МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»   
(Самарский университет)

*Борисов А.Н.*

**СОГЛАШЕНИЯ О ВЫЗОВАХ. ОСНОВЫ ДЕКОМПИЛЯЦИИ**

*Методические указания к лабораторной работе 3*

Самара, 2022

СОДЕРЖАНИЕ

[Цели и задачи лабораторной работы 3](#_Toc98163076)

[1 Введение 3](#_Toc98163077)

[1.1 Исторические сведения 3](#_Toc98163078)

[1.2 Основные определения 4](#_Toc98163079)

[2 Соглашения о вызовах 6](#_Toc98163080)

[2.1 Соглашения для х86 6](#_Toc98163081)

[2.2 Соглашения для х86-64 7](#_Toc98163082)

[2.2.1 Соглашение System V 8](#_Toc98163083)

[2.2.2 Соглашение Microsoft 9](#_Toc98163084)

[3 Компиляция, дизассемблирование и декомпиляция 11](#_Toc98163085)

[3.1 Общая схема 11](#_Toc98163086)

[3.2 Статические библиотеки как зависимость при компоновке 12](#_Toc98163087)

[3.2.1 В CMake 12](#_Toc98163088)

[3.2.2 В SASM 12](#_Toc98163089)

[3.3 Символы в NASM 13](#_Toc98163090)

[3.3.1 Глобальные символы 13](#_Toc98163091)

[3.3.2 Внешние символы 13](#_Toc98163092)

[3.3.3 Локальные метки 13](#_Toc98163093)

[3.4 Искажение имен (mangling) 15](#_Toc98163094)

[3.4.1 Искажение имен в C 15](#_Toc98163095)

[3.4.2 Искажение имен в С++ 15](#_Toc98163096)

[3.5 Получение объектного файла из статической библиотеки 16](#_Toc98163097)

[4 Задание на лабораторную работу 17](#_Toc98163098)

[Общие замечания 17](#_Toc98163099)

[Задание 1 17](#_Toc98163100)

[Задание 2 17](#_Toc98163101)

# Цели и задачи лабораторной работы

Цель лабораторной работы: изучение взаимодействия с внешним кодом, методов компиляции и компоновки исполняемых файлов, основ дизассемблирования.

Задание на лабораторную работу: реализовать две программы согласно варианту на лабораторную работу.

# Введение

## Исторические сведения

Еще со времен существования ассемблера, как основного способа программирования программистам приходилось вырабатывать некие общие соглашения о том, как им следует писать программный код, чтобы остальные программисты могли далее с ним работать. Как правило, подобные соглашения были своими для каждой фирмы и явно документировались при передаче программного кода за пределы фирмы.

С появлением компиляторов данная проблема стала решаться автоматически, однако данные соглашения по-прежнему были своими для каждого компилятора. К примеру, компилятор языка Pascal мог использовать свое соглашение о вызовах, отличное от компилятора языка C, а компилятор для языка С от Wacom мог использовать соглашение, отличное от компилятора С от Microsoft.

Начиная с 2000-х годов на рынке компиляторов С/С++ доминирующими силами остались компиляторы Microsoft Visual C++ Compiler и семейство компиляторов GNU Compiler Collection (включает в себя компиляторы g++ и gcc). Каждая из сторон использует собственные соглашения и нацелена на свою платформу (Windows и Linux/UNIX), хотя GCC может, при необходимости, собирать код для Windows. Остальные компиляторы: Intel C++ Compiler, Clang, Embarcadero C++ Builder по умолчанию подстраиваются под одну из этих сторон.

Помимо компиляторов, существенное влияние на программные соглашения также определяет целевая операционная система. В частности, ОС регламентирует форматы исполняемых файлов и способы взаимодействия программы и ОС.

Суммарно набор всех программных соглашений, действующих на уровне двоичного кода, называется ABI (Application Binary Interface, двоичный интерфейс приложений). ABI включает в себя форматы файлов (исполняемые файлы, статические и динамические библиотеки), требования к точке входа в программу, механизм системных вызовов и пр. В рамках данной л/р наиболее важной составляющей ABI являются **соглашения о вызовах** – набор правил, по которым происходит вызов функций.

Существовал целый ряд соглашений о вызовах для 32-битных систем (cdecl, stdcall, pascal, fastcall, thiscall). С приходом 64-битных систем осталось только 2 основных соглашения о вызовах: Microsoft x64 и System V (“System 5”) x64.

## Основные определения

**Соглашение о вызове** - набор правил, регламентирующих вызов функций.

**Неизменяемые регистры** – регистры, содержание которых до и после вызова функции неизменно.

**Изменяемые регистры** – регистры, содержание которых до и после вызова функции может (но не обязано) отличаться.

**POD-структура** (Plain Old Data) – структура, имеющая тривиальные обычный конструктор, деструктор, конструктор копирования и оператор присваивания. *Все структуры языка С являются POD-структурами*. Классы и структуры языка С++ являются POD-структурами при указанных выше ограничениях. Ссылка на точное определение: <https://en.cppreference.com/w/cpp/named_req/PODType>

**Единица трансляции –** исходный код, являющийся результатом обработки одного файла исходного кода препроцессором (т.е., исходный код, получаемый из текста файла после расширения всех макросов и выполнения всех директив препроцессора).

**Объектный файл** – файл, являющийся результатом компиляции единицы трансляции. Объектный файл содержит машинный код и таблицы символов.

**Символ** – уникальное в пределах единицы трансляции имя с ассоциированным значением**.** Символами являются имена функций и глобальных переменных.

**Таблица символов** – таблица, сопоставляющая символы и их определения.

**Таблица импорта** – таблица, в которой перечислены символы, используемые в исполняемом файле, но определение которых в самом файле отсутствует.

**Компоновка** – процесс сборки программы из объектных файлов и других исполняемых файлов.

**Статическая компоновка** – компоновка, происходящая во время создания исполняемого файла программы.

**Динамическая компоновка** – компоновка, происходящая во время загрузки программы из исполняемого файла и во время исполнения самой программы.

# Соглашения о вызовах

## Соглашения для х86

Соглашения, действовавшие во времена 32-битных платформ интенсивно используют стек для передачи параметров. Возврат результата происходит в регистре EAX (char/shot/int или указатель), паре регистров [EDX:EAX] (long long/POD-структуры размером ≤ 8 байт) или ST0 (вещественные числа).

Характеристики соглашений о вызовах приведены в таблице 2.1

Таблица 2.1 Соглашения о вызовах для х86.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Соглашение** | cdecl(UNIX) | | cdecl(Windows) | stdcall | | thiscall |
| **Изменяемые регистры** | eax, ecx, edx,  st0-st7 (см. ниже), xmm0-7 | | | | | |
| **Неизменяемые**  **регистры** | esp, ebp, ebx, esi, edi,  mxcsr, xmm7-15,  управляющие регистры х87 | | | | | |
| **Передача аргументов** | На стеке в обратном порядке (см. ниже) | | | | | |
| **Указатель this** | 1-ый аргумент | | | | | ecx |
| **Очистка стека от аргументов** | Вызывающая функция | | | | Вызываемая функция | |
| **Возврат результата** | | | | | | |
| **char/short/int** | eax | | | | | |
| **Указатель** | eax | | | | | |
| **long long** | edx:eax | | | | | |
| **float/double** | st0 | | | | | |
| **POD-структура менее 8 байт** | в буфер возврата  (первый *скрытый* аргумент, убирает вызываемая функция) | edx:eax | | | | |
| **Остальные структуры** | в буфер возврата (второй после this аргумент) | | | | в буфер возврата (первый аргумент) |

Аргумент на стеке не может занимать менее 4 байт. Даже если аргумент – char, на стек будет положен int (4 байта). Если структура занимает 7 байт – на стек будет положено 8 байт (2\*4).

Особые указания относятся к стеку сопроцессора. По общему правилу, стек должен быть пуст в момент вызова функции, а в момент выхода из функции должен либо содержать результат в ST0, либо быть пустым

Примеры результатов компиляции для MSVC x86:

<https://godbolt.org/z/haErnoz7h>

Примеры результатов компиляции для GCC:

<https://godbolt.org/z/r4aETo9qP>

## Соглашения для х86-64

Соглашения для 64-битных платформ более интенсивно используют регистры для передачи параметров.

Общим для обоих соглашений является **требование к выравниванию стека:** *стек в момент вызова функции обязан быть выровнен по границе 16 байт:* RSP % 16 == 0. Т.е., в момент входа в функцию справедливо равенство RSP % 16 == 8, т.к. в момент вызова функции стек был выровнен по границе 16 байт, а на стек положили адрес возврата, размером 8 байт.

Характеристики соглашений о вызовах приведены в таблице 2.2

Таблица 2.2 Соглашения о вызовах для х86-64.

|  |  |  |
| --- | --- | --- |
| **Соглашение** | System V | Microsoft |
| **Изменяемые регистры** | rax, rcx, rdx, rsi, rdi, r8-r11, xmm0-15 | rax, rcx, rdx, r8-r11, xmm0-xmm5, |
| **Неизменяемые**  **регистры** | rbx, rsp, rbp, r12-15 | rbx, rsi, rdi, rsp, rbp, r12-r15, xmm6-xmm15 |
| **Передача целочисленных**  **аргументов** | rdi, rsi, rdx, rcx, r8, r9, далее на стеке | rcx, rdx, r8, r9+shadow space,  далее на стеке |
| **Передача вещественных**  **аргументов** | xmm0-xmm7,  далее на стеке (см. ниже) | xmm0-xmm3, далее на стеке (см. ниже) |
| **Указатель this** | Первый аргумент | |
| **Тривиально копируемая структура не более 8 байт** | В регистре или на стеке, согласно порядковому номеру аргумента | |
| **Тривиально копируемая структура не более 16 байт** | Если есть 2 свободных регистра – то в них. Иначе – см. Microsoft | Через временную копию, аргумент – указатель на копию |
| **Остальные структуры** | через временную копию, аргумент – указатель на копию | |
| **Очистка стека от аргументов** | Вызывающая функция | |
| **Возврат результата** | | |
| **char/short/int** | rax | |
| **Указатель** |
| **long long** |
| **float/double** | xmm0 | |
| **POD-структура не более 8 байт** | rax | |
| **POD-структура не более 16 байт** | младшая часть – rax/xmm0  старшая часть – rdx/xmm1 | в буфер на стеке,  указатель на буфер – второй после this аргумент |
| **Остальные структуры** | в буфер на стеке,  указатель на буфер – первый аргумент |

### Соглашение System V

Передача простых аргументов в соглашении System V описывается кратко: аргументы передаются в регистрах по порядку в таблице 2.2, если все регистры заполнены – на стеке.

Если функция имеет переменное число параметров (например, как printf/scanf), то дополнительно в RAX передается количество занятых вещественными аргументами XMM-регистров.

Пример вызова printf: <https://godbolt.org/z/G4sxxETe8>

Если структура имеет тривиальный конструктор перемещения/копирования и тривиальный деструктор (обычный конструктор может быть нетривиальным), то она может быть передана в регистрах. При этом:

* Если структура имеет размер 8 или менее байт, и имеет *только вещественные* поля, она передается в следующем свободном XMM-регистре.
* Если структура имеет размер 8 или менее байт, и имеет *целочисленные* или *целочисленные и вещественные* поля, она передается в следующем свободном регистре общего назначения.
* Если структура имеет размер от 9 до 16 байт, она разбивается на 2 части по 8 байт, каждая из которых передается, согласно правилу для 8-байтных структур.
* Если половина структуры поместилась, а половина - нет, она целиком передается через указатель на копию.
* Если структура не может быть передана в регистрах, то она копируется на стек, а в качестве аргумента передается указатель на копию.

Возврат структур происходит по тем же правилам (если возможно - в RAX/XMM0 и RDX/XMM1, иначе – как указатель на буфер возврата).

Примеры передачи структур: [httпps://godbolt.org/z/K59ojWK4d](https://godbolt.org/z/K59ojWK4d)

В System V ABI существует понятие **красной зоны -** области в 128 байт ниже текущей вершины стека. Данная зона, хотя и формально находится за пределами текущего стека, защищена от изменения обработчиками сигналов и прерываний (т.е. эта зона - «красная» для ОС, а для программиста – вполне «зеленая»).

*Если ваша функция не вызывает другие функции, то она* ***может*** *считать красную зону своим кадром стека.*При этом а) нет необходимости прибавлять/вычитать RSP, если вам достаточно 120 байт; б) нет необходимости устанавливать указатель кадра стека в RBP => экономятся инструкции пролога и эпилога.

Пример использования красной зоны: <https://godbolt.org/z/sdvnrnzo4>

### Соглашение Microsoft

Передача аргументов в соглашении Microsoft происходит менее эффективно, по сравнению с соглашением System V. В регистрах передается 4 первых аргумента: целочисленные аргументы передаются, по порядку, в RCX, RDX, R8, R9; вещественные – в XMM0-XMM3. Хотя используется 8 регистров, передать можно только 4 аргумента. Например, если первый аргумент – int, а второй – float, то первый аргумент будет передан в RCX, а второй – в XMM1, хотя XMM0 не занят.

Пятый и далее аргументы передаются на стеке. При этом для первых 4 аргументов резервируется теневое пространство (**shadow space**) размером 32 байта (по 8 байт на аргумент). Остальные аргументы идут после теневого пространства. Теневая зона используется для сохранения первых 4 аргументов, если возникает необходимость освободить регистр. Кроме того, аргументы сбрасываются в теневую зону при сборке в Debug-конфигурации для облегчения работы отладчика. *Теневое пространство резервируется вызывающей функцией, даже если аргументов у вызываемой функции нет.*

Если функция имеет переменное число параметров (например, как printf/scanf), то значение вещественных параметров в XMM-регистрах должно дублироваться в целочисленных регистрах без приведения типов (например, инструкциями MOVD/MOVQ). Например, если занят регистр XMM1, то в точности то же значение должно находиться в RDX.

Пример вызова printf: <https://godbolt.org/z/cx64vv1PG>

Правила передачи структур более простые и менее эффективные, по сравнению с System V. Если структура имеет тривиальный конструктор перемещения/копирования и тривиальный деструктор (обычный конструктор может быть нетривиальным) и имеет размер 8 или менее байт, она передается в регистре по значению. Иначе структура копируется на стек, а качестве аргумента передается указатель на копию.

Пример передачи структур: <https://godbolt.org/z/E6xnhY84r>

Возврат структур происходит по тому же правилу: в регистре RAX либо через указатель на буфер возврата.

# Компиляция, дизассемблирование и декомпиляция

## Общая схема

В языках С/С++ основной единицей трансляции является файл исходного кода, обычно имеющий расширение .c/.cpp/.cxx. Каждый файл исходного кода содержит определение некоторых символов – глобальных переменных или функций. Заголовочные файлы, по общему правилу, должны содержать только объявления символов. Результатом компиляции является набор объектных файлов.

По умолчанию все символы в пределах единицы трансляции являются глобальными, т.е. видимыми из других единиц трансляции (в других языках это может быть не так). Неглобальными (недоступными извне модуля) являются только символы, определенные с модификатором static. Все символы заносятся в таблицу символов результирующего объектного файла в ходе компиляции (рисунок 3.1).

Если в пределах единицы трансляции символ был объявлен, но не определен, то он помечается, как не имеющий определенного значения (**внешний символ**). Значение такого символа должно быть определено на этапе компоновки.

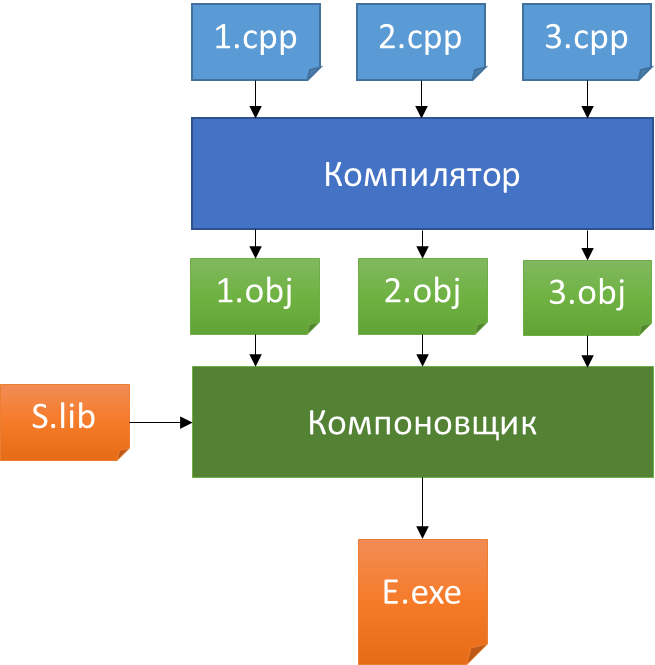


Рисунок 3.1 – Общая схема создания исполняемого файла

*Если целью компиляции является исполняемый файл или динамическая библиотека*, то после компиляции наступает этап статической компоновки. На этапе статической компоновки можно указать дополнительные зависимости сборки – статические и динамические библиотеки. Таблицы символов всех объектных файлов и библиотек объединяются. Если в общей таблице возникают дубликаты с разными значениями, то компоновка завершается с ошибкой, т.к. в программе есть два одинаковых символа. Если хотя бы одна запись в таблице символов остается незаполненной, и при этом отсутствует в таблице динамического импорта, то компоновка завершается с ошибкой.

*Если итоговой целью компиляции является статическая библиотека* (.lib/.a), то набор объектных файлов просто объединяется в архив вместе с некоторой дополнительной информацией. Т.е. статическую библиотеку можно легко открыть архиватором.

Особым случаем является ситуация, при которой требуемый символ находится в динамической библиотеке (.dll/.so). В этом случае символ считается найденным и помещается в *таблицу динамического импорта исполняемого файла* вместе с именем динамической библиотеки, в которой он находится. Во время загрузки исполняемого файла на этапе динамической компоновки операционная система загружает также все динамические библиотеки, упомянутые в таблице импорта, находит в ней адреса импортируемых функций и переменных, и далее программа использует уже эти адреса.

## Статические библиотеки как зависимость при компоновке

### В CMake

В CMake за добавление внешней статической библиотеки в качестве зависимости при компоновке отвечает директива [target\_link\_libraries](https://cmake.org/cmake/help/latest/command/target_link_libraries.html).

Если мы собираем проект с именем Lab3, и наша библиотека находится по пути “C:\PathToLib\my.lib”, то итоговая директива будет иметь вид

target\_link\_libraries(Lab3 PUBLIC “C:\PathToLib\my.lib”)

### В SASM

В SASM при сборке проекта за компоновку отвечает программа, указанная в поле «Компоновщик». Компоновщику передается на вход несколько объектных файлов.

Добавить библиотеку для компоновки можно с помощью аргумента *-l:путь\_до\_библиотеки*. Следите за лишними пробелами.

*Файлы, предоставляемые для выполнения лабораторной работы, скомпилированы с помощью MinGW и не вызовут проблем*. Файлы .lib, создаваемые компилятором MSVC могут вызывать проблемы при компоновке, в этом случае приходится извлекать объектный файл и передавать путь к нему без ключа *-l*.

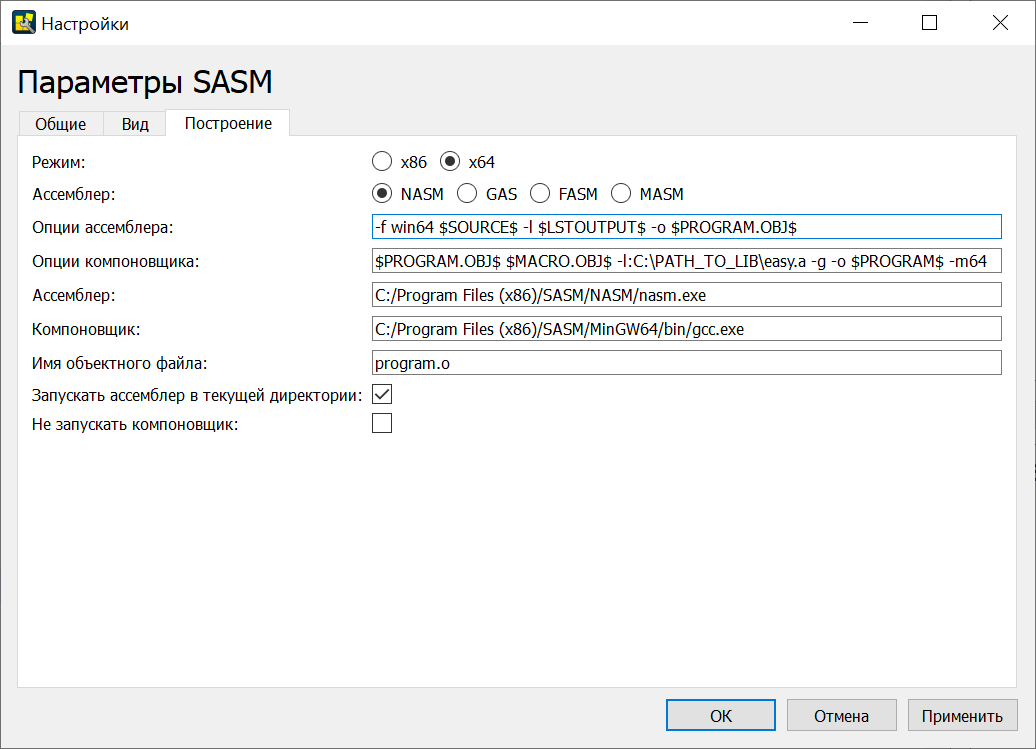


Рисунок 3.2 – Аргументы компоновщика

## Символы в NASM

### Глобальные символы

Любая метка в NASM является символом. Однако по умолчанию ни один символ не является глобальным. Для того, чтобы символ был виден при компоновке, требуется явно указать его, как global. Наиболее частый пример – функция main, которая обязана быть глобальной. Поэтому в примерах всегда присутствует строчка:

global main

Помните, что нет различий между символами-переменными и символами-функциями.

### Внешние символы

Для того, чтобы указать, что данный символ является внешним, в языке NASM используется ключевое слово extern. Например, объявить printf, как внешний символ можно строкой

extern printf

Однако здесь в силу вступает различие между искажением имен для x86 и его отсутствием в x86-64. Поэтому в набор макросов SASM включен макрос CEXTERN, который учитывает данный момент и автоматически подстраивается.

CEXTERN printf ; \_printf on x86, printf on x86-64

### Локальные метки

Поскольку метка является символом, ее имя должно быть уникальным. Это может представлять неудобства. Рассмотрим следующий код:

int foo(int x){

    for(int i = 0; i < 10; ++i)

        x+=x;

    return x;

}

int bar(int x){

    for(int i = 0; i < 10; ++i)

        x+=x;

    return x;

}

Данный код может быть реализован в NASM, как:

foo:

mov eax, edi

mov edx, 10

**foo\_loop:**

add eax, eax

sub edx, 1

jne foo\_loop

ret

bar:

mov eax, edi

mov edx, 10

**bar\_loop:**

add eax, eax

sub edx, 1

jne bar\_loop

ret

Из-за того, что каждая метка должна быть уникальной, метки начала циклов тоже должны отличаться. Можно вручную писать уникальны имена, как в примере выше, а можно использовать локальные метки.

Локальная метка – специальным метка, имя которой начинается с точки. Пример: .local\_label. Локальная метка связана с предыдущей нелокальной меткой.

Используя локальные метки, пример выше можно переписать, как

foo:

mov eax, edi

mov edx, 10

**.loop: ;<------------------**

add eax, eax ;**|**

sub edx, 1 ;**|**

jne **.loop ;^**

ret

bar:

mov eax, edi

mov edx, 10

**.loop: ;<------------------**

add eax, eax ;**|**

sub edx, 1 ;**|**

jne **.loop ;^**

ret

При этом первая локальная метка связана с меткой foo и имеет полное имя foo.loop. Вторая локальная метка связана с меткой bar и имеет полное имя bar.loop.

## Искажение имен (mangling)

### Искажение имен в C

По историческим причинам, имя символа-функции может быть искажено в зависимости от используемого соглашения о вызовах.

Компилятор Microsoft применяет следующие правила иска

Символы-функции, использующие соглашение **cdecl**, предваряются префиксом \_. То есть, функции void foo() будет соответствовать символ \_foo.

Символы-функции, использующие соглашение **stdcall**, предваряются префиксом \_ и дополняются суффиксом @, за которым следует общий размер фргументов. То есть, функции void foo(int x, float y) будет соответствовать символ \_foo@12.

В х86-64 искажение имен в С не используется.

### Искажение имен в С++

Поскольку в языке С отсутствовали перегрузки функций, имя функции (а значит и символ) всегда были уникальны.

В языке С++ с появлением перегрузок функций возникла проблема: нужно было как-то назначить разным перегрузкам функций разные символы. Сделано это было с помощью искажения имен.

В С++, каждый компилятор имеет свою схему искажения имен, но общая схема совпадает. К началу любого символа (в т.ч переменной) прибавляется префикс, содержащий информацию о пространстве имен и классе, с которым связан символ. Если символ является функцией, то для символа добавляется суффикс, содержащий информацию о аргументах. Отменить искажение имен можно ключевой комбинацией extern “C”.

|  |  |  |
| --- | --- | --- |
| Функция | Символ (MSVC x86) | GCC |
| void f(int x, float y) | ?f@@YAHHM@Z | \_Z3fif |
| extern “C”  void f(int x, float y) | \_f | f |

Все современные средства отладки и анализа кода обращают искажение имени (demangling) автоматически.

Для того, чтобы провести или обратить искажение имени, можно использовать и отдельные инструменты, например *undname* для Windows, *с++filt* для Linux.

## Получение объектного файла из статической библиотеки

Статическая библиотека (.lib/.a) является не более, чем архивом, включающим в себя объектные файлы и метаданные. Как следствие, данный архив легко открыть любым распространенным архиватором, в т.ч. WinRAR или [7-Zip](https://www.7-zip.org). К примеру, файл библиотеки для легкого уровня в 7-Zip выглядит следующим образом:

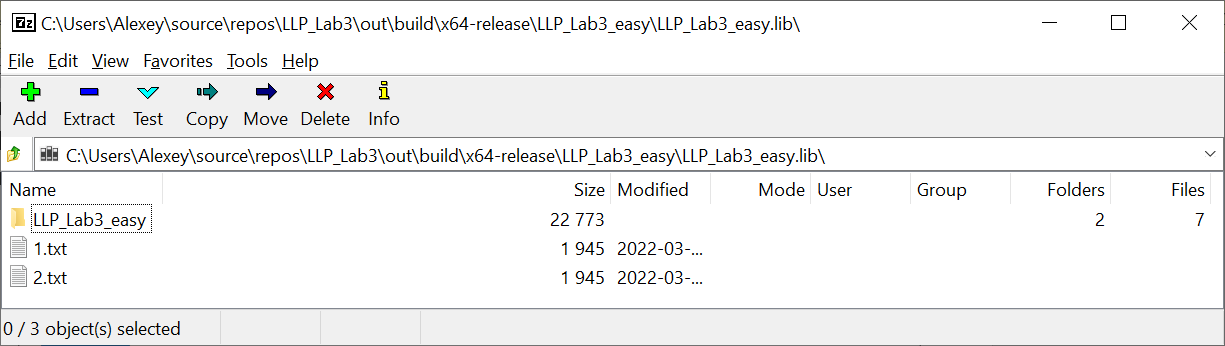


Рисунок 3.3 – Файл статической библиотеке в 7-Zip

Интересующий объектный файл может быть извлечен и подвергнут дизассемблированию.

# Задание на лабораторную работу

## Общие замечания

В данной лабораторной работе предполагается использование специализированного ПО для анализа кода исполняемых файлов. Итоговый выбор ПО и метода анализа исполняемого файла не ограничивается –анализировать файл можно как угодно и чем угодно, главное - чтобы поставленная задача была решена.

## Задание 1

В программе, полученной в ходе выполнения л/р 1 или л/р 2 (по вашему выбору), заменить все макросы ввода-вывода SASM на вызовы функций printf/scanf, все переменные, не являющиеся массивами – на локальные переменные в стеке/регистрах. Если ввод-вывод с консоли не был реализован, его следует добавить. Для среднего и сложного уровня запрещается использовать буферы в .data/.bss, память должна выделяться динамически через malloc и освобождаться через free. Помните про макрос CEXTERN.

*Программу нужно написать в 2 вариантах - для архитектуры х86-32 и для х86-64.*

## Задание 2

Дана статическая библиотека, содержащая некоторый метод проверки доступа. Вам необходимо:

1. реализовать программу, проходящую данную проверку (добиться вывода Access granted в результате вызова этой функции);
2. восстановить примерный код взломанной функции на языке С (в т.ч. состав полей структур/объектов для среднего/сложного уровня).

*Для сдачи лабораторной необходимо выполнить выбранный уровень и задачи с уровня ниже (т.е., средний+легкий или сложный+средний).*

Для анализа и отладки программы предлагается использовать [IDA Free](https://hex-rays.com/ida-free/) ([альтернативная ссылка](https://disk.yandex.ru/d/YUoCTCBkwRfo2g)) Аналогами являются открытый проект radare2 с интерфейсом [iaito](https://github.com/radareorg/iaito) и открытый проект Ghidra от АНБ (отсутствует отладчик, пригоден для анализа). В самом простом варианте можно просто дизассемблировать объектный файл с помощью [онлайн-дизассемблера](https://onlinedisassembler.com) (объектный файл придется извлечь из библиотеки) или напрямую с помощью objdump из каталога *<SASM\_DIR>/MinGW64/bin* командой *objdump.exe -M intel,x86-64 -rdF -j .text -j .rdata --no-show-raw-insn FILE > OUT\_FILE* .

К сожалению, бесплатная IDA Free из-за ограничений может анализировать только исполняемые файлы, но не статические библиотеки или объектные файлы. Поэтому перед анализом функций необходимо создать исполняемый файл.

Порядок выполнения:

1. Открыть в архиваторе файл библиотеки, найти в текстовых файлах список экспортируемых символов, выбрать вашу функцию.
2. В SASM указать библиотеку как зависимость, указать вашу функцию, как CEXTERN <ваш символ> после global CMAIN.
3. В SASM вызвать вашу функцию. Программа не будет работать, но в исполняемый файл данная функция будет включена.
4. Сохранить исполняемый файл (Файл -> Сохранить .exe)
5. Открыть .EXE в IDA/iaito/Ghidra.
6. Проанализировать функцию, определить ее параметры, а также условия, при которых проверка будет пройдена (в IDA/iaito можно воспользоваться отладчиком).
7. Если необходимо – исправить программу в SASM, goto 3.

**Легкий уровень:**

Функция проверки имеет вид:

extern "C" void access**N**(T1 a, T2 b, T3 c, T4 d);

где N – номер варианта, T1-T4 – некоторые простые числовые типы (не указатели).

В случае необходимости по запросу может быть предоставлен ассемблерный код функции. Ассемблерный код выдается по запросу студента.

**Средний уровень:**

Код варианта имеет следующий прототип:

namespace var**N** {

        struct S {

          /\*\*/

        };

        extern "C" void access**N**(T1 a, T2 b, T3 c );

}

где N – номер варианта; T1, T2, T3 – фундаментальные типы, S или const S&; поля структуры S являются числами или массивами чисел, но не указателями.

**Сложный уровень:**

Код варианта имеет следующий прототип:

namespace var**N** {

        struct S {

            /\*\*/

        };

        class C {

            /\*\*/

        };

extern "C" void access**N**(T1 , T2);

}

Где N – номер варианта, T1 и T2 – S, const S&, С или const С&.