**Seminar – 9**

# Calling assembly code from C (25 minutes)

The file string.asm contains functions print\_string and world:

**Question** What does the function world do? Explain how it works.

The file hello.c contains a function main that calls print\_string and outputs “hello” with it in this way:

The function main also calls world.

**Answer:** The world function consists of a single line mov rdi, message. The usual transfer to the register of the address where the string is stored. But there is no return at the end, which means that after this transfer, print\_string with the message argument will be immediately called. That is, as a result, the contents of the message will be displayed on the screen.

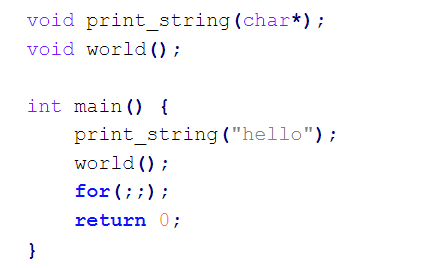
**Question** These files are missing a few lines so that you can interact with each other's code. Append files so that the functions print\_string and hello the caller and check the result. Hint: remember what it takes to call code from another file from one C code file.

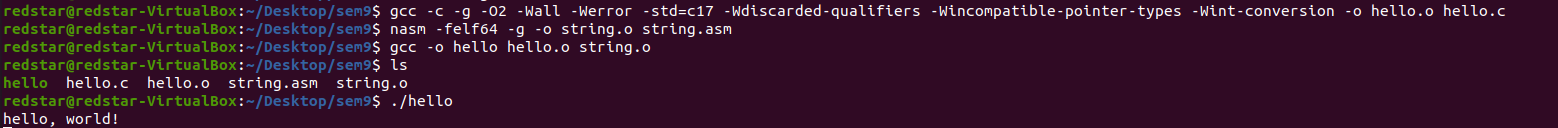
**Answer:**

First, we need to make the assembler functions global.

global print\_string

global world

Secondly, we will add declarations of called functions to the .c file. At the linking stage, the definitions of these functions will be loaded from the string.o object file



**Question** Please study Makefileto build this program (see directory contents task). What is the meaning $@, $^?

**Answer:**

$@ substitutes the target name. If there are several targets, then the one that caused the script to run is substituted.

$^ substitutes the names of all prerequisites separated by a space.

**Question** What are sections .rodata and .bss?

**Answer:**

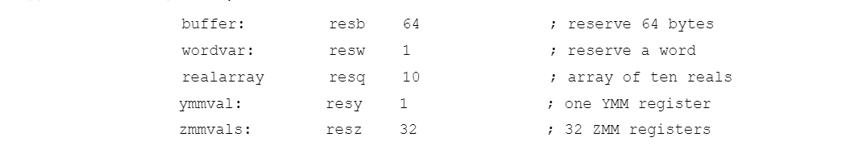
.rodata = read-only data - usually constant data is stored there

.bss = block starting symbol - this is a section of code where statically allocated variables are stored (i.e. those that are allocated at the compilation stage and live the entire runtime) that have been declared, but have not yet been defined.

**Question** What do nasm directives resb, resq and other [fn: 1]?

**Answer:**

RESB, RESW, RESD, RESQ, REST, RESO, RESY и RESZ needed for use in .bss sections. With the help of them, an uninitialized place in memory is allocated.



buffer:         res

RESB, RESW, RESD, RESQ and REST are designed to be used in the BSS section of a module: they declare uninitialised storage space. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve. As stated in **section 2.2.7**, NASM does not support the MASM/TASM syntax of reserving uninitialised space by writing DW ? or similar things: this is what it does instead. The operand to a RESB-type pseudo-instruction is a critical expression: see **section 3.7**.

**Question** In the file hello.c, in which section will the line = ”hello” = be highlighted?

**Answer:**

Output the contents of the sections using the objdump -s hello.o commandИзображение выглядит как текст

Автоматически созданное описание

The string is located in the .rodata section, which is logical, because it is a constant hardcoded (literal) value that we passed to the print\_string function parameter.

**ELF files (25 minutes)**

In the previous step, during the compilation process, we received the files:

* string.o from an assembly file string.asm;
* hello.o from a file with C code hello.c;
* hello, executable file.

Let's examine these files in more detail using readelf. To facilitate the research process, record all results.

The ELF file has three headers:

File header

main header, contains general information about the file and links to the other two headers.

Section header

a list of sections (the same ones that are in assembler, and many others for service purposes)

Program header

list of *segments* . Each segment describes a memory region into which one or more sections will be loaded.

**Question** Please enter readelfno arguments. Read the output and determine what keys are needed to display the three file headers.

**Answer:**

Изображение выглядит как текст

Автоматически созданное описание

**Question** Print the file header for the file hello.o. What is Entry point address and why is it 0?

**Answer:**

Entry point address is the address of the entry point to the program, that is, the address (in our case virtual), starting from which instructions will be executed after the program is started. Apparently, this is the address of the \_start label. Since we are studying the object file (that is, the non-linked part of the program), there are stubs in the places of the addresses, which explains the value 0. For the sake of interest, let's run the same command for the assembled hello executable file and make sure that the value of the Entry point address field is different from zero:Изображение выглядит как текст

Автоматически созданное описание

**Question** Output the program header for the file hello.o. Explain the result. 

Program header is an array of structures describing program segments. It includes such information as, for example, the indentation in bytes from the beginning of the file to the beginning of the segment, the virtual address at which the first byte of the segment is located, and much more. Such a header makes sense only for executable files, because object files have stubs instead of specific addresses. Therefore, the program header is missing from the object files.

**Question** Determine the download addresses of sections .text and .rodata.

Enter the readelf -S hello command. We get a table, the second column of which is the name of the section, the fourth is the address. We find the necessary addresses.



**Question** Output the program header for the file hello. What segments do .textand fall into .rodata? What are the download addresses of these segments?

.rodata is the beginning of the segment with the number 04, .text lies in the segment 03

Now we will output section-headers using the -S key:



.rodata - the beginning of segment 04, the address is specified in the third column

Segment 03 starts with the .init section:1

# File upload (20 minutes)

Modify your C program so that it goes into an infinite loop. This will allow us to test our guesses about the section and segment addresses.

**Question** Display the memory regions map for the running program we are working with today. Map segments containing sections .rodata and .text memory regions. Is it true that only one segment corresponds to one region of memory?

**Answer:**

Let's run our program in the background and display a map of the regions:

Next, we see the process number, go to this address in the /proc/ID/ directory)

And output the maps file:Изображение выглядит как текст

Автоматически созданное описание

Let's compare it with the section headings and by the method of a close look we will understand that, apparently, this does not always happen (readelf -S hello).

Изображение выглядит как текст

Автоматически созданное описание

the first section of the next segment starts with a new region.

**Question** What regions correspond to parts of dynamic libraries? Dynamic link libraries in Linux have the extension \*.so.

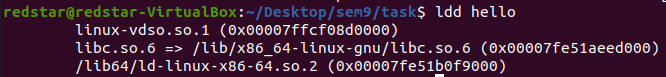
**Answer:**

Изображение выглядит как текст

Автоматически созданное описание

**Question** Execute ldd hello. Explain the result.

**Answer:**



By entering man ldd, we make sure that this command outputs all object dynamic dependencies (libraries).

The first line is linux-vdso.so.1 is not a library, but a Linux mechanism in the form of a library that allows you to reduce the number of context switches to the kernel by transferring system calls to the user context.

The second line is the standard C language library

The third line is a dynamic linker (it links dynamic libraries with the program during runtime).

The address where the corresponding library is loaded is written in parentheses.**Question** Why do you think the C standard library is implemented as a dynamic library, and not included in the executable file statically, like regular .ofiles?

**Answer:**

If the standard C library was implemented as a static one, then we would need to rebuild every single exe file with its own COPY of this library, which is not very good. Dynamic libraries are also called shared, indicating that many programs can connect to the same physical library at the same time.

**Seminar – 10**

**Question** What will be displayed in stdout? If this is an unexpected result, how do you rewrite it dblto get rid of the unwanted and unexpected behavior?

**Answer:**

the value 9 is output (as if one 3 is taken and multiplied by two \*2 before another 3 is added to it). But it should output (3+3)\*2=12.

Let’s fix it:

A small change:

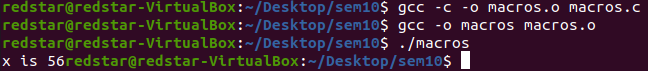
#define dbl(y) y \* 2   -> #define dbl(y) ***(****y****)*** \* 2

**Question** Write a program that will use this macro to output a variable. Test it with the -Egcc / clang switch .

**Answer:**

Изображение выглядит как текст

Автоматически созданное описание



**Question** What will happen if you write print\_var(42)?

**Answer:**

Изображение выглядит как текст

Автоматически созданное описание



**Question** Test this program using the key -E for gcc / clang.

**Answer:** gcc -E print.c

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст, устройство

Автоматически созданное описание

**Question** Test this program using the key -E for gcc / clang. Which is revealed earlier: \_Genericor #define?

**Answer:** gcc -E generic.c

Изображение выглядит как текст

Автоматически созданное описание