ЛЕКЦИЯ 18
МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЯ

ЛЕКТОР ФУРМАВНИН С.А.

CASE STUDY: MYARRAY

Допустим, у вас есть интерфейс **IBuffer**, использованный в **Array**

```
class Array {
protected:
  IBuffer *buf_;
public:
  explicit Array(IBuffer *buf) : buf_(buf) {}
  // something interesting
};
```

Вы реализовали ваш собственный превосходный класс `MyBuffer`, наследующий от `IBuffer`

Как написать класс `MyArray`, наследующий от Array и использующий `MyBuffer`?

ПЕРВАЯ ПОПЫТКА: ДВОЙНОЕ ВКЛЮЧЕНИЕ

Мы можем просто сохранить `MyBuffer` внутри

```
class MyArray : Array {
protected:
  MyBuffer *mbuf_;
public:
  explicit MyArray(int size) : mbuf_(size), Array(mbuf_) {}
  // something MORE interesting
};
```

Это не будет работать, так как буфер нельзя инициализировать раньше базового класса

Но и переставить инициализаторы местами мы не можем

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

```
class MyArray : protected MyBuffer, public Array {
public:
  explicit MyArray(int size) : MyBuffer(size), Array(???) {}
  // something MORE interesting
};
```

Синтаксис наследования: все базовые классы с модификаторами через запятую

Здесь наследование защищенное потому что:

- мы не хотим прятать защищенную часть MyBuffer и не можем сделать его приватным
- Мы не хотим показывать MyBuffer наружу и не можем сделать его публичным

Но есть небольшая проблема: что написать вместо знаков вопроса?

РЕШЕНИЕ ПРОКСИ-КЛАСС

```
struct ProxyBuf {
  MyBuffer buf;
  explicit ProxyBuf(int size) : buf(size) {}
};
class MyArray : protected ProxyBuf, public Array {
public:
  explicit MyArray(int size) : ProxyBuf(size),
                               Array(&ProxyBuf::buf) {}
  // something more interesting
};
```

Теперь всё срастается

ОБСУЖДЕНИЕ СОМНИТЕЛЬНОЙ ИДЕИ

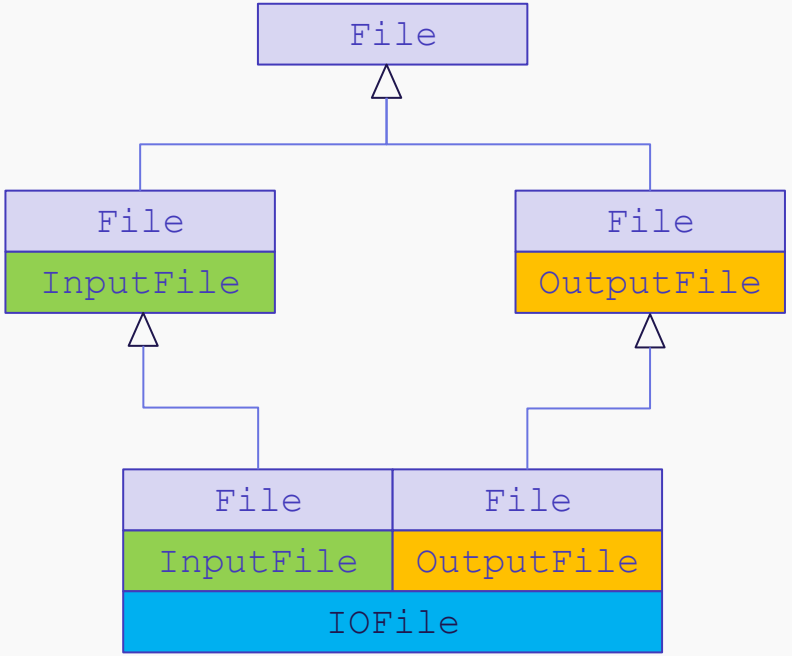
Множественное наследование интерфейса не вызывает вопросов

```
class Man : public ITwoLegs, public INoFeather {
public:
  // Two-Legs methods
public:
  // NoFeather methods
// other
};
```

Но в довольно большом количестве языков запрещено множественное наследование реализации. И сделано это неспроста.

РОМБОВИДНЫЕ СХЕМЫ

```
struct File {int a; };
struct InputFile :
  public File( int b; };
struct OutputFile :
  public File { int c; };
struct IOFile :
  public InputFile,
  public OutputFile {
    int d; };
```



ПРОБЛЕМЫ РОМБОВИДНЫХ СХЕМ

Поскольку в объект нижнего класса входят два верхних подобъекта, доступ к переменным неочевиден.

```
IOFile f{11};
int x = f.a; // CE
int y = f.InputFile::a; // OK, но это боль
```

Кроме того, в принципе `f.IntputFile::a` и `f::OutputFile::a` могут разойтись в процессе работы.

В качестве решения хотелось бы иметь один экземпляр базового класса сколькими бы путями он ни пришел в производный
Такие базовые классы называются **виртуальными**

ВИРТУАЛЬНЫЕ БАЗОВЫЕ КЛАССЫ

Виртуальное наследование – это поддержка в языке

```
struct File { …. };
struct InputFile : virtual public File { …. };
struct OutputFile : virtual public File { …. };
struct IOFile : public InputFile, public OutputFile { …. };

IOFile f{11};
int x = f.a; // OK
int y = f.InputFile::a; // OK, тоже работает
```

Конечно, тут сразу возникает масса вопросов

ВИРТУАЛЬНЫЕ БАЗОВЫЕ КЛАССЫ

- Что если класс виртуальный не по всем путям?
- Что, если базовый класс виртуальный, но в нижнем подобъекте всего один?
- В каком порядке и когда конструируются обычные и виртуальные подобъекты?

ВИРТУАЛЬНЫЕ БАЗОВЫЕ КЛАССЫ

- Что если класс виртуальный не по всем путям?

никаких проблем, вниз попадет один со всех виртуальных путей и по одному с каждого невиртуального пути

- Что, если базовый класс виртуальный, но в нижнем подобъекте всего один?

никаких проблем, можно хоть все базовые классы всегда делать виртуальными, будет работать, как обычное наследование*

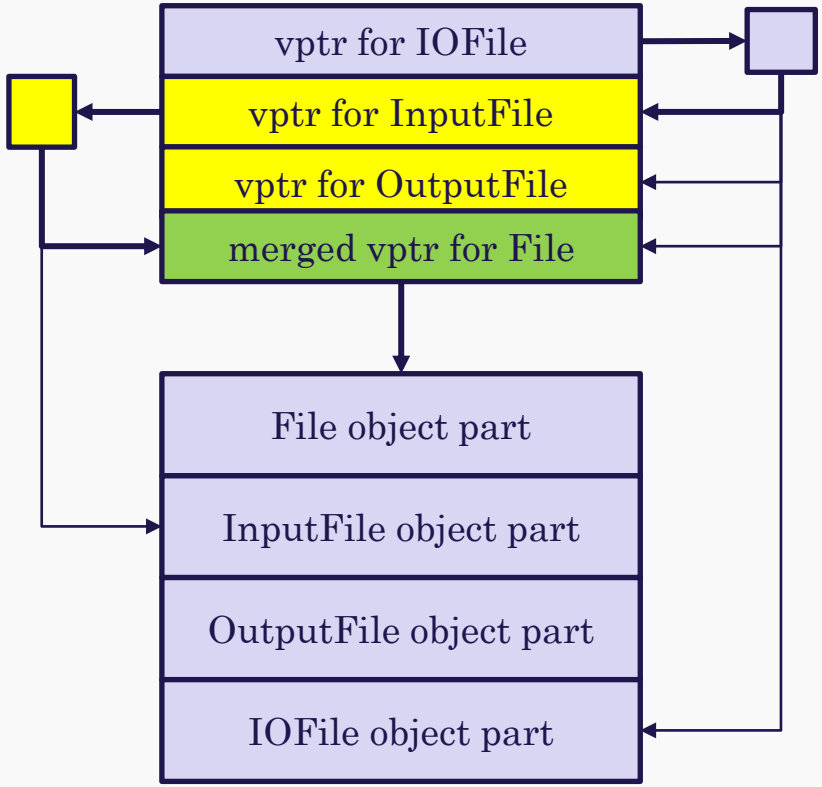
- В каком порядке и когда конструируются обычные и виртуальные подобъекты?

такое чувство, что сначала должны конструироваться все виртуальные, а потом все остальные

ВИРТУАЛЬНЫЕ БАЗОВЫЕ КЛАССЫ

Вызов виртуальной функции при множественном наследовании должен пройти через дополнительный уровень диспетчеризации

А при виртуальном наследовании через **еще один дополнительный уровень** из-за того, что таблицы для виртуальных подобъектов должны быть отдельно смержены



СПИСКИ ИНИЦИАЛИЗАЦИИ

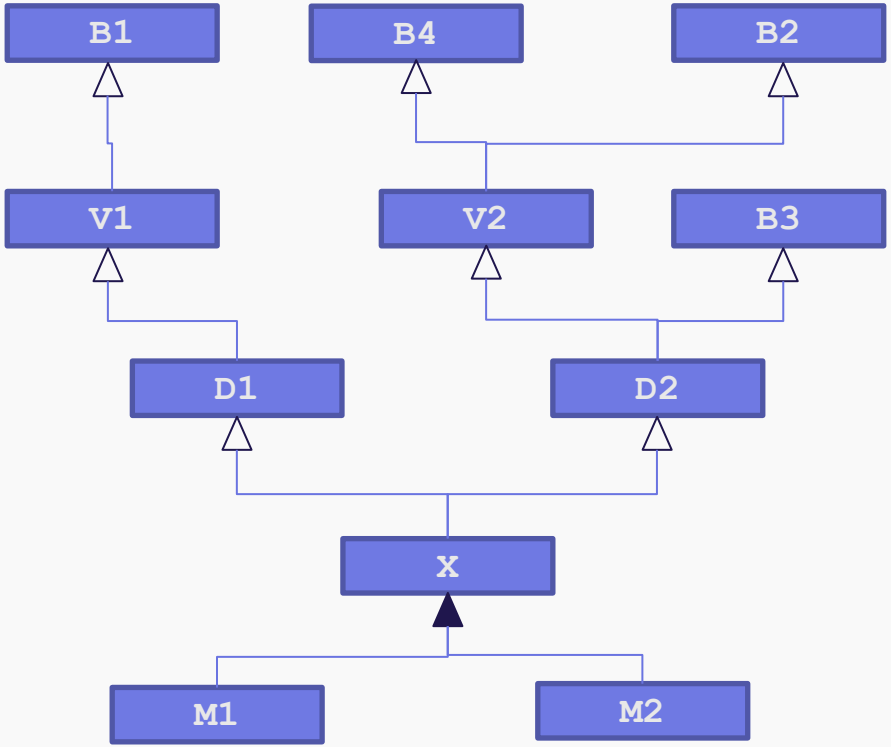
Виртуальный базовый класс обязан появиться в списке инициализации самого нижнего подобъекта

```
struct InputFile : virtual public File {
  InputFile() : File(smth1} {};
};
struct OutputFile : virtual public File {
  OutputFile() : File(smth2) {};
};
struct IOFile : public InputFile, public OutputFile {
  IOFile() : File(smth3), InputFile(), OutputFile() {};
};

IOfile f; // вызовет File(smth3)
```

СПИСКИ ИНИЦИАЛИЗАЦИИ

Порядок инициализации?



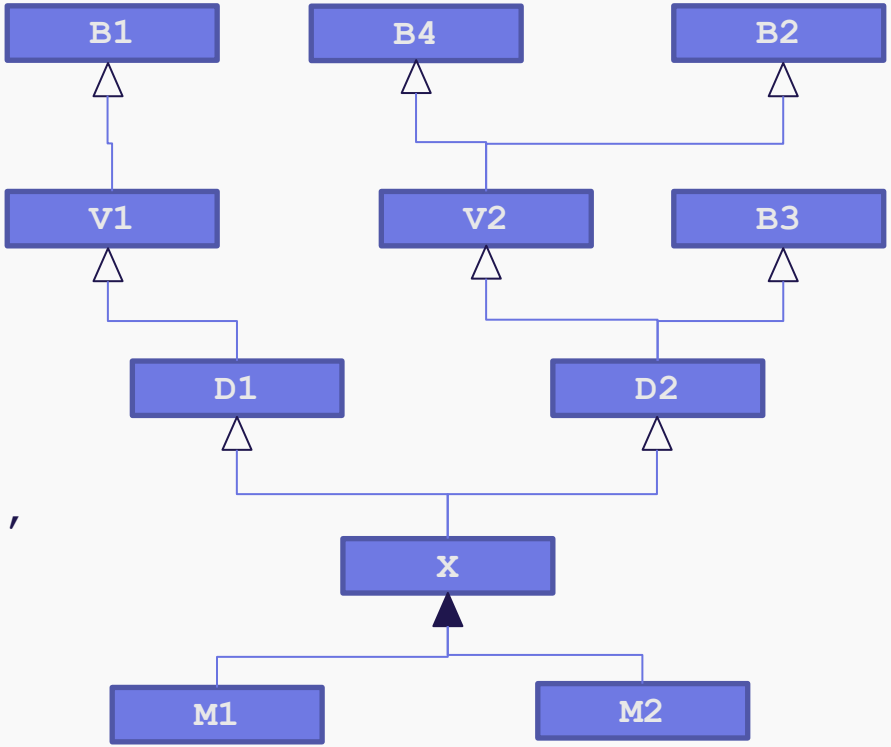
СПИСКИ ИНИЦИАЛИЗАЦИИ

Порядок инициализации?

```
V1: B1, V1
V2: B4, B2, V2
D1: D1
D2: B3, D2
X: M1, M2, X
```

Итого:

```
B1, V1, B4, B2, V2, D1, B3,
D2, M1, M2, X
```



ОБСУЖДЕНИЕ

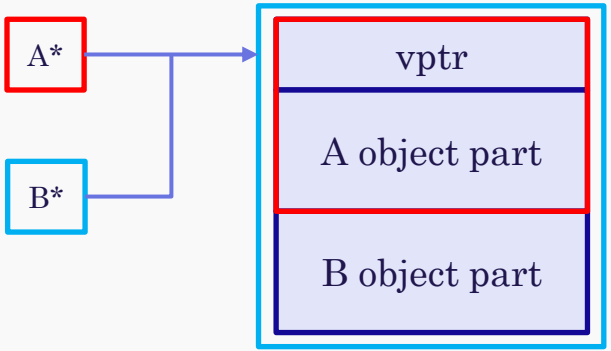
- Множественное наследование уже кажется мрачным?
- Это мы еще не дошли до по-настоящему мрачных вещей.
- Дело в том, что проблемы возможны не только с ромбовидными схемами.

ПРОБЛЕМА ПРЕОБРАЗОВАНИЙ

- Для того, чтобы при одиночном наследовании преобразовать вверх или вниз по указателю или ссылке достаточно static_cast

```
struct Base {};
struct Derived : Base {};
Derived *pd = new Derived{};
Base *pb = static_cast<Base*>(pd); // OK
pd = static_cast<Derived*>(pb); // OK
```

- Сработает ли такой подход при множественном наследовании?



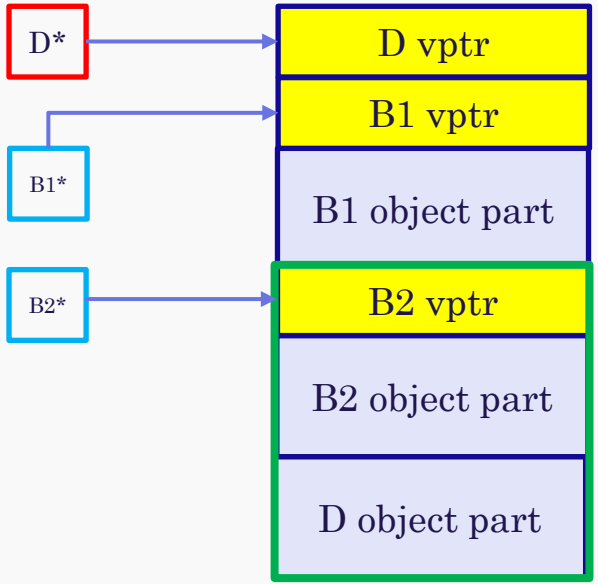
ПРОБЛЕМА ПРЕОБРАЗОВАНИЙ

- Как ни странно все магическим образом прекрасно работает при касте вверх

```
struct B1 {};
struct B2 {};
struct D : B1, B2 {};

D *pd = new D{};
B1 *pb1 = static_cast<B1*>(pd); // OK
B2 *pb2 = static_cast<B2*>(pd); // OK
```

- Мало того, все магическим образом работает и вниз



ОБСУЖДЕНИЕ

Такое чувство, что при виртуальном наследовании из-за смерженных таблиц не должен работать каст вниз?

RUNTIME TYPE INFORMATION (RTTI)

Для разрешения насущных вопросов (например «какой у меня динамический тип») и свободного хождения вниз-вверх по иерархиям классов, программа на C++ должна во время исполнения поддерживать особые невидимые программисту структуры данных.

Это очень странное решение для C++, потому что оно противоречит идеологии языка

В языке ровно два таких механизма: RTTI и исключения

Много раз делались попытки завести к ним какой-нибудь третий, но других ошибок с 1998 года комитет ни разу не делал

И конечно основа RTTI – это typeinfo

ВОЗМОЖНОСТИ TYPEID

Оператор `typeid` возвращает объект `std::typeinfo` который можно сравнивать и можно выводить на экран
Этот объект представляет собой динамический или статический тип.

```
OutputFile *pof = new IOFile{5};
assert(typeid(*pof) == typeid(IOFile)); // динамический тип
```

`typeid` может брать type или expression, если он берет expression, то динамический тип выводится только если это lvalue expression объекта с хотя бы одной виртуальной функцией

```
assert(typeid(pof) != typeid(IOFile*)); // статический тип
```

ВОЗМОЖНОСТИ DYNAMIC_CAST

Самым распространенным (и самым накладным) механизмом RTTI является `dynamic_cast`. Он может приводить типы внутри иерархий.

```
IOFile *piof = new IOFile{}; // File это виртуальная база
File *pf = static_cast<File *>(piof); / OK
InputFile *pif = dynamic_cast<InputFile*>(pf); // OK
OutputFile *pof = dynamic_cast<OutputFile*>(pf); // OK
pif = dynamic_cast<InputFile*>(pof); // OK
```

Обратите внимание, возможно приведение к сестринскому (братскому) типу

ОГРАНИЧЕНИЯ

- `dynamic_cast` ходит по всем путям, в том числе виртуальным. Время его работы может превышать время работы `static_cast` **на порядки**!
- К тому же затраты на `dynamic_cast` могут изменяться при изменении иерархий наследования
- При отсутствии таблиц виртуальных функций, `dynamic_cast` ведет себя как `static_cast` и это наиболее безумное его использование
- `dynamic_cast` работает только для указателей и ссылок
- Причем он работает для них по разному

ПОВЕДЕНИЕ DYNAMIC_CAST ПРИ ОШИБКЕ

- В случае, если `dynamic_cast` не может привести указатель, он возвращает нулевой указатель

```
OutputFile *pof = new OutputFile{13};
InputFile *pif = dynamic_cast<InputFile*>(pof);
assert(pif == nullptr);
```

- Но что он может сделать, если он используется для ссылок?

```
OutputFile &rof = *pof;
InputFile &rif = dynamic_cast<InputFile&>(rof);
```

- Ведь нет никакой «нулевой» ссылки

ПОВЕДЕНИЕ DYNAMIC_CAST ПРИ ОШИБКЕ

- На самом деле накопилось уже несколько вопросов
- Что делать `dynamic_cast`, если он работает для ссылок?
- Что возвращать `typeid`, если он вызван для `nullptr`?
- Как вернуть код ошибки из перегруженного оператора сложения?
- Похоже, в языке должен быть некий фундаментальный механизм, отвечающий за такие вещи. И этот механизм – **исключения**.
- Но о них в следующий раз.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Bjarne Stroustrup – The C++ Programming Language (4th Edition), 2013
2. Grady Booch – Object-oriented Analysis and Design with Applications, 2007
3. Скотт Мейерс, Эффективный современный C++: 42 способа улучшить ваше использование C++11 и C++14
4. Joshua Gerrard – The dangers of C-style casts, CppCon, 2015
5. Ben Deane – Operator Overloading: History, Principles and Practice, CppCon, 2018
6. Titus Winters – Modern C++ Design, CppCon 2018
7. Дональд Кнут, Искусство программирования. Том 1. Основные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 721 с.
8. Дональд Кнут, Искусство программирования. Том 2. Получисленные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 743 с.
9. Дональд Кнут, Искусство программирования. Том 3. Сортировка и поиск / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 767 с.