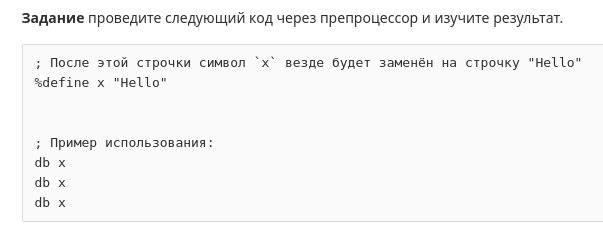
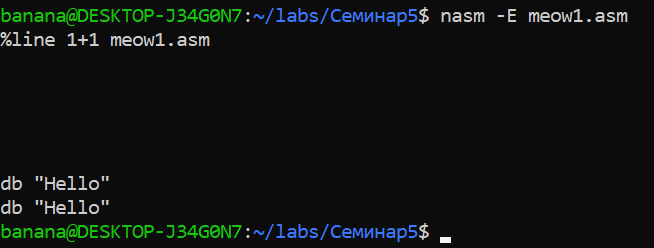
На этом этапе изучаем примеры с помощью утилиты nasm и ключа -E

“nasm -E programm.nasm”

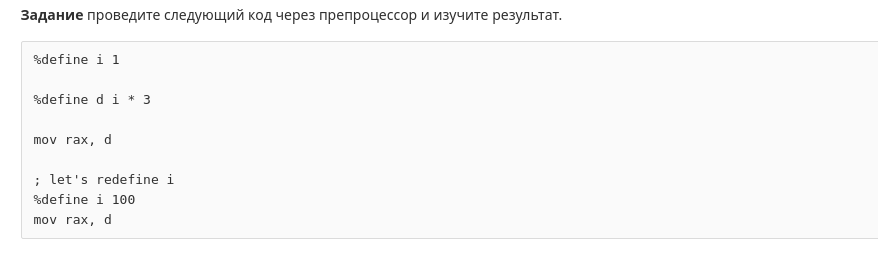


проверяем:

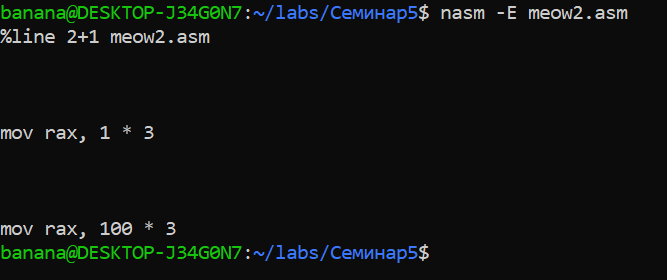


изучаем:

Действительно, после препроцессинга на место каждого x подставилось “Hello”.



проверяем:



изучаем:

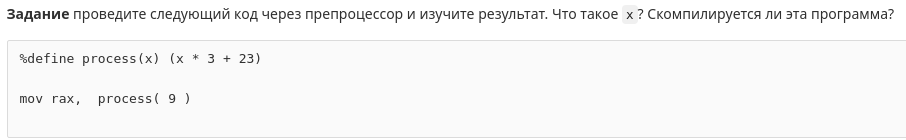
На этапе препроцессинга сначала на место i поставилась 1,

далее определилась d = i \* 3 = 1 \* 3 Это значение d и кинули в rax. Все логично.

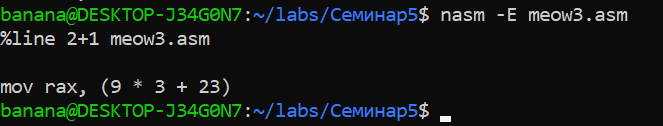
Потом мы видим, что i переопределили как i = 100,

соответственно поменялась и d = i \* 3 = 100 \* 3

То есть видим, что значение будет записано последнее из полученных.



проверяем:

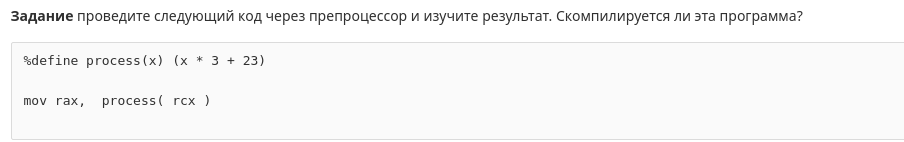


изучаем:

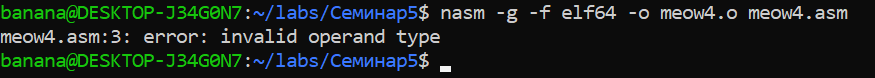
Тут с помощью директивы %define определили макрос-функцию process(x)

Когда в коде будет встречаться конструкция вида “process(x)” будет выполняться подстановка x \* 3 + 23

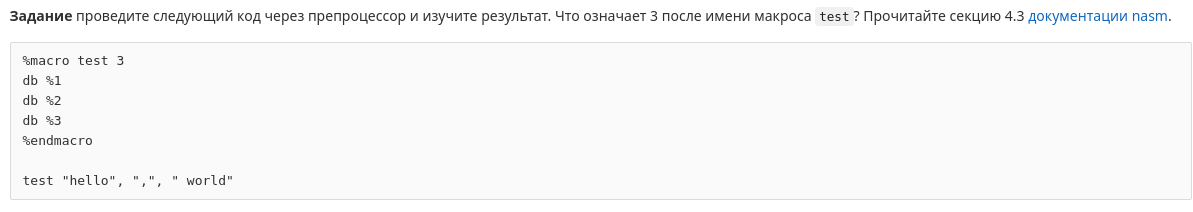
Программа смогла скомпилировалась и в регистре rax ожидаемо оказалось значение 50



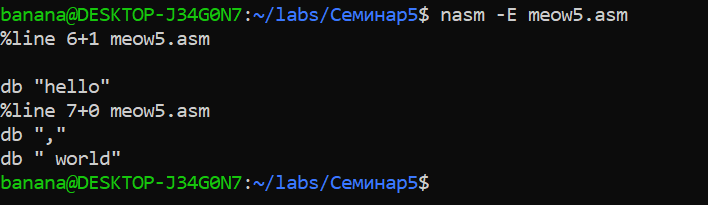
Суть программы та же, только в качестве аргумента передается регистр rcx



Программа не скомпилировалась.. Ожидалось, что мы подставим число.



проверяем:



изучаем:

Тут мы имеем дело с многострочным макросом. По сути, та же макрос-функция, но записана в несколько строк.

Число 3 тут означает кол-во передаваемых параметров.

В самом макросе, чтобы взаимодействовать с параметрами, необходимо указывать их порядковый номер (начиная с единицы) после %. Например, %1 %2 … %n

В целом многострочная макрос-функция имеет синтаксис:

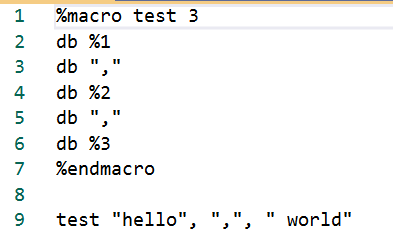
%macro имя\_функции кол\_во\_параметров

…

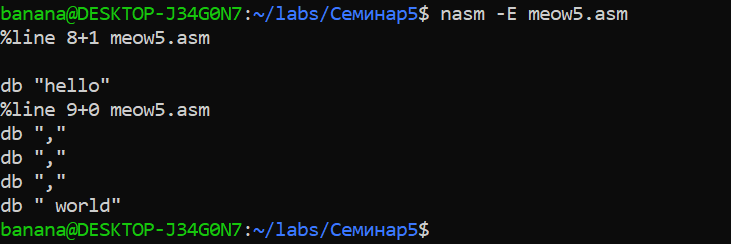
…

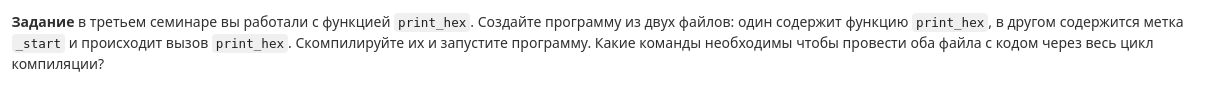
%endmacro





Убеждаемся, что все работает:





**файлы:**

**print\_hex.asm**

**start.asm**

**вопрос**: как провести оба файла через цикл компиляции, чтоб получить один исполняемый файл?

**ответ**:

1. сначала транслируем файлы с ассемблерным кодом в объектные файлы по отдельности с помощью команд

**nasm -g -f elf64 -o print\_hex.o print\_hex.asm**

**nasm -g -f elf64 -o start.o start.asm**

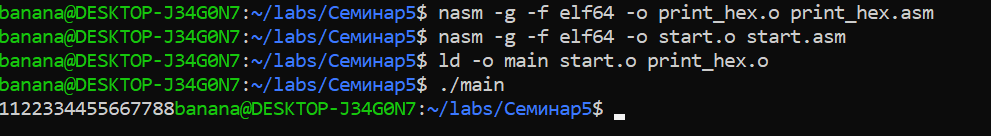
2)Таким образом мы получим два объектных файла: start.o и print\_hex.o, которые нам необходимо скомпоновать в один исполняемый файл

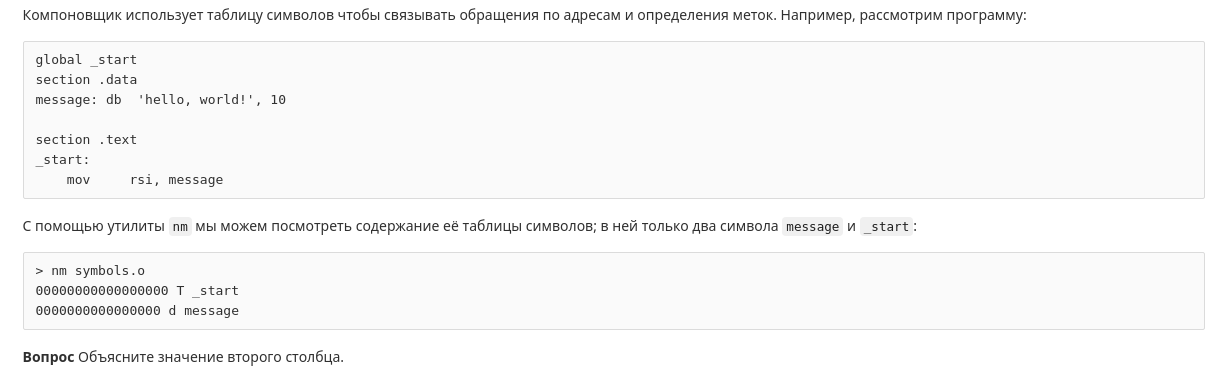
делаем это так:

**ld -o main start.o print\_hex.o**

И получаем готовый исполняемый файл main

3)Проверяем что все работает





Второй столбец хранит информацию о свойствах меток

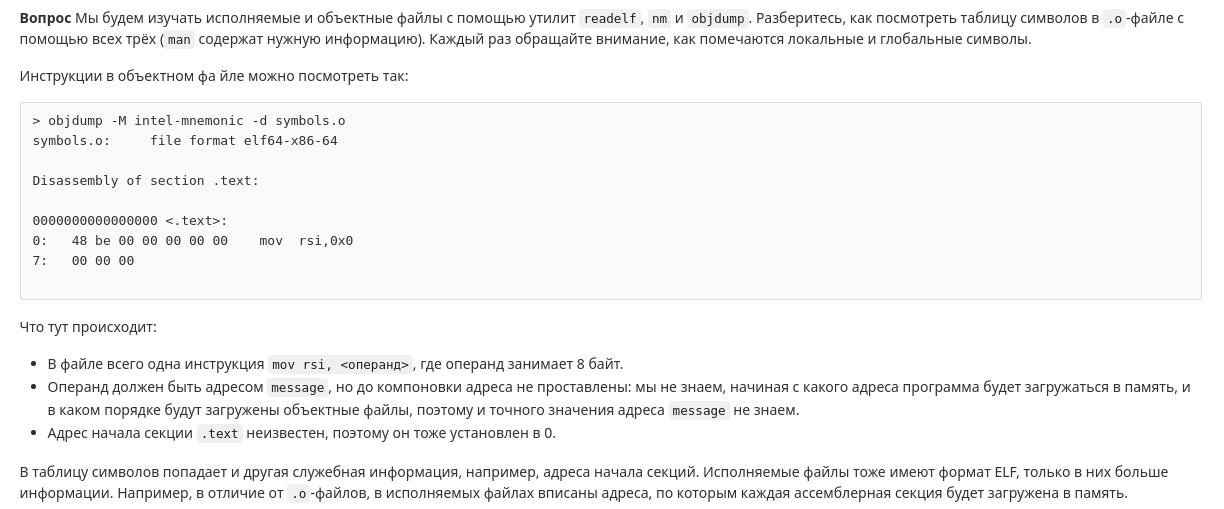
Например,

символ T означает, что метка \_start определена в секции .text

символ d означает, что метка message определена в секции .data

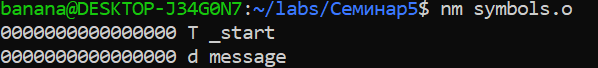
Если взглянуть на код файла, то можно убедиться в правильности этой информации

Нам нужно создать файл symbols.asm, скопировать в него код, и транслировать его в объектный файл symbols.o, чтоб дальше самим с ним экспериментировать



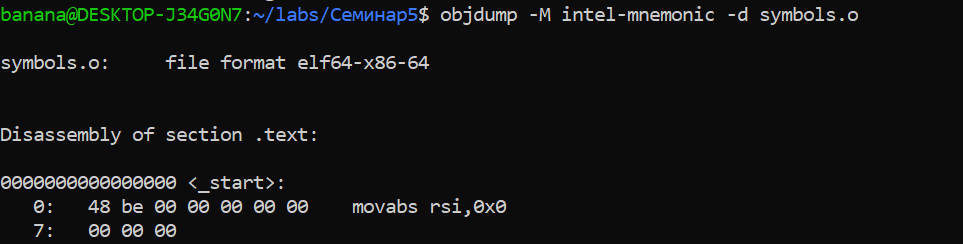
Изучим утилиты nm, objdump, readelf

Утилита nm самая простая и показывает меньше всего информации. Она удобна, если нужно посмотреть только на таблицу символов. (nm symbols.o).



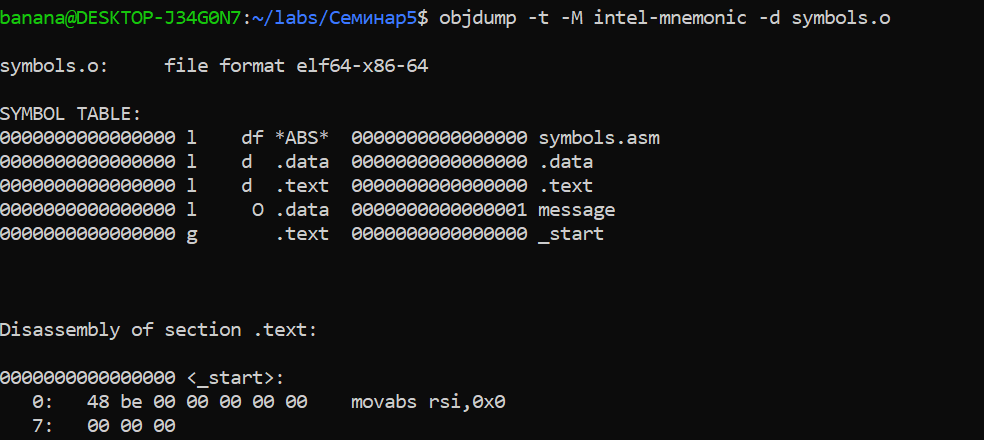
z

Посмотрим на утилиту objdump. Это основной инструмент для изучения объектных файлов, который показывает всю основную информацию о файле. Важно, что утилита может работать не только с объектными файлами типа ELF, но и с другими форматами. (objdump -M intel-mnemonic -d symbols.o).



Тут мы можем посмотреть даже дизассемблирование разделов (по сути, это дает нам возможность посмотреть на соответствие ассемблерных команд и их представлений в машинном коде).

Используя различные ключи с этой утилитой, мы можем более детально изучить объектные файлы.

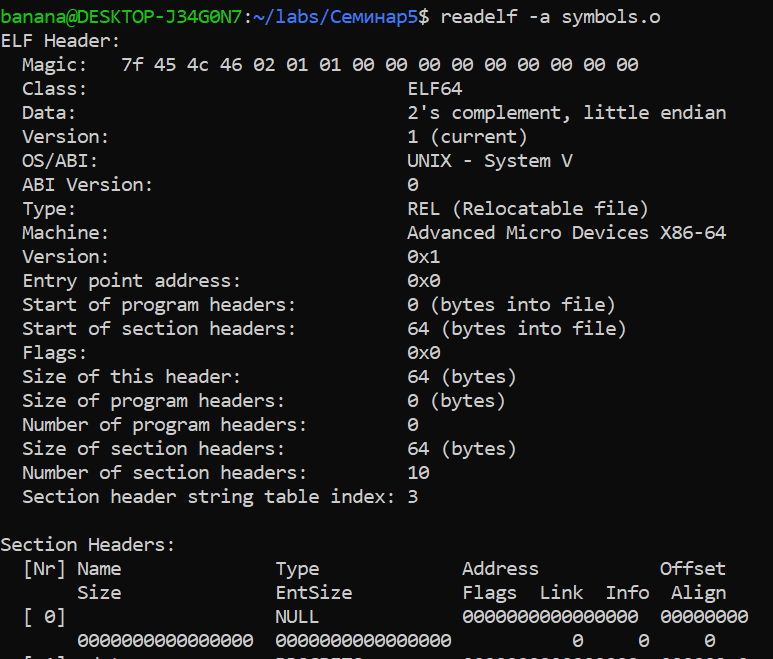


Например, добавив ключик -t мы можем посмотреть таблицу меток, с еще большим количеством информации о метках. (objdump -t -M intel-mnemonic -d symbols.o).

Перейдем к последней утилите readelf. Она создана для изучения объектных файлов только типа ELF. Т.е. если мы знаем, что какой-либо объектный файл точно относится к типу ELF, то этот инструмент будет наиболее предпочтительным, так как позволит получить всю необходимую информацию о нем.

*оффтоп: все объектные файлы что мы по настоящий момент получали, мы транслировали в ELF, так как мы всегда явно задавали параметр, указывающий на тип объектного файла (-f elf64) в утилите nasm.*

В общем, написав **readelf -a имя\_объектного\_файла,** мы получим всю информацию о нем



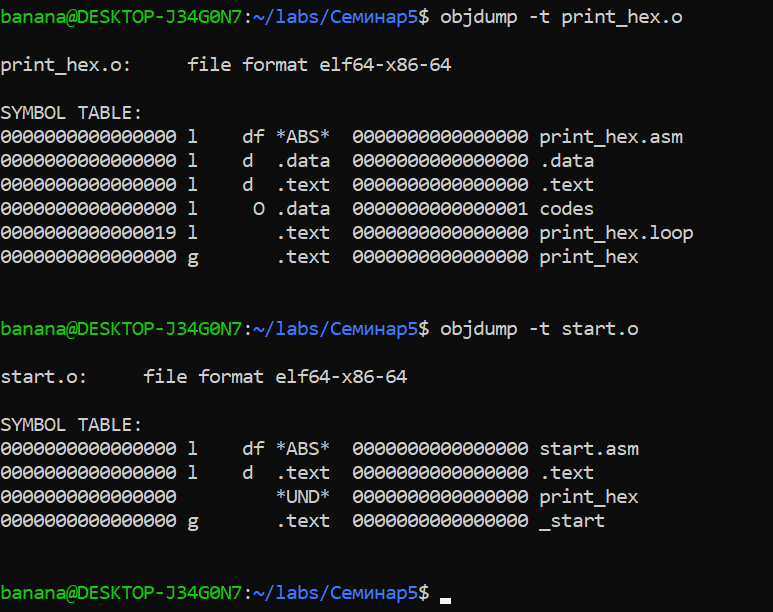
…и так далее, тут оч много инфы :>

(readelf -a symbols.o).

Нет никакой необходимости учить все опции этих (да и вообще всех) утилит

Их предназначение всегда можно посмотреть с помощью man имя\_утилиты )





Можем заметить, что во втором столбце глобальные метки (т.е. помеченные с помощью global в самих файлах) отмечены буквой **g**. Локальные метки напротив отмечены буквой **l**

У метки print\_hex, что мы импортировали в файле start (с помощью extern) во втором столбце вообще ничего не указано, что логично, так как она не является ни локальной, ни глобальной. В третьем столбце у этой импортированной метки, есть строчка \*UND\*, которая означает, что эта метка не определена ни в какой из секций в файле start (все так, потому, используя extern мы просто ссылаемся на метку из другого файла, а не копируем к себе).



Метка \_start является точкой входа программы, так что мы обязаны сделать ее глобальной, чтоб компоновщик понимал откуда начинается программа.

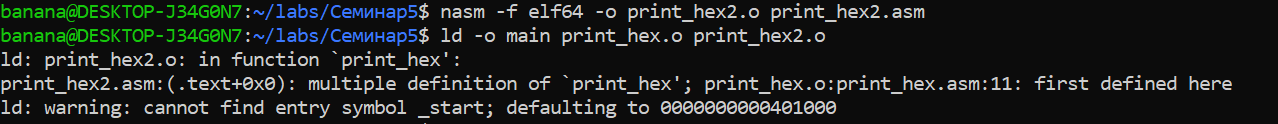
Он использует некоторый адрес по умолчанию, так что программа может и запустится, но ругаться компоновщик будет и не зря, так как в другой ситуации адрес по умолчанию может и не спасти, тогда программа, конечно, не запустится.





Не могут! Ибо возникнет ошибка в неоднозначности файлов(функций).

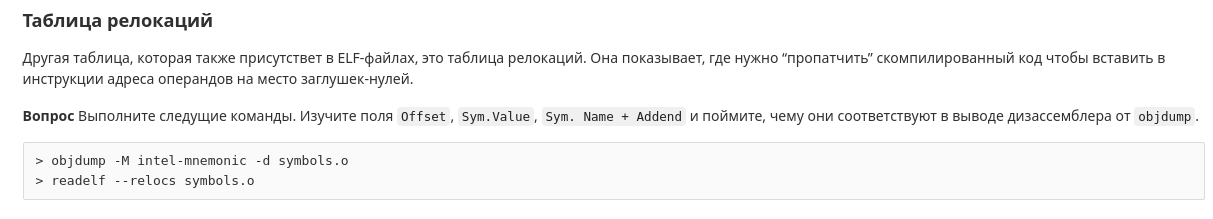
Для проверки я сделала копию print\_hex как print\_hex2 и запустила файлы вместе:



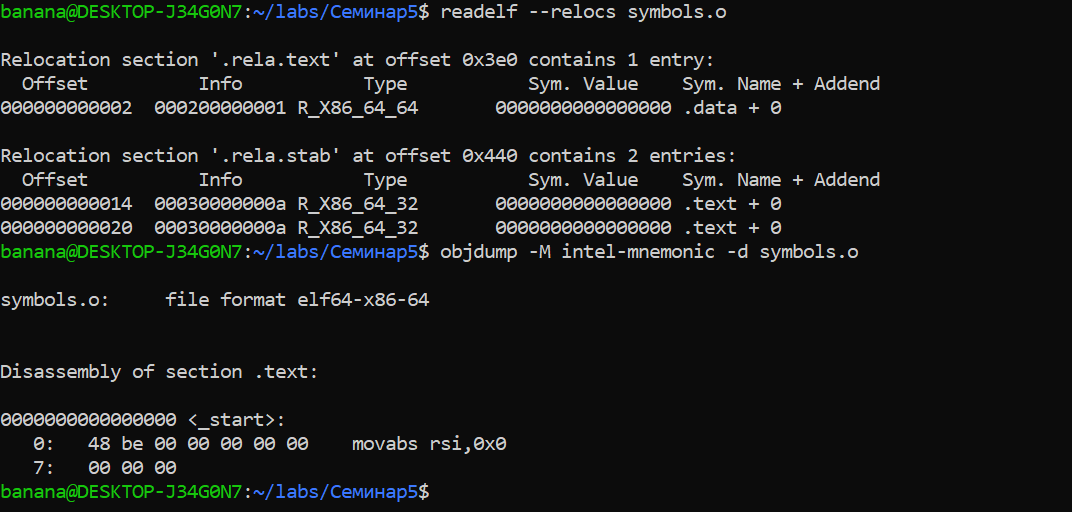
Однако, если не делать оба файла глобальными и не делать 2 точки старта (ибо они будут видны везде и компоновщик увидит 2 одинаковые метки и заругается), а локально можно делать одинаковые названия,

* если не локальные, то могут.

Таблица релоксаций патчит значения адресов при загрузке программы в память.



посмотрим, что нам покажут эти утилиты:



Видим, что тут информация о двух перемещениях(релокациях), один из которых необходим меткам отладчика. Первая секция (текст) – то, что происходит в секции текст и только это интересует нас. Информация об этом находится в дисассемблере.

А вторая секция – информация добавочная об окончании файла\метинформация о заголовках.

В целом таблица релокаций нужна, чтобы понимать, где в памяти находятся инструкции и их операнды (относительно секций), чтоб далее их верно сопоставить.

Вот пример: mov rdi, message

тут mov инструкция, а message операнд

Нас интересуют колонки:

Offset (Смещение)

Sym.Value (Знач.симв.)

Sym.Name + Addend (Имя симв. + Addend)

Значение Offset - относительный адрес инструкции. В дизассемблированном коде первое число до двоеточия соответствует значению offset – 2. (Указывает на смещение от начала программы, т. е. куда нужно будет подгрузить адрес, которого ещё нет.).

Info – содержит данные о флагах (состояние процессора).

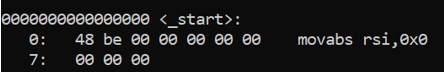
Type – нужен для определения типов в подстановке, например, для задания call используется PC32, которая показывает, что надо будет выделить 4 байта под ОТНОСИТЕЛЬНЫЙ АДРЕС при вызове функции.

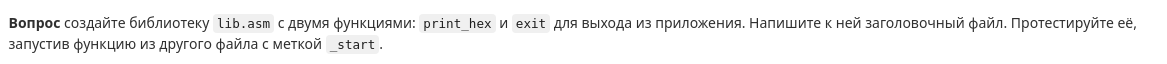
Sym.Value судя по-всему, соответствует второй строчке дизассемблированного кода.

(Характеризует нужный адрес на данный момент, которого, безусловно, нет, ведь ещё программа не загружена в память.

Sym.Name + Addend говорит об адресе операнда относительного его секции.

Этот адрес будет подставлен на место заглушки-нуля





Тут все просто. Немного редактируем те два файла print\_hex.asm и start.asm, ну и переименуем print\_hex.asm в lib.asm, а start.asm в start2.asm

проверяем работоспособность **lib.asm** и **start2.asm**

