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

Лабораторная 7

Изнаночка наших любимых плюсов

Рогозин Владимир
Группа Б03-106

Пункт 1: Методы и поля классов в листинге

Первым пунктом посмотрим на классы, их поля и методы. Напишем класс с парой полей, конструкторов, методов, посмотрим как формируются их названия.

```cpp
#include <iostream>

class TestClass {
private:
        int MyFirstPrivateVariable;
public:
        int MyFirstPublicVariable;

        TestClass(int a) {MyFirstPrivateVariable = a; MyFirstPublicVariable = 0;}

        TestClass(int priv, int publ): MyFirstPrivateVariable(priv), MyFirstPublicVariable(publ) {}

        int GetPriv() const {
                return MyFirstPrivateVariable;
        }
        int IncPriv() {
                MyFirstPrivateVariable++;
                return 1;
        }
        void IncPubl() {
                MyFirstPublicVariable++;
        }
};

int main() {
        TestClass A(1, 2);
        TestClass B(1);
        int a = A.IncPriv();
        A.IncPubl();
        int c = B.GetPriv();
        B.IncPubl();
        return 0;
}
```

Рис. 1: Простейший класс с двумя конструкторами, методами и полями

Сгенерируем листинг, посмотрим на названия методов в листинге.

```
_ZN9TestClassC2Ei:
```

(a) Первый конструктор

```
_ZN9TestClassC2Eii:
```

(b) Второй конструктор

```
call    _ZN9TestClassC1Eii
leaq    -16(%rbp), %rax
movl    $1, %esi
movq    %rax, %rdi
call    _ZN9TestClassC1Ei
```

Рис. 3: Вызов конструктора

В названиях конструкторов сначала идёт непонятная приписка _ZN9, затем идёт название класса (по сути название функции, так как это конструктор), потом C2E, затем приписка i и ii соответственно (int и int int), что явно показывает типы аргументов функции.

Примечательно то, что при вызове конструктора его название передаётся не таким, как оно записано при объявлении функции (там C2 в названии, а не C1). Как будто компилятор сам переименовал функцию, иначе бы он не смог создать экземпляр класса. Далее смотрим на методы класса.

```
_ZNK9TestClass7GetPrivEv:
```

(a) Константный метод

```
_ZN9TestClass7IncPrivEv:
```

(b) Неконстантный метод

Теперь два метода, оба не принимают аргументов, один из которых const. Начало имени такое же, только в константном методе добавляется буква K (будто бы хотели написать Konst ха-ха). После этого идёт название класса, цифра 7 и название метода. После названия функции, как и в случае с конструкторами, идёт буква E, потом v, что означает отсутствие аргументов функции.

```
_ZN9TestClass7IncPublEv:
```

Рис. 5: Неконстантный метод

В последнем методе все как и в предыдущих. Стоит ещё отметить, что в названиях функций никаким образом не указано что они должны (или ничего не должны) возвращать, указаны только типы данных аргументов функции.

Пункт 2: Как же он узнаёт с каким экземпляром класса работать?

В этом пункте, так как метод класса это просто функция, посмотрим как она понимает какой экземпляр класса её вызывает. Для этого возьмём программу с прошлого пункта, посмотрим на примере конструктора. Ниже приведён его листинг.

```
main:
.LFB1531:
        .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
        leaq    -24(%rbp), %rax
        movl    $2, %edx
        movl    $1, %esi
        movq    %rax, %rdi
        call    _ZN9TestClassC1Eii
```

(a) Листинг вызова конструктора

```
_ZN9TestClassC2Eii:
.LFB1526:
        .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)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movq    -8(%rbp), %rax
        movl    -12(%rbp), %edx
        movl    %edx, (%rax)
        movq    -8(%rbp), %rax
        movl    -16(%rbp), %edx
        movl    %edx, 4(%rax)
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Листинг конструктора

Видим, что при вызове метода класса неявным аргументом в функцию передаётся адрес того экземпляра класса, который вызвал этот метод. Передаётся адрес ячейки памяти через регистр *rdi*, именно таким образом машина понимает о каком именно объекте идёт речь.

Пункт 3: Раскрываем тайну загадочного слова *this*
Создадим функцию, использующую *this*.

```
int IncPriv() {
        this->MyFirstPrivateVariable++;
        return this->MyFirstPrivateVariable;
}
```

Рис. 7: Функция с *this*

Теперь смотрим на её листинг.

```
_ZN9TestClass7IncPrivEv:
.LFB1529:
        .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    -8(%rbp), %rax
        movl    (%rax), %eax
        leal    1(%rax), %edx
        movq    -8(%rbp), %rax
        movl    %edx, (%rax)
        movq    -8(%rbp), %rax
        movl    (%rax), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Рис. 8: Листинг функции выше

Из ассемблерного листинга видно, что *this* работает как указатель, который в данном случае, чтобы функция вернула значение поля экземпляра класса, передаёт адрес объекта, потом по этому адресу уже берётся значение переменной конкретного экземпляра класса, и функция возвращает это значение.

Пункт 4: Конструкторы и деструкторы

Посмотрим где и как вызываются конструкторы/деструкторы для локальных, глобальных, а также переменных, созданных в куче.

Сначала разберёмся с локальными переменными. При вызове конструктор принимает ещё один неявный аргумент – адрес ячейки памяти где будет храниться объект (а точнее его первое поле). Передаётся этот адрес через регистр *rdi*. С деструктором также, он дополнительно принимает адрес экземпляра класса, и если было какое-либо выделение памяти, то освобождает её.

Если создавать объект в куче, то ситуация такая же, только на вход конструктору/деструктору приходит адрес памяти, которая выделяется в куче.

Деструкторы вызываются в порядке, обратном конструкторам. Если экземпляр находится в куче, то сначала вызывается деструктор, затем срабатывает *delete* (если он конечно присутствует в коде).

```cpp
#include <iostream>

class TestClass {
private:
        int MyFirstPrivateVariable;
        int* HeapVariables;
public:
        int MyFirstPublicVariable;

        TestClass(int priv, int publ): MyFirstPrivateVariable(priv), MyFirstPublicVariable(publ) {
                HeapVariables = new int[5];
        }
        ~TestClass() {delete [] HeapVariables;}
};

TestClass GlobalClass(111, 111);


int main() {
        TestClass LocalClass(11, 11);
        TestClass* HeapClass = new TestClass(1, 1);
        delete HeapClass;
        return 0;
}
```

Рис. 9: Код программы

```
_ZN9TestClassC2Eii:
.LFB1523:
        .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
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movq    -8(%rbp), %rax
        movl    -12(%rbp), %edx
        movl    %edx, (%rax)
        movq    -8(%rbp), %rax
        movl    -16(%rbp), %edx
        movl    %edx, 16(%rax)
        movl    $20, %edi
        call    _Znam@PLT
        movq    %rax, %rdx
        movq    -8(%rbp), %rax
        movq    %rdx, 8(%rax)
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(a) Листинг конструктора

```
_ZN9TestClassD2Ev:
.LFB1526:
        .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
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rax
        movq    8(%rax), %rax
        testq   %rax, %rax
        je      .L4
        movq    -8(%rbp), %rax
        movq    8(%rax), %rax
        movq    %rax, %rdi
        call    _ZdaPv@PLT
.L4:
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Листинг деструктора

С глобальной переменной всё чуть иначе. Она, как и положено, объявляется в начале листинга. Но сам вызов конструктора и деструктора описан в конце листинга, после *main*'а, так ещё и вызываются дополнительные непонятные функции. Конструктор вызывается как функция с помощью команды *call*, а деструктор, как обычная переменная, кладётся в регистр *rdi* и, судя по всему, потом с этим регистром что-то делает непонятная функция, которая вызывается после.

```
.LFE1526:
        .size   _ZN9TestClassD2Ev, .-_ZN9TestClassD2Ev
        .weak   _ZN9TestClassD1Ev
        .set    _ZN9TestClassD1Ev,_ZN9TestClassD2Ev
        .globl  GlobalClass
        .bss
        .align 16
        .type   GlobalClass, @object
        .size   GlobalClass, 24
GlobalClass:
        .zero   24
```

(a) Объявление глобальной переменной

```
call    __cxa_atexit@PLT
movl    $111, %edx
movl    $111, %esi
leaq    GlobalClass(%rip), %rdi
call    _ZN9TestClassC1Eii
leaq    __dso_handle(%rip), %rdx
leaq    GlobalClass(%rip), %rsi
leaq    _ZN9TestClassD1Ev(%rip), %rdi
call    __cxa_atexit@PLT
```

(b) Вызов конструктора и деструктора

Ещё стоит отметить, что если поля идут в порядке *int*, *int**, *int*, то, так как размер указателя равен 8-ми байтам, идёт выравнивание байтов, и каждому *int*'у выделяется не 4, а 8 байт, всего один объект занимает 24 байта. если же поля идут в порядке *int**, *int*, *int*, то тогда каждой целочисленной переменной выделяется 4 байта и суммарное место, занимаемое экземпляром класса, сокращается до 16 байтов. То есть, записывая поля класса в правильном порядке, можно сэкономить немалое количество памяти.

Пункт 5: Инкапсуляция. *Public* и *private* поля и методы

Посмотрим как выглядит инкапсуляция в листинге. Для этого добавим приватный метод, сгенерируем листинг.

```
#include <iostream>

class TestClass {
private:
        int MyFirstPrivateVariable;
        void IncPriv() {MyFirstPrivateVariable++;}
public:
        int MyFirstPublicVariable;

        TestClass(int priv, int publ): MyFirstPrivateVariable(priv), MyFirstPublicVariable(publ) {}

        void IncAll() {
                IncPriv();
                MyFirstPublicVariable++;
        }
};


int main() {
        TestClass LocalClass(11, 11);
        LocalClass.IncAll();
        return 0;
}
```

Рис. 12: Класс с приватным полем и методом

Из листинга приватного метода который меняет *private* поле видно, что с точки зрения ассемблера разницы никакой между *private* и *public* нету. Метод меняет поле точно также, как если бы оно было публичным. В функцию неявно передаётся указатель на объект, затем она получает доступ к любому, будь то *private* или *public*, полю.

```
_ZN9TestClass7IncPrivEv:
.LFB1522:
        .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    -8(%rbp), %rax
        movl    (%rax), %eax
        leal    1(%rax), %edx
        movq    -8(%rbp), %rax
        movl    %edx, (%rax)
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Рис. 13: Листинг *private* функции, меняющей *private* поле

Пункт 6: Наследование

Напишем некоторую иерархию классов, затем создадим экземпляр дочернего класса и посмотрим, как раскрываются конструкторы и в каком порядке лежат в памяти поля.

```
#include <iostream>

class Animal {
protected:
        bool isPredator;
public:
        Animal(bool predator): isPredator(predator) {}
};

class Predator: public Animal {
protected:
        int numberOfTeeth;
        int numberOfLegs;
public:
        Predator(int teeth, int legs): Animal(true) {
                numberOfTeeth = teeth;
                numberOfLegs = legs;
        }
};

class Cat: public Predator {
protected:
        bool isItCute = true;
        bool isItPet;
public:
        Cat(bool isPet): Predator(30, 4), isItPet(isPet) {}
};


int main() {
        Cat kitty{true};
        return 0;
}
```

Рис. 14: Код программы

```
_ZN8PredatorC2Eii:
.LFB1526:
        .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
        movq    %rdi, -8(%rbp)
        movl    %esi, -12(%rbp)
        movl    %edx, -16(%rbp)
        movq    -8(%rbp), %rax
        movl    $1, %esi
        movq    %rax, %rdi
        call    _ZN6AnimalC2Eb
        movq    -8(%rbp), %rax
        movl    -12(%rbp), %edx
        movl    %edx, 4(%rax)
        movq    -8(%rbp), %rax
        movl    -16(%rbp), %edx
        movl    %edx, 8(%rax)
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(a) Конструктор класса Predator

```
_ZN3CatC2Eb:
.LFB1529:
        .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
        movq    %rdi, -8(%rbp)
        movl    %esi, %eax
        movb    %al, -12(%rbp)
        movq    -8(%rbp), %rax
        movl    $4, %edx
        movl    $30, %esi
        movq    %rax, %rdi
        call    _ZN8PredatorC2Eii
        movq    -8(%rbp), %rax
        movb    $1, 12(%rax)
        movq    -8(%rbp), %rax
        movzbl  -12(%rbp), %edx
        movb    %dl, 13(%rax)
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Конструктор класса Cat

Из листинга видим, что в конструкторе дочернего класса сначала вызывается конструктор родительского, а только после этого инициализируются поля дочернего класса. И поэтому в памяти сначала будут лежать поля базового класса, затем дочернего ему класса, затем дочернего дочернего и так далее до класса, экземпляр которого создаётся. Это также объясняет порядок срабатывания конструкторов – от базового и вниз по иерархии.

Пункт 7: Откуда же полиморфизм

Ну на этот вопрос теперь ответить не сложно. Возвращаясь к первому пункту этой лабы, понимаем, что названия функций, которые генерирует компилятор, включают в себя информацию о типах данных которые принимают функции в качестве аргументов. Таким образом, функция то не только название, но и аргументы, которые она принимает. Отсюда и получаем полиморфизм.

Пункт 8: *Static* поля и методы

Посмотрим как выглядят *static* поля и методы в ассемблере. Для этого добавим в класс такое поле и метод. Код приведён ниже. Сгенерируем листинг и проанализируем его.

Из листинга видим, что статиковые поля это что-то среднее между глобальными переменными и нестатиковыми полями класса. Объявляется такая переменная как глобальная, но в названии указана принадлежность к конкретному классу, поэтому можно создавать в разных классах статические переменные с одинаковым названием. Компилятор сам отслеживает законность/незаконность доступа к переменной вне класса (публичное ли поле или нет). Хранится, понятное дело, одна копия поля для всех экземпляров класса, статический метод, который есть обычная функция, обращается к этому полю как к глобальной переменной, поэтому при вызове этого метода нет нужды передавать адрес объекта, который вызывал статический метод – адрес и не передаётся, а значит *static* метод не может взаимодействовать с нестатическими полями и методами класса.

```cpp
#include <iostream>

class TestClass {
protected:
        static int someData1;
        int someData2;
public:
        TestClass(int var): someData2(var) {}

        static void setStatic(int stat) {
                someData1 = stat;
        }
};

int TestClass::someData1;

int main() {
        TestClass object{2};
        object.setStatic(22);
        return 0;
}
```

Рис. 16: Класс со статическим полем и методом

```
        .globl  _ZN9TestClass9someData1E
        .bss
        .align 4
        .type   _ZN9TestClass9someData1E, @object
        .size   _ZN9TestClass9someData1E, 4
_ZN9TestClass9someData1E:
        .zero   4
```

Рис. 17: Статическое поле листинг

```
_ZN9TestClass9setStaticEi:
.LFB1525:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, _ZN9TestClass9someData1E(%rip)
        nop
        popq    %rbp
```

Рис. 18: Статический метод листинг

Пункт 9: Перегруженные операторы

Перегрузим оператор сложения и посмотрим как он выглядит в ассемблерном листинге.

```
#include <iostream>

class TestClass {
public:
        int someData;
public:
        TestClass(int var): someData(var) {}

        TestClass operator+(const TestClass& right) const {
                TestClass result {this->someData + right.someData};
                return result;
        }
};

int main() {
        TestClass obj1{2};
        TestClass obj2{3};
        TestClass sum = obj1 + obj2;
        return 0;
}
```

Рис. 19: Перегрузка оператора

```
_ZNK9TestClassplERKS_:
.LFB1525:
        .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    %rdi, -24(%rbp)
        movq    %rsi, -32(%rbp)
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movq    -24(%rbp), %rax
        movl    (%rax), %edx
        movq    -32(%rbp), %rax
        movl    (%rax), %eax
        addl    %eax, %edx
        leaq    -12(%rbp), %rax
        movl    %edx, %esi
        movq    %rax, %rdi
        call    _ZN9TestClassC1Ei
        movl    -12(%rbp), %eax
        movq    -8(%rbp), %rcx
        xorq    %fs:40, %rcx
        je      .L4
        call    __stack_chk_fail@PLT
```

Рис. 20: Листинг перегруженного оператора

Вызываются операторы как функции, и из листинга видно, что это действительно просто функции. В названии функции также указано имя оператора (*pl* означает плюс), а также типы данных аргументов.

Пункт 10: Шаблоны

В этом пункте посмотрим на шаблонизацию функций и классов.

```cpp
#include <iostream>

template<typename T>
class TestClass {
protected:
        T someData;
public:
        TestClass(T var): someData(var) {}
};

template<typename T>
T func(T arg) {
        return arg;
}

int main() {
        char c = 'a';
        TestClass<int> obj1{2};
        TestClass<char> obj2{c};
        std::cout << func<int>(6) << std::endl;
        return 0;
}
```

Рис. 21: Шаблонизированные класс и фунция

```
_ZN9TestClassIiEC2Ei:
.LFB1762:
        .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)
        movl    %esi, -12(%rbp)
        movq    -8(%rbp), %rax
        movl    -12(%rbp), %edx
        movl    %edx, (%rax)
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(a) Конструктор класса с шаблоном *int*

```
_ZN9TestClassIcEC2Ec:
.LFB1765:
        .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)
        movl    %esi, %eax
        movb    %al, -12(%rbp)
        movq    -8(%rbp), %rax
        movzbl  -12(%rbp), %edx
        movb    %dl, (%rax)
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Конструктор класса с шаблоном *char*

Получаем, что при создании экземпляра, когда шаблон раскрывается, компилятор создаёт конструктор и все функции, уже зная что будет вместо шаблона, поэтому объекты класса с различным раскрытием шаблонов как бы являются экземплярами различных классов. То есть генерируется столько различных классов, сколько было уникальных раскрытий шаблона. Это же видно и из названия конструктора. После

названия класса идёт буква *I*, что судя по всему указывает на то, что это шаблонный класс (или функция), потом идёт информация о том, что конкретно подставляется в шаблон, в конце, как и у обычных функций, указаны типы данных аргументов функции.

Пункт 11: *Rvalue*-ссылки

Ещё раз взглянем на *rvalue*-ссылки в ассемблерном представлении.

```cpp
#include <iostream>

int func(int& arg) {
        return arg + 1;
}

int func(int&& arg) {
        return arg + 22;
}

int main() {
        int a = 11;
        int b = func(a);
        int c = func(a + 1);
        return 0;
}
```

Рис. 23: Функции, принимающие ссылку и *rvalue*-ссылку

```
_Z4funcRi:
```

(a) Обычная ссылка

```
_Z4funcOi:
```

(b) *Rvalue*-ссылка

Как и должно быть в конце названия функции указан тип данных аргументов, в первом случае это обычная *int* ссылка *Ri*, *rvalue*-ссылка на *int* обозначается через *Oi*. Перед вызовом функции с аргументом *rvalue*-ссылкой результат промежуточной операции кладётся в стек как локальная переменная, затем уже через регистр передаётся в функцию, именно благодаря этому и можно экономить время, избегая глубокого копирования там, где оно не нужно.

Пункт 12: *Enum*

Посмотрим на конструкцию *enum* в ассемблерном листинге.

```cpp
#include <iostream>

enum seasons {
        summer = 1,
        autumn,
        winter,
        spring
};

int main() {
        seasons Season = autumn;
        std::cout << Season << std::endl;
        return 0;
}
```

(a) Код с *enum* 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    $16, %rsp
        movl    $2, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    _ZSt4cout(%rip), %rdi
```

(b) Ассемблерный листинг программы

Как видно из листинга, компилятор сам заменяет переменную на соответствующую ей числовую константу, то есть в ассемблерном листинге получаем переменную типа *int* (в данном случае).