C++ Russia 2023

~~Back~~ Deep to basics: Наследование и виртуальность в C++

kaspersky

Евгений Ерохин

Чего не будет в этом докладе

- Конструкции языка
- Синтаксис
- Все что вы и так видели в учебниках

О чем этот доклад

- ABI(в основном *nix) платформ
- Как раскладывает данные компилятор
- Как располагаются структуры и классы в памяти
- Виртуальные функции и V-таблицы

Basics Basics Basics

Базовые определения

Класс описывает множество объектов, имеющих общую структуру и обладающих одинаковым поведением. Класс - это шаблон кода, по которому создаются объекты.

POD

POD (Plain Old Data) – C++ тип который имеет эквивалент в Си, и использует те же правила **инициализации**, **копирования**, **размещения** и **адресации**.

Любые “фишки” C++ превращают объект в не POD.

- Конструктор/деструктор
- Ссылки
- Указатели на членов
- Виртуальные функции
- Наличие базового класса
- ...

https://isocpp.org/wiki/faq/intrinsic-types#pod-types

POD или нePOD.

В C++20 это понятие отменили:

Deprecating the notion of “plain old data” (POD). It has been replaced with two more nuanced categories of types, **“trivial”** and **“standard-layout”**. “POD” is equivalent to “trivial and standard layout”, but for many code patterns, a narrower restriction to just “trivial” or just “standard layout” is appropriate; to encourage such precision, the notion of “POD” was therefore deprecated. The library trait is_pod has also been deprecated correspondingly.

https://botondballo.wordpress.com/2017/11/20/trip-report-c-standards-meeting-in-albuquerque-november-2017/

Standard layout

Класс со стандартным выравниванием

Класс со стандартным выравниванием это класс, который

- **не имеет** нестатических элементов данных типа класса **с нестандартным выравниванием** (или массива таких типов) или ссылки на него
- **не имеет виртуальных функций** и виртуальных базовых классов,
- имеет **одинаковый контроль** доступа для всех нестатических элементов данных,
- **не** имеет **базовых классов с нестандартным** выравниванием,
- только один класс в иерархии имеет нестатические элементы данных, и
- Неформально ни один из базовых классов не имеет того же типа, что и первый нестатический элемент данных. Или, формально: для класса S, не имеет элемент множества типов M(S) в качестве базового класса, где M(X) для типа X определяется как:
  - Если X является типом класса не объединения без (возможно, унаследованных) нестатических элементов данных, множество M(X) пусто.
  - Если X является типом класса не объединения, первый нестатический элемент данных которого имеет тип X0 (где указанный элемент может быть анонимным объединением), множество M(X) состоит из X0 и элементов M(X0).
  - Если X является типом объединения, множество M(X) является объединением всех M(Ui) и набора, содержащего все Ui, где каждый Ui является типом i-го нестатического элемента данных X.
  - Если X является типом массива с типом элементов Xe, множество M(X) состоит из Xe и элементов M(Xe).
  - Если X не относится к типу класса и массива, множество M(X) пусто.

https://en.cppreference.com/w/cpp/language/classes#Standard-layout_class

Standard layout

Класс со стандартным выравниванием

Класс со стандартным выравниванием это класс, который

- **не имеет** нестатических элементов данных типа класса **с нестандартным выравниванием** (или массива таких типов) или ссылки на него
- **не имеет виртуальных функций** и виртуальных базовых классов,
- имеет **одинаковый контроль** доступа для всех нестатических элементов данных,
- **не** имеет **базовых классов с нестандартным** выравниванием,

https://en.cppreference.com/w/cpp/language/classes#Standard-layout_class

Basics Basics

Data layout (natural alignment)

Type	Size	Alignment
int8_t	1	1
int16_t	2	2
int32_t	4	4
int64_t	8	8*

- Выравнивание данных происходит на их размер
- Требование исходит из микропроцессорной архитектуры
- **Невыровненный доступ может** приводить к падению или деградации производительности
- На некоторых архитектурах требование выравнивания **может ослабляться**

Выравнивание структур

Структура/класс выравнивается по самому **большему выравниванию** ее элементов.

Data layout

```
struct B
{
    char a;
    int b;
    short c;
    long long d;
};
```

24 bytes

padding

LIFEHACK!: сортируем по размеру

```
struct B
{
    long long d;
    int b;
    short c;
    char a;
};
```

16 bytes

Структура выравнивается по наибольшему выравниванию элементов

Упаковка (ВНИМАНИЕ НЕСТАНДАРТНАЯ ФИЧА!)

Мы можем попросить компилятор не делать паддинги.
В зависимости от архитектуры, компилятор может:

- вставлять код **сборки данных** по частям
- специальные инструкции
- или обычные для Intel платформ

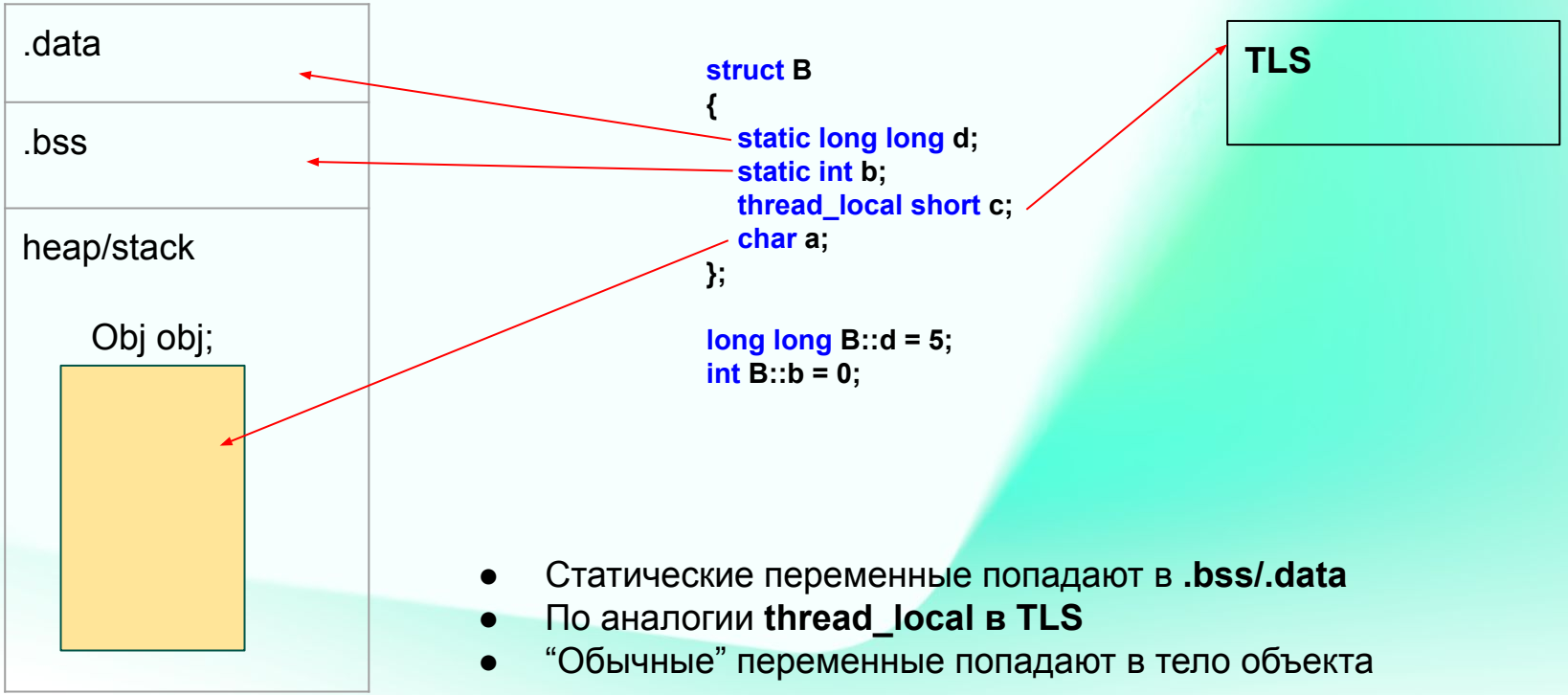
Как это сделать:

- помещаем структуру между
  #pragma pack(push, 1)
  struct …
  #pragma pack(pop)

- задать атрибут ***__attribubte__((packed))***

- задать атрибут ***[[gnu::packed]]***

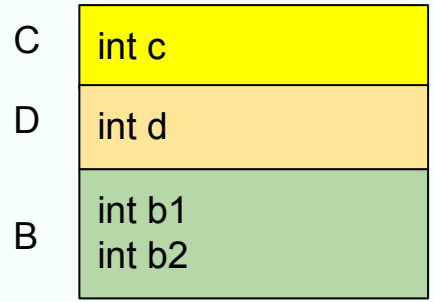
Data layout: static members



- Статические переменные попадают в **.bss/.data**
- По аналогии **thread_local в TLS**
- “Обычные” переменные попадают в тело объекта

Наследование

```
struct C
{
    int c;
};
struct D
{
    int d;
};
struct B : C, D
{
    int b1;
    int b2;
};
```



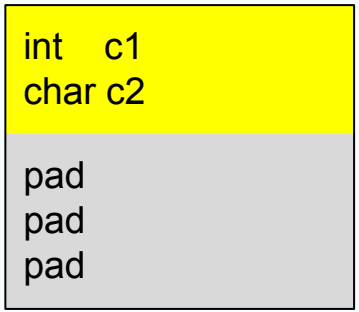
- Память классов родителей идет в начале, в **последовательности** указанной после **“:”**
- **Приведение** к базовому классу **не требует действий**
- Приведение ко **второму базовому** классу, уже требует прибавить смещение

Общее правило: последовательность размещения наследуемых классов строится обходом дерева наследования DFS (Depth First, в глубину).

Наследование и выравнивание

```
struct C
{
    int c1;
    char c2;
};
struct D
{
    char d;
};
struct B : C, D
{
    char b1;
    char b2;
};
```

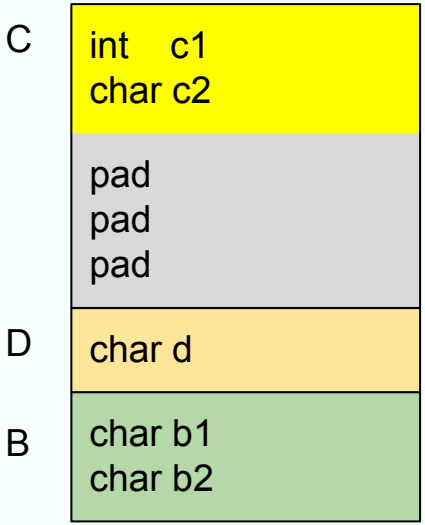
C



Как мы должны разложить следующие классы?

Наследование и выравнивание

```
struct C
{
    int c1;
    char c2;
};
struct D
{
    char d;
};
struct B : C, D
{
    char b1;
    char b2;
};
```



C является **POD**, следовательно мы можем передавать его в Си функции.

Следовательно, мы должны полностью поддерживать семантику Си

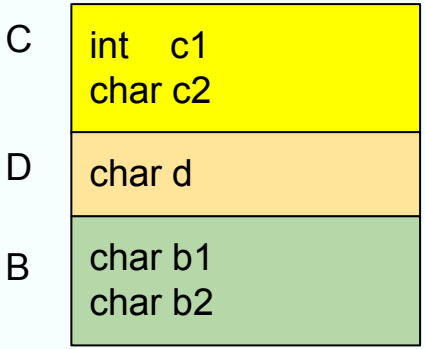
В Си нет понятия “наследование”

Компилятор Си может считать что в падингах ничего нет, и оптимизировать работу с ними

На самом деле так!
Но можно ли что-то с этим сделать?

Наследование и выравнивание

```
class C
{
    int c1;
    char c2;
};
struct D
{
    char d;
};
struct B : C, D
{
    char b1;
    char b2;
};
```



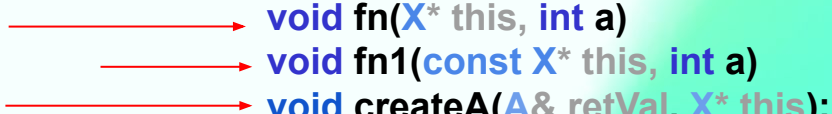
Заменив **struct** на **class**, **C** перестает быть **POD**, и мы можем работать с ним “по новому”.

На самом деле, вместо замены мы можем сделать любые другие действия, которые делают его не-POD.

Вызов метода

Для компилятора метод, это функция, которая первым параметром берет указатель на объект.

```
class X
{
        void fn(int a);
        void fn1(int a) const;
        A createA();
};
```



Basics

Полиморфизм

In programming language theory and type theory, polymorphism is the provision of a **single interface** to entities of different types or the use of a **single symbol** to represent multiple different types.

Есть разные подходы к классификации, и наиболее принятая классификация:

- **Ad hoc polymorphism** (специализация)
  перегрузка имен функций

- **Parametric polymorphism**
  шаблоны

- **Subtyping**
  переопределение при наследование

https://en.wikipedia.org/wiki/Polymorphism_(computer_science)
https://accu.org/journals/overload/25/141/pamudurthy_2424/

Полиморфизм

Также можно классифицировать по времени “применения”:

- **Полиморфизм времени компиляции (статический)**
  Работает только с типами известными на этапе компиляции. **CRTP**

- **Полиморфизм времени выполнения (динамический)**
  Работает как с типами известными на стадии компиляции, так и с известными только на времени выполнения.

Последний из двух, является целью этого доклада.

https://en.wikipedia.org/wiki/Polymorphism_(computer_science)
https://accu.org/journals/overload/25/141/pamudurthy_2424/

Динамический полиморфизм

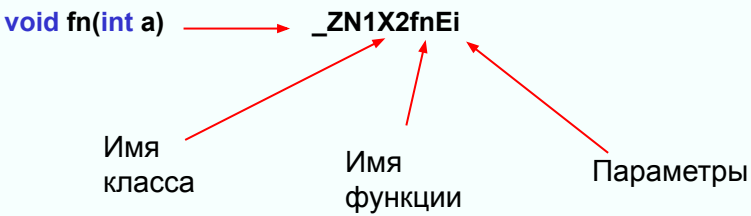
Что мы хотим получить на выходе:

- **Базовый тип** определяет (декларирует) **поведение**
- **Наследованные** типы могут его **переопределять**
- **Вызывающая сторона не знает** во время исполнения, с каким именно **типом** имеет дело

```
class X
{
    virtual void fn() {...}
    virtual void fn(int a) {...}
};
class Y : X
{
    virtual void fn(int a) {...}
};
void caller(X& x)
{
    x.fn(5);
}
```

Термины: сигнатура

Сигнатура функции - это мангленное имя функции.



Динамическая диспетчеризация

Метод которым мы будем достигать поставленной цели называется **“Динамическая диспетчеризация”**.

В чем идея:

- Для каждого класса создаем таблицу (**V-table**) указателей на функции
- Для каждой **virtual** функции отводится место (**jump slot**) в порядке следования.
- Если в наследованном классе есть функция с такой же **сигнатурой**, то оно заменяет соответствующий слот в таблице этого класса.
- Если такой сигнатуры нет - выделяем **новый слот**
- Вызов виртуальной функции идет через указатель в этой таблице

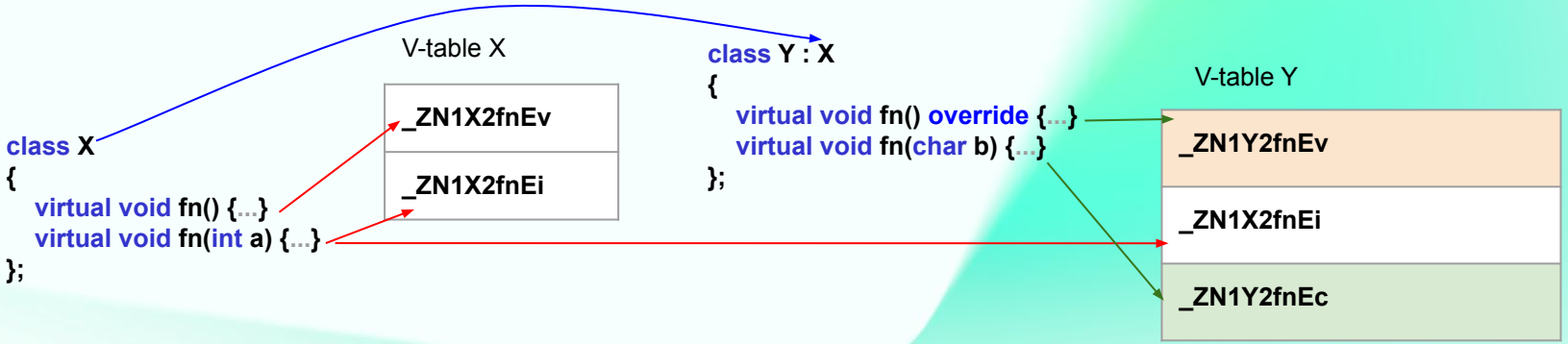
V-table

fn()
fn(int)
fn1()

Динамическая диспетчеризация

Для наследуемого класса

- **Копируем** таблицу
- Если сигнатура есть - **заменяем**
- Если сигнатуры нет - **выделяем** новый слот

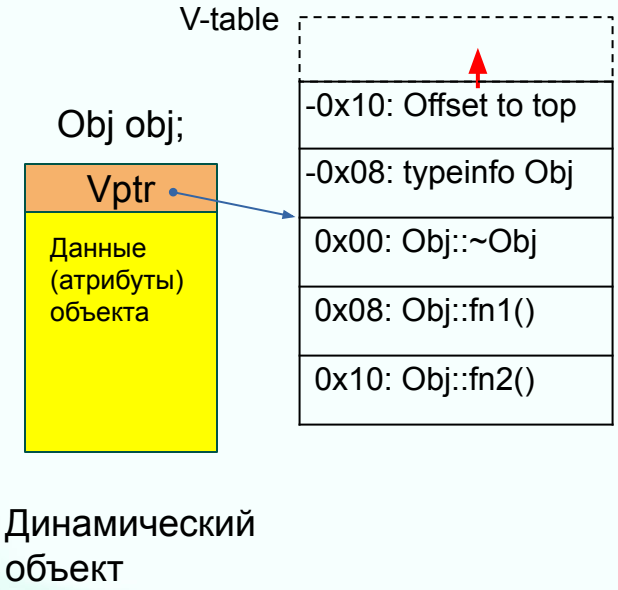


Термины: динамический класс

Динамический класс - класс имеющий **V-таблицу**:

- имеет виртуальные **функции**, и/или
- имеет виртуальный **базовый класс**

VTable (Itanium ABI)



В **Itanium ABI** (сейчас используется на всех *nix платформах):

- В начале каждого объекта с виртуальными функциями есть указатель **vptr** на V-таблицу
- **vptr** указывает на элемент с индексом 2* (от начала)
- **vptr[0]** содержит указатель на самую первую виртуальную функцию
- Виртуальные функции раскладываются в **vptr[n]** в порядке объявления
- **vptr[-1]** указывает на структуру **typeinfo** объекта
- **vptr[-2]** содержит смещение до начала объекта (обсудим позже)
- После vptr идут данные класса, в порядке объявления
- **Неявно созданные деструкторы** и т.п. располагают в конце

Но это еще не всё ;-)

Термины: почти пустой класс

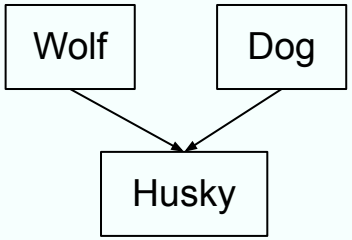
Почти пустой класс (nearly empty class) - динамический класс без данных.

Термины: Most Derived Object

Most Derived Object (MDO) - наиболее унаследованный объект.

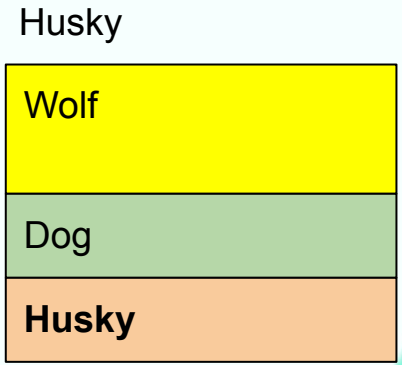
```
class A { … };

class B { … };

class C: public A, public B { … };

…
auto* mdo = new C;

…
C mdo;
```

Множественное наследование



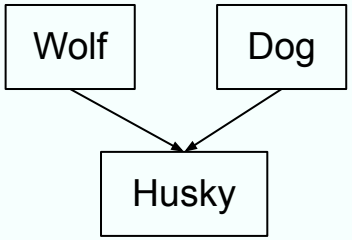
Класс Husky обладает свойствами обоих базовых классов.

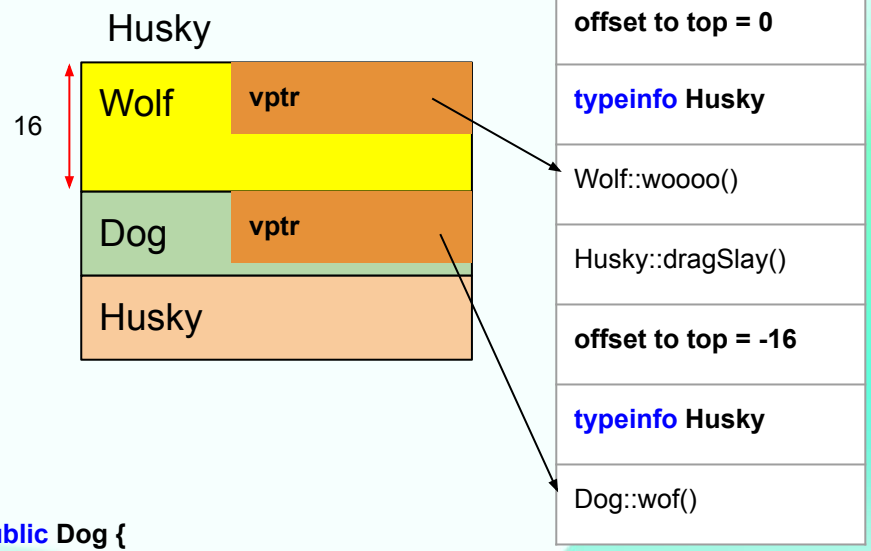
```
class Wolf
{...};
class Dog
{...};
class Husky : public Wolf, public Dog
{...};
```



Подобъекты **Wolf** и **Dog** являются полноценными объектами!

Множественное наследование V-таблица



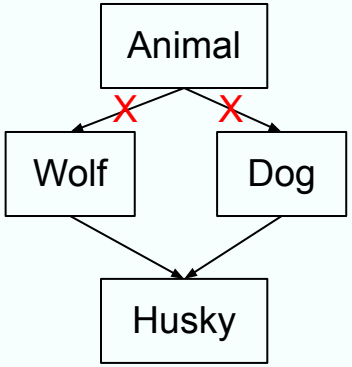



```
class Wolf {
   virtual void woooo();
};
class Dog {
   virtual void wof();
};
class Husky : public Wolf, public Dog {
   virtual void dragSlay();
};
```

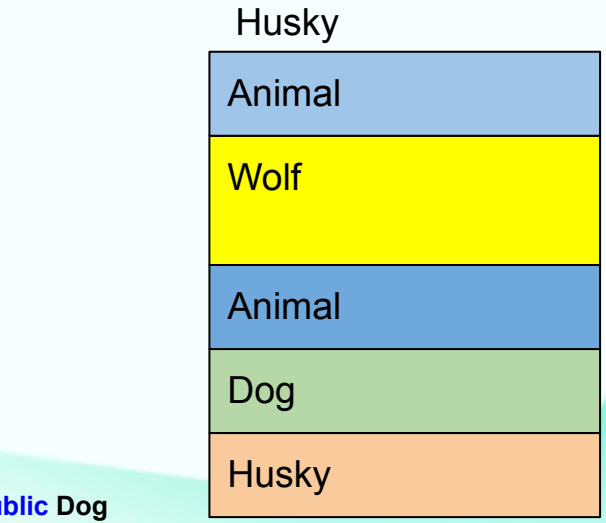
typeinfo у всех таблиц одна.

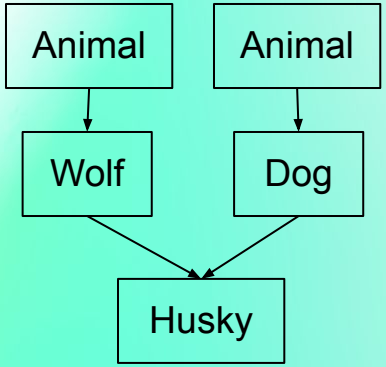
offset to top хранит смещение объекта от начала **MDO (Most Derived Object)**.

Множественное наследование?



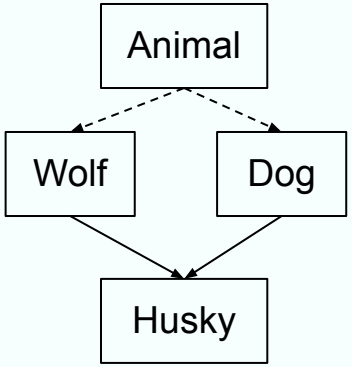
Попробуем продлить родословную.



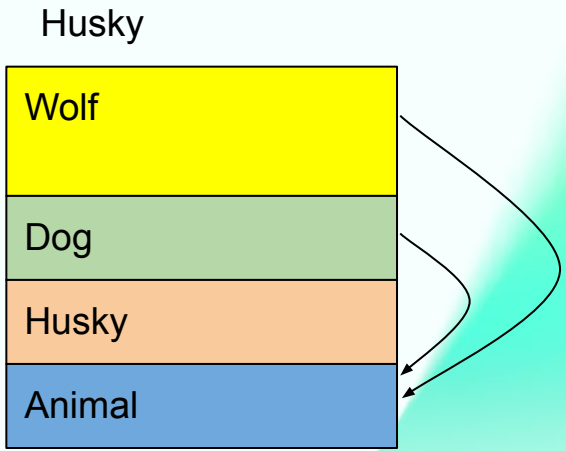



```
class Animal
{...};
class Wolf : public Animal
{...};
class Dog : public Animal
{...};
class Husky : public Wolf, public Dog
{...};
```

Виртуальное наследование



Мы хотим иметь только одну копию данных



Компилятор знает где находится виртуальный класс и корректирует **this** для методов **Animal**.

```
class Animal
{...};
class Wolf : virtual public Animal
{...};
class Dog : virtual public Animal
{...};
class Husky : public Wolf, public Dog
{...};
```

Не совсем корректно

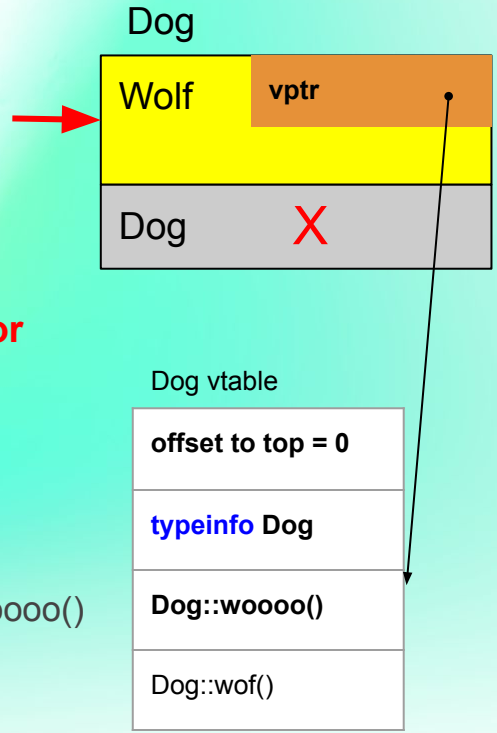
Виртуальные функции в конструкторе/деструкторе

В V-таблице объекта класса всегда находится **метод** переопределенный в **наиболее-унаследованном** классе!

- Мы находимся в **Wolf::Wolf()** соответственно **Dog** еще не сконструирован
- Вызываем виртуальный **woooo()** -> **Dog::woooo()**
- **Dog::woooo()** обращается к данным в Dog = **Undefined Behavior**

У **деструктора** ситуация с точностью до наоборот!

```
class Wolf
{
   virtual void woooo();
   …
};
class Dog
{
   virtual void woooo() override;
   …
};
```



Not so Basics

Раскладка данных и V-таблицы класса

Главные правила!

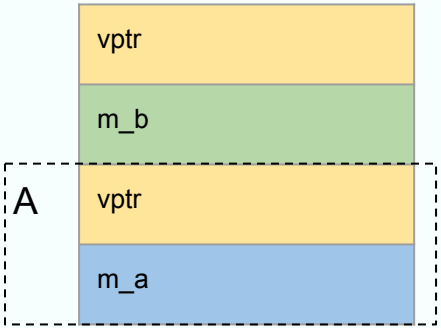
- **Данные** в под-объектах должны быть разложены также как если бы они были автономны.
- **V-таблица** под-объекта в родительской V-таблице должна полностью соответствовать автономной.

Эти правила позволяют делать приведение к базовому типу “бесплатным”.

Primary base class

```
class A
{
public:
   virtual void fn1();
   int m_a;
};
class B : public A
{
public:
   virtual void fn2();
   int m_b;
};
```

Если разложить: *наследник / наследуемый*



- Дублирование vptr
- При вызове наследованной функции надо корректировать **this** (B->A)

…

Если разложить: *наследуемый / наследник*

A - primary base class

- Общий vptr
- Приведение B->A бесплатно
- Можем задействовать паддинг

Полный алгоритм раскладки класса **(Primary base class)**

Выбираем **Основной базовый класс (Primary base class).**
Тот класс с которым мы разделяем его V-таблицу.

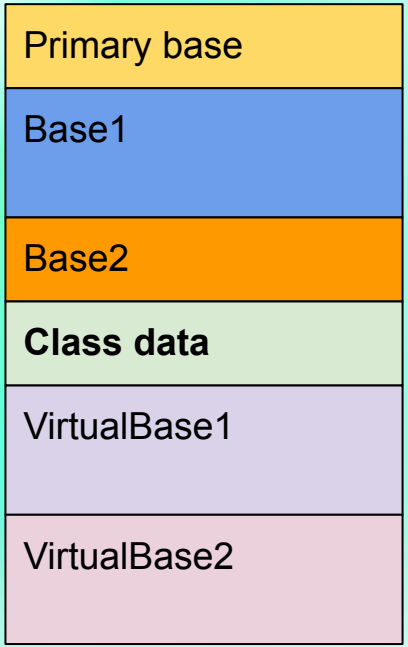
1. **Отложим** все **виртуально**-наследуемые классы

2. Пытаемся в порядке непосредственного наследования выбрать первый **динамический** класс

3. Если нет, “**Nearly empty virtual base class**” - нет данных, только V-таблица. Выбирается первый в графе наследования.
   - Который не является чьим-то **primary**
   - Если такого нет, то **первый primary**

Полный алгоритм раскладки класса

1. Если есть **primary-base** - аллоцируем его
2. Если **нет primary-base** и он **динамический** - аллоцируем **vptr**
3. Аллоцируем **НЕ ВИРТУАЛЬНЫЕ** базовые классы в порядке объявления (сами классы аллоцируются тем же алгоритмом, без виртуальной базы)
4. Аллоцируем **данные класса**
5. Аллоцируем **виртуальные базовые классы** (прямые и непрямые) - **НО не primary (**они уже аллоцированны**)**



Раскладка V-table

1. Раскладываем таблицу **основного-базового** класса
2. Раскладываем таблицу **самого класса**
3. Раскладываем таблицы **остальных базовых** классов

Раскладка виртуальных функций класса

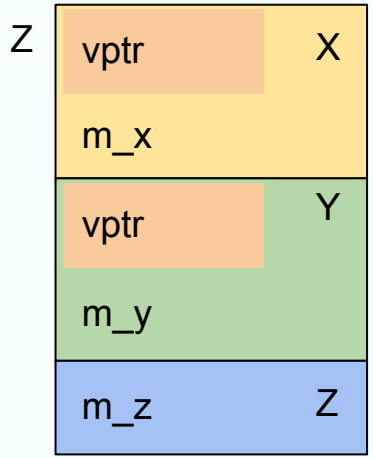
- Функции раскладываются в порядке декларации
- Implicit-defined **assignment operator**
- Implicit-defined **move operator**
- Implicit-defined **destructors**

Деструктор занимает 2 слота:

- вызывающий **delete()**
- и **не** вызывающий **delete()**

Ситуация

```
class X
{
public:
  int m_x = 1;
  virtual int xfn();
};
class Y
{
public:
  int m_y = 5;
  virtual int fn();
};
class Z : public X, public Y
{
public:
  int m_z;
  virtual int fn();
};
void call(Y* y) { y->fn(); }
```



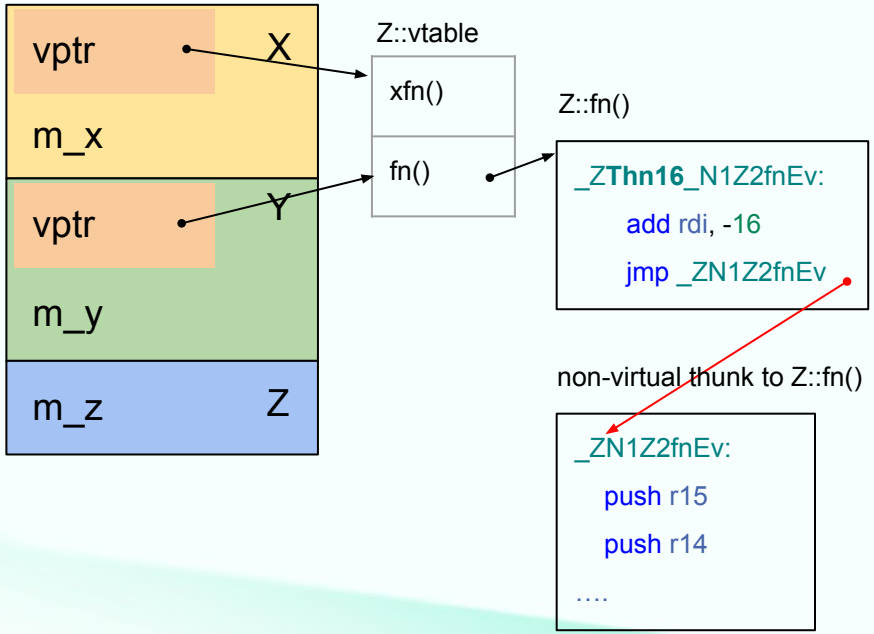

```
call(Y* y)
{
  y->fn();
```

```
int Z::fn(Z* this)
{
  m_x--;
  return m_x + 7;
}
```

- Имеем динамические классы **X**, **Y**
- Множественно унаследованный **Z**
- в **Z** мы переопределяем **fn()** из **Y**
- **call**() знает только о **Y**
- вызов идет через V-table
- Но! Переопределенная **Z::fn** в качестве **this** ожидает **Z***

https://godbolt.org/z/jrajazo98

Non-virtual thunk



Перед тем как виртуальная **не-primary base class** функция будет вызвана, надо скорректировать **this**.

Для этого заводим небольшую промежуточную функцию, коррекции this, или **thunk**. Манглится с префиксом **Thn<adj>_**. ‘**n**’ - negative.

Компилятор может “инлайнить” в thunk.

Исходная функция генерируется для не-виртуального вызова.

https://godbolt.org/z/jrajazo98

RTTI

В C++ typeinfo используется для:

- **typeid**
- Матчинг **исключений**
- **dynamic_cast**

Объект динамического класса **type_info** определяет **RTTI** для класса.

```
class type_info
{
   ...
private:
   const char* __type_name;
};
```

Содержит “заменгленное” имя класса

type_info является базовым классом, для других, которые описывают специфические классы:

abi::__fundamental_type_info
abi::__array_type_info
abi::__function_type_info
abi::__enum_type_info
abi::__class_type_info
abi::__si_class_type_info
abi::__vmi_class_type_info
abi::__pbase_type_info
abi::__pointer_type_info
abi::__pointer_to_member_type_info

RTTI пример

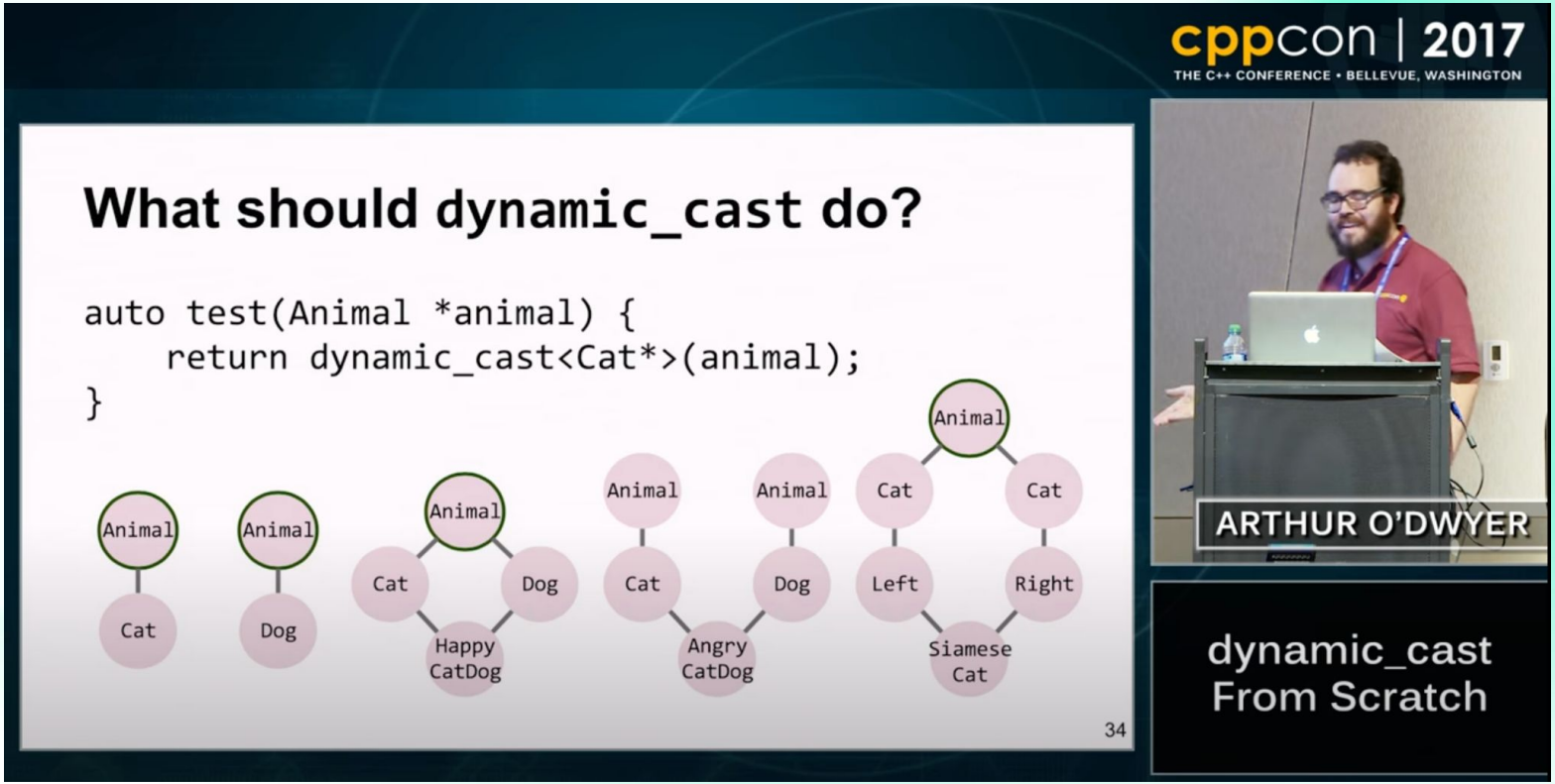
```
class type_info
{
   ...
private:
   const char* __type_name;
};

class X
{
public:
   virtual int xfn();
};
```

abi::__class_type_info

“**Заменгленное**” имя класса

```
_ZTV1X: // V-table X
        .quad   0
        .quad   _ZTI1X
        .quad   _ZN1X3xfnEv

_ZTI1X:
        .quad   _ZTVN10__cxxabiv117__class_type_infoE+16
        .quad   _ZTS1X

_ZTS1X:
        .asciz  "1X"
```

RTTI что посмотреть (Домашка)



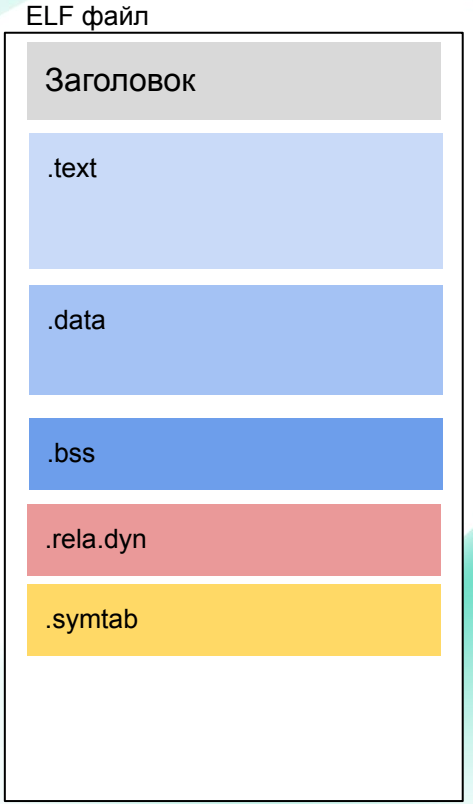
https://youtu.be/QzJL-8WbpuU

Галопом по Европам

Давайте бегло рассмотрим модели кода и их проблематику:

- Как выглядит исполняемый файл
- Позиционно зависимый код
- Позиционно независимый код
- Релокации

Как выглядит исполняемый файл



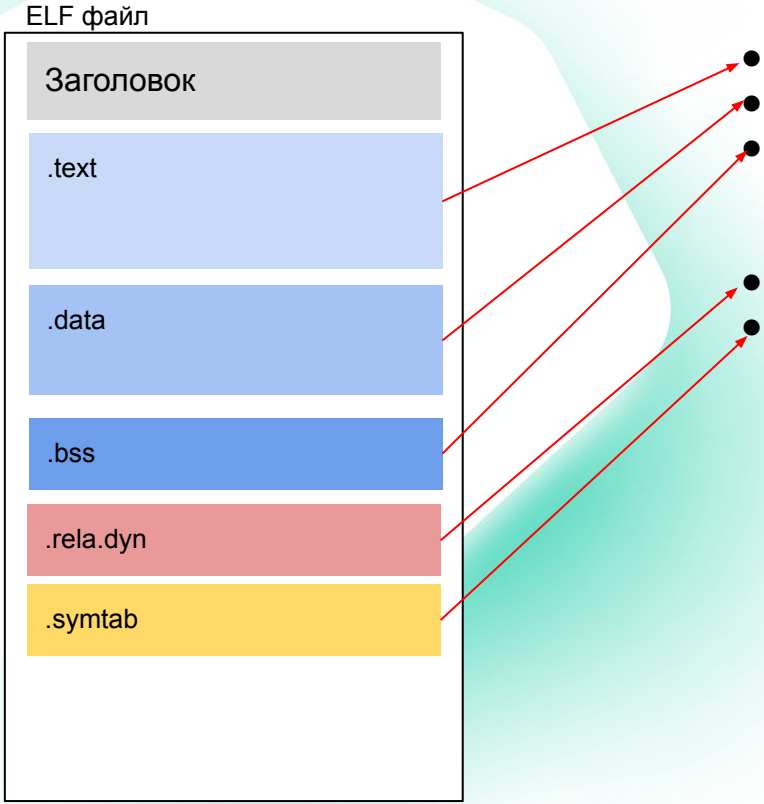
Исполняемый файл состоит (упрощенно) из:

- Заголовка
- И секций

Каждая секция так же состоит из:

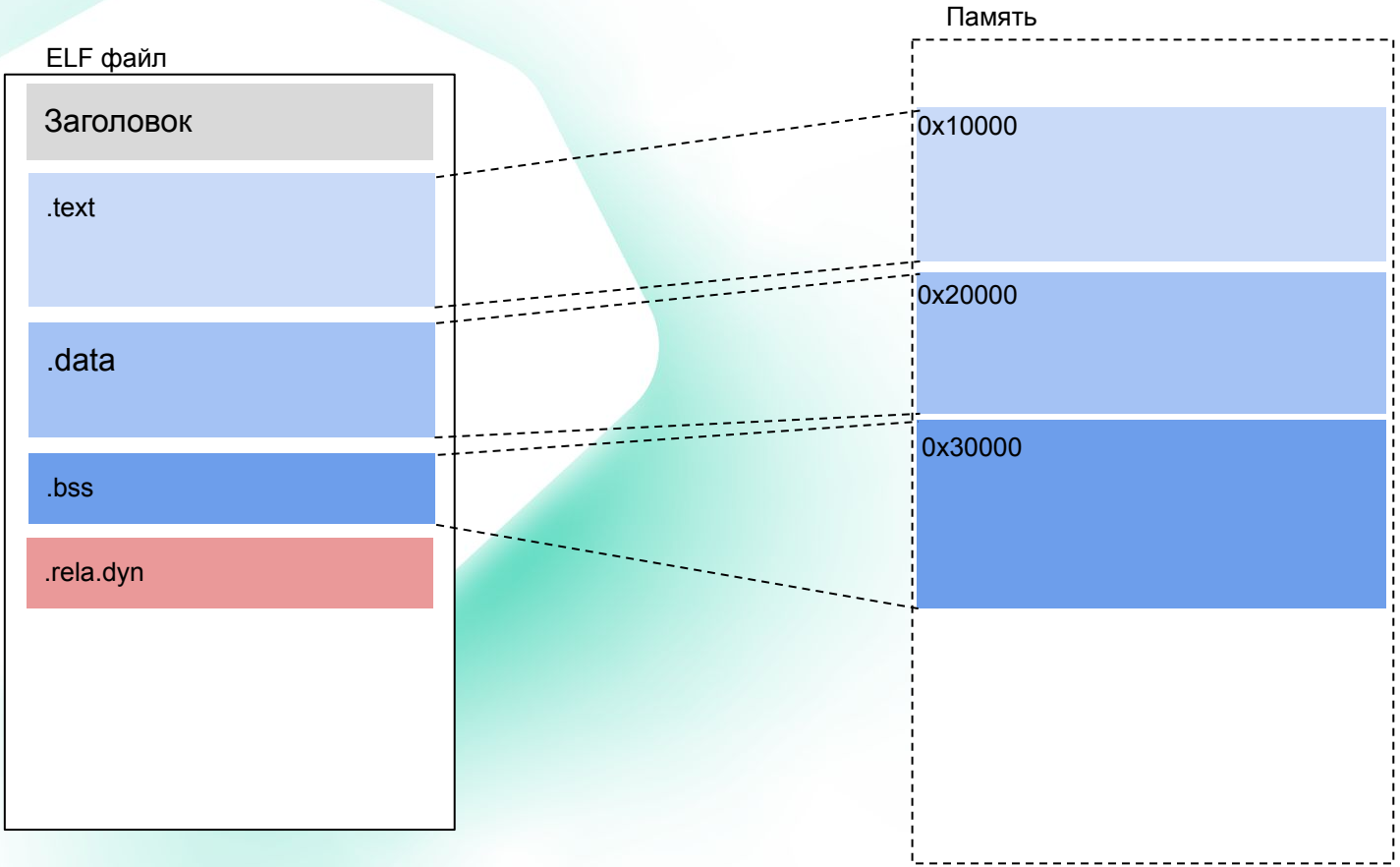
- Заголовка, который содержит адрес загрузки, размер, различные флаги и т.п.
- Данные(опционально)

Назначение секций

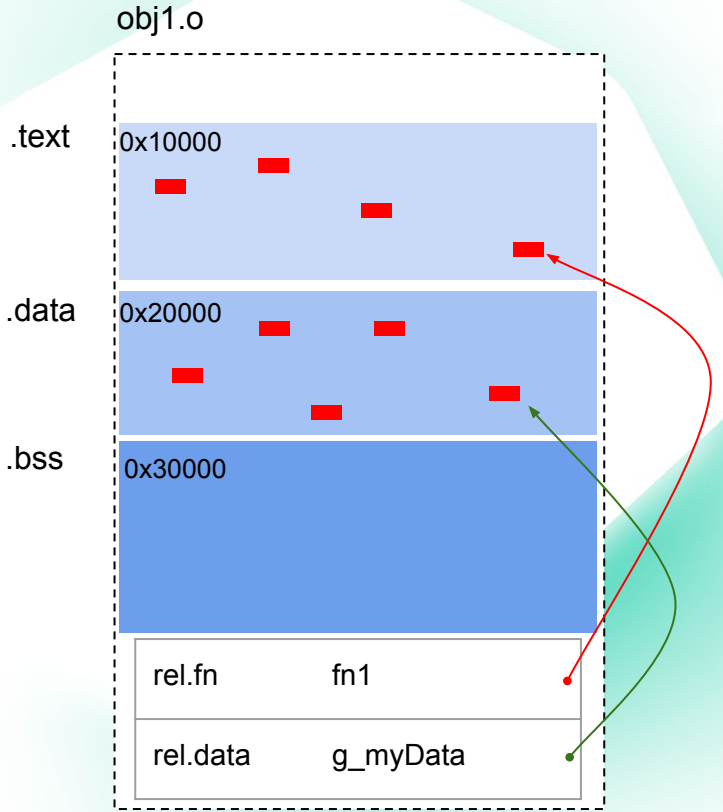


- **.text** исполняемый код.
- **.data** инициализированные данные.
- **.bss** неинициализированные данные (на самом деле данных не хранит, только размер).
- **.rela.dyn** динамические релокации.
- **.symtab** таблица символов

При загрузке: отображаем в память

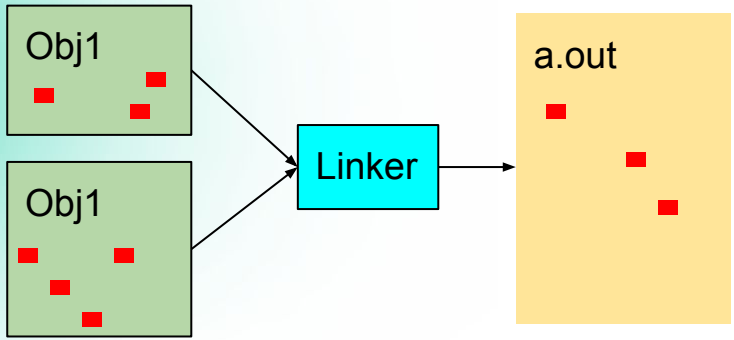


Релокации



После того как компилятор собрал объектник:

- в нем остаются **“дыры”** - неизвестные на стадии компиляции адреса и прочие данные.
- **Линковщик** заполнит что сможет
- Все что осталось - внешние символы, для них остаются **динамические релокации**

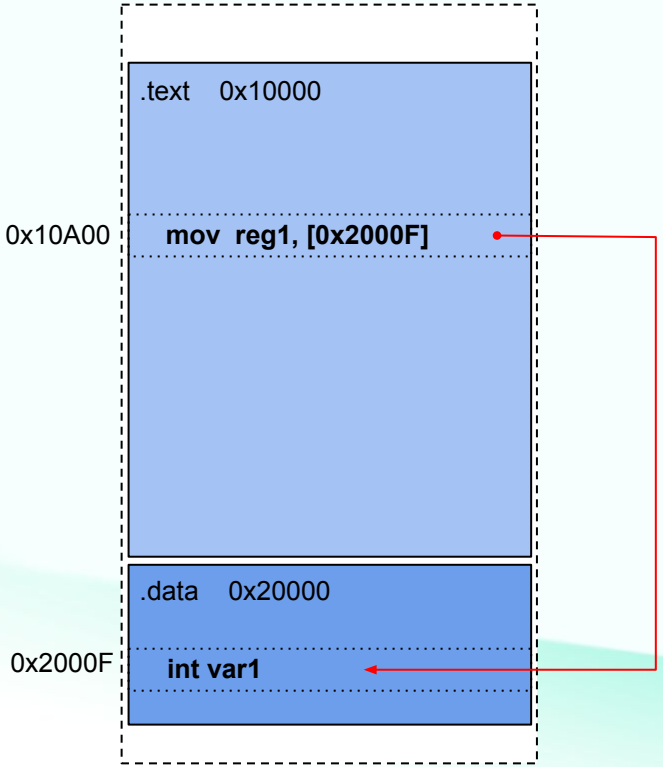


Динамические релокации

Для того чтобы программа могла выполняться, в эти **“дыры”** надо поместить **реальные данные/адреса**.

Загрузчик, после того как секции отображены в память, проходит по таблице релокаций и применяет указанные в ней действия.

Позиционно зависимый код



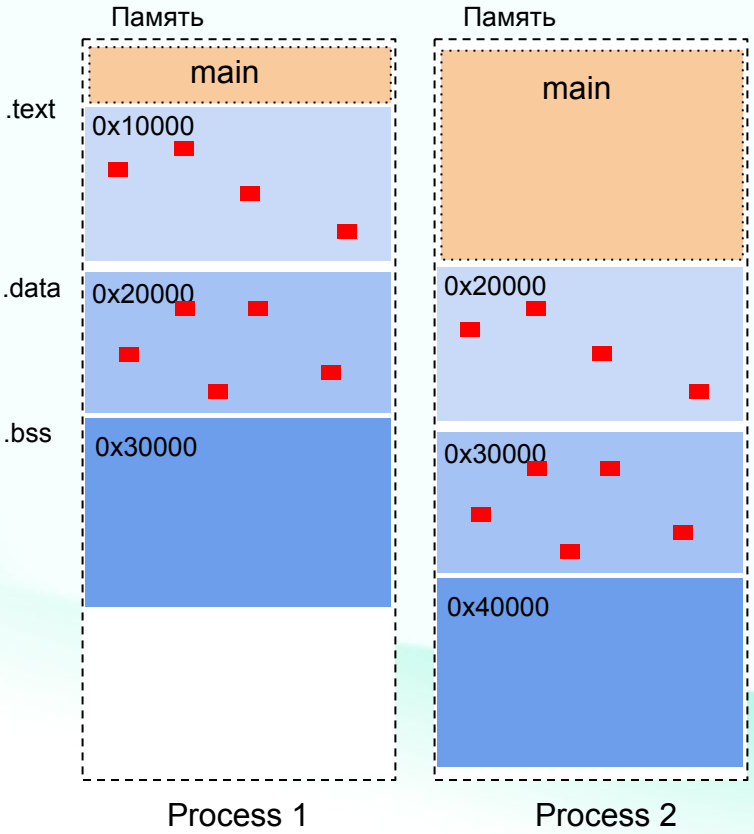
В позиционно зависимом коде инструкции используют непосредственно адреса данных и кода.

Адреса должны быть известны либо:

- на момент **компиляции**
- на момент **загрузки**, тогда нужны **релокации**

При загрузке должны быть применены релокации, “пропатчить” код и другие данные, чтобы на момент выполнения “неизвестных” не было.

Что не так с позиционно зависимым кодом



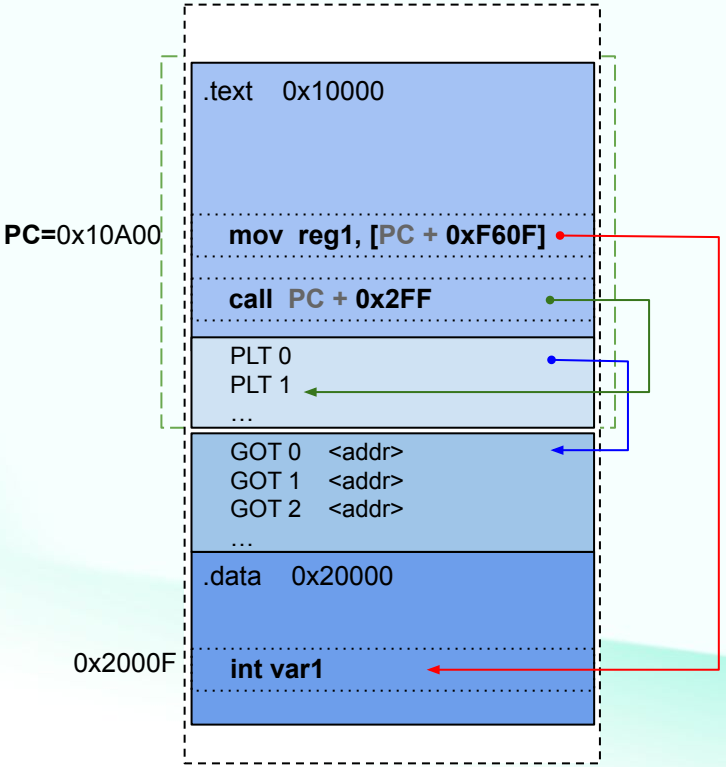
1. Если у нас разделяемая библиотека:
   В разных процессах она скорее всего будет на разных адресах.

2. **ASLR** (Address Space Layout Randomization):
   Чтобы усложнить хакеру, ОС может размещать основную программу и разделяемые библиотеки по разным адресам.

Те страницы памяти которые неизменны, мы можем хранить в физической памяти в единичном экземпляре, и отображать на разные виртуальные адреса в разных процессах.

НО секция .text “пропатчена” адресами одного из процессов! Значит разделение не возможно.

Позиционно (не)зависимый код **PIC**



Position Independent Code (PIC)

Мы не используем абсолютные адреса, адресация идет относительно адреса самой инструкции **PC** (Program Counter).

- Все локальные обращения идут через относительную адресацию
- Внешние адреса выносятся в специальную таблицу **Global Offset Table (GOT)**. Ее нам заполняет динамический линковщик.
- Вызовы внешних функций через небольшие кусочки кода в таблице **Procedure Linkage Table (PLT)**. Которая берет адрес из GOT.
- Теперь **.text** и **.plt** становятся разделяемыми.

Еще секции

- **.rodata** (Read Only data) - содержит константные данные. Память может разделяться.
- **.relro** (Relocate and Read Only) - загрузчик применяет все релокации и меняет защиту страниц на Read Only. Память не разделяемая.

Где разместить V-таблицу

-0x10: Offset to top
-0x08: typeinfo Obj
0x00: Obj::~Obj
0x08: Obj::fn1()
0x10: Obj::fn2()

- V-таблица, это **массив указателей**
- Для каждого элемента будет создана **релокация**

В какой секции разместить
(-) .text - мы стараемся выкинуть все релокации
(-) .rodata - релокации недопустимы
(-+) .data - можно, но хакер может попробовать подменять адреса
(+).relro - подходит оптимально. Жаль что не разделяема :-(

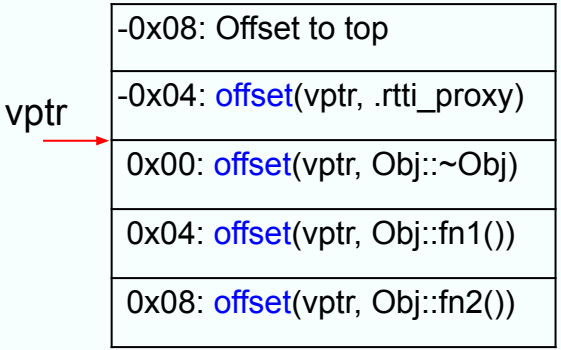
А что нового: **Relative V-table**

Идея нового подхода - **А давайте хранить не абсолютные адреса в V-таблице, а смещения!**

Clang: **-fexperimental-relative-c++-abi-vtables**

- Разработано с активным участием Google
- Несмотря на префикс “experimental” фича уже отлаженная
- Фича включена по умолчанию под ОС Fuchsia

Relative V-table



- В каждом элементе хранится не адрес, а **смещение**
- Для модели памяти medium на 64 бит архитектурах размер элемента сокращается с 8 до **4 байта**

Relative V-table

```
vtable for A:
        .long   0                                       # 0x0
        .long   (typeinfo for A [clone .rtti_proxy]-vtable for A)-8
vptr    .long   (A::fn1()@PLT-vtable for A)-8
        .long   (A::~A() [base object destructor]@PLT-vtable for A)-8
        .long   (A::~A() [deleting destructor]@PLT-vtable for A)-8
```

- Точкой отсчета является место куда указывает **vptr**
- На стадии линковки отсчет идет от стартового символа таблицы (это корректирует **-8**)
- В каждом слоте у нас сохраняется смещение от точки куда указывает vptr до вызываемого метода
- В случае внешнего символа, смешение до его **PLT** слота
- Для RTTI добавляется прокси **.rtti_proxy**

https://godbolt.org/z/KaW5d81Ez

RTTI и proxy

- RTTI может быть в другой единице трансляции, поэтому мы используем небольшой индирект **.rtti_proxy**, куда уже помещается адрес
- Это добавляет 1 релокацию

Вызов с relative V-table

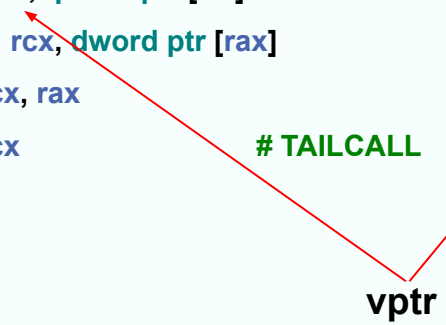
```
void call(Y* y)
{
  y->fn();
}
```

Relative:

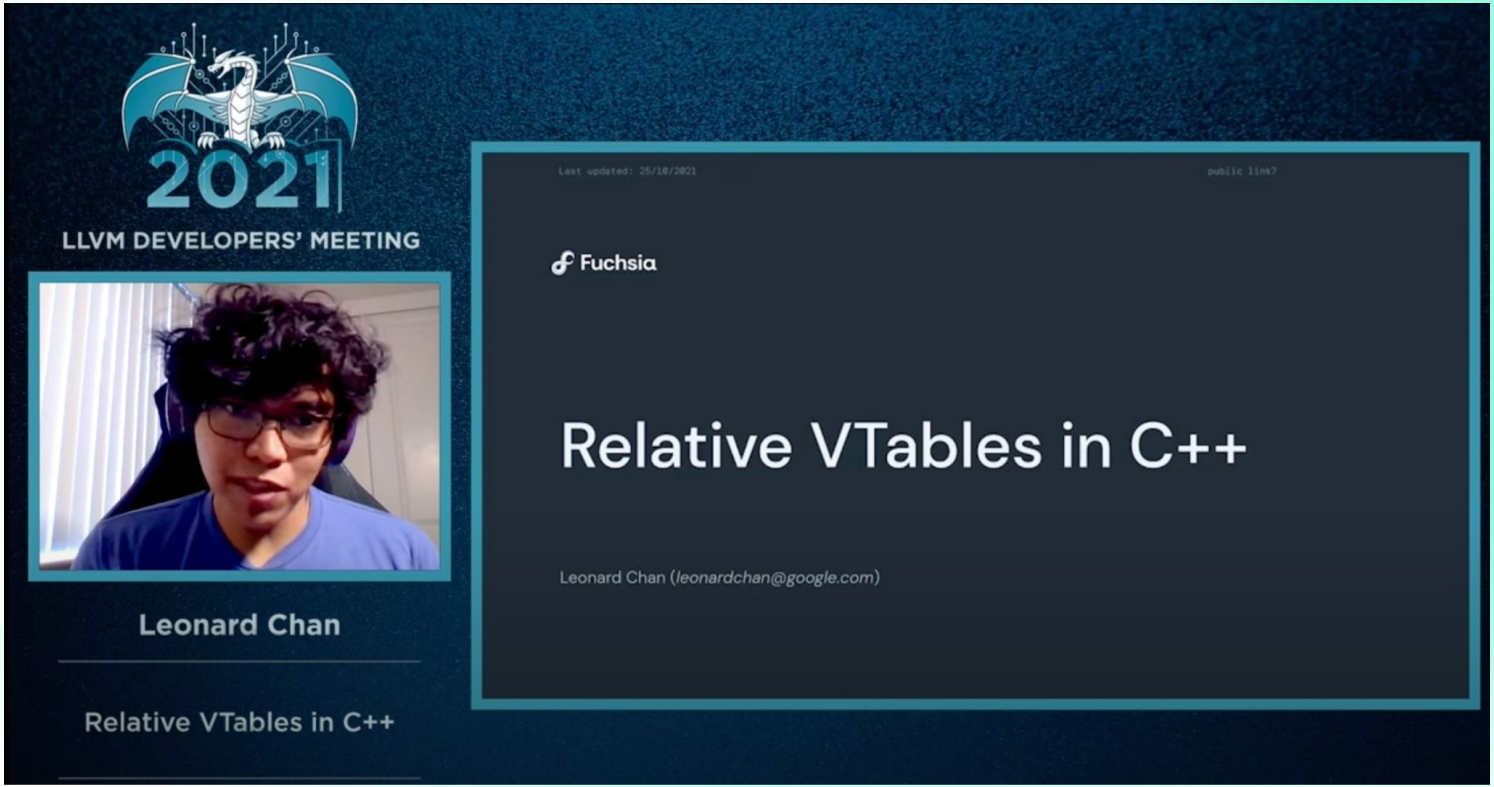
```
call(Y*):                    # @call(Y*)
    mov     rax, qword ptr [rdi]
    movsxd  rcx, dword ptr [rax]
    add     rcx, rax
    jmp     rcx                  # TAILCALL
```

Classic:

```
call(Y*):                    # @call(Y*)
    mov     rax, qword ptr [rdi]
    jmp     qword ptr [rax]          # TAILCALL
```



Relative V-table (домашка 2)



https://youtu.be/9HGKlDiJy8E

Выводы по relative V-tables

- **V-таблица** становится **read only**
- Разработчики из Google репортят неплохую **экономию памяти**
- Также репортят что **скорость** исполнеия не снизилась
- У нас уменьшается количество релокаций **->** увеличиваем **скорость загрузки**
- Эта фича является **ABI-breaking**

https://youtu.be/9HGKlDiJy8E

Рефакторинг и ABI

Что меняет **ABI**:

- Изменение **порядка** функций
- **Добавление** виртуальных функций в нефинальный класс
- **+-** Изменение **прототипа**

Патерн решения проблем с V-таблицей:

```
class IOUSBHostInterface : public IOUSBInterface
{
   OSDeclareDefaultStructors(IOUSBHostInterface)

public:
   static IOUSBHostInterface* withDescriptors(const
StandardUSB::ConfigurationDescriptor* configurationDescriptor, const
StandardUSB::InterfaceDescriptor* interfaceDescriptor);

protected:
   virtual bool initWithDescriptors(const StandardUSB::ConfigurationDescriptor*
configurationDescriptor, const StandardUSB::InterfaceDescriptor* interfaceDescriptor);
   // Pad slots for future expansion
   OSMetaClassDeclareReservedUnused(IOUSBHostInterface, 0);
   OSMetaClassDeclareReservedUnused(IOUSBHostInterface, 1);
   OSMetaClassDeclareReservedUnused(IOUSBHostInterface, 2);
   …

#define OSMetaClassDeclareReservedUnused(className, index)      \
    virtual void _RESERVED ## className ## index ()
```

Заключение (Часть 1)

Что хочется сказать про механизм V-таблиц:

- Механизм который иногда **кажется сложным**, и иногда это так
- При этом **весьма эффективный**, но надо применять с умом
- Любая **динамика** в рантайме, требует добавления разного рода **“индиректов”**

И на следующую часть у нас еще остается много чего интересного!

Спасибо!

kaspersky