ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)
ФАКУЛЬТЕТ АЭРОКОСМИЧЕСКИХ ТЕХНОЛОГИЙ

Лабораторная 3
После пузырька можно уже всё

Рогозин Владимир
Группа Б03-106

Пункт 1: а) Уже из самого первого примера на C с простейшей функуцией без аргументов и возвращаемых значений видно, что в 32-х битной версии много дополнительных действий, а именно пуши регистров и действия с ними.

```
func:
.LFB0:
        .cfi_startproc
        endbr32
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        pushl   %ebx
        subl    $4, %esp
        .cfi_offset 3, -12
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        subl    $12, %esp
        leal    .LC0@GOTOFF(%eax), %edx
        pushl   %edx
        movl    %eax, %ebx
        call    puts@PLT
        addl    $16, %esp
        nop
        movl    -4(%ebp), %ebx
        leave
        .cfi_restore 5
        .cfi_restore 3
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   func, .-func
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        endbr32
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        call    func
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

(a) на C 32-х битная система

```
func:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   func, .-func
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        call    func
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) на C 64-х битная система

```c
#include <stdio.h>

void func() {
        printf("a b c\n");
}

int main() {
        func();
        return 0;
}
```

(a) Код программы на C

```cpp
#include <iostream>

void func() {
        std::cout << "a b c" << std::endl;
}

int main() {
        func();
        return 0;
}
```

(b) Код программы на C++

```
.LFB1522:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rsi
        leaq    _ZSt4cout(%rip), %rdi
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT
        movq    %rax, %rdx
        movq    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%rip), %rax
        movq    %rax, %rsi
        movq    %rdx, %rdi
        call    _ZNSolsEPFRSoS_E@PLT
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1522:
        .size   _Z4funcv, .-_Z4funcv
        .globl  main
        .type   main, @function
main:
.LFB1523:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        call    _Z4funcv
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Рис. 3: Листинг программы на C++ 64-х битной системы

На C++ ситуация примерно такая же, в 32-х битной версии присутствуют какие-то дополнительные пуши регистров, в 64-х битной версии такого почти нету, за исключением пушей, связанных, как мне кажется, с $std :: cout$ и $std :: endl$.

b) Для функции без аргумеетов, но с возвращаемым значением типа int дела обстоят практически одинаково что для C++, что для C, независимо от разрядности системы. Везде под возвращаемый int выделяется 16 байт (почему-то), только в C++ немного хуже дела обстоят с названиями (с человеческой точки зрения).

c) Далее, функция, возвращающая int, с одним аргументом того же типа. Здесь видны различия между 64-х и 32-х битными системами. В 32-х битной аргумент, который потом передаётся в функцию, перед вызовом функции кладётся в стек, и потом лежит в стеке перед функцией, причём если внутри функции есть арифметические операции с аргументом, то они производятся со значением, лежащим в стеке перед функцией. В 64-х битной системе аргумент запоминается в отдельный регистр (у меня был edi), после вызова функции значение с этого регистра копируется в стек, и операции проводятся со значением, лежащим в стеке после функции. Разницы между C и C++ опять же почти нету. Под результат функции (одно число int) и там и там выделяется 16 байт.

d) Теперь функция, которая возвращает int и принимает два аргумента типа int. Всё почти так же как и в предыдущем случае. В 32-х битном варианте резервируется под возвращаемое значение место в стеке (опять 16 байтов на один int), затем в стек пушатся аргументы функции, затем функция, потом производятся все арифметические операции, значения для которых передаются в регистры из стека (оттуда куда до этого пушились аргументы), после завершения работы функции на вершине стека остаётся возвращаемое значение. В 64-х битной системе, как и в предыдущем случае, используются дополнительные регистры (теперь edi и esi) для хранения значений аргументов. Значения их этих регистров копируются в стек функции, там же и проделываются операции с числами. Результат работы программы после её завершения также остаётся на вершине стека, на один int выделяется 16 байт. Различия между C и C++

всё также только в названиях переменных.

Пункт 2: a) Разберёмся с локальными переменными. Сначала посмотрим листинг с одной локальной переменной.

```
.LFB2000:
        .cfi_startproc
        endbr32
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        pushl   %ebx
        subl    $4, %esp
        .cfi_offset 3, -12
        call    __x86.get_pc_thunk.bx
        addl    $_GLOBAL_OFFSET_TABLE_, %ebx
        cmpl    $1, 8(%ebp)
        jne     .L5
        cmpl    $65535, 12(%ebp)
        jne     .L5
        subl    $12, %esp
        leal    _ZStL8__ioinit@GOTOFF(%ebx), %eax
        pushl   %eax
        call    _ZNSt8ios_base4InitC1Ev@PLT
        addl    $16, %esp
        subl    $4, %esp
        leal    __dso_handle@GOTOFF(%ebx), %eax
        pushl   %eax
        leal    _ZStL8__ioinit@GOTOFF(%ebx), %eax
        pushl   %eax
        movl    _ZNSt8ios_base4InitD1Ev@GOT(%ebx), %eax
        pushl   %eax
        call    __cxa_atexit@PLT
        addl    $16, %esp
.L5:
        nop
        movl    -4(%ebp), %ebx
        leave
        .cfi_restore 5
        .cfi_restore 3
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE2000:
        .size   _Z41__static_initialization_and_destruction_0ii, .-_Z41__static_initialization_and_destruction_0ii
        .type   _GLOBAL__sub_I_main, @function
```

Рис. 4: Функции после *main*, 32-х битная система

В языке C все достаточно предсказуемо, локальная переменная кладётся на вершину стека *main*'а, различия между 32-х и 64-х разрядной системой почти нету. Однако дело обстоит совсем по-другому с C++. В функции *main* все без изменений по сравнению с C, но после *main*'а что в 32-х битной, что в 64-х битной появляются ещё дополнительные функции, судя по некоторым различимым словам, это похоже на конструктор и деструктор класса *int*.

b) Теперь на очереди программа с парой локальных переменных типа *int*. В C, как в 32-х так и в 64-х битной системе, обе переменные просто кладутся на вершину стека *main*'а. В *main*'е листинга C++ всё идентично листингу на C, но после него опять идут дополнительные функции, которые абсолютно такие же, что и в случае одной локальной переменной, они на скринах листинга ниже и выше этого текста.

```
.LFB2003:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        cmpl    $1, -4(%rbp)
        jne     .L5
        cmpl    $65535, -8(%rbp)
        jne     .L5
        leaq    _ZStL8__ioinit(%rip), %rdi
        call    _ZNSt8ios_base4InitC1Ev@PLT
        leaq    __dso_handle(%rip), %rdx
        leaq    _ZStL8__ioinit(%rip), %rsi
        movq    _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rax
        movq    %rax, %rdi
        call    __cxa_atexit@PLT
.L5:
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2003:
        .size   _Z41__static_initialization_and_destruction_0ii, .-_Z41__static_initialization_and_destruction_0ii
        .type   _GLOBAL__sub_I_main, @function
_GLOBAL__sub_I_main:
.LFB2004:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $65535, %esi
        movl    $1, %edi
        call    _Z41__static_initialization_and_destruction_0ii
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Рис. 5: Функции после *main*, 64-х битная система

c) При создании статического массива из *int*'ов на C++ после *main*'а присутствуют те же самые дополнительные функции (похоже конструктор деструктор), которых нету на C, однако основной интерес представляют изменения в *main*'е. И в C, и в C++, и в 32-х битной, и в 64-х в стеке выделяется место под элементы массива (опять больше чем надо), но перед этим выполняется команда *movq* с первым операндом $\%fs:40$. Вся эта цепочка действий нужна для того, чтобы в конце работы функции проверить стек на переполненность, в регистре по адресу $fs:40$ лежит определенная константа, значения которой сравниваются в начале и в конце программы, если что-то идёт не как должно, то вызывается функция stack chk fail. В 32-х битной системе используется адрес $gs:20$.

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movl    $1, -32(%rbp)
        movl    $2, -28(%rbp)
        movl    $3, -24(%rbp)
        movl    $4, -20(%rbp)
        movl    $5, -16(%rbp)
        movl    $0, %eax
        movq    -8(%rbp), %rdx
        xorq    %fs:40, %rdx
        je      .L3
        call    __stack_chk_fail@PLT
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(a) Статический массив на C

```
main:
.LFB1522:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movl    $1, -32(%rbp)
        movl    $2, -28(%rbp)
        movl    $3, -24(%rbp)
        movl    $4, -20(%rbp)
        movl    $5, -16(%rbp)
        movl    $0, %eax
        movq    -8(%rbp), %rdx
        xorq    %fs:40, %rdx
        je      .L3
        call    __stack_chk_fail@PLT
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Статический массив на C++

d) Далее, динамический массив. Разницы между C и C++ опять почти нету, но большие отличия в листингах для различных систем. В 32-х битной перед и после пуша регистров, указывающих на вершину и дно стека (*esp* и *ebp* соответственно) кладутся в стек ещё и *ebx* и *ecx*, причем до этого в *main*'е регистр *ebx* вообще никак не используется, то есть по сути там лежит ненужный мусор. В каждой из систем в стеке хранится адрес на первый элемент выделенной памяти по которому и можно обращаться к любому элементу дин. массива. В 32-х битной системе количество выделенных байт (число) пушится в стек перед вызовом *new*(*malloc*), в 64-х – хранится в отдельном регистре (здесь это *edi*).

```
main:
.LFB1519:
        .cfi_startproc
        endbr32
        leal    4(%esp), %ecx
        .cfi_def_cfa 1, 0
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        .cfi_escape 0x10,0x5,0x2,0x75,0
        movl    %esp, %ebp
        pushl   %ebx
        pushl   %ecx
        .cfi_escape 0xf,0x3,0x75,0x78,0x6
        .cfi_escape 0x10,0x3,0x2,0x75,0x7c
        subl    $16, %esp
        call    __x86.get_pc_thunk.bx
        addl    $_GLOBAL_OFFSET_TABLE_, %ebx
        subl    $12, %esp
        pushl   $12
        call    _Znaj@PLT
        addl    $16, %esp
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        addl    $4, %eax
        movl    $56, (%eax)
        cmpl    $0, -12(%ebp)
        je      .L2
        subl    $12, %esp
        pushl   -12(%ebp)
        call    _ZdaPv@PLT
        addl    $16, %esp
.L2:
        movl    $0, %eax
        leal    -8(%ebp), %esp
        popl    %ecx
        .cfi_restore 1
        .cfi_def_cfa 1, 0
        popl    %ebx
        .cfi_restore 3
        popl    %ebp
        .cfi_restore 5
        leal    -4(%ecx), %esp
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

(a) Динамический массив на C++, 32 бита

```
main:
.LFB1522:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $12, %edi
        call    _Znam@PLT
        movq    %rax, -8(%rbp)
        movq    -8(%rbp), %rax
        addq    $4, %rax
        movl    $56, (%rax)
        cmpq    $0, -8(%rbp)
        je      .L2
        movq    -8(%rbp), %rax
        movq    %rax, %rdi
        call    _ZdaPv@PLT
.L2:
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Динамический массив на C++, 64 бита

Пункт 3: a) Двигаемся к структурам, сначала, не создавая экземпляра, просто объявим структуру с четырьмя различными полями *int*, *char*, *float* и *short*. В обоих языках в листинге просто по сути пустой *main*(что и логично), видимо пока нету экземпляра класса это никак не отражается на листинге. В C++ *main* также пуст, но функции после него (которые я считал конструктором и деструктором класса *int*) остались абсолютно такими же, вероятно это все же что-то другое, продолжим это выяснять дальше.

b) Теперь создадим глобальную переменную структуры и попробуем изменить пару её полей в *main*'е. Обращение происходит как и в случае с глобальным массивом, причем выделяется под каждое поле 4 байта, хотя некторые из них весят меньше . Перед *main*'ом объявляется глобальная переменная переменная. Принципиальных различий между C и C++ все также нет.

```
        .type   S, @object
        .size   S, 16
S:
        .long   1
        .byte   97
        .zero   3
        .long   1082130432
        .value  2
        .zero   2
        .text
        .globl  main
        .type   main, @function
main:
.LFB1519:
        .cfi_startproc
        endbr32
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        movl    S@GOTOFF(%eax), %edx
        addl    $1, %edx
        movl    %edx, S@GOTOFF(%eax)
        movb    $65, 4+S@GOTOFF(%eax)
        movl    $0, %eax
        popl    %ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

(a) Глобальная структура на C++, 32 бита

```
        .type   S, @object
        .size   S, 16
S:
        .long   1
        .byte   97
        .zero   3
        .long   1082130432
        .value  2
        .zero   2
        .text
        .globl  main
        .type   main, @function
main:
.LFB1522:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    S(%rip), %eax
        addl    $1, %eax
        movl    %eax, S(%rip)
        movb    $65, 4+S(%rip)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Глобальная структура на C++, 64 бита

c) Теперь в ту же структуры добавим статический массив. Ничего сверхъестественного не получаем, статический массив размера N по сути эквивалентен N переменным одного типа, так его и представляет компилятор, как бы развертывает массив в N новых полей одного типа. Принципиальных различий в разных системах и языках нету.

d) Тут передадим структуру в функцию (пока что не по указателю или ссылке). В обеих системах и обоих языках перед функцией в стек пушатся копии всех полей структуры (элементы массива пушатся поочереди) таким образом, что наверху оказывается первое по счёту поле. Различие в том, что в 32-х битной системе это делается с помощью команды *pushl*, а в 64-х битной это делается командой *movq*, используя адрес ячейки в стеке, к тому же, даже если переменные весят меньше 8 байт, к примеру две по 4 байта, их слепляют в одну восьмибайтовую, достать одно из двух значений конечно можно дописав к команде *mov* окончание *l* и указав правильный адрес.

e) Если сделать возвращаемым значением сделать структуру, то в этом случае также, как и когда функция возвращала целое число, в стеке перед вызовом функции резервируется место под возвращаемое значение, затем в стек пушатся все аргументы, потом вызывается функция. после завершения работы на вершине стека оказывается возвращаемая структура. Различие между системами только в способе заполнения стека, в 32-х битной с помощью команды *pushl*, в 64-х – *movq*.

f) В этом пункте взглянем на методы класса в C++. При создании экземпляра класса вызывается его конструктор, если у класса есть поля, под их значения выделяется место в стеке перед вызовом самой функции (конструктора). После этого вызывается функция, куда передаются адреса для записи значений полей, куда в итоге и попадают значения полей экземпляра класса. Если вызывается метод класса, то, если он использует поля класса, в функцию передаются как бы неявные аргументы – поля, метод работает

как обчная функция, если он меняет поля, то по сути это принимающая на вход указатель и меняющая по нему значение функция (что опять же логично очень даже).

g) Операторы. Операторы это те же функции. Работают также, результат операции отправляется в заранее заготовленную ячейку памяти. Единственное, в 32-х битной системе результат записывается в нужную ячейку в конце выполнения функции, в 64-х битной результат возвращается в регистре, после чего записывается из регистра в нужную часть памяти.

Пункт 4: a) Структуру в функцию мы уже передавали в пункте 3d), поэтому тут все тоже самое.

b) Теперь передадим структуру по указателю. Здесь все также достаточно ожидаемо, просто теперь функция использует не скопированные значения полей, а адреса ячеек, в которых лежат эти значения. Обращаясь по этим адреса, функция меняет сам объект, а не его копию. Отличие между 32-х и 64-х битной системами в том, что в 64-х битной когда полей мало, то их адреса передаются в функцию через регистры(адрес записывается в регистр, потом переписывается из регистра в стек функции после её вызова), а в 32-х битной всегда адреса пушатся в стек перед вызовом функции.

c) Ссылки на очереди. Передача в функцию по ссылке в ассемблерном листинге никоим образом не отличается от передачи по указателю. Все абсолютно также, как и в предыдущем пункте.

d) В этом пункте рассмотрим передачу в функцию с помощью rvalue-ссылок. В этом случае просто результат промежуточного действия записывается в одну из ячееку стека, адрес которой потом передаётся в функцию.

Пункт 5: a) Создадим структуру со статическим массивом в 100000 элементов значением 0, передадим её аргументом в функцию. Листинги программы приведены ниже:

```
        pushl   %ecx
        .cfi_escape 0xf,0x3,0x75,0x78,0x6
        .cfi_escape 0x10,0x3,0x2,0x75,0x7c
        leal    -397312(%esp), %eax
.LPSRL0:
        subl    $4096, %esp
        orl     $0, (%esp)
        cmpl    %eax, %esp
        jne     .LPSRL0
        subl    $2704, %esp
        call    __x86.get_pc_thunk.bx
        addl    $_GLOBAL_OFFSET_TABLE_, %ebx
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        leal    -400012(%ebp), %eax
        movl    $400000, %edx
        subl    $4, %esp
        pushl   %edx
        pushl   $0
        pushl   %eax
        call    memset@PLT
        addl    $16, %esp
        subl    $400000, %esp
        movl    %esp, %eax
        movl    %eax, %edx
        leal    -400012(%ebp), %eax
        movl    $400000, %ecx
        subl    $4, %esp
        pushl   %ecx
        pushl   %eax
        pushl   %edx
        call    memcpy@PLT
        addl    $16, %esp
        call    _Z4func6Custom
        addl    $400000, %esp
        movl    $0, %eax
        movl    -12(%ebp), %ecx
        xorl    %gs:20, %ecx
        je      .L4
        call    __stack_chk_fail_local
```

(a) Жирная структура в аргументе C++, 32 бита

```
main:
.LFB1523:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    -397312(%rsp), %r11
.LPSRL0:
        subq    $4096, %rsp
        orq     $0, (%rsp)
        cmpq    %r11, %rsp
        jne     .LPSRL0
        subq    $2704, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        leaq    -400016(%rbp), %rax
        movl    $400000, %edx
        movl    $0, %esi
        movq    %rax, %rdi
        call    memset@PLT
        subq    $400000, %rsp
        movq    %rsp, %rax
        movq    %rax, %rcx
        leaq    -400016(%rbp), %rax
        movl    $400000, %edx
        movq    %rax, %rsi
        movq    %rcx, %rdi
        call    memcpy@PLT
        call    _Z4func6Custom
        addq    $400000, %rsp
        movl    $0, %eax
        movq    -8(%rbp), %rcx
        xorq    %fs:40, %rcx
        je      .L4
        call    __stack_chk_fail@PLT
```

(b) Жирная структура в аргументе C++, 64 бита

Как видно, сначала при создании экземпляра структуры выделяется 400000 байт, значения задаются с

помощью функции memset, которая принимает на вход адрес первого элемента массива, значение, которое нужно присвоить элементам массива(здесь это 0), и длину массива. Различие между системами в том, что в 32-х битной эти аргументы функции memset пушатся в стек, а затем функция уже берет их оттуда, а в 64-х битной системе аргументы кладутся в регистры, откуда потом функция их использует. Но это относится к созданию экземпляра класса, что касается его передачи в функцию (точнее его копии), то тут дело обстоит примерно также, за исключением того, что для копирования массива при передачи в функцию используется функция memcpy. Она принимает на вход адрес куда копировать, адрес откуда копировать и количество элементов. Аргументы передаются в функцию таким же образом, по-разному в разных системах. Также заметим, что первое выделение места под структуру происходит не единовременным вычитанием 400000 байт из $rsp(esp)$, а в цикле, вычитая по 4096 байт (вот это выглядит странным, плюс тот факт, что с числом, лежащим в ячейке на которую указывает промежуточное значение $rsp(esp)$, прозводится логическое или, где второй операнд есть 0).

b) Теперь огромную структуру будем возвращать из функции, принимающей на вход тоже большую структуру. По сути, ничего принципиально отличающегося от предыдущего случая тут нету, просто больше копирований туда-сюда. Заранее выделяется место в стеке под возвращаемое значение (причем оно оказалось ниже в стеке, чем значение полей изначально созданной структуры, копия которой потом передаётся в функцию), с помощью memcpy сначала копируется на вершину весь массив как аргумент функции, затем, после операций, используя тот же memcpy, измененные значения копируются на дно стека, после этого работа функции заканчивается.

c) Если создать большую структуру внутри функции, то с помощью команды memset в стеке функции будет создан экземпляр структуры точно также, как и в стеке *main*'а.

d) При изменении размера структуры вроде как в обеих системах ничего, кроме размера выделяемой памяти, не меняется.

Пункт 6: Напишем рекурсивную функцию и посмотрим на её листинг. Просто из функции вызывается эта же функция.

```
func:
.LFB6:
        .cfi_startproc
        endbr32
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $8, %esp
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        cmpl    $1, 8(%ebp)
        jle     .L2
        movl    8(%ebp), %eax
        subl    $1, %eax
        subl    $12, %esp
        pushl   %eax
        call    func
        addl    $16, %esp
        jmp     .L3
.L2:
        movl    8(%ebp), %eax
.L3:
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

(a) Рекурсия на C, 32 бита

```
_Z4funci:
.LFB1522:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        cmpl    $1, -4(%rbp)
        jle     .L2
        movl    -4(%rbp), %eax
        subl    $1, %eax
        movl    %eax, %edi
        call    _Z4funci
        jmp     .L3
.L2:
        movl    -4(%rbp), %eax
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Рекурсия на C++, 64 бита