Семинар 1

- компиляция, gdb, gcc, sanitizers, strace

Вводная часть

Про меня

- Манаков Данила
- МФТИ
- работаю в Тинькофф
- можно писать в тг @DanilaManakov в любое время
- не факт, что я вам отвечу, но писать можно

Чат групп(ы?)

<Даня, не забудь втавить сюда QR-код>

Как закрыть курс

1. Инструменты

- Linux (хотяб Ubuntu)
  - лучше не WSL
  - да, можно виртуалку
- bash / zsh
- gcc + gas

2. Оценка

1. делаете задачи
2. защищаете задачи
3. оценка ставится по сумме баллов

~~Вводная часть~~

Сборка проекта

Код проекта → *???* → EFL

```c
#include <stdio.h>

int main() {
    printf("Hello, world!"\n);
    return 0;
}
```

```
$ file a.out
a.out: ELF 64-bit LSB pie executable, ...
```

Этапы сборки проекта

В процессе сборки проекта последовательно запускаются:

1. Препроцессор
    - из кода на C делает код на C
2. Компилятор
    - из кода на C делает код на ASM
3. Ассемблер
    - из кода на ASM делает op-коды
4. Линкер
    - собирает разные .o файлы вместе
    - настраивает адресацию функций
    - настраивает адресацию библиотек

1. Препроцессор

- выполняет простые операции с текстом (вырезать + вставить)
- обрабатывает инструкции, которые начинаются с #
  - #define
  - #include
  - #ifndef
  - #ifdef
  - #else
  - #end
  - ...

1. Препроцессор

```c
#include <stdio.h>

// у нас так НЕ ПРИНЯТО
// за такое будем бить
#define PI 3.1415926  // use `enum` instead

#define MAX(a, b) (((a) >= (b)) ? (a) : (b))

int main() {
    // ...
}
```

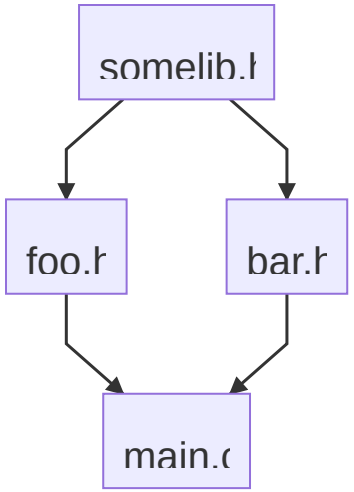
1. Препроцессор

```c
// somelib.h
void sml() {
  // ...
}
```

```c
// foo.h
#include "somelib.h"

void foo() {
  sml();
}
```

```c
// bar.h
#include "somelib.h"

void bar() {
  sml();
}
```

```c
#include "foo.h"
#include "bar.h"

int main() {
  foo();
  bar();
}
```



1. Препроцессор

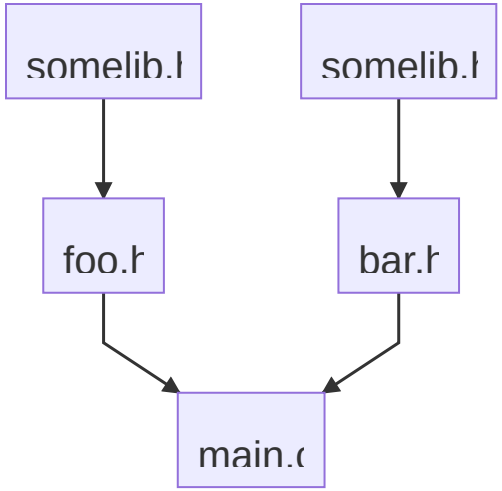
Скомпилировать такое не выйдет :(

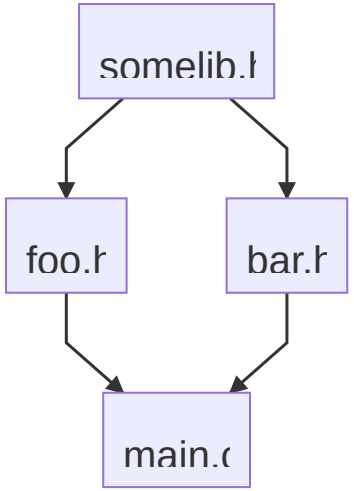
```
$ gcc main.c
In file included from bar.h:1,
                 from main.c:2:
somelib.h:1:6: error: redefinition of ‘sml’
    1 | void sml() {
      |      ^~~
In file included from foo.h:1,
                 from main.c:1:
somelib.h:1:6: note: previous definition of ‘sml’ with type ‘void()’
    1 | void sml() {
      |
```

1. Препроцессор

Потому что на самом деле там не ромбик, а такая штука

Хотя мы очень хотим, чтобы был ромбик





1. Препроцессор

Решается это как-то так

```
// somelib.h
#ifndef SOMELIB_H
#define SOMELIB_H

void sml() {
  // ...
}

#endif // SOMELIB_H
```

```
// foo.h
#ifndef FOO_H
#define FOO_H

#include "somelib.h"

void foo() {
  sml();
}

#endif // FOO_H
```

(аналогично `bar.h`)

1. Препроцессор

и такое чудо даже сработает

```
// gcc -E main.c
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c"
# 1 "foo.h" 1
# 1 "somelib.h" 1



void sml() {
}
# 2 "foo.h" 2

void foo() {
    sml();
}
# 2 "main.c" 2
# 1 "bar.h" 1


void bar() {
    sml();
}
# 3 "main.c" 2

int main() {
    foo();
    bar();
}
```

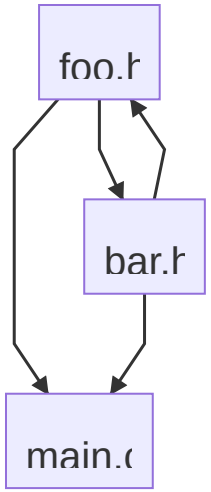
1. Препроцессор

Ну или чуть менее жизненный пример (в таком виде не компилируется)

```c
// foo.h
#include "bar.h"

void external_foo() { /* ... */ }

void foo() { external_bar(); }
```

```c
// bar.h
#include "foo.h"

void external_bar() { /* ... */ }

void bar() { external_foo(); }
```

```c
#include "foo.h"
#include "bar.h"

int main() {
  foo();
  bar();
}
```



1. Препроцессор

На самом деле чаще встречается что-то такое

```c
// foobar.h
#ifndef FOOBAR_H
#define FOOBAR_H

void foobar() {
    // ...
}

#endif // FOOBAR_H
```

```c
// main.c
#include "foobar.h"
#define PI 3.1415926

int main() {
    foobar();
    printf("%lf\n", PI);
    return 0;
}
```

```
$ gcc -E main.c > main.i
```

1. Препроцессор

```c
// foobar.h
#ifndef FOOBAR_H
#define FOOBAR_H

void foobar() {
    // ...
}

#endif // FOOBAR_H
```

```c
// main.c
#include "foobar.h"
#define PI 3.1415926

int main() {
    foobar();
    printf("%lf\n", PI);
    return 0;
}
```

```c
// gcc -E main.c
// и да, ниже валидный код на C
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c"
# 1 "foobar.h" 1
// (письмена выше понимать не надо)


void foobar() {
    // ...
}
# 2 "main.c" 2


int main() {
    foobar();
    printf("%lf\n", 3.1415926);
    return 0;
}
```

2. Компилятор

- это тот, который из кода на C делает код на языке ассемблера

2. Компилятор

```
// gcc -E main.c
// и да, ниже валидный код на C
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c"
# 1 "foobar.h" 1
// (письмена выше понимать не надо)


void foobar() {
    // ...
}
# 2 "main.c" 2


int main() {
    foobar();
    printf("%lf\n", 3.1415926);
    return 0;
}
```

```
        .file   "main.c"
        .intel_syntax noprefix
        .text
        .globl  foobar
        .type   foobar, @function
foobar:
.LFB0:
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        nop
        pop     rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   foobar, .-foobar
        .section        .rodata
.LC1:
        .string "%lf\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        mov     eax, 0
        call    foobar
        mov     rax, QWORD PTR .LC0[rip]
        movq    xmm0, rax
        lea     rax, .LC1[rip]
        mov     rdi, rax
        mov     eax, 1
        call    printf@PLT
        mov     eax, 0
        pop     rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   main, .-main
        .section        .rodata
        .align 8
.LC0:
        .long   1293080650
        .long   1074340347
        .ident  "GCC: (GNU) 12.2.1 20230201"
        .section        .note.GNU-stack,"",@progbits
```

```
$ gcc -S -masm=intel main.i
```

2. Компилятор

```
// gcc -E main.c
// и да, ниже валидный код на C
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c"
# 1 "foobar.h" 1
// (письмена выше понимать не надо)


void foobar() {
    // ...
}
# 2 "main.c" 2


int main() {
    foobar();
    printf("%lf\n", 3.1415926);
    return 0;
}
```

```
# ...
foobar:
.LFB0:
        push    rbp
        mov     rbp, rsp
        nop
        pop     rbp
        ret
# ...
main:
.LFB1:
        push    rbp
        mov     rbp, rsp
        mov     eax, 0
        call    foobar
        mov     rax, QWORD PTR .LC0[rip]
        movq    xmm0, rax
        lea     rax, .LC1[rip]
        mov     rdi, rax
        mov     eax, 1
        call    printf@PLT
        mov     eax, 0
        pop     rbp
        ret
# ...
```

```
$ gcc -S -masm=intel main.i
```

3. Ассемблер

- `ассемблер` - это программа
- `язык ассемблера` - это 'язык'
- `ассемблер` переводит код на `языке ассемблера` в op-коды

3. Ассемблер

Продолжаем издевательства

```
$ gcc -c main.s
```

Получаем ELF-файлик (не совсем правда)

```
$ file main.o
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

- файлик содержит op-коды
- но не содержит необходимой 'обертки' для запуска
- (в MS-DOS такое уже можно было запускать)

3. Ассемблер

Можно посмотреть, что там внутри этого файла

```
$ objdump -d -M intel main.o

main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foobar>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   90                      nop
   5:   5d                      pop    rbp
   6:   c3                      ret

0000000000000007 <main>:
   7:   55                      push   rbp
   8:   48 89 e5                mov    rbp,rsp
   b:   b8 00 00 00 00          mov    eax,0x0
  10:   e8 00 00 00 00          call   15 <main+0xe>
  15:   48 8b 05 00 00 00 00    mov    rax,QWORD PTR [rip+0x0]        # 1c <main+0x15>
  1c:   66 48 0f 6e c0          movq   xmm0,rax
  21:   48 8d 05 00 00 00 00    lea    rax,[rip+0x0]        # 28 <main+0x21>
  28:   48 89 c7                mov    rdi,rax
  2b:   b8 01 00 00 00          mov    eax,0x1
  30:   e8 00 00 00 00          call   35 <main+0x2e>
  35:   b8 00 00 00 00          mov    eax,0x0
  3a:   5d                      pop    rbp
  3b:   c3                      ret
```

4. Линкер

- собирает несколько .o файлов вместе
- высчитывает ссылки на функции
- 'линкует' динамические библиотеки

4. Линкер

Дальше запускаем простую команду

```
$ ld /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crtn.o /usr/lib/x86_64-linux-gnu/crt1.o -lc main.o -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o main_ELF_executable
```

4. Линкер

Дальше запускаем простую команду

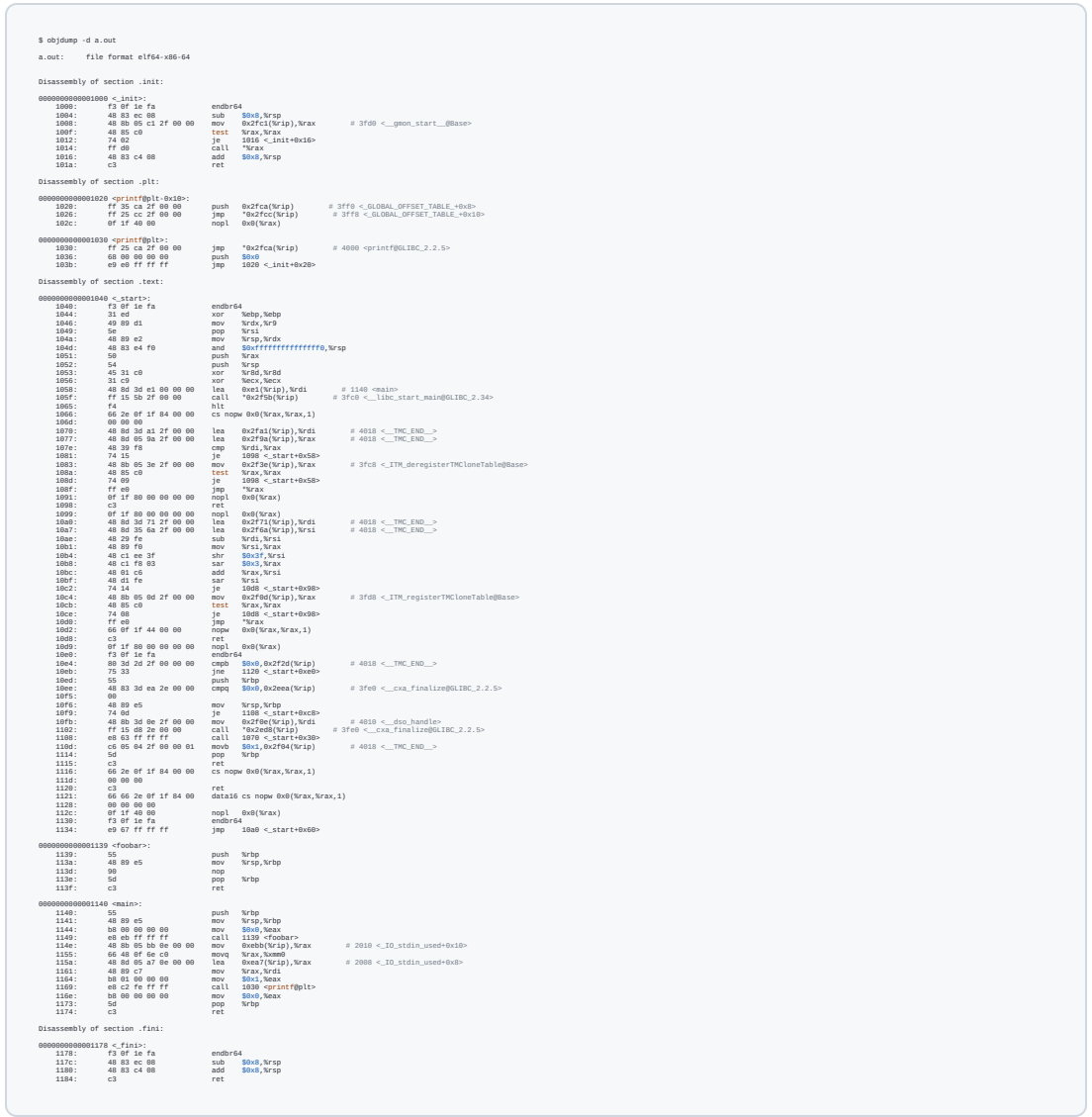
~~`$ ld /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crtn.o /usr/lib/x86_64-linux-gnu/crt1.o -lc main.o -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o main_ELF_executable`~~

```
$ gcc main.o
$ ./a.out
3.141593
```

4. Линкер

Для сравнения посмотрим на оба файлика

```
$ objdump -d -M intel main.o

main.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foobar>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   90                      nop
   5:   5d                      pop    rbp
   6:   c3                      ret

0000000000000007 <main>:
   7:   55                      push   rbp
   8:   48 89 e5                mov    rbp,rsp
   b:   b8 00 00 00 00          mov    eax,0x0
  10:   e8 00 00 00 00          call   15 <main+0xe>
  15:   48 8b 05 00 00 00 00    mov    rax,QWORD PTR [rip+0x0]        # 1c <main+0x15>
  1c:   66 48 0f 6e c0          movq   xmm0,rax
  21:   48 8d 05 00 00 00 00    lea    rax,[rip+0x0]        # 28 <main+0x21>
  28:   48 89 c7                mov    rdi,rax
  2b:   b8 01 00 00 00          mov    eax,0x1
  30:   e8 00 00 00 00          call   35 <main+0x2e>
  35:   b8 00 00 00 00          mov    eax,0x0
  3a:   5d                      pop    rbp
  3b:   c3                      ret
```



4. Линкер

Там есть уже знакомые нам куски

```
0000000000001139 <foobar>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       90                      nop
    113e:       5d                      pop    %rbp
    113f:       c3                      ret
```

```
0000000000001140 <main>:
    1140:       55                      push   %rbp
    1141:       48 89 e5                mov    %rsp,%rbp
    1144:       b8 00 00 00 00          mov    $0x0,%eax
    1149:       e8 eb ff ff ff          call   1139 <foobar>
    114e:       48 8b 05 bb 0e 00 00    mov    0xebb(%rip),%rax        # 2010 <_IO_stdin_used+0x10>
    1155:       66 48 0f 6e c0          movq   %rax,%xmm0
    115a:       48 8d 05 a7 0e 00 00    lea    0xea7(%rip),%rax        # 2008 <_IO_stdin_used+0x8>
    1161:       48 89 c7                mov    %rax,%rdi
    1164:       b8 01 00 00 00          mov    $0x1,%eax
    1169:       e8 c2 fe ff ff          call   1030 <printf@plt>
    116e:       b8 00 00 00 00          mov    $0x0,%eax
    1173:       5d                      pop    %rbp
    1174:       c3                      ret
```

4. Линкер

А есть что-то дикое (часть из этого коснёмся в будущем)

```
0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    # ...

0000000000001020 <printf@plt-0x10>:
    1020:       ff 35 ca 2f 00 00       push   0x2fca(%rip)        # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       ff 25 cc 2f 00 00       jmp    *0x2fcc(%rip)        # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000001030 <printf@plt>:
    1030:       ff 25 ca 2f 00 00       jmp    *0x2fca(%rip)        # 4000 <printf@GLIBC_2.2.5>
    1036:       68 00 00 00 00          push   $0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <_init+0x20>


0000000000001040 <_start>:
    1040:       f3 0f 1e fa             endbr64
    1044:       31 ed                   xor    %ebp,%ebp
    1046:       49 89 d1                mov    %rdx,%r9
    1049:       5e                      pop    %rsi
    104a:       48 89 e2                mov    %rsp,%rdx
    104d:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    1051:       50                      push   %rax
    1052:       54                      push   %rsp
    # ...

0000000000001178 <_fini>:
    1178:       f3 0f 1e fa             endbr64
    117c:       48 83 ec 08             sub    $0x8,%rsp
    1180:       48 83 c4 08             add    $0x8,%rsp
    1184:       c3                      ret
```

Подытог процесса сборки

1. Препроцессор

```
gcc -E main.c > main.i
```

2. Компилятор

```
gcc -S -masm=intel main.i
```

3. Ассемблер

```
gcc -c main.s
```

4. Линкер

```
gcc main.o
```

Подытог процесса сборки

Хотя в жизни мы с вами будем делать как-то так

```
gcc main.c -o hw1_ex1
```

ну или так

```
gcc main.c
```

Подытог процесса сборки

А почему везде `gcc`?

```
# GCC - GNU C Compiler
# GCC - GNU Compiler Collection
# C / C++
# ASM
# Fortran
# Pascal
# ...

# gnu c compiler
gcc -E main.c > main.i

# gnu c compiler (again)
gcc -S -masm=intel main.i

# gnu assembler
gcc -c main.s

# ld - The GNU linker
gcc main.o
```

GDB

- gnu debugger

Шпаргалка по основным командам

- `r` / `run`
- `b` / `break`
- `c` / `continue`
- `n` / `next`
- `s` / `step`
- `layout src` / `layout asm` / `layout regs`
- `print`