut ekranu pokazujący co dokładnie eksportujemy:

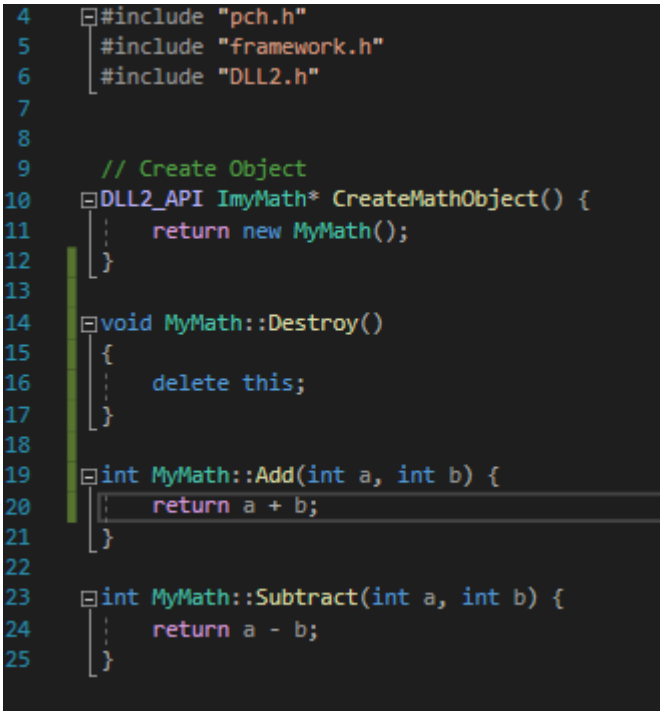
```
ordinal hint RVA      name

      1    0 00011154 CreateMathObject = @ILT+335(_CreateMathObject)
```

To jedyny symbol jaki musimy wyeksportować aby mieć dostęp do szeregu funkcji oraz zmiennych zawartych w wygenerowanej przez funckję klasę.

Dzięki temu musimy w pliku programu dokonać importu tylko raz, nieważne do ilu funkcji chcemy mieć dostęp. Przy 3 funkcjach i 2 zmiennych może nie stanowić to jeszcze dużej bariery, ale jeżeli program musi korzystać z 30 różnych funkcji, to na pewno kod stanie się bardziej przejrzysty oraz wygenerujemy go mniej.

Plik .cpp



```cpp
#include "pch.h"
#include "framework.h"
#include "DLL2.h"


// Create Object
DLL2_API ImyMath* CreateMathObject() {
    return new MyMath();
}

void MyMath::Destroy()
{
    delete this;
}

int MyMath::Add(int a, int b) {
    return a + b;
}

int MyMath::Subtract(int a, int b) {
    return a - b;
}
```

Plik .cpp to definicje metod. Tutaj widać implementację wyżej wspomnianej metody Destroy() i sposób na zniszczenie klasy. CreateMathObject() natomiast to również wspominany wzorzec Factory.

Plik main

```cpp
#include <iostream>
#include <Windows.h>
#include "../DLL2/DLL2.h"
using namespace std;

typedef ImyMath* (*CREATE_MATH) ();

int main()
{
    HINSTANCE hDLL = LoadLibrary(L"DLL2.dll");

    if (hDLL == NULL) {
        std::cout << "Failed to load library.\n";
    }
    else {
        CREATE_MATH pEntryFunction =
            (CREATE_MATH)GetProcAddress(hDLL, "CreateMathObject");
        ImyMath* pMath = pEntryFunction();

        pMath->a = 1;
        pMath->b = 2;

        cout << pMath->a + pMath->b + 1 << endl;

        int a = 0, b = 0;
        cout << "a: "; cin >> a;
        cout << "b: "; cin >> b;

        cout << a << "+" << b << "=" << pMath->Add(a, b) << std::endl;
        cout << a << "-" << b << "=" << pMath->Subtract(a, b) << std::endl;

        pMath->Destroy();

        FreeLibrary(hDLL);
    }

```

Nie będę opisywał całego pliku, schemat pozostaje praktycznie taki sam. Tym razem potrzebne jest dołączenie pliku .h w celu rozpoznania nazwy interfejsu. Nasz wskaźnik na funkcję CreateMathObject pozwala nam uzyskanie instancji klasy, którą przypisujemy do obiektu pMath. Tym sposobem obiekt pMath staje się dokładnie klasą MyMath, która zawiera metody oraz zmienne. Od tego momentu możemy używać pMath'a jak zwykłej klasy.

Dalsza część pliku to parę testów, które dowiadują, że faktycznie obiekt jest w pełni sprawny, a jego parametry dostępne.

Na sam koniec nisczymy obiekt aby zapobiec niechcianemu wyciekowi danych.

Zwalniamy również bibliotekę.

Opracował Filip Wawrzyniak 3rok studiów inż. Na WFiIS UŁ, w roku 2021.

Komentarz prowadzącego zajęcia: Powyższe rozwiązanie będzie dobrze pracowało przy jedynie typach prostych w klasie. Jeśli są typy złożone i to oparte o STL to trzeba założyć, że eksport/import będzie działał w oparciu o tylko jeden kompilator. Będzie działał tak długo aż nie zmieni się wersja typu złożonego.