extra2. Размещение объекта в памяти в случае ромбовидного наследования от полиморфных типов. Независимость проблемы ромбовидного наследования от наличия виртуальных функций. Количество указателей на vtable. Устройство таблицы для Mother-in-Son и для Father-in-Son, разница между этими таблицами. Понятие top_offset. Устройство vtable в случае виртуального наследования с виртуальными функциями. Значения virtual_offset и top_offset.

Классы, у которых есть виртуальные функции, называются полиморфными. Пусть есть типичное ромбовидное наследование, но только все классы имеют virtual функцию.

```
struct Granny {
    virtual void fg();
    int g;
};

struct Mother : public Granny {
    virtual void fm();
    int m;
};
struct Father : public Granny {
    virtual void ff();
    int f;
};
struct Son : Mother, Father {
    virtual void fs();
    int s;
};
```

Заметим, что если в такой конструкции у Son попытаться вызвать fg(), то проблема ромбовидного наследования останется, так как компилятор:

Сначала выбирает всех кандидатов на fg()

Потом понимает, как из них — наилучшая перегрузка

Затем проверяет приватность

И только потом заканчивает компиляцию и исполняет программу

Не трудно заметить, что, независимо от виртуальности функции gf(), CE будет уже на втором этапе, так как есть две идентичные перегрузки — из мамы и из папы.

Как мы помним без виртуальности их расположение в памяти было бы:

$[g][m][g][f][s]$

Но теперь у нас должны появиться указатели на vtable, вероятнее всего, где-нибудь в начале. Но сколько же конкретно их нам понадобится? 1? 2? 3? Попробуем реализовать лишь с одним:

$[vptr][g][m][g][f][s]$

Но что же тогда будет, если мы кастанем сына к отцу (Son sptr; Father* fptr = sptr;)? По тем правилам, что мы изучали ранее мы должны сузиться до [g][f][s], у которого vptr в начале нету. Но если мы захотим вызвать ff() — компилятор попробует найти vptr для виртуальной функции, но не сможет. Потому корректная реализация такая:

$[vptr1][g][m][vptr2][g][f][s]$

Но остается загадка: будут ли vptr1 и vptr2 указывать на одну и ту же таблицу? Ответ — нет. Но почему? Пусть vptr1 и vptr2 равны, но давайте вспомним о dynamic_cast. Если указатели одинаковы, то type_id будет один и тот же — нет никакого способа различить Mother и Father. Как следствие dynamic_cast не сможет понять, в чем разница между кастом к Mother и кастом к Father.

Разберемся чуть подробнее с устройством таблицы для Mother-in-Son и для Father-in-Son. Вспомним устройство vtable:

$[type\_info][\&S :: fg]...$

Где vptr указывает ровно на первый блок [&S::fg]. Собственно разница между Mother-in-Son и Father-in-Son будет в том, что в одном классе написано "Я — Mother, чтоб кастануть меня к Mother — делать ничего не надо, а чтоб кастануть к Father — сдвинься на 16 байт вправо". И аналогично во втором.

Теперь рассмотрим случай виртуального наследования:

```
struct Granny {
    virtual void foo();
    int g;
};

struct Mother : virtual public Granny {
    int m;
};
struct Father : public virtual Granny {
    int f;
};
struct Son : Mother, Father {
    int s;
};
```

Тогда устройство в памяти такое:

$[vptr][m][vptr][f][s][g]$

Как же теперь устроен vtable? Вот так:

$[virtual\_offset][top\_offset][type\_info][foo]$

Что же такое top_offset? Это то, насколько байт вправо сдвинута текущая функция от начала. То есть для Father это было бы 16, а для Mother — 0. То есть это помогает понять, куда двигаться, если нас кастуют к другому классу в иерархии, или хотят вызвать функцию у Granny. Для virtual_offset — идея аналогична. Это число, которое надо пройти, чтоб дойти до виртуальных классов.

P.S.: на лекции было сказано, что это устройство для g++. Для других компиляторов, вероятнее всего, если и не тоже и самое, то что-то аналогичное.