Лабораторная работа 7

Изнанка языка C++

Задание 1

Разберёмся как выглядят методы и поля классов. Для этого напишем класс **Porsche_911**. Выбор такого названия естественно обусловлен тем, что перед покупкой надо разобраться в характеристиках автомобиля, а ездить на чём то хуже, мне, как физтеху, просто не позволительно.

Создадим несколько приватных переменных, конструктор, деструктор, геттер и сеттер. Деструктор будет пустым, так как нам не надо освобождать память при уничтожении экземпляра. Но для того чтобы ему не было совсем скучно, мы скажем ему напечатать одну строчку (См Рис. 1).

```cpp
#include <iostream>

class Porsche_911
{
private:
    const int power;
    const int max_speed;
    const int fuel_consumption;
    const int acceleration_0_100;
    const int fuel_capacity;
    int fuel_volume;

public:
    Porsche_911() : power(385), max_speed(293),
                    fuel_consumption(9.4),
                    acceleration_0_100(4.2),
                    fuel_capacity(64),
                    fuel_volume(0)
    {}

    void set_fuel_volume(int new_fuel_volume)
    {
        fuel_volume = new_fuel_volume;
    }

    int get_fuel_volume()
    {
        return fuel_volume;
    }

    ~Porsche_911()
    {
        printf("Destructor finished");
    }
};

int main()
{
    Porsche_911 car1{};
    Porsche_911 car2{};
    car1.set_fuel_volume(60);
    car1.get_fuel_volume();
    car2.set_fuel_volume(60);
    car2.get_fuel_volume();
}
```

Рисунок 1: Программа ***ex1_1.cpp***

```
14  _ZN11Porsche_911C2Ev:
15  .LFB1523:
16      .cfi_startproc
17      endbr64
18      pushq   %rbp
19      .cfi_def_cfa_offset 16
20      .cfi_offset 6, -16
21      movq    %rsp, %rbp
22      .cfi_def_cfa_register 6
23      movq    %rdi, -8(%rbp)
24      movq    -8(%rbp), %rax
25      movl    $385, (%rax)
26      movq    -8(%rbp), %rax
27      movl    $293, 4(%rax)
28      movq    -8(%rbp), %rax
29      movl    $9, 8(%rax)
30      movq    -8(%rbp), %rax
31      movl    $4, 12(%rax)
32      movq    -8(%rbp), %rax
33      movl    $64, 16(%rax)
34      movq    -8(%rbp), %rax
35      movl    $0, 20(%rax)
36      nop
37      popq    %rbp
38      .cfi_def_cfa 7, 8
39      ret
40      .cfi_endproc
```

*Рисунок 3: Ф-я **_ZN11Porsche_911C2Ev** из файла **ex1_1.s***

```
49  _ZN11Porsche_91115set_fuel_volumeEi:
50  .LFB1525:
51      .cfi_startproc
52      endbr64
53      pushq   %rbp
54      .cfi_def_cfa_offset 16
55      .cfi_offset 6, -16
56      movq    %rsp, %rbp
57      .cfi_def_cfa_register 6
58      movq    %rdi, -8(%rbp)
59      movl    %esi, -12(%rbp)
60      movq    -8(%rbp), %rax
61      movl    -12(%rbp), %edx
62      movl    %edx, 20(%rax)
63      nop
64      popq    %rbp
65      .cfi_def_cfa 7, 8
66      ret
67      .cfi_endproc
```

*Рисунок 2: Ф-я **_ZN11Porsche_91115set_fuel_volumeEi** из файла **ex1_1.s***

```
74  _ZN11Porsche_91115get_fuel_volumeEv:
75  .LFB1526:
76      .cfi_startproc
77      endbr64
78      pushq   %rbp
79      .cfi_def_cfa_offset 16
80      .cfi_offset 6, -16
81      movq    %rsp, %rbp
82      .cfi_def_cfa_register 6
83      movq    %rdi, -8(%rbp)
84      movq    -8(%rbp), %rax
85      movl    20(%rax), %eax
86      popq    %rbp
87      .cfi_def_cfa 7, 8
88      ret
89      .cfi_endproc
```

*Рисунок 4: Ф-я **_ZN11Porsche_91115get_fuel_volumeEv** из файла **ex1_1.s***

```
 99  _ZN11Porsche_911D2Ev:
100  .LFB1528:
101      .cfi_startproc
102      .cfi_personality 0x9b,DW.ref.__gxx_personality_v0
103      .cfi_lsda 0x1b,.LLSDA1528
104      endbr64
105      pushq   %rbp
106      .cfi_def_cfa_offset 16
107      .cfi_offset 6, -16
108      movq    %rsp, %rbp
109      .cfi_def_cfa_register 6
110      subq    $16, %rsp
111      movq    %rdi, -8(%rbp)
112      leaq    .LC0(%rip), %rdi
113      movl    $0, %eax
114      call    printf@PLT
115      nop
116      leave
117      .cfi_def_cfa 7, 8
118      ret
119      .cfi_endproc
```

*Рисунок 5: Ф-я **_ZN11Porsche_911D2Ev** из файла **ex1_1.s***

```
137 main:
138 .LFB1530:
139     .cfi_startproc
140     endbr64
141     pushq   %rbp
142     .cfi_def_cfa_offset 16
143     .cfi_offset 6, -16
144     movq    %rsp, %rbp
145     .cfi_def_cfa_register 6
146     subq    $64, %rsp
147     movq    %fs:40, %rax
148     movq    %rax, -8(%rbp)
149     xorl    %eax, %eax
150     leaq    -64(%rbp), %rax
151     movq    %rax, %rdi
152     call    _ZN11Porsche_911C1Ev
153     leaq    -32(%rbp), %rax
154     movq    %rax, %rdi
155     call    _ZN11Porsche_911C1Ev
156     leaq    -64(%rbp), %rax
157     movl    $60, %esi
158     movq    %rax, %rdi
159     call    _ZN11Porsche_91115set_fuel_volumeEi
160     leaq    -64(%rbp), %rax
161     movq    %rax, %rdi
162     call    _ZN11Porsche_91115get_fuel_volumeEv
163     leaq    -32(%rbp), %rax
164     movl    $60, %esi
165     movq    %rax, %rdi
166     call    _ZN11Porsche_91115set_fuel_volumeEi
167     leaq    -32(%rbp), %rax
168     movq    %rax, %rdi
169     call    _ZN11Porsche_91115get_fuel_volumeEv
170     leaq    -32(%rbp), %rax
171     movq    %rax, %rdi
172     call    _ZN11Porsche_911D1Ev
173     leaq    -64(%rbp), %rax
174     movq    %rax, %rdi
175     call    _ZN11Porsche_911D1Ev
176     movl    $0, %eax
177     movq    -8(%rbp), %rdx
178     xorq    %fs:40, %rdx
179     je  .L8
180     call    __stack_chk_fail@PLT
```

*Рисунок 6: Ф-я **main** из файла **ex1_1.s***

Теперь посмотрим на фрагменты ассемблерного листинга программы **ex1_1.cpp** (См. Рис. 2-6) и подробно разберём всё представленные названия методов класса **Porsche_911**.

Мы получили следующие названия:

- Конструктор - **_ZN11Porsche_911C2Ev**
- Деструктор - **_ZN11Porsche_911D2Ev**
- Сеттер - **_ZN11Porsche_91115set_fuel_volumeEi**
- Геттер - **_ZN11Porsche_91115get_fuel_volumeEv**

Как можно видеть названия всех функций начинаются с **_ZN** и номера **11.** Число здесь означает количество символов в названии класса. Далее идёт само название класса **Porsche_911**. У конструктора и деструктора далее идут буквы **C** и **D**, которые означают «**construct**» и «**destruct**» соответственно. Последняя буква в названиях сеттера и геттера означает возвращаемые типы — **i** - «**int**», **v** - «**void**» соответственно.

Задание 2

Как мы можем видеть, в вышеприведённом коде (См. Рис. 1) мы создаём два объекта **car1** и **car2** внутри функции main(). Ассемблерный листинг функции main приведён на Рис. 6. В строках 152 и 155 происходит вызов конструкторов для соответствующих экземпляров класса с помощью команды **call**. С подобным мы уже встречались в предыдущих работах. Особый интерес здесь представляет то, что происходит на две строки выше: с помощью команды **leaq** мы извлекаем из стека некий адрес и помещаем его в регистр **rdi**.

Теперь посмотрим на саму реализацию конструктора (См. Рис. 3). Здесь мы в 23 строке извлекаем из регистра **rdi** некоторое значение и далее работаем с ним. Отсюда мы можем понять, что работа с разными экземплярами класса происходит с помощью передачи адреса на расположение этого экземпляра в памяти. А в самой функции с помощью различных сдвигов мы уже получаем доступ ко всем необходимы полям. Тогда мы можем сказать что экземпляр **car1** находится по адресу **-64(%rbp)**, а **car2** - **-32(%rbp)**. Также становится ясным, что отдельный экземпляр лежит в памяти в виде непрерывного куска данных, иначе работа функций без дополнительной информации была бы просто невозможной.

Задание 3

При вызове функции **car1.set_fuel_volume(60)** C++ понимает, что функция **set_fuel_volume()** работает с объектом **car1**. Рассмотрим детально как это всё работает.

Возьмём к примеру следующую строку:

car1.set_fuel_volume(60);

Хотя на первый взгляд кажется,что у нас здесь только один аргумент, но на самом деле их два. В время компиляции строка **car1.set_fuel_volume(60)** конвертируется компилятором в следующее:

set_fuel_volume(&car1, 60);

Теперь это всего лишь стандартный вызов функции, а объект **car1** (который ранее был отдельным объектом и находился перед точкой) теперь передается по адресу в качестве аргумента функции.

Но это только половина дела. Поскольку в вызове функции теперь есть два аргумента, то и метод нужно изменить соответствующим образом (чтобы он принимал два аргумента). Следовательно, следующий метод:

```
void set_fuel_volume(int new_fuel_volume)
{
        fuel_volume = new_fuel_volume;
}
```

Конвертируется компилятором в:

```
void set_fuel_volume(Porsche_911* const this,  int new_fuel_volume)
{
        this->fuel_volume = new_fuel_volume;
}
```

При компиляции обычного метода, компилятор неявно добавляет к нему параметр ***this**. Указатель ***this** — это скрытый константный указатель, содержащий адрес объекта, который вызывает метод класса. Отсюда мы можем сделать вывод о том, что все те процедуры, которые были описаны в задании 2 связаны как раз с передачей указателя ***this**.

Задание 4

Теперь поговорим поподробнее о конструкторах и деструкторах и сравним то, в каком порядке они вызываются для глобальных и локальных экземпляров класса. Для этого создадим глобальную переменную **global_car** и оставим одну из локальных переменных **car1** (См. Рис. 8).

```
139 global_car:
140     .zero   24
141     .text
142     .globl  main
143     .type   main, @function
144 main:
145 .LFB1530:
146     .cfi_startproc
147     endbr64
148     pushq   %rbp
149     .cfi_def_cfa_offset 16
150     .cfi_offset 6, -16
151     movq    %rsp, %rbp
152     .cfi_def_cfa_register 6
153     subq    $32, %rsp
154     movq    %fs:40, %rax
155     movq    %rax, -8(%rbp)
156     xorl    %eax, %eax
157     leaq    -32(%rbp), %rax
158     movq    %rax, %rdi
159     call    _ZN11Porsche_911C1Ev
160     leaq    -32(%rbp), %rax
161     movl    $60, %esi
162     movq    %rax, %rdi
163     call    _ZN11Porsche_91115set_fuel_volumeEi
164     leaq    -32(%rbp), %rax
165     movq    %rax, %rdi
166     call    _ZN11Porsche_91115get_fuel_volumeEv
167     movl    $60, %esi
168     leaq    global_car(%rip), %rdi
169     call    _ZN11Porsche_91115set_fuel_volumeEi
170     leaq    global_car(%rip), %rdi
171     call    _ZN11Porsche_91115get_fuel_volumeEv
172     leaq    -32(%rbp), %rax
173     movq    %rax, %rdi
174     call    _ZN11Porsche_911D1Ev
175     movl    $0, %eax
176     movq    -8(%rbp), %rdx
177     xorq    %fs:40, %rdx
178     je  .L8
179     call    __stack_chk_fail@PLT
```

*Рисунок 7: Фрагмент файла **ex4_1.s***

```
37  Porsche_911 global_car{};
38
39  int main()
40  {
41      Porsche_911 car1{};
42      car1.set_fuel_volume(60);
43      car1.get_fuel_volume();
44      global_car.set_fuel_volume(60);
45      global_car.get_fuel_volume();
46  }
```

*Рисунок 8: Фрагмент файла **ex4_1.cpp***

Теперь посмотрим на ассемблерный листинг данной программы (См. Рис. 7). Здесь представлена лишь функция main, так как весь остальной код остался либо посимвольно идентичным тому, что было ранее, либо с небольшими малоинтересными корректировками.

Мы можем видеть, что для локального экземпляра класса всё осталось без изменений: последовательно вызываются конструктор, сеттер, геттер и деструктор, для каждого из них со стека извлекается адрес и передаётся в каждую из этих функций (функций в смысле ассемблера естественно).

Для глобальной переменной всё происходит несколько иначе. В листинге для неё появился отдельный блок, который описывает некоторые необходимые начальные параметры. Внутри функции main адрес для передачи в функцию извлекается сразу же по имени самой переменной. Но с этим всем мы уже были знакомы. Интересным является то, что в отличие от локального экземпляра класса, для **global_car** внутри функции main вызываются только сеттер и геттер, а конструктор и деструктор — нет.

Но так как при выполнении программы **ex4_1.cpp** в выводе мы дважды видим сообщение «**Destructor finished**», то деструктор , очевидно, срабатывает и для глобальных переменных. Просто похоже что это происходит не так явно, как для локальных экземпляров класса. И это действительно так. Рассмотри это несколько подробнее.

```
226 _GLOBAL__sub_I_global_car:
227 .LFB2012:
228     .cfi_startproc
229     endbr64
230     pushq   %rbp
231     .cfi_def_cfa_offset 16
232     .cfi_offset 6, -16
233     movq    %rsp, %rbp
234     .cfi_def_cfa_register 6
235     movl    $65535, %esi
236     movl    $1, %edi
237     call    _Z41__static_initialization_and_destruction_0ii
238     popq    %rbp
239     .cfi_def_cfa 7, 8
240     ret
241     .cfi_endproc
```

*Рисунок 9: Ф-я **_GLOBAL__sub_I_global_car** из файла **ex4_1.s***

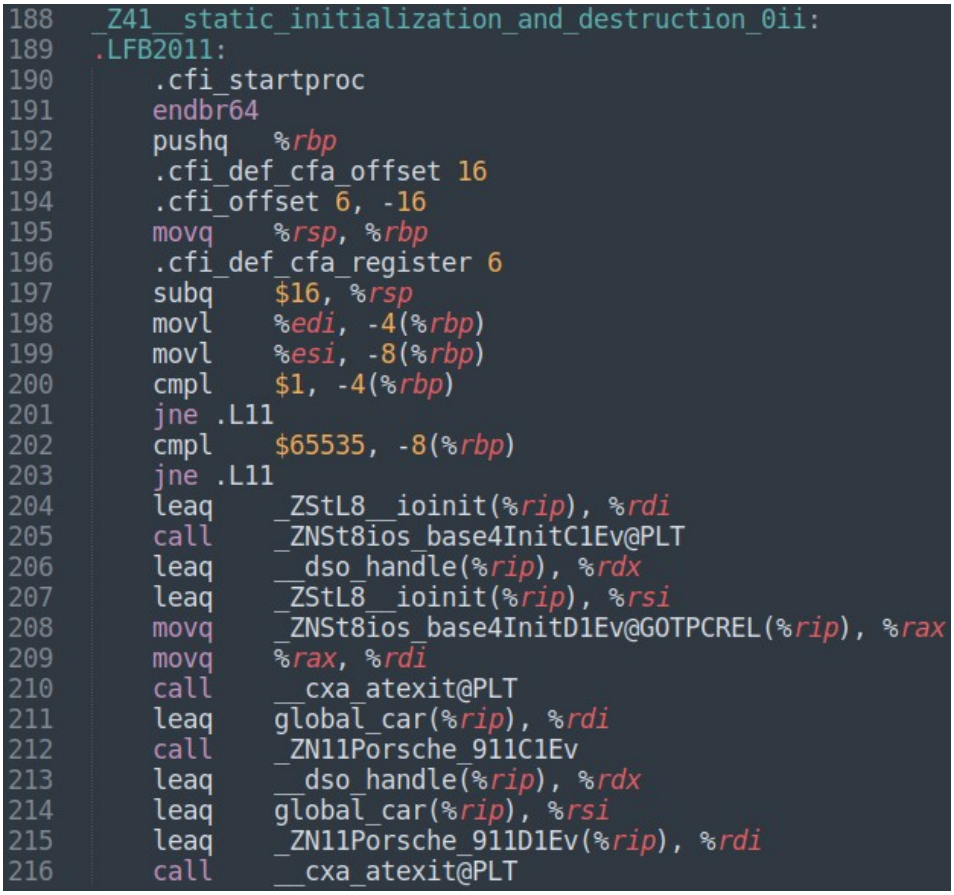


*Рисунок 10: Ф-я **_Z41__static_initialization_and_destruction_0ii** из файла **ex4_1.s***

В файле **ex4_1.s** ниже функции **main** мы можем заметить функцию **_GLOBAL__sub_I_global_car** (См. Рис. 9), в которой происходит вызов функции **_Z41__static_initialization_and_destruction_0ii** (строка 327). А уже в ней происходит вызов конструктора и деструктора для глобальной переменной **global_car**. (См. Рис. 10, строки 212 и 215). Сам вызов происходит по той же схеме, что была рассмотрена раньше.

Задание 5

Посмотрим на то как работает инкапсуляция. Для этого создадим новый приватный метод **get_fuel_of_volume()**, а для того чтобы он появился в листинге мы изменим публичный метод **set_fuel_volume()** так, чтобы он использовал **get_fuel_of_volume()**.

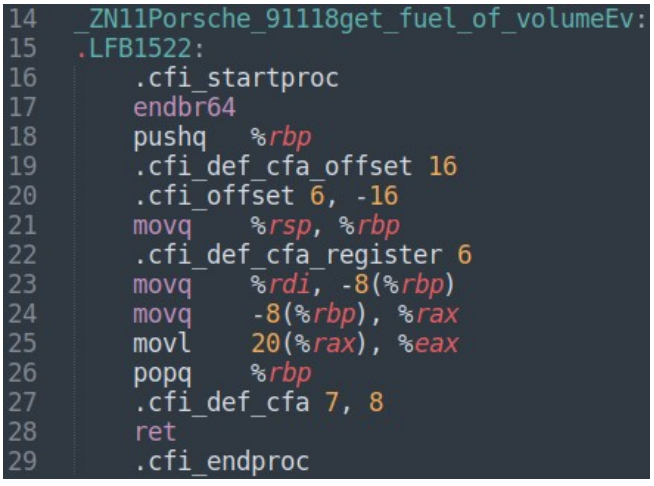


*Рисунок 11: Ф-я **_ZN11Porsche_91118get_fuel_of_volumeEv** из файла **ex5_1.s***

*Рисунок 12: Ф-я **_ZN11Porsche_91115get_fuel_volumeEv** из файла **ex5_1.s***

Теперь сравним полученные ассемблерные листинги функций **get_fuel_of_volume()** и **get_fuel_volume()** (См. Рис. 11 и 12). Как можно видеть в представленных приватной и публичной функциях ничего кроме названия не отличается, а потому мы можем заключить, что на уровне ассемблера инкапсуляции не существует. Здесь методы любого уровня доступа представляются в виде обычных функций, обращение к которым происходит с помощью меток. Тогда мы можем заключить, что возможности инкапсуляции в языке C++ предоставляются нам компилятором, который будет бить по рукам, если кто то решит обратиться к приватному полю класса непосредственно, а не через публичную функцию.

Задание 6

Теперь поговорим про наследование. Для этого напишем два класса: родительский класс **Human** и дочерний класс **BasketballPlayer** (См. Рис. 13).

```cpp
#include <iostream>

class Human
{
public:
    std::string name;
    int age;

    Human(std::string name = "", int age = 0)
        : name(name), age(age)
    {
        printf("Human constructor working\n");
    }

    std::string getName()
    {
        return name;
    }

    int getAge() const
    {
        return age;
    }

    ~Human()
    {
        printf("Human destructor working\n");
    }

};

class BasketballPlayer : public Human
{
public:
    double game_average;
    int points;

    BasketballPlayer(double gameAverage = 0.0, int points = 0)
        : game_average(game_average), points(points)
    {
        printf("BasketballPlayer constructor working\n");
    }

    double get_game_average()
    {
        return game_average;
    }

    int get_points()
    {
        return points;
    }

    ~BasketballPlayer()
    {
        printf("BasketballPlayer destructor working\n");
    }
};

int main()
{
    BasketballPlayer player1(31.2, 44);
    player1.get_points();
}
```

*Рисунок 13: Программа **ex6_1.cpp***

Оба класса имеют конструктор и деструктор, которые выводят сообщения о своей работе, два публичных поля и метода. Вывод данной программы следующий:

```
Human constructor working
BasketballPlayer constructor working
BasketballPlayer destructor working
Human destructor working
```

*Рисунок 14: Вывод программы **ex6_1.cpp***

Таким образом мы понимаем, что сначала отрабатывает конструктор родительского класса, а потом дочернего. Порядок работы деструкторов обратный.

Теперь посмотрим на то, как это реализовано внутри (См. Рис. 14 - 19). В функции main всё организовано интуитивно понятым способом (См. Рис. 14). Мы получили ровно то, чего хотели: создали экземпляр класса, вызвали метод и всё уничтожили деструктором.

Но если взглянуть на конструктор дочернего класса **BasketballPlayer** (См. Рис. 16), то мы можем увидеть, что внутри него происходит вызов конструктора родительского класса - **Human**. Здесь происходит инициализация необходимых полей и вывод проверочного сообщения (См. Рис 17). После поток исполнения возвращается обратно к конструктору дочернего класса **BasketballPlayer** и происходит вывод его сообщения.

```
main:
.LFB1538:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $64, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movq    .LC5(%rip), %rdx
    leaq    -64(%rbp), %rax
    movl    $44, %esi
    movq    %rdx, %xmm0
    movq    %rax, %rdi
    call    _ZN16BasketballPlayerC1Edi
    leaq    -64(%rbp), %rax
    movq    %rax, %rdi
    call    _ZN16BasketballPlayer10get_pointsEv
    leaq    -64(%rbp), %rax
    movq    %rax, %rdi
    call    _ZN16BasketballPlayerD1Ev
    movl    $0, %eax
    movq    -8(%rbp), %rcx
    xorq    %fs:40, %rcx
    je  .L20
    call    __stack_chk_fail@PLT
```

*Рисунок 15: Ф-я **main** из файла **ex6_1.s***

На рисунках 16 и 17 асссемблерный код представлен до момента печати нашего проверочного сообщения ввиду громоздкости листинга.

С деструкторами дело обстоит точно также, не считая уже оговоренного обратного порядка их работы. Но ассемблерный код, описывающий работу деструкторов в разы меньше, чем тот что описывает конструкторы. Код представлен на рисунках 18 и 19.

```
_ZN16BasketballPlayerC2Edi:
.LFB1531:
    .cfi_startproc
    .cfi_personality 0x9b,DW.ref.__gxx_personality_v0
    .cfi_lsda 0x1b,.LLSDA1531
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    pushq   %rbx
    subq    $104, %rsp
    .cfi_offset 3, -24
    movq    %rdi, -88(%rbp)
    movsd   %xmm0, -96(%rbp)
    movl    %esi, -100(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -24(%rbp)
    xorl    %eax, %eax
    movq    -88(%rbp), %rbx
    leaq    -65(%rbp), %rax
    movq    %rax, %rdi
    call    _ZNSaIcEC1Ev@PLT
    leaq    -65(%rbp), %rdx
    leaq    -64(%rbp), %rax
    leaq    .LC2(%rip), %rsi
    movq    %rax, %rdi
.LEHB3:
    call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1EPKcRKS3_@PLT
.LEHE3:
    leaq    -64(%rbp), %rax
    movl    $0, %edx
    movq    %rax, %rsi
    movq    %rbx, %rdi
.LEHB4:
    call    _ZN5HumanC2ENSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEi
.LEHE4:
    leaq    -64(%rbp), %rax
    movq    %rax, %rdi
    call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev@PLT
    leaq    -65(%rbp), %rax
    movq    %rax, %rdi
    call    _ZNSaIcED1Ev@PLT
    movq    -88(%rbp), %rax
    movsd   40(%rax), %xmm0
    movq    -88(%rbp), %rax
    movsd   %xmm0, 40(%rax)
    movq    -88(%rbp), %rax
    movl    -100(%rbp), %edx
    movl    %edx, 48(%rax)
    leaq    .LC3(%rip), %rdi
.LEHB5:
    call    puts@PLT
```

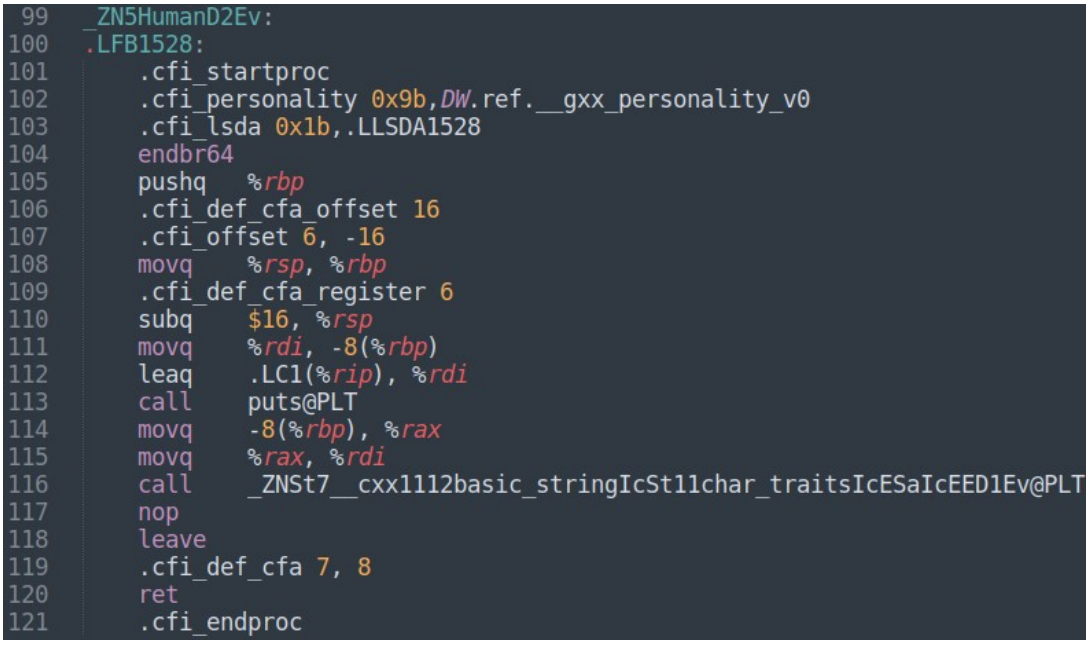
*Рисунок 16: Ф-я **_ZN16BasketballPlayerC2Edi** из файла **ex6_1.s***

```
16  _ZN5HumanC2ENSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEi:
17  .LFB1523:
18      .cfi_startproc
19      .cfi_personality 0x9b,DW.ref.__gxx_personality_v0
20      .cfi_lsda 0x1b,.LLSDA1523
21      endbr64
22      pushq   %rbp
23      .cfi_def_cfa_offset 16
24      .cfi_offset 6, -16
25      movq    %rsp, %rbp
26      .cfi_def_cfa_register 6
27      pushq   %rbx
28      subq    $40, %rsp
29      .cfi_offset 3, -24
30      movq    %rdi, -24(%rbp)
31      movq    %rsi, -32(%rbp)
32      movl    %edx, -36(%rbp)
33      movq    -24(%rbp), %rax
34      movq    -32(%rbp), %rdx
35      movq    %rdx, %rsi
36      movq    %rax, %rdi
37  .LEHB0:
38      call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1ERKS4_@PLT
39  .LEHE0:
40      movq    -24(%rbp), %rax
41      movl    -36(%rbp), %edx
42      movl    %edx, 32(%rax)
43      leaq    .LC0(%rip), %rdi
44  .LEHB1:
45      call    puts@PLT
```

Рисунок 17: Ф-я ***_ZN5HumanC2ENSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEi*** *из файла* ***ex6_1.s***

```
300 _ZN16BasketballPlayerD2Ev:
301 .LFB1536:
302     .cfi_startproc
303     .cfi_personality 0x9b,DW.ref.__gxx_personality_v0
304     .cfi_lsda 0x1b,.LLSDA1536
305     endbr64
306     pushq   %rbp
307     .cfi_def_cfa_offset 16
308     .cfi_offset 6, -16
309     movq    %rsp, %rbp
310     .cfi_def_cfa_register 6
311     subq    $16, %rsp
312     movq    %rdi, -8(%rbp)
313     leaq    .LC4(%rip), %rdi
314     call    puts@PLT
315     movq    -8(%rbp), %rax
316     movq    %rax, %rdi
317     call    _ZN5HumanD2Ev
318     nop
319     leave
320     .cfi_def_cfa 7, 8
321     ret
322     .cfi_endproc
```

Рисунок 18: Ф-я ***_ZN16BasketballPlayerD2Ev*** *из файла* ***ex6_1.s***



```
 99  _ZN5HumanD2Ev:
100  .LFB1528:
101      .cfi_startproc
102      .cfi_personality 0x9b,DW.ref.__gxx_personality_v0
103      .cfi_lsda 0x1b,.LLSDA1528
104      endbr64
105      pushq   %rbp
106      .cfi_def_cfa_offset 16
107      .cfi_offset 6, -16
108      movq    %rsp, %rbp
109      .cfi_def_cfa_register 6
110      subq    $16, %rsp
111      movq    %rdi, -8(%rbp)
112      leaq    .LC1(%rip), %rdi
113      call    puts@PLT
114      movq    -8(%rbp), %rax
115      movq    %rax, %rdi
116      call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev@PLT
117      nop
118      leave
119      .cfi_def_cfa 7, 8
120      ret
121      .cfi_endproc
```

*Рисунок 19: Ф-я **_ZN5HumanD2Ev** из файла **ex6_1.s***

Задание 7

Теперь скажем несколько слов о полиморфизме в C++. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает понижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий.

Как мы можем понять по предыдущему опыту на уровне ассемблера полиморфизма не существует. В листинге также как и раньше просто будут существовать функции, имеющие разные названия. Выбор же конкретной функции, в зависимости от ситуации, будет возлагаться на компилятор.

Задание 8

Разберёмся со статическими полями. Для этого вернёмся к нашему классу **Porsche_911**, добавив в код некоторые изменения. Сделаем все поля публичными и добавим одно новое статическое поле:

static bool cabriolet

Оно будет отвечать за то, нужна ли нам машина с откидным верхом или без него. Ответ очевиден — с откидным, а значит внутри функции **main()** присвоим этому полю экземпляра **car1** значение **true**. Теперь у всех экземпляров этого класса данная переменная будет принимать такое значение (См. **ex8_1.cpp**).

Хотя мы можете получить доступ к статическим членам через разные объекты класса, но статические члены существуют, даже если объекты класса не созданы. Подобно глобальным переменным, они создаются при запуске программы и уничтожаются, когда программа завершает свое выполнение.

Следовательно, статические члены принадлежат классу, а не объектам этого класса. Поскольку **cabriolet** существует независимо от любых объектов класса, то доступ к нему осуществляется напрямую через имя класса и оператор разрешения области видимости. В данном случае, через **Porsche_911::cabriolet** (См. Рис. 20).

```
37  bool Porsche_911::cabriolet;
38
39  int main()
40  {
41      Porsche_911 car1{};
42      car1.set_fuel_volume(60);
43      car1.get_fuel_volume();
44      Porsche_911::cabriolet = true;
45  }
```

*Рисунок 20: Фрагмент программы **ex8_1.cpp***

Когда мы объявляем статическую переменную внутри тела класса, то мы сообщаем компилятору о существовании статической переменной, но не о её определении (аналогией является предварительное объявление). Поскольку статические переменные не являются частью отдельных объектов класса (они обрабатываются аналогично глобальным переменным и инициализируются при запуске программы), то мы должны явно определить статический член вне тела класса — в глобальной области видимости. Теперь посмотрим на ассемблерный листинг и рассмотрим то, что же произошло внутри.

```
138  _ZN11Porsche_9119cabrioletE:
139      .zero   1
140      .text
141      .globl  main
142      .type   main, @function
143  main:
144  .LFB1530:
145      .cfi_startproc
146      endbr64
147      pushq   %rbp
148      .cfi_def_cfa_offset 16
149      .cfi_offset 6, -16
150      movq    %rsp, %rbp
151      .cfi_def_cfa_register 6
152      subq    $32, %rsp
153      movq    %fs:40, %rax
154      movq    %rax, -8(%rbp)
155      xorl    %eax, %eax
156      leaq    -32(%rbp), %rax
157      movq    %rax, %rdi
158      call    _ZN11Porsche_911C1Ev
159      leaq    -32(%rbp), %rax
160      movl    $60, %esi
161      movq    %rax, %rdi
162      call    _ZN11Porsche_91115set_fuel_volumeEi
163      leaq    -32(%rbp), %rax
164      movq    %rax, %rdi
165      call    _ZN11Porsche_91115get_fuel_volumeEv
166      movb    $1, _ZN11Porsche_9119cabrioletE(%rip)
167      leaq    -32(%rbp), %rax
168      movq    %rax, %rdi
169      call    _ZN11Porsche_911D1Ev
170      movl    $0, %eax
171      movq    -8(%rbp), %rdx
172      xorq    %fs:40, %rdx
173      je  .L8
174      call    __stack_chk_fail@PLT
```

*Рисунок 21: Фрагмент файла **ex8_1.s***

Как мы можем видеть из рисунка 21, статические поля класса действительно инициализируются и обрабатываются как глобальные переменные.

У статических методов есть две интересные особенности. Во-первых, поскольку статические методы не привязаны к объекту, то они не имеют скрытого указателя ***this**. В этом есть смысл, так как указатель ***this** всегда указывает на объект, с которым работает метод. Статические методы могут не работать через объект, поэтому и указатель ***this** им не нужен.

Во-вторых, статические методы могут напрямую обращаться к другим статическим членам (переменным или функциям), но не могут напрямую обращаться к нестатическим членам. Это связано с тем, что нестатические члены принадлежат объекту класса, а статические методы — нет.

Задание 9

Теперь мы переопределим оператор сравнения для нашего класса **Porsche_911**. Для этого мы напишем новый метод **bool operator<**. Сравнивать наши машины мы будем по количеству топлива в бензобаке, то есть по полю **fuel_volume**. Весь код представлен на рисунке 22.

```cpp
#include <iostream>

class Porsche_911
{
public:
    const int power;
    const int max_speed;
    const int fuel_consumption;
    const int acceleration_0_100;
    const int fuel_capacity;
    int fuel_volume;

    Porsche_911() : power(385), max_speed(293),
                    fuel_consumption(9.4),
                    acceleration_0_100(4.2),
                    fuel_capacity(64),
                    fuel_volume(0)
    {}

    void set_fuel_volume(int new_fuel_volume)
    {
        fuel_volume = new_fuel_volume;
    }

    int get_fuel_volume()
    {
        return fuel_volume;
    }

    bool operator<(const Porsche_911& other2)
    {
        return this->fuel_volume < other2.fuel_volume;
    }

    ~Porsche_911()
    {
        printf("Destructor finished");
    }
};

int main()
{
    Porsche_911 car1{};
    Porsche_911 car2{};
    car1.set_fuel_volume(50);
    car2.set_fuel_volume(60);
    if (car1 < car2)
    {
        printf("It works!!!");
    }
}
```

*Рисунок 22: Программа **ex9_1.cpp***

```
_ZN11Porsche_911ltERKS_:
.LFB1527:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    movq    -8(%rbp), %rax
    movl    20(%rax), %edx
    movq    -16(%rbp), %rax
    movl    20(%rax), %eax
    cmpl    %eax, %edx
    setl    %al
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

*Рисунок 23: Ф-я **_ZN11Porsche_911ltERKS_** из файла **ex9_1.s***

Теперь посмотрим на ассемблерный листинг (См. Рис. 23 и 24). Внутри функции main (См. Рис. 24) теперь вызывается функция **_ZN11Porsche_911ltERKS_**. Если мы посмотрим на её реализацию (См. Рис. 23), то убедимся в том, что в этой функции действительно происходит сравнение, а результат сравнения помещается в регистр, с которым после происходят взаимодействия внутри main (См. Рис. 24, строка 181).

Исходя из представленных результатов, мы можем сказать, что листинг перегруженного оператора ничем не отличается от листинга обычной функции, а его вызов от вызова обычной функции.

```
145  main:
146  .LFB1531:
147      .cfi_startproc
148      .cfi_personality 0x9b,DW.ref.__gxx_personality_v0
149      .cfi_lsda 0x1b,.LLSDA1531
150      endbr64
151      pushq   %rbp
152      .cfi_def_cfa_offset 16
153      .cfi_offset 6, -16
154      movq    %rsp, %rbp
155      .cfi_def_cfa_register 6
156      pushq   %rbx
157      subq    $72, %rsp
158      .cfi_offset 3, -24
159      movq    %fs:40, %rax
160      movq    %rax, -24(%rbp)
161      xorl    %eax, %eax
162      leaq    -80(%rbp), %rax
163      movq    %rax, %rdi
164      call    _ZN11Porsche_911C1Ev
165      leaq    -48(%rbp), %rax
166      movq    %rax, %rdi
167      call    _ZN11Porsche_911C1Ev
168      leaq    -80(%rbp), %rax
169      movl    $50, %esi
170      movq    %rax, %rdi
171      call    _ZN11Porsche_91115set_fuel_volumeEi
172      leaq    -48(%rbp), %rax
173      movl    $60, %esi
174      movq    %rax, %rdi
175      call    _ZN11Porsche_91115set_fuel_volumeEi
176      leaq    -48(%rbp), %rdx
177      leaq    -80(%rbp), %rax
178      movq    %rdx, %rsi
179      movq    %rax, %rdi
180      call    _ZN11Porsche_911ltERKS_
181      testb   %al, %al
182      je  .L7
183      leaq    .LC1(%rip), %rdi
184      movl    $0, %eax
```

*Рисунок 24: Ф-я **main** из файла **ex9_1.s***

Задание 10

```
1  #include <iostream>
2
3  template <typename T>
4  T max(T a, T b)
5  {
6      if(a > b)
7      {
8          return a;
9      }
10     else
11     {
12         return b;
13     }
14 }
15
16 int main()
17 {
18     printf("%d\n", max(5, 6));
19     printf("%f\n", max(5.0, 6.0));
20 }
```

*Рисунок 25: Программа **ex10_1.cpp***

Поговорим о шаблонах. Напишем шаблон простой функции, которая возвращает максимум из двух чисел (См. Рис. 25).

Теперь посмотрим на ассемблерный листинг. Внутри функции **main** (См. Рис. 26) происходит обращение к двум функциям: **_Z3maxIiET_S0_S0_** и **_Z3maxIdET_S0_S0_**. Анализируя их реализации (См. Рис. 27 и 28) мы можем понять, что их суть совершенно идентична — они просто сравнивают два числа. Но первая работает с целыми числами, а вторая с числами с плавающей запятой. Отсюда мы можем сделать вывод о том, что генерация шаблонов в **C++** превращается в генерацию различных функций с разными типами данных.

```
17  main:
18  .LFB1523:
19      .cfi_startproc
20      endbr64
21      pushq   %rbp
22      .cfi_def_cfa_offset 16
23      .cfi_offset 6, -16
24      movq    %rsp, %rbp
25      .cfi_def_cfa_register 6
26      movl    $6, %esi
27      movl    $5, %edi
28      call    _Z3maxIiET_S0_S0_
29      movl    %eax, %esi
30      leaq    .LC0(%rip), %rdi
31      movl    $0, %eax
32      call    printf@PLT
33      movsd   .LC1(%rip), %xmm0
34      movq    .LC2(%rip), %rax
35      movapd  %xmm0, %xmm1
36      movq    %rax, %xmm0
37      call    _Z3maxIdET_S0_S0_
38      leaq    .LC3(%rip), %rdi
39      movl    $1, %eax
40      call    printf@PLT
41      movl    $0, %eax
42      popq    %rbp
43      .cfi_def_cfa 7, 8
44      ret
45      .cfi_endproc
```

*Рисунок 27: Ф-я **main** из файла **ex10_1.s***

```
51  _Z3maxIiET_S0_S0_:
52  .LFB1760:
53      .cfi_startproc
54      endbr64
55      pushq   %rbp
56      .cfi_def_cfa_offset 16
57      .cfi_offset 6, -16
58      movq    %rsp, %rbp
59      .cfi_def_cfa_register 6
60      movl    %edi, -4(%rbp)
61      movl    %esi, -8(%rbp)
62      movl    -4(%rbp), %eax
63      cmpl    -8(%rbp), %eax
64      jle .L4
65      movl    -4(%rbp), %eax
66      jmp .L5
```

*Рисунок 26: Ф-я **_Z3maxIiET_S0_S0_** из файла **ex10_1.s***

```
79  _Z3maxIdET_S0_S0_:
80  .LFB1761:
81      .cfi_startproc
82      endbr64
83      pushq   %rbp
84      .cfi_def_cfa_offset 16
85      .cfi_offset 6, -16
86      movq    %rsp, %rbp
87      .cfi_def_cfa_register 6
88      movsd   %xmm0, -8(%rbp)
89      movsd   %xmm1, -16(%rbp)
90      movsd   -8(%rbp), %xmm0
91      comisd  -16(%rbp), %xmm0
92      jbe .L11
93      movsd   -8(%rbp), %xmm0
94      jmp .L9
```

*Рисунок 28: Ф-я **_Z3maxIdET_S0_S0_** из файла **ex10_1.s***

```
1   #include <iostream>
2
3   template <typename T>
4   class Porsche_911
5   {
6   private:
7       T fuel_volume;
8   public:
9       Porsche_911() : fuel_volume(60)
10      {}
11
12      T get_fuel_volume()
13      {
14          return fuel_volume;
15      }
16  };
17
18  int main()
19  {
20      Porsche_911<int> car1{};
21      Porsche_911<float> car2{};
22      printf("%d\n", car1.get_fuel_volume());
23      printf("%f\n", car2.get_fuel_volume());
24  }
```

*Рисунок 29: Программа **ex10_2.cpp***

Теперь создадим шаблонный класс с уже родным для нас названием **Porsche_911** (См. Рис. 29). Теперь мы можем задать количество топлива не только целым числом, но ещё и числом с плавающей точкой, что и делает конструктор.

Посмотрим на ассемблерный листинг. Функция **main** представлена на рисунке 30. Из неё становится понятно, что мы получили 2 реализации конструктора (См. Рис. 31 и 32) и 2 реализации геттера (См. Рис. 33 и 34). Все они логически идентичны, а отличаются лишь типом. То есть генерация шаблонов классов происходит путём создания различных методов (или функций в терминах ассемблера) с разными типами.

```
17  main:
18  .LFB1524:
19      .cfi_startproc
20      endbr64
21      pushq   %rbp
22      .cfi_def_cfa_offset 16
23      .cfi_offset 6, -16
24      movq    %rsp, %rbp
25      .cfi_def_cfa_register 6
26      subq    $16, %rsp
27      movq    %fs:40, %rax
28      movq    %rax, -8(%rbp)
29      xorl    %eax, %eax
30      leaq    -16(%rbp), %rax
31      movq    %rax, %rdi
32      call    _ZN11Porsche_911IiEC1Ev
33      leaq    -12(%rbp), %rax
34      movq    %rax, %rdi
35      call    _ZN11Porsche_911IfEC1Ev
36      leaq    -16(%rbp), %rax
37      movq    %rax, %rdi
38      call    _ZN11Porsche_911IiE15get_fuel_volumeEv
39      movl    %eax, %esi
40      leaq    .LC0(%rip), %rdi
41      movl    $0, %eax
42      call    printf@PLT
43      leaq    -12(%rbp), %rax
44      movq    %rax, %rdi
45      call    _ZN11Porsche_911IfE15get_fuel_volumeEv
46      cvtss2sd    %xmm0, %xmm0
47      leaq    .LC1(%rip), %rdi
48      movl    $1, %eax
49      call    printf@PLT
50      movl    $0, %eax
51      movq    -8(%rbp), %rdx
52      xorq    %fs:40, %rdx
53      je  .L3
54      call    __stack_chk_fail@PLT
```

*Рисунок 30: Ф-я **main** из файла **ex10_2.s***

```
66  _ZN11Porsche_911IiEC2Ev:
67  .LFB1762:
68      .cfi_startproc
69      endbr64
70      pushq   %rbp
71      .cfi_def_cfa_offset 16
72      .cfi_offset 6, -16
73      movq    %rsp, %rbp
74      .cfi_def_cfa_register 6
75      movq    %rdi, -8(%rbp)
76      movq    -8(%rbp), %rax
77      movl    $60, (%rax)
78      nop
79      popq    %rbp
80      .cfi_def_cfa 7, 8
81      ret
82      .cfi_endproc
```

*Рисунок 31: Ф-я **_ZN11Porsche_911IiEC2Ev** из файла **ex10_2.s***

```
91  _ZN11Porsche_911IfEC2Ev:
92  .LFB1765:
93      .cfi_startproc
94      endbr64
95      pushq   %rbp
96      .cfi_def_cfa_offset 16
97      .cfi_offset 6, -16
98      movq    %rsp, %rbp
99      .cfi_def_cfa_register 6
100     movq    %rdi, -8(%rbp)
101     movq    -8(%rbp), %rax
102     movss   .LC2(%rip), %xmm0
103     movss   %xmm0, (%rax)
104     nop
105     popq    %rbp
106     .cfi_def_cfa 7, 8
107     ret
108     .cfi_endproc
```

*Рисунок 32: Ф-я **_ZN11Porsche_911IfEC2Ev** из файла **ex10_2.s***

```
117 _ZN11Porsche_911IiE15get_fuel_volumeEv:
118 .LFB1767:
119     .cfi_startproc
120     endbr64
121     pushq   %rbp
122     .cfi_def_cfa_offset 16
123     .cfi_offset 6, -16
124     movq    %rsp, %rbp
125     .cfi_def_cfa_register 6
126     movq    %rdi, -8(%rbp)
127     movq    -8(%rbp), %rax
128     movl    (%rax), %eax
129     popq    %rbp
130     .cfi_def_cfa 7, 8
131     ret
132     .cfi_endproc
```

*Рисунок 33: Ф-я **_ZN11Porsche_911IiE15get_fuel_volumeEv** из файла **ex10_2.s***

```
139 _ZN11Porsche_911IfE15get_fuel_volumeEv:
140 .LFB1768:
141     .cfi_startproc
142     endbr64
143     pushq   %rbp
144     .cfi_def_cfa_offset 16
145     .cfi_offset 6, -16
146     movq    %rsp, %rbp
147     .cfi_def_cfa_register 6
148     movq    %rdi, -8(%rbp)
149     movq    -8(%rbp), %rax
150     movss   (%rax), %xmm0
151     popq    %rbp
152     .cfi_def_cfa 7, 8
153     ret
154     .cfi_endproc
```

*Рисунок 34: Ф-я **_ZN11Porsche_911IfE15get_fuel_volumeEv** из файла **ex10_2.s***

Задание 11

Теперь поговорим про **enum**. Напишем небольшую программу (См. Рис. 35) где перечислим несколько вариантов автомобилей уже полюбившейся (я надеюсь) читателю марки.

```
1  #include <iostream>
2
3  enum Porsche_911
4  {
5      CARRERA,
6      CARRERA_CABRIOLET,
7      CARRERA_4,
8      CARRERA_4_CABRIOLET,
9      CARRERA_S,
10     CARRERA_S_CABRIOLET,
11     CARRERA_4S,
12     CARRERA_4S_CABRIOLET,
13     Targa_4,
14     Targa_4S,
15     CARRERA_GTS,
16     CARRERA_GTS_CABRIOLET,
17 };
18
19 int main()
20 {
21     int car1 = CARRERA;
22     int car2 = CARRERA_CABRIOLET;
23     int car3 = CARRERA_4;
24     int car4 = CARRERA_4_CABRIOLET;
25     int car5 = CARRERA_S;
26     int car6 = CARRERA_S_CABRIOLET;
27     int car7 = CARRERA_4S;
28     int car8 = CARRERA_4S_CABRIOLET;
29     printf("%d\n", car1);
30     printf("%d\n", CARRERA);
31 }
```

*Рисунок 35: Программа **ex11_1.cpp***

```
15 main:
16 .LFB1522:
17     .cfi_startproc
18     endbr64
19     pushq   %rbp
20     .cfi_def_cfa_offset 16
21     .cfi_offset 6, -16
22     movq    %rsp, %rbp
23     .cfi_def_cfa_register 6
24     subq    $32, %rsp
25     movl    $0, -32(%rbp)
26     movl    $1, -28(%rbp)
27     movl    $2, -24(%rbp)
28     movl    $3, -20(%rbp)
29     movl    $4, -16(%rbp)
30     movl    $5, -12(%rbp)
31     movl    $6, -8(%rbp)
32     movl    $7, -4(%rbp)
33     movl    -32(%rbp), %eax
34     movl    %eax, %esi
35     leaq    .LC0(%rip), %rdi
36     movl    $0, %eax
37     call    printf@PLT
38     movl    $0, %esi
39     leaq    .LC0(%rip), %rdi
40     movl    $0, %eax
41     call    printf@PLT
42     movl    $0, %eax
43     leave
44     .cfi_def_cfa 7, 8
45     ret
46     .cfi_endproc
```

*Рисунок 36: Ф-я **main** из файла **ex11_1.s***

Посмотрим на ассемблерный листинг данной программы (См. Рис. 36). Объявление перечислений не требует выделения памяти. Только когда переменная перечисляемого типа определена, тогда выделяется память для этой переменной.

Каждому перечислителю автоматически присваивается целочисленное значение в зависимости от его позиции в списке перечисления. По умолчанию, первому перечислителю присваивается целое число 0, а каждому следующему — на единицу больше, чем предыдущему.

Перечисляемые типы считаются частью семейства целочисленных типов, и компилятор сам определяет, сколько памяти выделять для переменных типа **enum**. По стандарту **C++** размер перечисления должен быть достаточно большим, чтобы иметь возможность вместить все перечислители. Но чаще всего размеры переменных **enum** будут такими же, как и размеры обычных переменных типа **int**.