ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

**Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»**

Московский институт электроники и математики им. А.Н. Тихонова

Департамент прикладной математики

Отчёт
по ребусу №1
по курсу «Алгоритмизация и программирование»

ФИО студента	Номер группы	Дата
Кейер Александр Петрович	БПМ-231	7 октября 2024 г.

Москва, 2024

Вопрос 1

main.cpp

```
#include "foo.h"
int main() {
  foo();
  return 0;
}

```

foo.h

```
void foo() {
  /* some code */
}

```

foo.cpp

```
#include "foo.h"

/* some code */

```

Проблема: в данном случае функция foo реализована в заголовочном файлу foo.h - это не очень хорошо, поскольку если этот заголовочный файл будет включаться в несколько единиц трансляции (в ребусе включается в main.cpp и в foo.cpp), то компоновщик во время линковки выбросит ошибку, ссылающуюся на ODR (One Definition Rule), которое гласит, что функция не может быть определена более одного раза (единственный источник правды).

Решение 1: если переписать foo.h в следующий вид,

```
inline void foo() {
  /* some code */
}

```

то описанная проблема пропадет. То есть:

1. Этап препроцессинга - все также вставим вместо #include ”foo.h” все содержимое файла (но уже с inline)

2. Этап трансляции (компиляции) - не совсем понятно как, но, видимо, благодаря inline здесь на уровне бинарных файлов единиц трансляции каким-то образом происходит разрешение коллизии.

3. Этап линковки - здесь уже не возникает ошибок, ссылающихся на ODR

Решение 2: хорошей практикой является лишь описание сигнатуры функции внутри .h файлов, а реализация этой функции должна попадать уже в зону ответственности соответствующей единицы трансляции.

Интересно: если проводить последовательно (руками) препроцессинг, трансляцию и линковку, то ошибка возникает. Если же давать компилятору g++ самостоятельно осуществить эти этапы, то ошибки не возникает, но на уровне ассемблерного кода можно обнаружить отличие в виде двух команд (см. фото ниже). Подробнее изучить, зачем это нужно, можно по ссылке.

```
        .file   "01_Keyer_BAM231.cpp"
        .text
        .globl  __Z3fooii
        .def    __Z3fooii;      .scl    2;      .type   32;     .endef
__Z3fooii:
LFB0:
        .cfi_startproc
        pushl   %ebp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        popl    %ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
LFE0:
        .def    ___main;        .scl    2;      .type   32;     .endef
        .globl  _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
LFB1:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
"main.s" 49L, 897B                                    1,1-8         Top
```

```
        .file   "01_Keyer_BAM231.cpp"
        .section        .text$_Z3fooii,"x"
        .linkonce discard
        .globl  __Z3fooii
        .def    __Z3fooii;      .scl    2;      .type   32;     .endef
__Z3fooii:
LFB0:
        .cfi_startproc
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        popl    %ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
LFE0:
        .def    ___main;        .scl    2;      .type   32;     .endef
        .text
        .globl  _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
LFB1:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
".\inline_main.s" 51L, 947B                           27,6          Top
```

Вопрос 2

int a = 0, b = 1; // Объявили и инициализировали на стеке две целочисленные переменные a и b.

int result = (10, (a = 3) + (b = 4)); // Используем оператор "запятая". Сначала вычисляем левый операнд - там 10, поэтому отбрасываем результат и переходим ко второму операнду. Присываиваем переменной a значение 3. Присываиваем переменной b значение 4. Вычисляем правый операнд = 3 + 4 = 7. В итоге: a = 3, b = 4, result = 7.

Вопрос 3

```
1     int arr[3] = { 0, 2, 3 };
2
3     if (arr[0] == *arr || (arr[0] = 1))
4       arr[1] = 0;
5
```

int arr[3] = 0, 2, 3 ; // Создали на стеке статический массив из 3 элементов.

if (arr[0] == *arr || (arr[0] = 1)) // arr[0] - взяли первый элемент массива. *arr взяли значения указателя arr, который указывает на первый элемент массива. Итого левая часть условия arr[0] == *arr верна. Оператор "ИЛИ"|| останавливается, когда получаем первый истинный результат, поэтому до правой части (arr[0] = 1) мы не доходим.

arr[1] = 0; // Условие оператора if ИСТИНА, поэтому сдвигаем указатель arr на 1 вправо и обновляем значение полученного указателя на 0. Итого получаем массив 0, 0, 3.