# **11 лаба DLL ; create start 02 03 04**

### Общее про DLL

DLL– программный модуль, который может быть загружен динамически (во время выполнения, в отличии от статической где во время компиляции) и содержать функции и данные.

Механизм проецирования – один и тот же образ в памяти(в оперативной) DLL используется несколькими процессами (*код – общий, данные по отдельности*).

Т.е работает по принципу File Mapping, проекция в памяти и проекция в адресное пространство каждого процесса.

При сборке dll у нас появляется 2 файла: **.dll** и **.lib**

Есть 2 способа подключить dll: динамически(LoadLibrary и GetProcAddress) и **статически** (через .lib библиотеку импорта)

**динамически**

#### **Использование DLL в программе на Visual C++**

<https://rsdn.org/article/baseserv/dlluse.xml>

<http://www.codenet.ru/progr/visualc/Using-DLL.php>

Использование DLL

Практически невозможно создать приложение Windows, в котором не использовались бы библиотеки DLL. В DLL содержатся все функции Win32 API и несчетное количество других функций операционных систем Win32.

Вообще говоря, DLL - это просто наборы функций, собранные в библиотеки. Однако, в отличие от своих статических родственников (файлов . lib), библиотеки DLL не присоединены непосредственно к выполняемым файлам с помощью редактора связей. В выполняемый файл занесена только информация об их местонахождении. В момент выполнения программы загружается вся библиотека целиком. Благодаря этому разные процессы могут пользоваться совместно одними и теми же библиотеками, находящимися в памяти. Такой подход позволяет сократить объем памяти, необходимый для нескольких приложений, использующих много общих библиотек, а также контролировать размеры ЕХЕ-файлов.

Однако, если библиотека используется только одним приложением, лучше сделать ее обычной, статической. Конечно, если входящие в ее состав функции будут использоваться только в одной программе, можно просто вставить в нее соответствующий файл с исходным текстом.

Чаще всего проект подключается к DLL статически, или неявно, на этапе компоновки. Загрузкой DLL при выполнении программы управляет операционная система. Однако, DLL можно загрузить и явно, или динамически, в ходе работы приложения.

Многие знают, что существует два основных способа подключить DLL к программе - явный и неявный.

При неявном подключении (implicit linking)

линкеру передается ***библиотека импорта*** (обычно имеет расширение lib), содержащая список переменных и функций DLL, которые могут использовать приложения.

Обнаружив, что программа обращается хотя бы к одной из них, линкер добавляет в целевой exe-файл ***таблицу импорта***.

Таблица импорта содержит список всех DLL, которые использует программа, с указанием конкретных переменных и функций, к которым она обращается

. Позже, когда exe-файл будет запущен, загрузчик проецирует все DLL, перечисленные в таблице импорта, на адресное пространство процесса; в случае неудачи весь процесс немедленно завершается.

При явном подключении (explicit linking) приложение вызывает функцию LoadLibrary, чтобы загрузить DLL, затем использует функцию GetProcAddress, чтобы получить указатели на требуемые функции (или переменные), а по окончании работы с ними вызывает FreeLibrary, чтобы выгрузить библиотеку и освободить занимаемые ею ресурсы.

Каждый из способов имеет свои достоинства и недостатки. В случае неявного подключения все библиотеки, используемые приложением, загружаются в момент его запуска и остаются в памяти до его завершения (даже если другие запущенные приложения их не используют). Это может привести к нерациональному расходу памяти, а также заметно увеличить время загрузки приложения, если оно использует очень много различных библиотек. Кроме того, если хотя бы одна из неявно подключаемых библиотек отсутствует, работа приложения будет немедленно завершена. Явный метод лишен этих недостатков, но делает программирование более неудобным, поскольку требуется следить за своевременными вызовами LoadLibrary и соответствующими им вызовами FreeLibrary, а также получать адрес каждой функции через вызов GetProcAddress.

Экспортирование функций из DLL

Чтобы приложение могло обращаться к функциям динамической библиотеки, каждая из них должна занимать строку в таблице экспортируемых функций DLL. Есть два способа занести функцию в эту таблицу на этапе компиляции.

Метод \_\_declspec (dllexport)

Можно экспортировать функцию из DLL, поставив в начале ее описания модификатор \_\_declspec (dllexport) . Кроме того, в состав MFC входит несколько макросов, определяющих \_\_declspec (dllexport), в том числе AFX\_CLASS\_EXPORT, AFX\_DATA\_EXPORT и AFX\_API\_EXPORT.

Метод \_\_declspec применяется не так часто, как второй метод, работающий с файлами определения модуля (.def), и позволяет лучше управлять процессом экспортирования.

#### **Неявное подключение**

Библиотеки импортирования

При статическом подключении DLL имя .lib-файла определяется среди прочих параметров редактора связей в командной строке или на вкладке "Link" диалогового окна "Project Settings" среды Developer Studio. Однако .lib-файл, используемый **при неявном подключении DLL,** - это не обычная статическая библиотека. Такие .lib-файлы называются **библиотеками импортирования** (import libraries). В них содержится не сам код библиотеки, а только ссылки на все функции, экспортируемые из файла DLL, в котором все и хранится. В результате библиотеки импортирования, как правило, имеют меньший размер, чем DLL-файлы. К способам их создания вернемся позднее. А сейчас рассмотрим другие вопросы, касающиеся неявного подключения динамических библиотек.

Загрузка неявно подключаемой DLL

При запуске приложение пытается найти все файлы DLL, неявно подключенные к приложению, и поместить их в область оперативной памяти, занимаемую данным процессом. Поиск файлов DLL операционной системой осуществляется в следующей последовательности.

* **Каталог, в котором находится ЕХЕ-файл.**
* **Текущий каталог процесса.**
* **Системный каталог Windows.**

Если библиотека DLL не обнаружена, приложение выводит диалоговое окно с сообщением о ее отсутствии и путях, по которым осуществлялся поиск. Затем процесс отключается.

Если нужная библиотека найдена, она помещается в оперативную память процесса, где и остается до его окончания. Теперь приложение может обращаться к функциям, содержащимся в DLL.

Это наиболее простой метод подключения DLL к нашей программе. Все, что нам нужно - это передать линкеру имя библиотеки импорта, чтобы он использовал ее в процессе сборки. Сделать это можно различными способами.

Во-первых, можно непосредственно добавить файл MyDll.lib в проект посредством команды Project->Add to project->Files... Во-вторых, можно указать имя библиотеки импорта в опциях линкера. Для этого откройте окно настроек проекта (Project->Settings...) и добавьте в поле Object/Library modules на вкладке Link имя MyDll.lib. Наконец, можно встроить ссылку на библиотеку импорта прямо в исходный код программы. Для этого используется директива #pragma c ключем comment. В нашем случае необходимо вставить в программу строчку:

|  |
| --- |
| #pragma comment(lib,"MyDll.lib") |

Теперь можно использовать в программе любые переменные, функции и классы, содержащиеся в DLL, как если бы они находились в статической библиотеке. Например:

|  |
| --- |
| #include <stdio.h>  #include "MyDll.h"  ...  Var = 123;  printf("Var = %d\n", Var);  Function(0, 0);  Class c;  c.SetA(312);  printf("c.a = %d\n", c.GetA());  c.VirtualFunc();  Class::StaticVar = 231;  printf("Class::StaticVar = %d\n", Class::StaticVar);  Class::StaticFunc(); |

#### **Явное подключение**

#### **Загрузка DLL**

Динамическая загрузка и выгрузка DLL

Вместо того, чтобы Windows выполняла динамическое связывание с DLL при первой загрузке приложения в оперативную память, можно связать программу с модулем библиотеки во время выполнения программы (при таком способе в процессе создания приложения не нужно использовать библиотеку импорта). В частности, можно определить, какая из библиотек DLL доступна пользователю, или разрешить пользователю выбрать, какая из библиотек будет загружаться. Таким образом можно использовать разные DLL, в которых реализованы одни и те же функции, выполняющие различные действия. Например, приложение, предназначенное для независимой передачи данных, сможет в ходе выполнения принять решение, загружать ли DLL для протокола TCP/IP или для другого протокола.

Как уже говорилось ранее, при явном подключении DLL программист должен сам позаботиться о загрузке библиотеки перед ее использованием. Для этого используется функция LoadLibrary, которая получает имя библиотеки и возвращает ее дескриптор. Дескриптор необходимо сохранить в переменной, так как он будет использоваться всеми остальными функциями, предназначенными для работы с DLL.

В нашем примере загрузка DLL выглядит так.

|  |
| --- |
| HMODULE hLib;  hLib = LoadLibrary("MyDll.dll"); |

#### **Согласование интерфейсов EXTERN ‘C’**

При использовании собственных библиотек или библиотек независимых разработчиков придется обратить внимание на согласование вызова функции с ее прототипом.

Если бы мир был совершенен, то программистам не пришлось бы беспокоиться о согласовании интерфейсов функций при подключении библиотек - все они были бы одинаковыми. Однако мир далек от совершенства, и многие большие программы написаны с помощью различных библиотек без C++.

По умолчанию в Visual C++ интерфейсы функций согласуются по правилам C++. Это значит, что параметры заносятся в стек справа налево, вызывающая программа отвечает за их удаление из стека при выходе из функции и расширении ее имени. Расширение имен (name mangling) позволяет редактору связей различать перегруженные функции, т.е. функции с одинаковыми именами, но разными списками аргументов. Однако в старой библиотеке С функции с расширенными именами отсутствуют.

Хотя все остальные правила вызова функции в С идентичны правилам вызова функции в C++, в библиотеках С имена функций не расширяются. К ним только добавляется впереди символ подчеркивания (\_).

Если необходимо подключить библиотеку на С к приложению на C++, все функции из этой библиотеки придется объявить как внешние в формате С:

extern "С" int MyOldCFunction(int myParam);

Объявления функций библиотеки обычно помещаются в файле заголовка этой библиотеки, хотя заголовки большинства библиотек С не рассчитаны на применение в проектах на C++. В этом случае необходимо создать копию файла заголовка и включить в нее модификатор extern "C" к объявлению всех используемых функций библиотеки. Модификатор extern "C" можно применить и к целому блоку, к которому с помощью директивы #tinclude подключен файл старого заголовка С. Таким образом, вместо модификации каждой функции в отдельности можно обойтись всего тремя строками:

extern "С"

{

#include "MyCLib.h"

}

В программах для старых версий Windows использовались также соглашения о вызове функций языка PASCAL для функций Windows API. В новых программах следует использовать модификатор winapi, преобразуемый в \_stdcall. Хотя это и не стандартный интерфейс функций С или C++, но именно он используется для обращений к функциям Windows API. Однако обычно все это уже учтено в стандартных заголовках Windows.

#### **Вызов функций**

После того как библиотека загружена, адрес любой из содержащихся в ней функций можно получить с помощью GetProcAddress, которой необходимо передать дескриптор библиотеки и имя функции. Затем функцию из DLL можно вызывать, как обычно. Например:

|  |
| --- |
| void (\*pFunction)(int, int);  (FARPROC &)pFunction = GetProcAddress(hLib, "Function");  pFunction(0, 0); |

Обратите внимание на приведение указателя к ссылке на тип FARPROC. FARPROC - это указатель на функцию, которая не принимает параметров и возвращает int. Именно такой указатель возвращает функция GetProcAddress. Приведение типа необходимо, чтобы умиротворить компилятор, который строго следит за соответствием типов параметров оператора присваивания. Альтернативный подход заключается в использовании оператора typedef с последующим приведением значения, возвращаемого GetProcAddress, к указателю на функцию с нужным прототипом.

#### **Доступ к переменным**

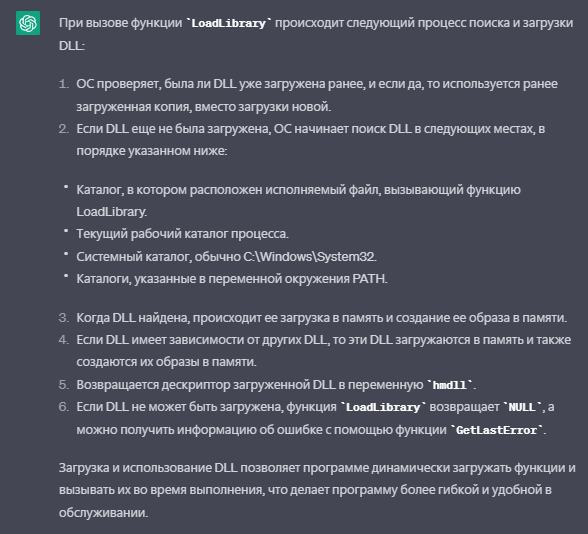
Хотя это не всегда очевидно из документации, получить указатель на переменную из DLL можно, используя все ту же функцию GetProcAddress. В нашем примере это выглядит так.

|  |
| --- |
| int \*pVar;  (FARPROC &)pVar = GetProcAddress(hLib, "Var");  \*pVar = 123; |

#### **Выгрузка библиотеки**

После того, как работа с библиотекой закончена, ее можно выгрузить, чтобы она не занимала системные ресурсы. Для этого используется функция FreeLibrary, которой следует передать дескриптор освобождаемой библиотеки. В нашем примере это выглядит так.

|  |
| --- |
| FreeLibrary(hLib); |



**FreeLibrary** операционная система освобождает выделенное виртуальное адресное пространство процесса, включая все выделенные в нем ресурсы, такие как данные и функции, связанные с DLL. Физическая память, используемая для хранения кода и данных DLL, освобождается только если нет других процессов, использующих эту же DLL.

### ===Что может спросить по лабе:

[Что такое динамическое подключение dll / статическое подключение dll (библиотека импорта)](#_Использование_DLL_в)

Динамическое подключение dll / статическое подключение dll (библиотека импорта) - это два способа использования библиотек в C++. Библиотека - это набор функций и классов, которые можно использовать в своем коде. В зависимости от того, как вы подключаете библиотеку, она может иметь разные преимущества и недостатки.

Динамическое подключение dll (dynamic linking dll) означает, что вы используете библиотеку в виде отдельного файла с расширением .dll (dynamic link library). Этот файл содержит код библиотеки, который загружается в память при запуске вашей программы или при первом вызове функции из библиотеки. Для того, чтобы использовать динамическую библиотеку, вам нужно:

* Указать компилятору, какие функции и классы вы хотите импортировать из библиотеки. Это делается с помощью директивы #include и макроса \_\_declspec(dllimport).
* Указать линкеру, где находится файл библиотеки. Это делается с помощью опции /LIBPATH или добавлением пути к файлу в настройках проекта.
* Распространять файл библиотеки вместе с вашей программой, чтобы она могла его найти и загрузить.

Преимущества динамического подключения dll:

* Уменьшение размера вашей программы, так как код библиотеки не включается в нее.
* Возможность обновлять библиотеку без перекомпиляции вашей программы, если интерфейс библиотеки не меняется.
* Возможность делить одну библиотеку между несколькими программами, что экономит память и ресурсы.

Недостатки динамического подключения dll:

* Зависимость от наличия файла библиотеки на целевой машине, что может вызвать проблемы совместимости или безопасности.
* Необходимость управлять версиями библиотеки и избегать конфликтов между разными версиями.
* Возможная потеря производительности из-за дополнительных операций загрузки и связывания кода.

Статическое подключение dll (static linking dll) означает, что вы используете библиотеку в виде файла с расширением .lib (library). Этот файл содержит информацию для линкера о том, какие функции и классы вы хотите экспортировать из библиотеки. Для того, чтобы использовать статическую библиотеку, вам нужно:

* Указать компилятору, какие функции и классы вы хотите экспортировать из библиотеки. Это делается с помощью директивы #include и макроса \_\_declspec(dllexport).
* Указать линкеру, где находится файл библиотеки. Это делается так же, как и для динамической библиотеки.
* Ничего не распространять вместе с вашей программой, так как код библиотеки уже включен в нее.

Преимущества статического подключения dll:

* Независимость от наличия файла библиотеки на целевой машине, что упрощает распространение и установку вашей программы.
* Гарантия того, что ваша программа использует ту же версию библиотеки, с которой она была скомпилирована, что исключает проблемы совместимости или безопасности.
* Возможное увеличение производительности из-за отсутствия дополнительных операций загрузки и связывания кода.

Недостатки статического подключения dll:

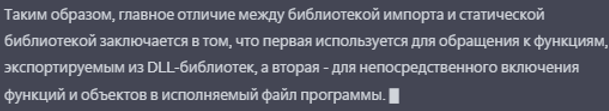
* Увеличение размера вашей программы, так как код библиотеки включается в нее.
* Невозможность обновлять библиотеку без перекомпиляции вашей программы, если интерфейс библиотеки меняется.
* Невозможность делить одну библиотеку между несколькими программами, что занимает больше памяти и ресурсов.

Про dll, что это, где хранится, что такое адресное пространство процесса, как dll в него попадает

[Что такое extern ‘C’](#_Согласование_интерфейсов_EXTERN)

По каким путям ищется dll (сначала рабочий каталог и где далее)

* Каталог, в котором находится ЕХЕ-файл.
* Текущий каталог процесса.
* Системный каталог Windows (обычно C:\Windows\System32).
* Директории, указанные в переменной окружения PATH



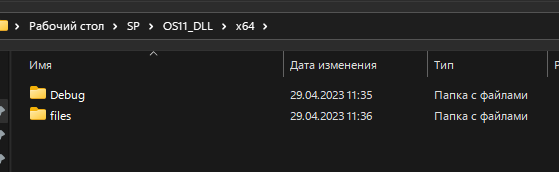
Логирование в файл показать

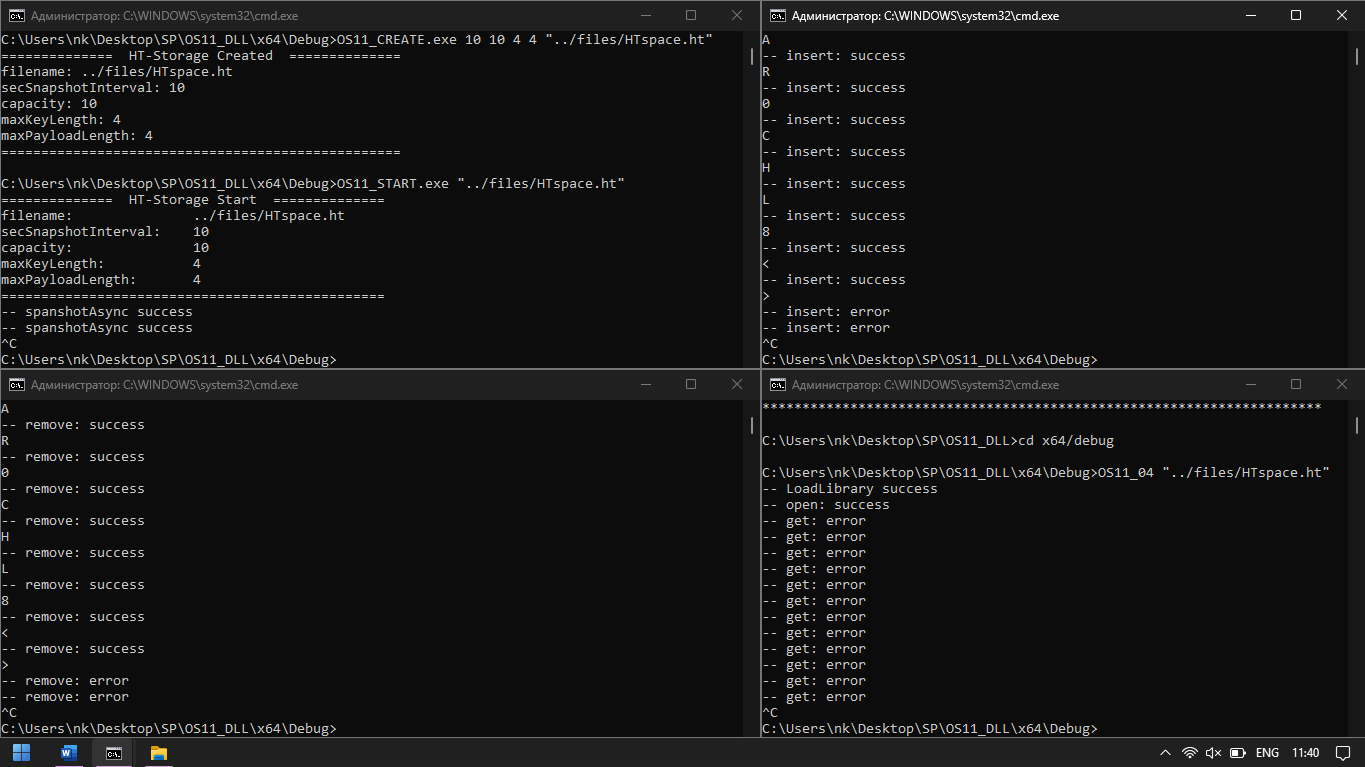
Create, Start, OS11\_02, OS11\_03, OS11\_04 – ввод параметров через консоль (4 консоли должно быть открыто)

Тестит лабу вводя неправильные значения (нулевой ключ, неправильное имя файла, больше вместимости и тд)

2000 3 4 4 "../files/HTspace.ht"

### Гайд по запуску лабы:

1. Пкм os11\_htapi в обозревателе решений – СОБРАТЬ , и так все другие проекты OS11\_02, ….
2. Создать папку files в папке x64 (либо просто пути подобрать в зависимости где files, но там багается)
3. OS11\_CREATE.exe 10 4 4 4 "../files/HTspace.ht" создаем
4. OS11\_START.exe "../files/HTspace.ht" запускаем
5. OS11\_02.exe "../files/HTspace.ht" ; OS11\_03.exe "../files/HTspace.ht ; OS11\_04.exe "../files/HTspace.ht вставка/удаление/get



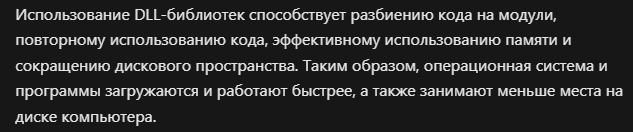
Ctrl + C остановка

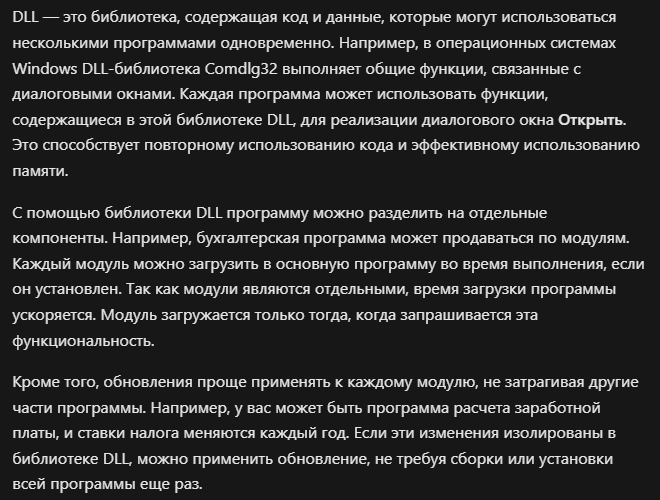
### Лекция\_09\_DLL

#### **Dynamic Link Library**

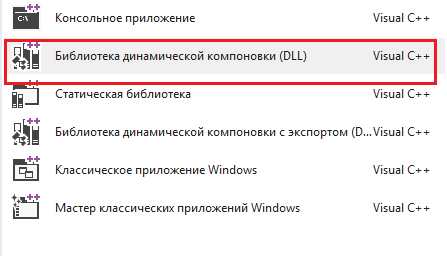
DLL– программный модуль, который может быть загружен динамически (во время выполнения, в отличии от статической где во время компиляции) и содержать функции и данные.

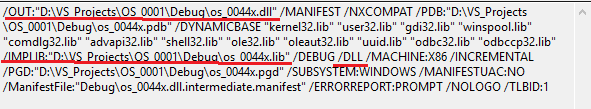
Механизм проецирования – один и тот же образ в памяти(в оперативной) DLL используется несколькими процессами (*код – общий, данные по отдельности*).

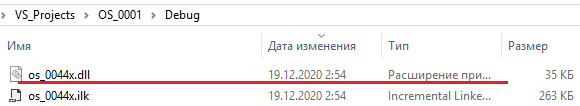




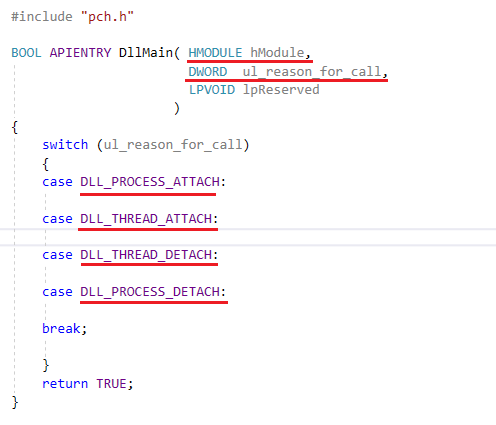
#### **OS: Windows: создание DLL, DDLmain, LoadLibrary, FreeLibrary**

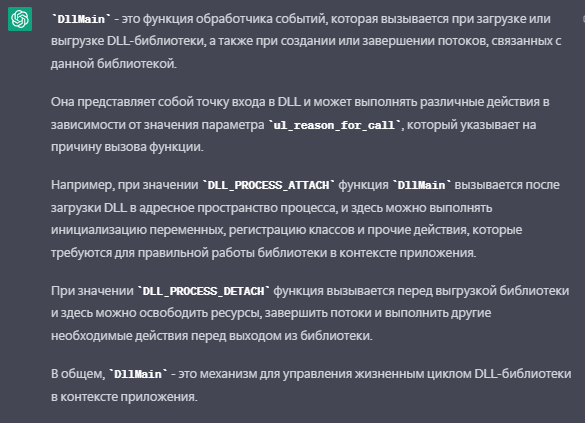
****

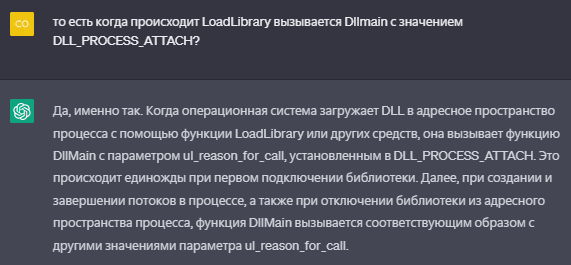
****

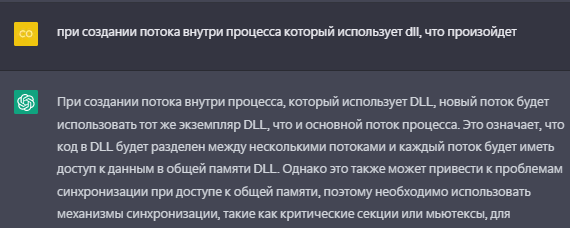
****

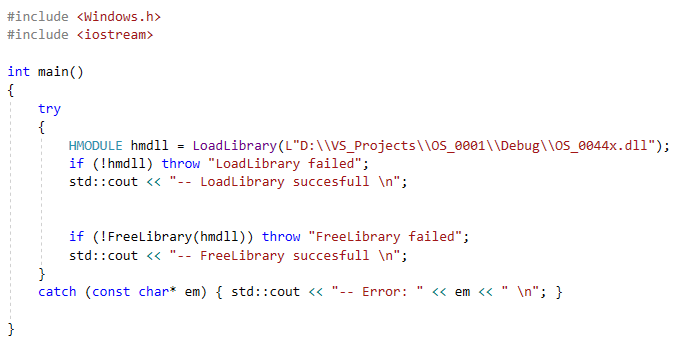
#### **Dllmain**

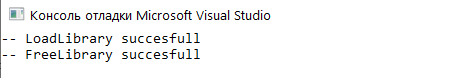
****



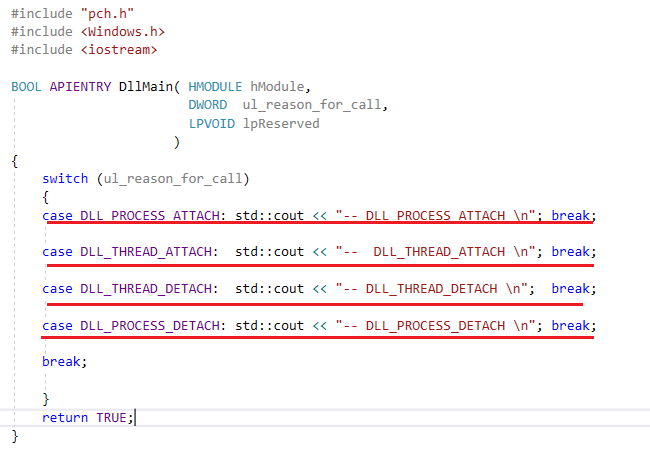


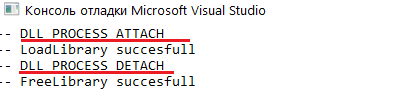


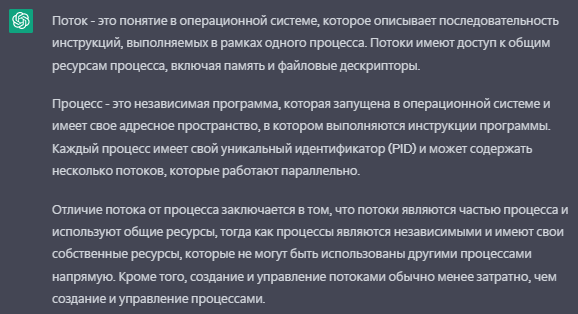
****

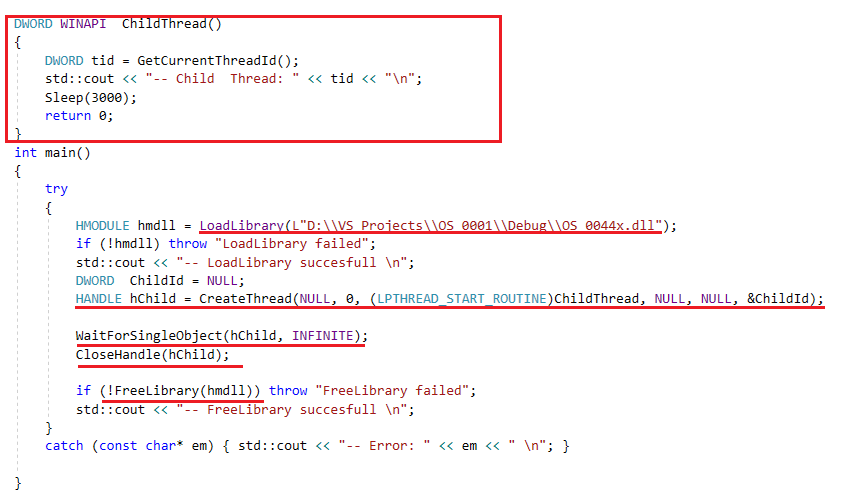
****

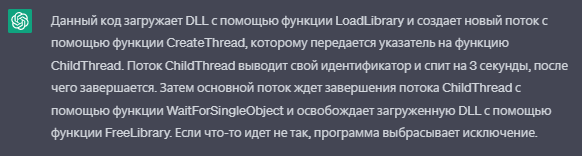
#### **OS: Windows: процессы и потоки**

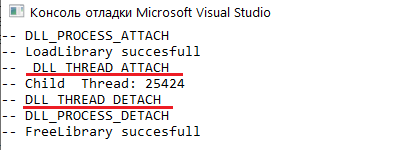
****

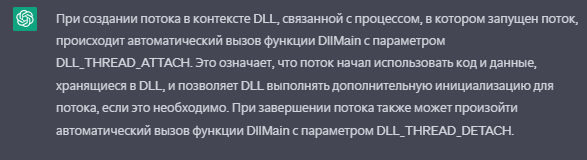
****

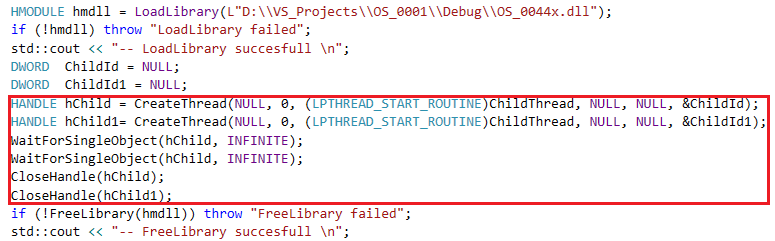


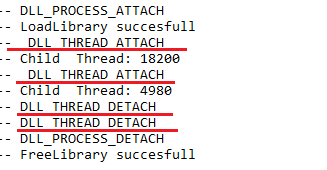
****



****



****

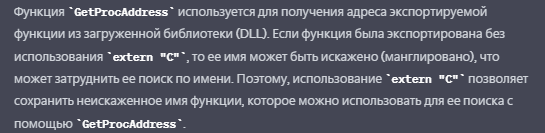
****

#### **OS: Windows: экспорт и импорт функций**

**extern “C”** – не искажает имени (в стиле С)

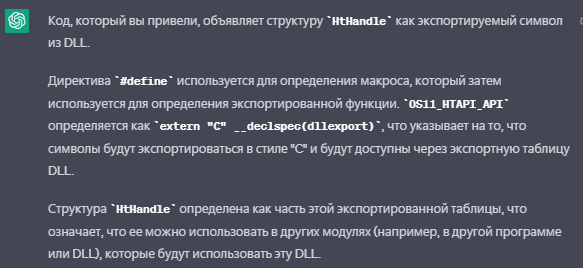
При импортеадрес функции можно получить по **GetProcAddress:**



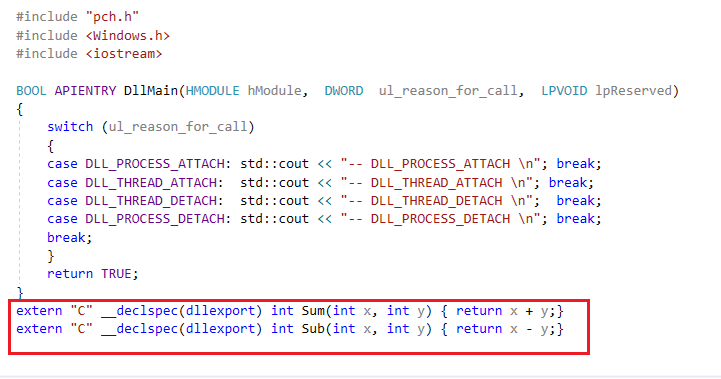
988888888888888888888+



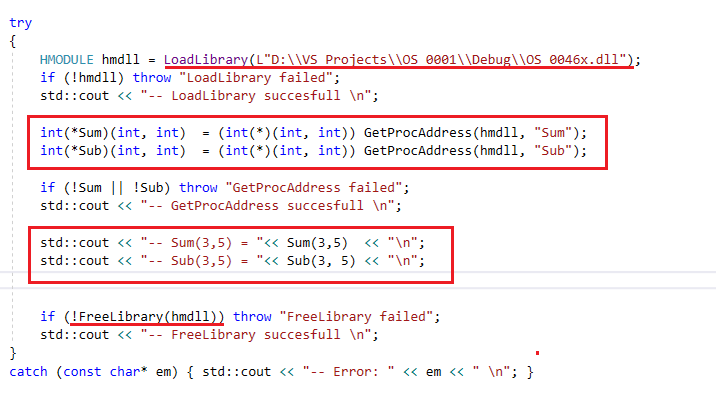


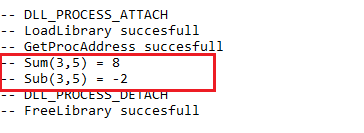




****

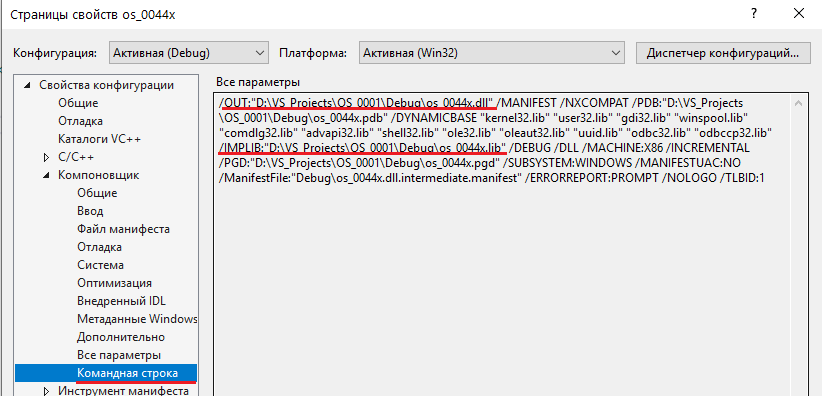


****

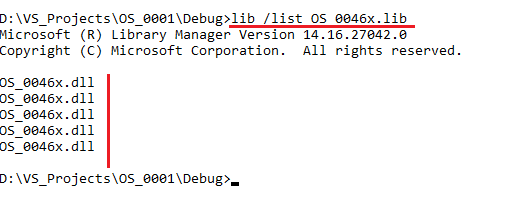
****

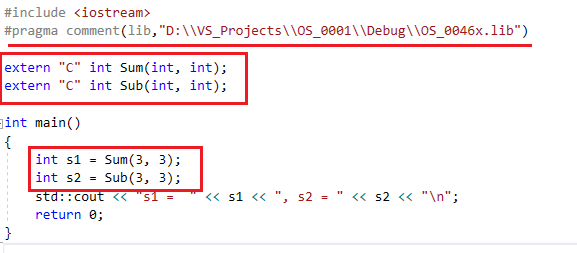
#### **OS: Windows: библиотека импорта**

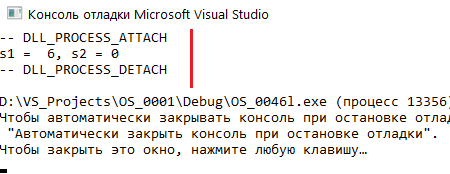
При сборке dll у нас появляется 2 файла: **.dll** и **.lib**:

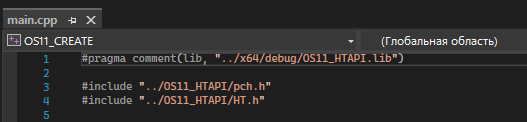


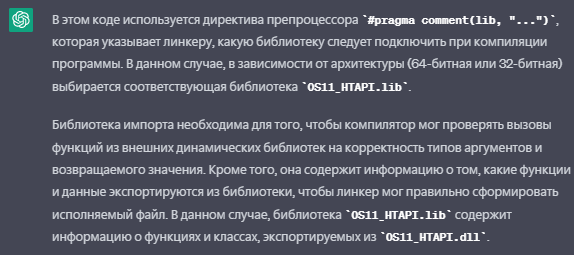
**Библиотека импорта** – просто содержит ссылки на адреса функций из dll.



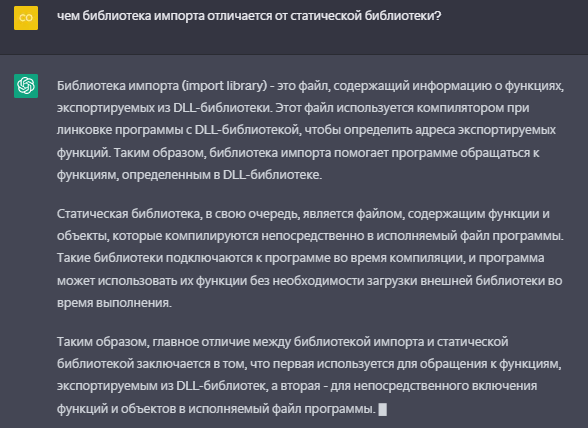








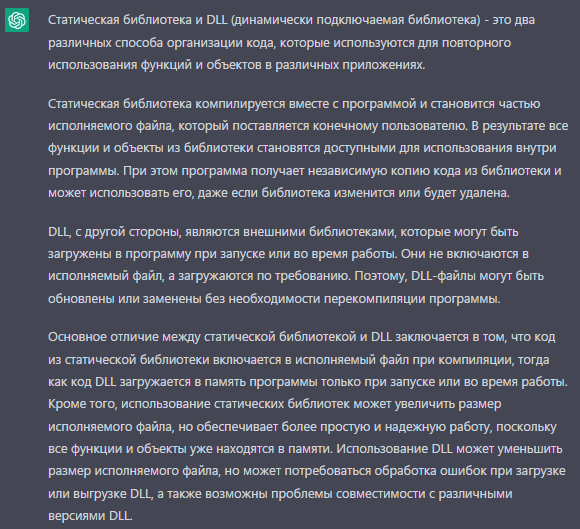




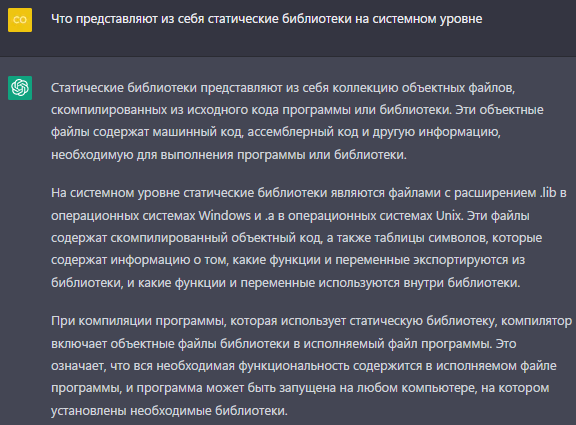


Отличия библиотек

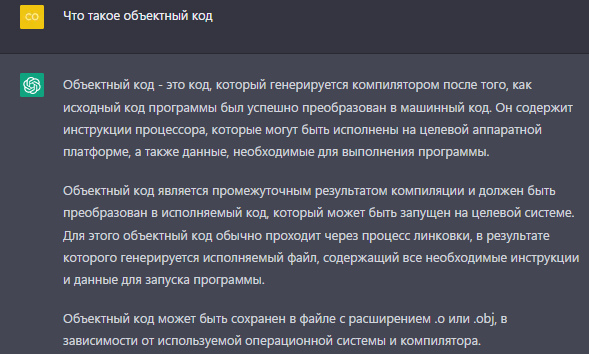
### ===Статическая библиотека (прошлая лаба)



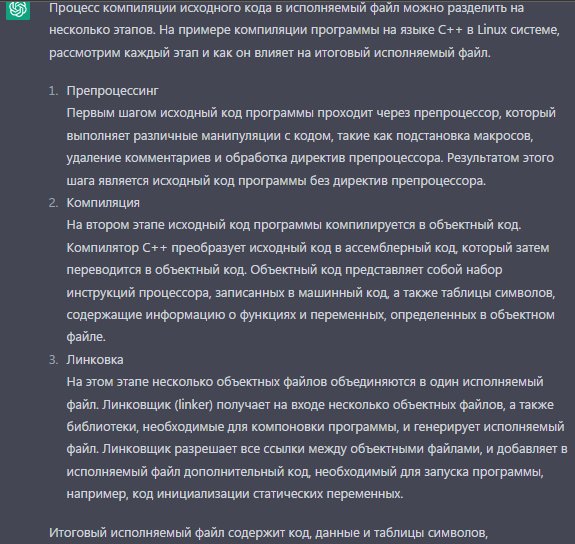
Статические библиотеки на системном уровне



Обьектный код

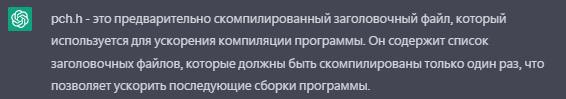


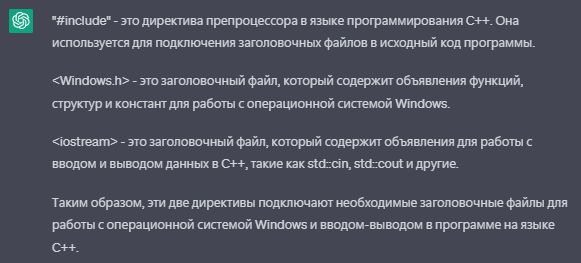
Процесс компиляции исходного кода в исполняемый файл

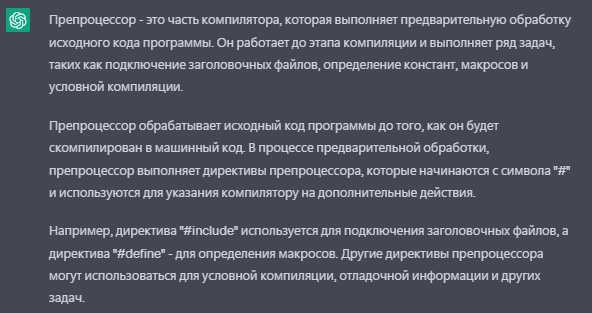


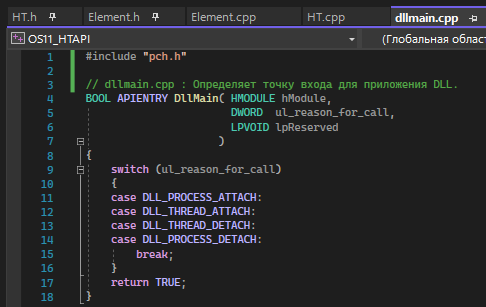
Был вопрос на каком этапе подкл статическая библиотека -> на этапе линковки (объединения в исполняемый файл)

### Код лабы



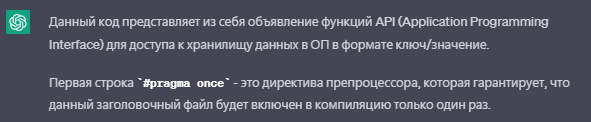


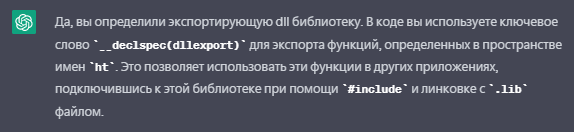




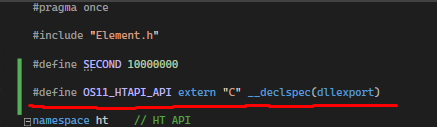
Файл dllmain.cpp - это файл, используемый для определения точки входа в динамическую библиотеку.

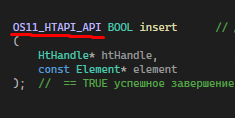
#### **Ht.h**

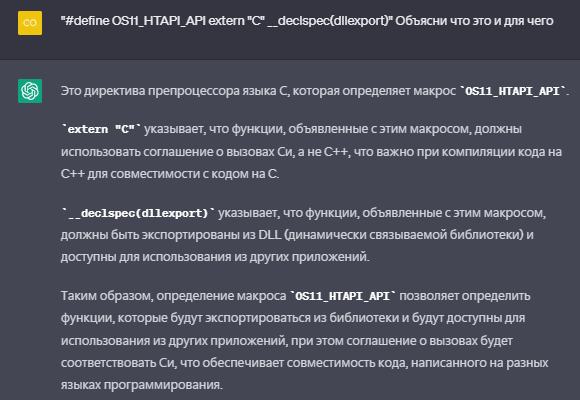


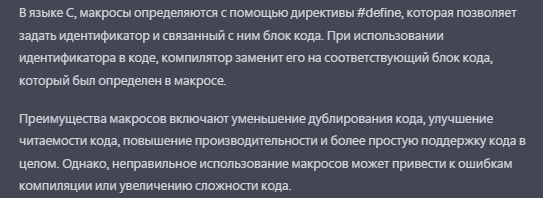






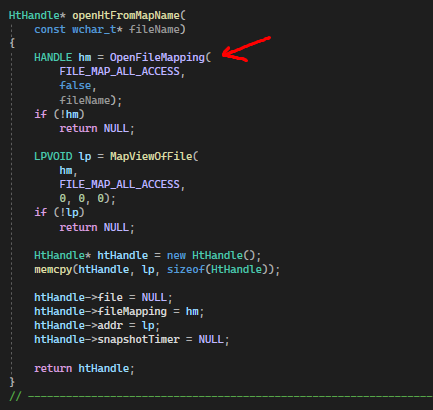






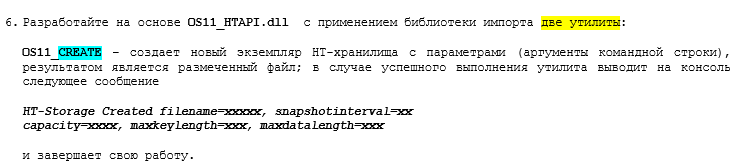
#### **Ht.cpp**



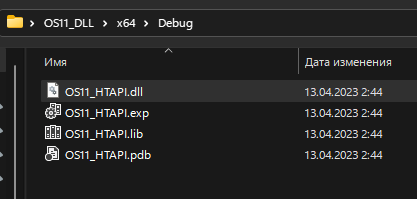


Межпроцессное взаимодействие с помощью FileMapping (открываем существующую файловую проекцию через OpenFileMapping)

#### **OS11\_CREATE**

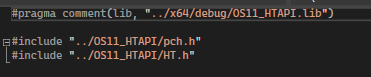


Пкм по проекту dll библиотеки OS11\_HTAPI -> собрать, в папке проекта появится:

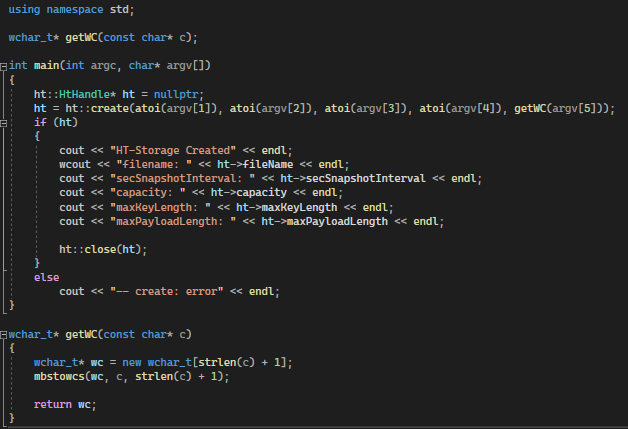


dll и библиотека импорта(содержит только ссылки на функции)

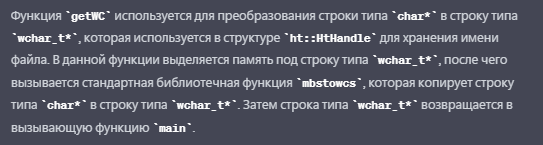
##### **main.cpp**

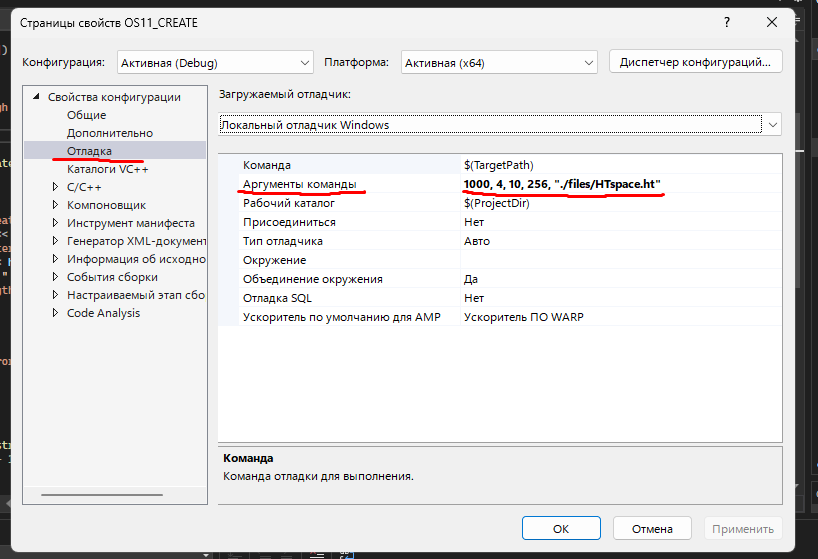


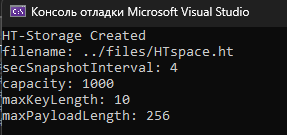
Используем библиотеку импорта



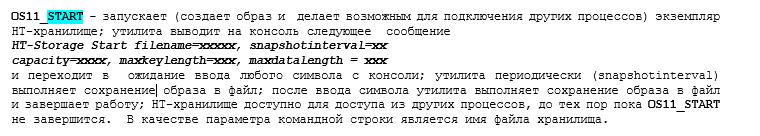
Аргументы командной строки **argc** и **argv** используются для передачи параметров функции **ht::create**. В этой функции создается указатель на структуру **ht::HtHandle** и заполняются его поля с помощью переданных параметров командной строки. Если хэш-таблица была успешно создана, то функция выводит некоторую информацию о ней и закрывает ее с помощью функции **ht::close**.



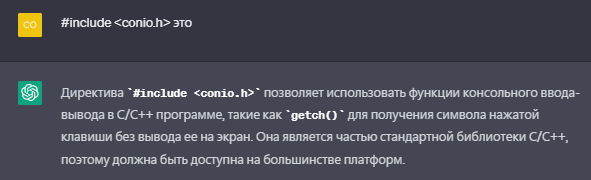


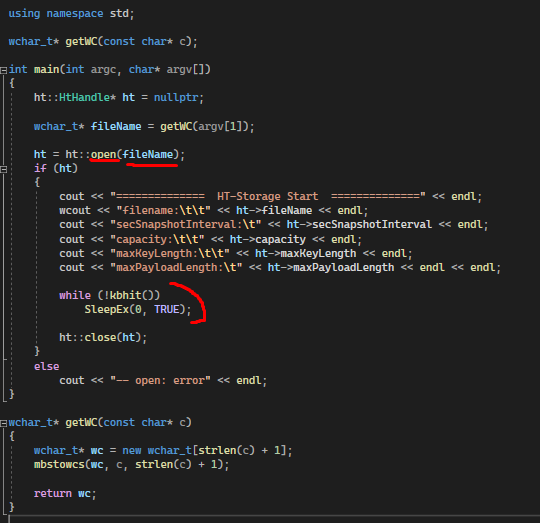


#### **OS11\_START**

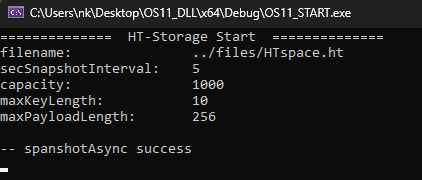


Создает file mapping по существующему файлу и сохраняет по интервалу на диск

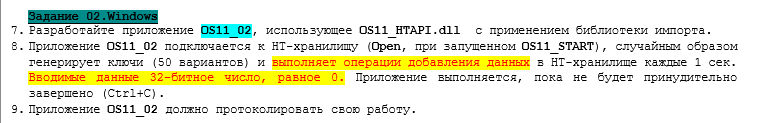


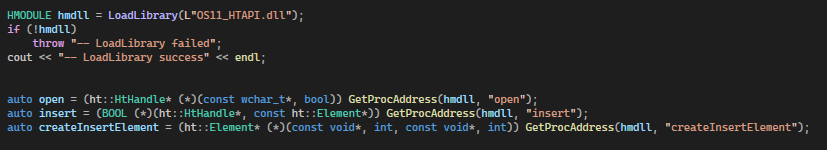


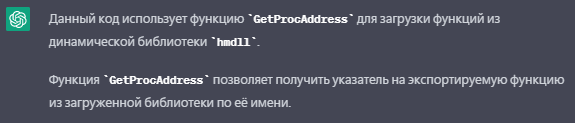
**kbhit** – цикл пока не будет нажата клавиша

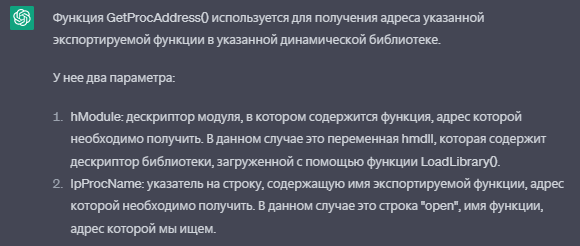


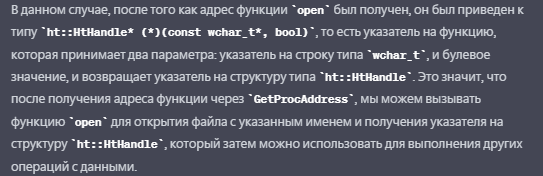
#### **OS11\_02**



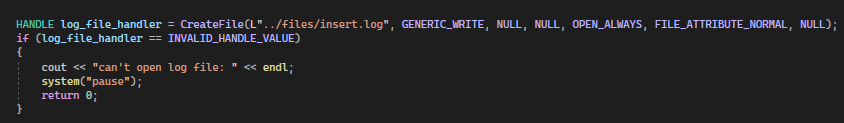




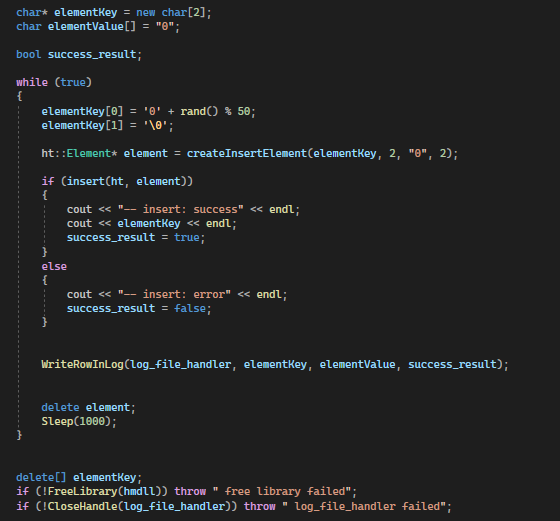




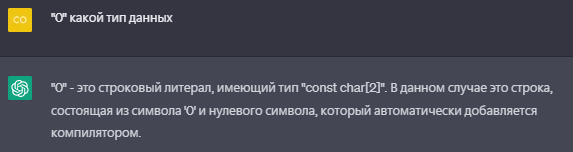


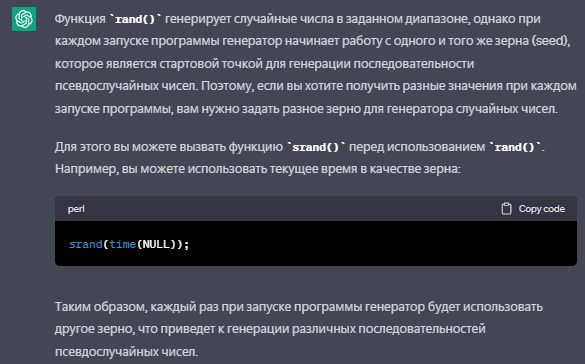


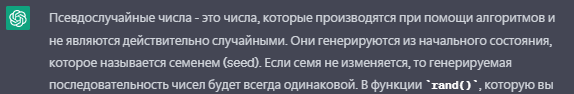
**Логи**



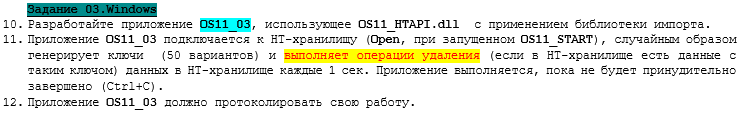
**Ключ = ASCI ‘0’ + Рандомное число от 0 до 49, потом insert элемента с этим ключом**

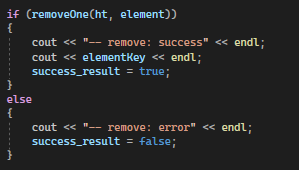






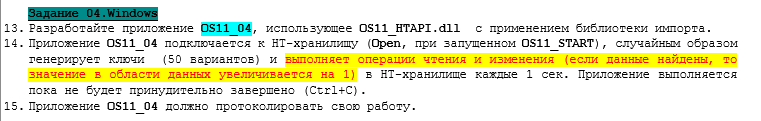
#### **OS11\_03**





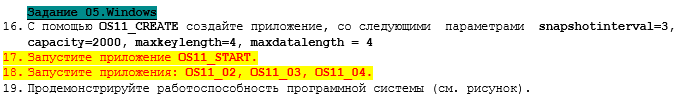
**Все то же самое, только вместо insert – removeOne**

#### **OS11\_04**



Получаем элемент и update его данные

#### **OS11\_05**



### Задания:

1. ***Внимание! Для работы с файловой системой использовать только OS API.***
2. В заданиях лабораторной работы используется программная система ***HT-хранилище***, разработанная в предыдущей лабораторной работе.

**Задание 01.Windows**

1. Разработайте приложение **OS11\_HTAPI**.
2. Приложение **OS11\_HTAPI** предоставляет собой DLL-библиотеку (**OS11\_HTAPI.dll**), реализующей интерфейс **HT API** и экспортирующей его функции.
3. Доработайте реализацию **OS11\_HTAPI** для того, чтобы с общим HT-хранилищем могли работать несколько независимых процессов операционной системы.
4. Разработайте на основе **OS11\_HTAPI.dll**  с применением библиотеки импорта две утилиты:

**OS11\_CREATE** – создает новый экземпляр HT-хранилища с параметрами (аргументы командной строки), результатом является размеченный файл; в случае успешного выполнения утилита выводит на консоль следующее сообщение

***HT-Storage Created filename=xxxxx, snapshotinterval=xx***

***capacity=xxxx, maxkeylength=xxx, maxdatalength=xxx***

и завершает свою работу.

**OS11\_START** - запускает (создает образ и делает возможным для подключения других процессов) экземпляр HT-хранилище; утилита выводит на консоль следующее сообщение

***HT-Storage Start*** ***filename=xxxxx, snapshotinterval=xx***

***capacity=xxxx, maxkeylength=xxx, maxdatalength = xxx***

и переходит в ожидание ввода любого символа с консоли; утилита периодически (snapshotinterval) выполняет сохранение образа в файл; после ввода символа утилита выполняет сохранение образа в файл и завершает работу; HT-хранилище доступно для доступа из других процессов, до тех пор пока **OS11\_START** не завершится. В качестве параметра командной строки является имя файла хранилища.

**Задание 02.Windows**

1. Разработайте приложение **OS11\_02**, использующее **OS11\_HTAPI.dll**  с применением библиотеки импорта.
2. Приложение **OS11\_02** подключается к HT-хранилищу (**Open**, при запущенном **OS11\_START**), случайным образом генерирует ключи (50 вариантов) и выполняет операции добавления данных в HT-хранилище каждые 1 сек. Вводимые данные 32-битное число, равное 0. Приложение выполняется, пока не будет принудительно завершено (Ctrl+C).
3. Приложение **OS11\_02** должно протоколировать свою работу.

**Задание 03.Windows**

1. Разработайте приложение **OS11\_03**, использующее **OS11\_HTAPI.dll**  с применением библиотеки импорта.
2. Приложение **OS11\_03** подключается к HT-хранилищу (**Open**, при запущенном **OS11\_START**), случайным образом генерирует ключи (50 вариантов) и выполняет операции удаления (если в HT-хранилище есть данные с таким ключом) данных в HT-хранилище каждые 1 сек. Приложение выполняется, пока не будет принудительно завершено (Ctrl+C).
3. Приложение **OS11\_03** должно протоколировать свою работу.

**Задание 04.Windows**

1. Разработайте приложение **OS11\_04**, использующее **OS11\_HTAPI.dll**  с применением библиотеки импорта.
2. Приложение **OS11\_04** подключается к HT-хранилищу (**Open**, при запущенном **OS11\_START**), случайным образом генерирует ключи (50 вариантов) и выполняет операции чтения и изменения (если данные найдены, то значение в области данных увеличивается на 1) в HT-хранилище каждые 1 сек. Приложение выполняется пока не будет принудительно завершено (Ctrl+C).
3. Приложение **OS11\_04** должно протоколировать свою работу.

**Задание 05.Windows**

1. С помощью **OS11\_CREATE** создайте приложение, со следующими параметрами **snapshotinterval=3,**

**capacity=2000, maxkeylength=4, maxdatalength = 4**

1. Запустите приложение **OS11\_START.**
2. Запустите приложения: **OS11\_02, OS11\_03, OS11\_04.**
3. Продемонстрируйте работоспособность программной системы (см. рисунок).

