C++ Russia 2023

~~Back~~ Deep to basics: Наследование и виртуальность в C++ (Часть 2)

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

Как компилируем примеры

clang++ -O2 -fno-inline

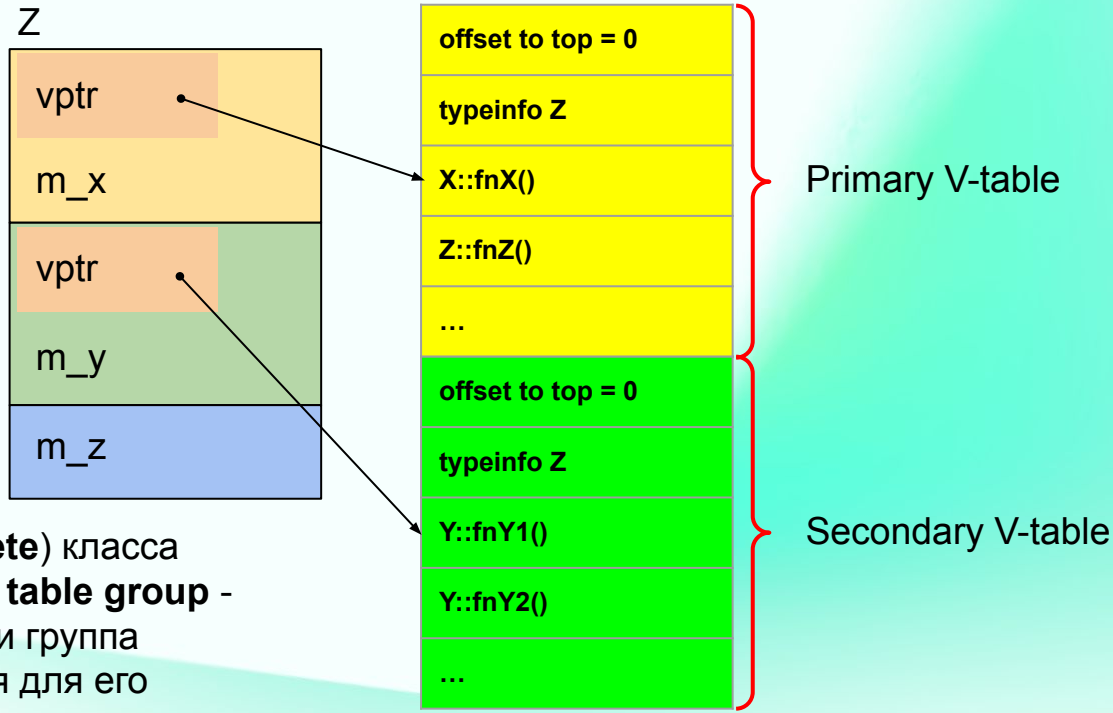
Где инстанцируются V-таблицы

Key function (ключевая функция) - первая **не-inline,** не-чистая, виртуальная функция декларированная функция.

- **V-таблицы** и все что им сопутствует эмитится в единице трансляции где находится тело **ключевая функция.**
- Если ключевой функции нет, то V-таблицы эмитится во все единицы трансляции в секцию **COMDAT**.

Определение: Virtual table group

```
class X
{ ... };
class Y
{ ... };
class Z : public X, class Y
{ ... };
```



Для полного (**complete**) класса формируется **Virtual table group** - непрерывно в памяти группа таблиц, необходимая для его иерархии наследования.

Конструкторы и деструкторы

Не совсем корректно

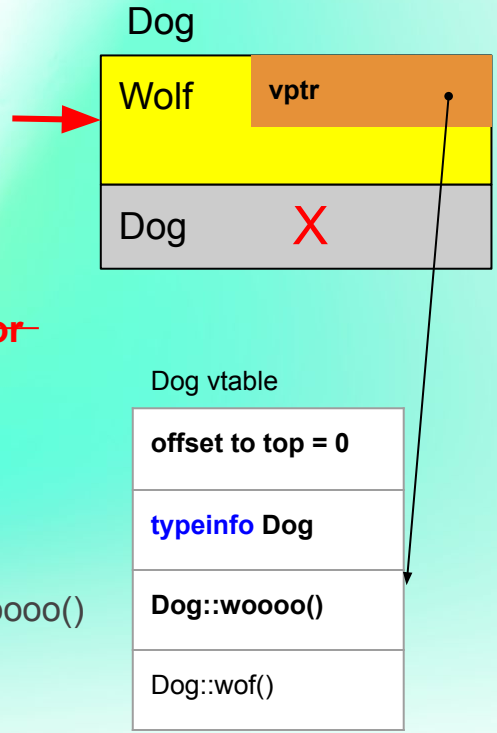
Виртуальные функции в конструкторе/деструкторе

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;
   …
};
```



Конструирование

```
class A
{
  int m_a;
public:
  A();
  virtual int fn();
};
class B : public A
{
  int m_b;
public:
  B();
  virtual int fn() override;
};
A::A() { m_a = 10; }
B::B() { m_b = 5; }
```

2 A::A() [base object constructor]:

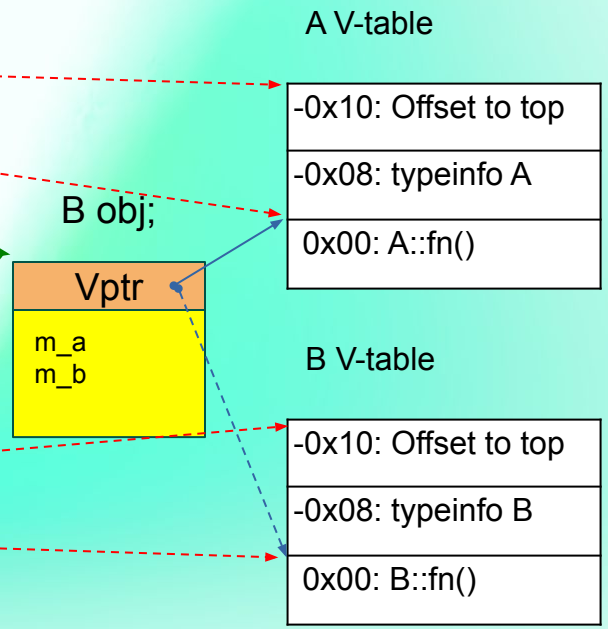
```
mov   rax, qword ptr [rip + vtable for A@GOTPCREL]
add   rax, 16
mov   qword ptr [rdi], rax
mov   dword ptr [rdi + 8], 10        this
ret
```

1 B::B() [base object constructor]:

```
push  rbx
mov   rbx, rdi
call  A::A() [base object constructor]
```

3

```
mov   rax, qword ptr [rip + vtable for B@GOTPCREL]
add   rax, 16
mov   qword ptr [rbx], rax
mov   dword ptr [rbx + 12], 5
pop   rbx
ret
```

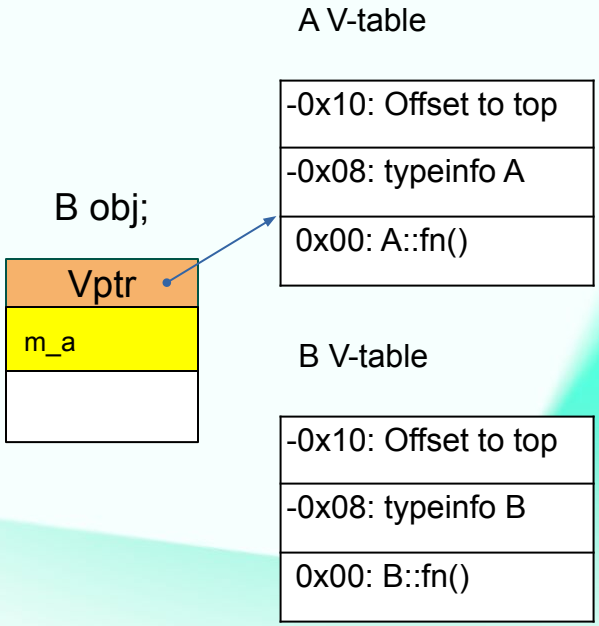


https://godbolt.org/z/rhz3Mc4P7

Конструирование

```
class A
{
  int m_a;
public:
  A();
  virtual int fn();
};
class B : public A
{
  int m_b;
public:
  B();
  virtual int fn() override;
};
A::A() { m_a = 10; }
B::B() { m_b = 5; }
```

Прежде чем стать полностью сконструированным объект переживает последовательно все стадии “своего развития”.



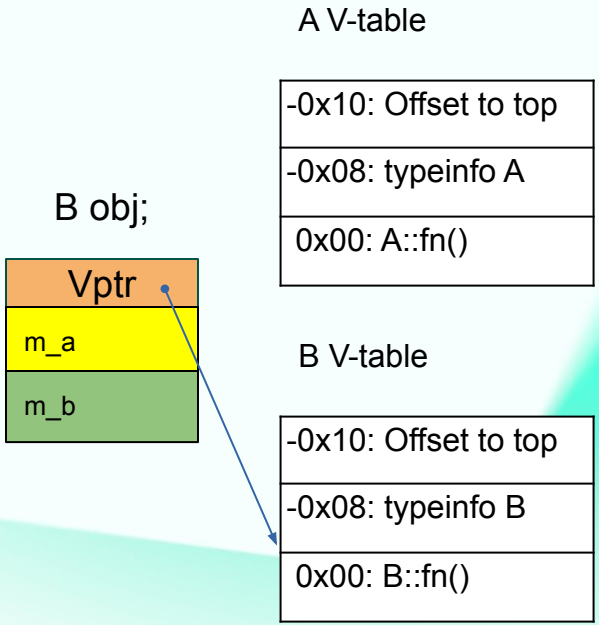
На каждом этапе конструирования объект видит картину мира **не выше текущего уровня**.

https://godbolt.org/z/rhz3Mc4P7

Конструирование

```
class A
{
  int m_a;
public:
  A();
  virtual int fn();
};
class B : public A
{
  int m_b;
public:
  B();
  virtual int fn() override;
};
A::A() { m_a = 10; }
B::B() { m_b = 5; }
```

Прежде чем стать полностью сконструированным объект переживает последовательно все стадии “своего развития”.



На каждом этапе конструирования объект видит картину мира **не выше текущего уровня**.

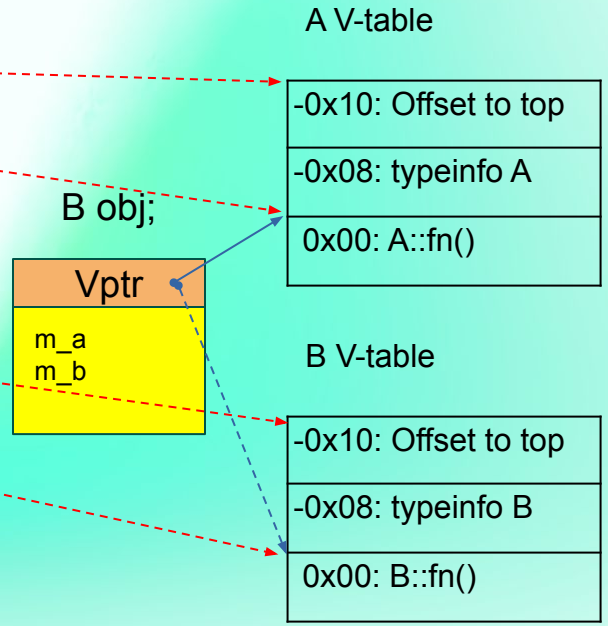
https://godbolt.org/z/rhz3Mc4P7

1
1

Деструкция

```
class A
{
  int m_a;
public:
  ~A();
  virtual int fn();
};
class B : public A
{
  int m_b;
public:
  ~B();
  virtual int fn() override;
};
A::~A() { m_a = 10; }
B::~B() { m_b = 5; }
```

```
2  A::~A() [base object destructor]:
       mov    rax, qword ptr [rip + vtable for A@GOTPCREL]
       add    rax, 16
       mov    qword ptr [rdi], rax
       mov    dword ptr [rdi + 8], 10
       ret
1  B::~B() [base object destructor]:
       mov    rax, qword ptr [rip + vtable for B@GOTPCREL]
       add    rax, 16
       mov    qword ptr [rdi], rax
       mov    dword ptr [rdi + 12], 5
       jmp    A::~A() [base object destructor] # TAILCALL
```



Каждый деструктор меняет свой vptr!

https://godbolt.org/z/dxoEPnbcx

Конструкция/Деструкция и ожидания

- Программист скорее ожидает что будет выполняться **наиболее верхнеуровневая логика** переопределенной виртуальной функции.

- По факту вызывается виртуальная функция не выше **текущего уровня**.

Как передаются объекты

Как передаются объекты по значению

```
struct A
{
  long a;
  long b;
  ~A() { printf("d"); }
};

long g_a, g_b;
void fn(A a)
{
  g_a = a.a;
  g_b = a.b;
}
```

```
fn(A):
    mov    rax, qword ptr [rdi]
    mov    qword ptr [rip + g_a], rax
    mov    rax, qword ptr [rdi + 8]
    mov    qword ptr [rip + g_b], rax
    ret
g_a:
    .quad  0                    # 0x0
g_b:
    .quad  0                    # 0x0
```

Эквивалент:
`void fn(const A& a)`

const - только с точки зрения внешних гарантий.
Модифицируем только локальную копию.

https://godbolt.org/z/eqYzb9bne

Как передаются объекты по значению

```
struct A
{
  long a;
  long b;
  //~A() { printf("d"); }
};

long g_a, g_b;
void fn(A a)
{
  g_a = a.a;
  g_b = a.b;
}
```

```
fn(A):
        mov     qword ptr [rip + g_a], rdi
        mov     qword ptr [rip + g_b], rsi
        ret
```

Для **standard layout** объектов Itanium ABI позволяет передавать объекты которые влазят в 2 регистра, через регистры.

https://godbolt.org/z/eqYzb9bne

Как возвращаются объекты по значению

```
struct A
{
  long a;
  long b;
  ~A() { printf("d"); }
};

A fn()
{
  A a{10, 15};
  return a;
}
```

```
fn():
        mov     rax, rdi
        mov     qword ptr [rdi], 10
        mov     qword ptr [rdi + 8], 15
        ret
```

Эквивалент:

```
void fn(A& ret);
```

https://godbolt.org/z/eqYzb9bne

Как возвращаются объекты по значению

```
struct A
{
  long a;
  long b;
  // ~A() { printf("d"); }
};

A fn()
{
  A a{10, 15};
  return a;
}
```

```
fn():
        mov     eax, 10
        mov     edx, 15
        ret
```

Для **standard layout** объектов Itanium ABI позволяет возвращать объекты которые влазят в 2 регистра, через регистры возврата.

https://godbolt.org/z/eqYzb9bne

Что это несет нашему любимому RAII

std::unique_ptr - умный УКАЗАТЕЛЬ.

Суть RAII - наличие пользовательского деструктора

->

не standard layout

->

Передается НЕ как обычный указатель через регистры,
а как **указатель на объект**.

Множественное наследование

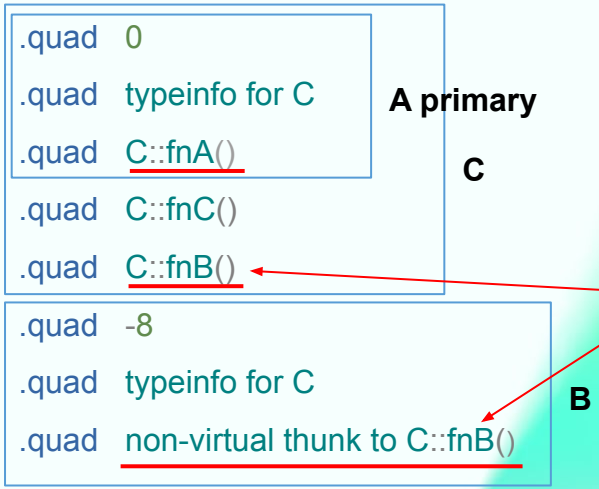
Дубликаты на множественном наследовании (replicated)

```
class A
{
public:
   virtual void fnA();
};

class B
{
public:
   virtual void fnB();
};

class C : public A, public B
{
public:
   virtual void fnC();
   void fnA() override;
   void fnB() override;
};
```

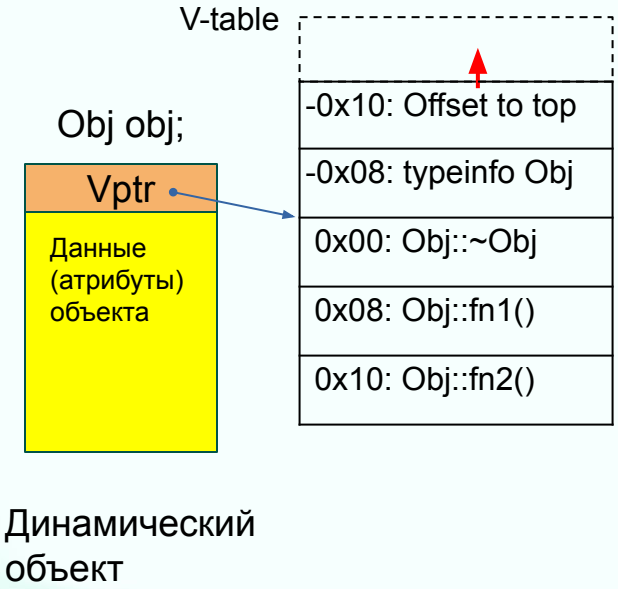
vtable for C:



- Thunk вызов происходит когда мы отдает подобъект B.
- Иначе используется не-thunk поле V-таблицы.
- Это позволяет избежать двойного преобразования **this**.

https://godbolt.org/z/raMq99167

Вспомним VTable (Itanium ABI)



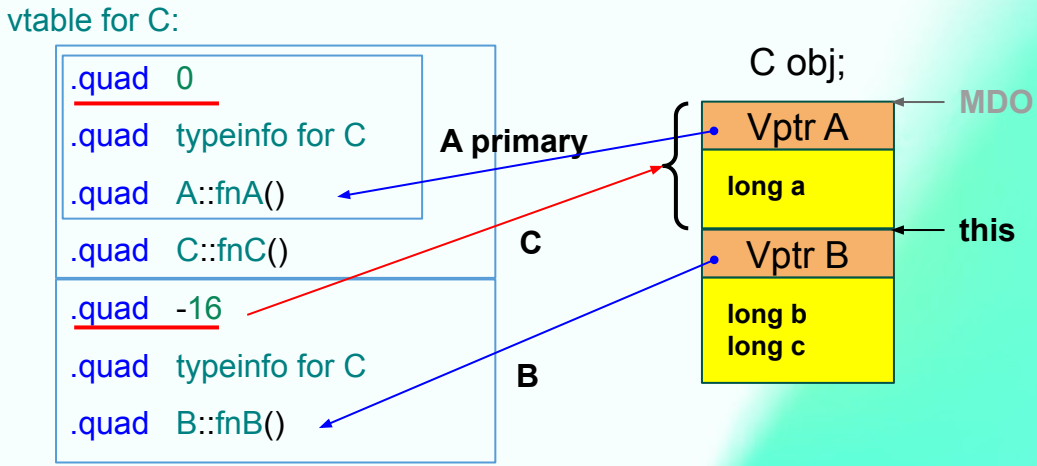
В **Itanium ABI** (сейчас используется на всех *nix платформах):

- 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 ;-)

Offset to top

```
class A
{
public:
  long a = 1;
  virtual void fnA();
};
class B
{
public:
  long b = 2;
  virtual void fnB();
};
class C : public A, public B
{
public:
  virtual void fnC();
  long c = 3;
};
```



Vtable[-2] - Offset to top

MDO = this + this[-2]

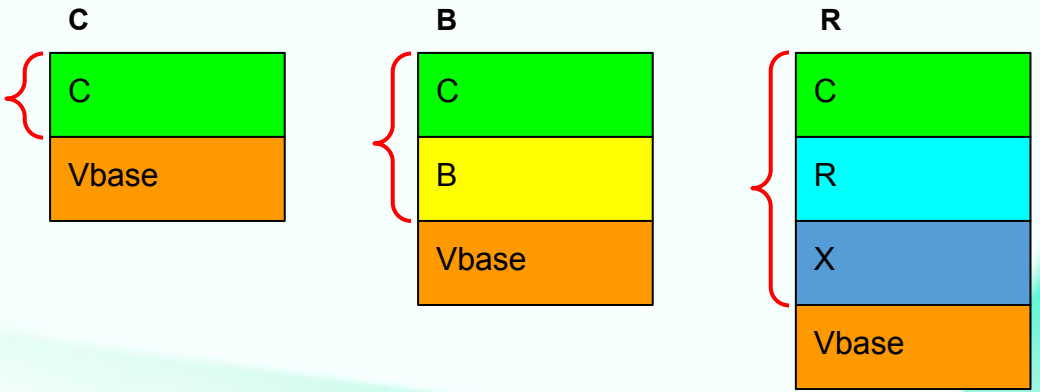
https://godbolt.org/z/heo35dzf4

Виртуальное наследование

Где же наш виртуальный предок

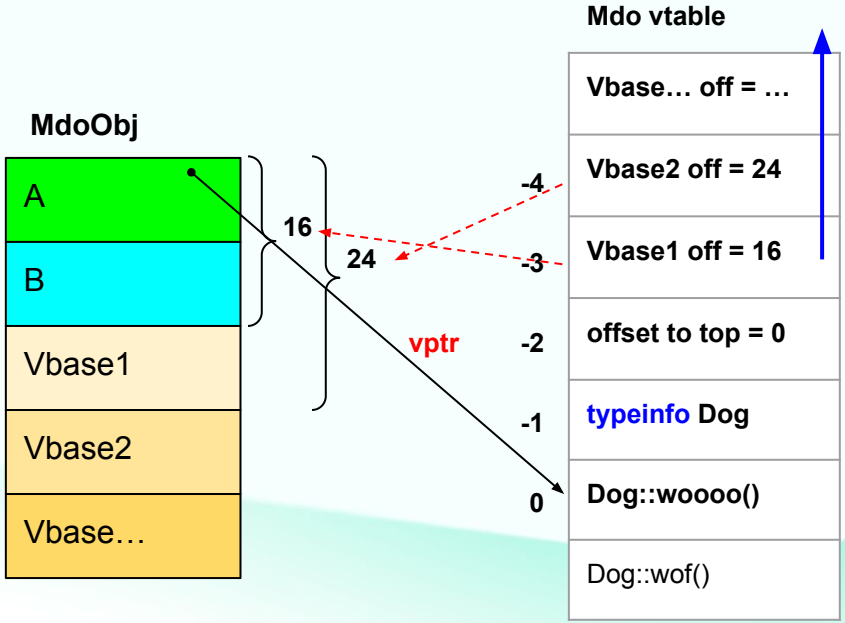
Мы уже обсуждали в прошлом докладе - виртуальные предки раскладываются в памяти объекта:

- в конце
- порядок (смещение) в разных MDO будет разный



Virtual Base (vbase) offset

Порядок следования виртуальных предков полностью определен - давайте в этом же порядке, но в отрицательном смещении разложим смещения в V-таблице.



Vbase offset - содержит смещение от начала объекта до соответствующего виртуального предка.

В зависимости от дерева наследования, и соответствующего размещения, может быть **отрицательное смещение**.

Virtual Base (vbase) offset пример

```
class A
{
public:
  long avirt = 1;
};
class B : virtual public A
{
public:
  long c = 3;
  virtual void fnB();
};
```

```
void B::fnB()
{
  avirt = 5;
}
```

Приводим **this MDO** к указателю на виртуального-предка **B**.

```
B::fnB():                          # @B::fnB()
        mov     rax, qword ptr [rdi]          // vptr = *this
        mov     rax, qword ptr [rax - 24]     // voff = vptr[-3]
        mov     qword ptr [rdi + rax], 5      // *(this + voff) = 5
        ret
```

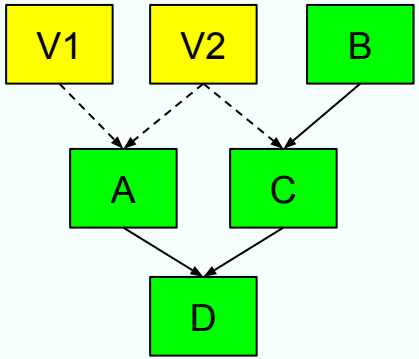
https://godbolt.org/z/r4qhPETn1

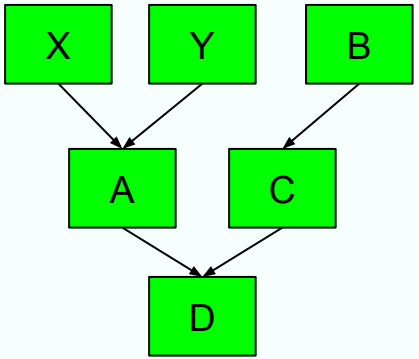
Конструирование виртуальных предков

С точки зрения C++, все виртуально наследуемые под-объекты принадлежат MDO, и MDO ответственен за их конструирование.

- Объект который конструируется как **MDO,** должен в конструкторе вызвать **конструкторы ВСЕХ виртуальных предков**.
- Объект который конструируется как **не-MDO (как подобъект), НЕ вызывает** в конструкторе **конструкторы виртуальных предков.**
- Для этого, для класса **с виртуальными предками** создаются 2 версии конструкторов: **base** и **complete.**
- При конструировании **MDO** с виртуальными предками, вызывается **complete** (только для корня графа наследования).
- **Complete** уже вызывает **base** конструкторы.
- Если у класса **нет виртуальных предков**, то **MDO** конструируется через **base**.

Конструкторы base и complete





D d;

D::D complete (&d)

- V1::V1 **base** (...)
- V1::V1 **base** (...)
- A::A **base** (...)
- C::C **base** (...)

D d;

D::D base (&d)

- A::A **base** (...)
- C::C **base** (...)

У классов без виртуальных предков,
есть только **base** конструктор.

Конструирование виртуальных предков (пример)

```
class Virt {
public:
  long virt = 1;
};
class A {
public:
  int a = 111;
};
class B : public A, virtual public Virt {
public:
  B();
  long c = 333;
  virtual void fnB();
};
```

```
B::B() [base object constructor]:
    push   r14
    push   rbx
    push   rax
    mov    rbx, rsi
    mov    r14, rdi
    add    rdi, 8
    call   A::A() [base object constructor]
    mov    rax, qword ptr [rbx]
    mov    qword ptr [r14], rax
    mov    qword ptr [r14 + 16], 333
    add    rsp, 8
    pop    rbx
    pop    r14
    ret
```

```
B::B() [complete object constructor]:
    push   rbx
    mov    rbx, rdi
    add    rdi, 24
    call   Virt::Virt() [base object constructor]
    lea    rdi, [rbx + 8]
    call   A::A() [base object constructor]
    lea    rax, [rip + vtable for B+24]
    mov    qword ptr [rbx], rax
    mov    qword ptr [rbx + 16], 333
    pop    rbx
    ret
```

При создании объекта, сначала конструируются **ВСЕ виртуальные предки.**

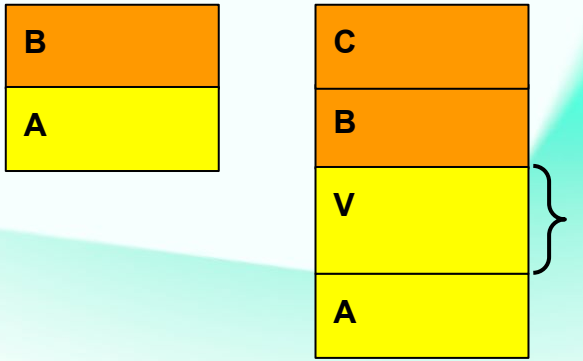
https://godbolt.org/z/h58E7Eqfc

Проблема переопределения метода

```
class A
{
public:
  long avirt = 1;
  virtual void fnA();
};
class B : virtual public A
{
  long c = 3;
public:
  virtual void fnA();
};
void B::fnA()
{
  c += avirt;
}
```

Мы переопределяем виртуальный метод виртуального предка:

- Его **this** должен указывать на переопределивший подобъект
- Сам виртуальный предок располагается по произвольному смещению
- Метод может вызываться функцией которая имеет только указатель объект виртуального предка
- **this должен быть приведен от указателя на предка к указателю на потомка!**

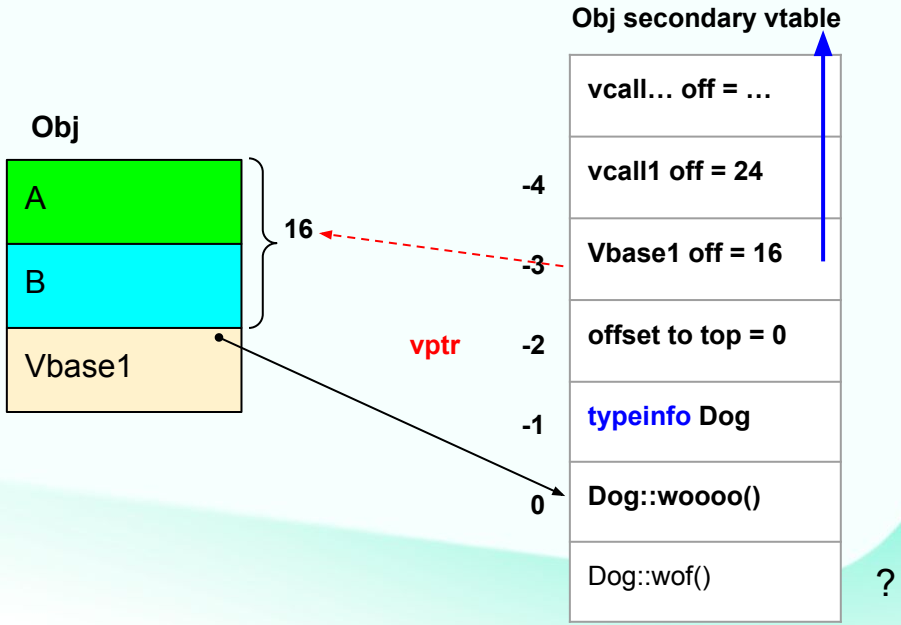


Варианты иерархий

```
void someFn(A& a)
{
  …
}
```

Virtual call (vcall) offset

Для каждого переопределенного метода из виртуального предка добавим слот со смещением внутри объекта, в V-таблицу **(в переопределяющем классе)**.



Vbase и **Vcall** смещения сначала от primary таблицы базового класса, потом собственные.

Virtual thunk

```
class A
{
public:
  long avirt = 1;
  virtual void fnA();
};
class B : virtual public A
{
  long c = 3;
public:
  virtual void fnA();
};
void B::fnA()
{
  c += avirt;
}
```

```
vtable for B:
  .quad   16
  .quad   0                    B
  .quad   typeinfo for B
  .quad   B::fnA()             <- Дубликат
  .quad   -16          // vcall offset
  .quad   -16          // offset to top      A
  .quad   typeinfo for B
  .quad   virtual thunk to B::fnA()   <- Virtual thunk

virtual thunk to B::fnA():
  mov     rax, qword ptr [rdi]      // vptr = *this
  add     rdi, qword ptr [rax - 24] // vcalloff = this + vptr[-3]
  jmp     B::fnA()  # TAILCALL
```

Мы хотим иметь унифицированный код в независимости от расположения.

Covariant return type

Covariant return types

```
class A
{
public:
  long count = 1;
  virtual A* clone();
};
class B : virtual public A
{
  long c = 3;
public:
  virtual B* clone() override;
};
B* B::clone() { count++; return new B; }
A* A::clone() { count++; return new A; }
```

C++ позволяет менять возвращаемый тип в переопределяемой функции если:

- Возвращается **указатель** или **ссылка** на объект
- Возвращаемый объект в переопределенном - является **потомком** (исходно возвращаемого)
- Возвращаемый в переопределенном - должен так же или менее **cv-qualified**

https://en.cppreference.com/w/cpp/language/virtual

Covariant return thunk

```
class A
{
public:
  long count = 1;
  virtual A* clone();
};
class B : virtual public A
{
  long c = 3;
public:
  virtual B* clone() override;
};
B* B::clone() { count++; return new B; }
A* A::clone() { count++; return new A; }
```

vtable for B:

```
B   .quad   16
    .quad   0
    .quad   typeinfo for B
    .quad   B::clone()
A   .quad   -16    // vcall offset
    .quad   -16
    .quad   typeinfo for B
    .quad   covariant return thunk to B::clone()
```

Дубликат

Уже знакомая корректировка **this**

covariant return thunk to B::clone():

```
push    rax
mov     rax, qword ptr [rdi]
add     rdi, qword ptr [rax - 24]
call    B::clone()                  // return B -> rax
mov     rcx, qword ptr [rax]    // vptrA = *rax
add     rax, qword ptr [rcx - 24] // rax = vptrA[-3]
pop     rcx
ret
```

B::clone() на выходе возвращает объект **B**,

Но вызывающий с базового класса, рассчитывает увидеть под-объект **A**.

```
void someFn(A& a)
{
   … = a.clone();
}
```

https://godbolt.org/z/G7PshjjP7

Covariant return thunk вариант 2

```
struct M { long m; };
struct N { long n; };
struct X: public M, public N
{ long x; };

class A
{
public:
  virtual N* clone();
};
class B : public A
{
public:
  virtual X* clone() override;
};
X* B::clone() { return new X; }
N* A::clone() { return new N; }
```

```
vtable for B:
    .quad   0
    .quad   typeinfo for B
    .quad   covariant return thunk to B::clone()
    .quad   B::clone()
```

Таблица primary класса

Дубликат

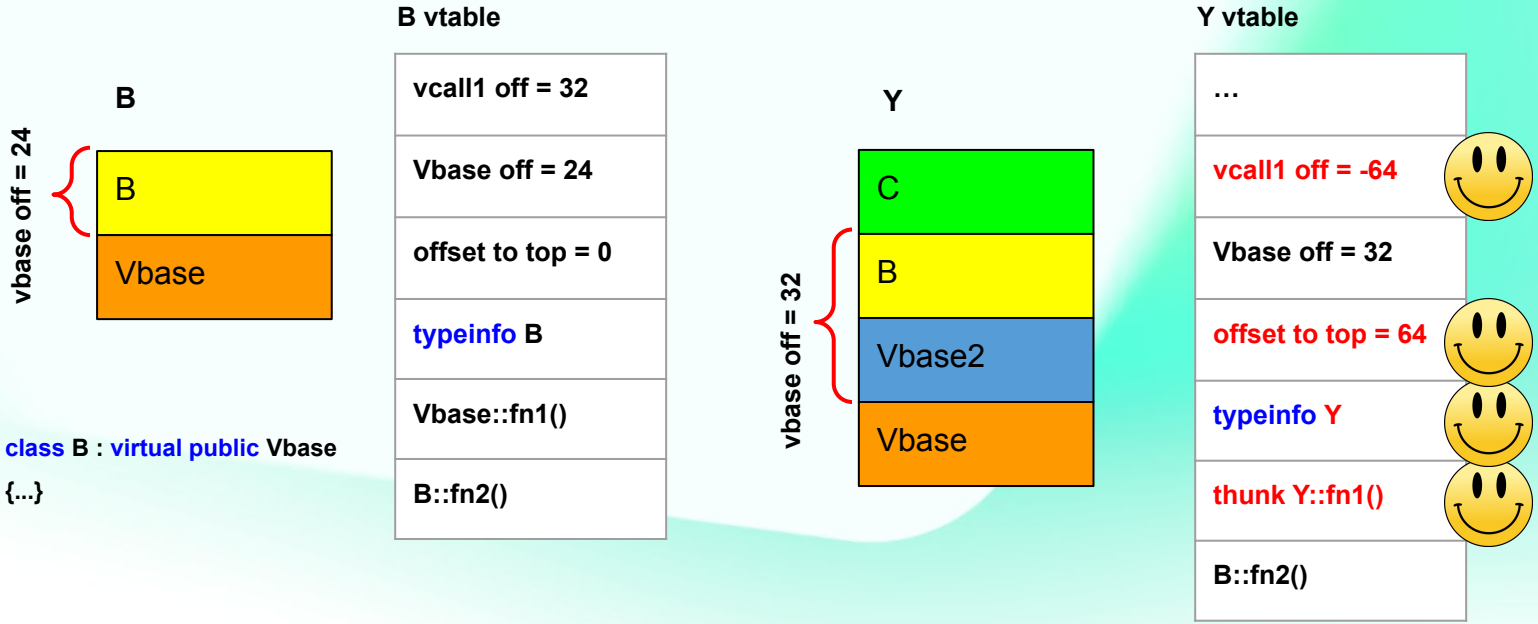
```
covariant return thunk to B::clone():
    push    rax
    call    B::clone() // return X -> rax
    add     rax, 8      // ptrN = rax + sizeof(M)
    pop     rcx
    ret
```

https://godbolt.org/z/nM3sx4Gj7

Конструируем виртуальных предков

Проблема конструирования виртуальных предков

Конструируем **B**, как подобъект в **Y**.
Какая должна использоваться **V-таблица** на **конструирования / деструкции**?



Construction V-table

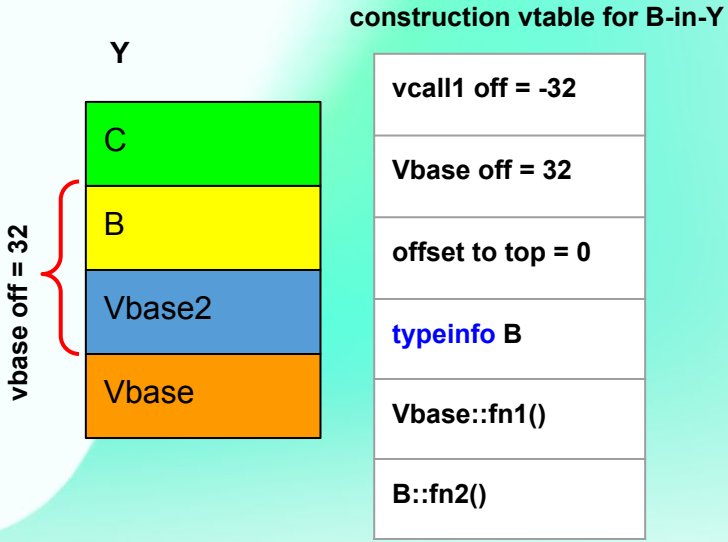
Компилятор создает дополнительную V-таблицу, специально для этапа конструирования / деструкции.

Берем из таблицы **B**

- адреса функций
- адрес RTTI
- Offset to top

Берем из таблицы **Y**

- Virtual base offsets
- Virtual call offsets (кроме переопределенных выше)



Construction V-таблицы являются свойством полного класса: **construction vtable for *-in-Y**

Только для под-классов имеющих виртуальных предков!

Проблема

- **Construction V-таблицы** - свойство **полного** класса.
- **->** Мы не можем хардкодить **construction V-таблицы** в конструкторы.

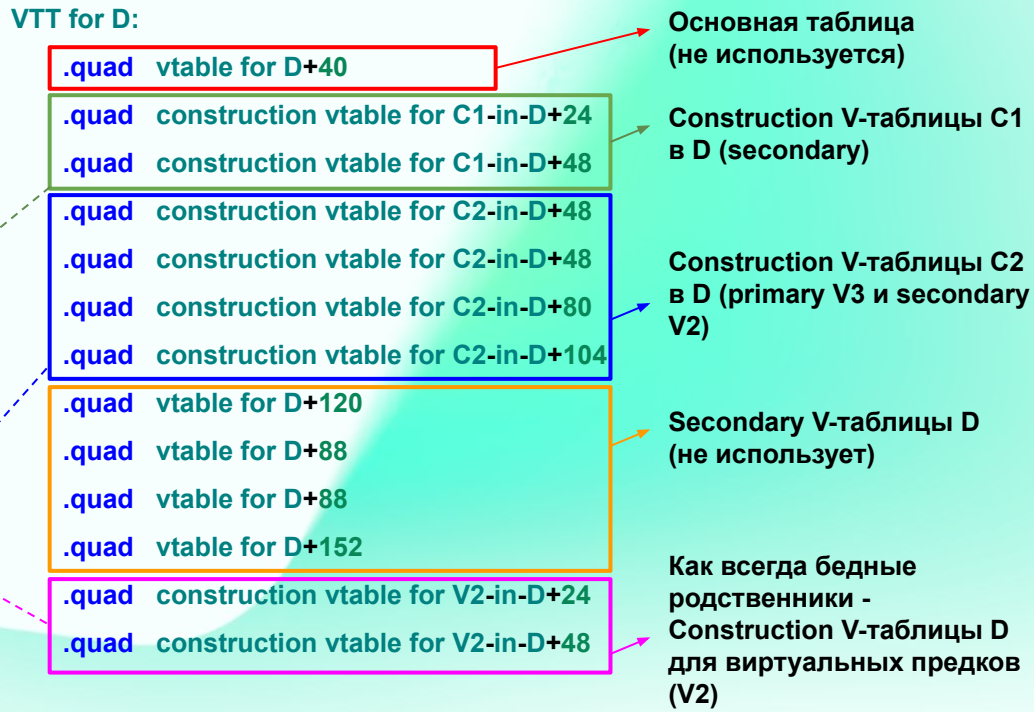
VTT (Virtual Table of Tables)

Одна таблица, чтобы контролировать все V-таблицы.

VTT (Virtual Table of Tables)

Одна таблица, чтобы контролировать все V-таблицы.

```
class A1 { int i; };
class A2 { int i; virtual void f(); };
class V1 : public A1, public A2 { int i; };
class B1 { int i; };
class B2 { int i; };
class V2 : public B1, public B2, public virtual V1
{ int i; };
class V3 {virtual void g(); };
class C1 : public virtual V1 { int i; };
// C1 has no primary base, V1 is secondary base
class C2 : public virtual V3, public virtual V2 { int i; };
// C2 has V3 primary (nearly-empty virtual) base, V2 is
secondary base
class X1 { int i; };
class C3 : public X1 { int i; };
class D : public C1, public C2, public C3 {...}
```



https://godbolt.org/z/Tca34a1z1

Что делать с VTT (Virtual Table of Tables)

При конструировании иерархии классов мы используем:

- **complete** конструктор - вызываем для MDO (полного объекта)
- **base** конструктор для классов имеющих виртуальных предков меняет сигнатуру на:

```
void C1(void * this, ….) //base constructor C1 Нет виртуальных предков
void ~C1(void * this) //base destructor C1 Нет виртуальных предков
void C2(void * this, void* vtt, ….) //base constructor C2 Есть виртуальные предки
void ~C2(void * this, void* vtt) //base destructor C2 Есть виртуальные предки
```

Неявный аргумент **vtt** - получает указатель на свою подгруппу таблиц в **VTT**.

Конструируем с VTT

Одна таблица, чтобы контролировать все V-таблицы.

```
D::D() [complete object constructor]:
        push    r15
        push    r14
        push    rbx
        mov     rbx, rdi        #D:this <- $rbx
        add     rdi, 40
        call    V1::V1() [base object constructor]
        lea     r14, [rbx + 16]    #D:this + 16
        mov     rdi, r14
        call    V3::V3() [base object constructor]
... #Сконструировали виртуальных предков
без не имеющих своих виртуальных предков
```

```
VTT for D:
        .quad   vtable for D+40
        .quad   construction vtable for C1-in-D+24
        .quad   construction vtable for C1-in-D+48
        .quad   construction vtable for C2-in-D+48
        .quad   construction vtable for C2-in-D+48
        .quad   construction vtable for C2-in-D+80
        .quad   construction vtable for C2-in-D+104
        .quad   vtable for D+120
        .quad   vtable for D+88
        .quad   vtable for D+88
        .quad   vtable for D+152
        .quad   construction vtable for V2-in-D+24
        .quad   construction vtable for V2-in-D+48
```

Основная таблица (не используется)

Construction V-таблицы C1 в D (primary и secondary)

Construction V-таблицы C2 в D (primary и secondary)

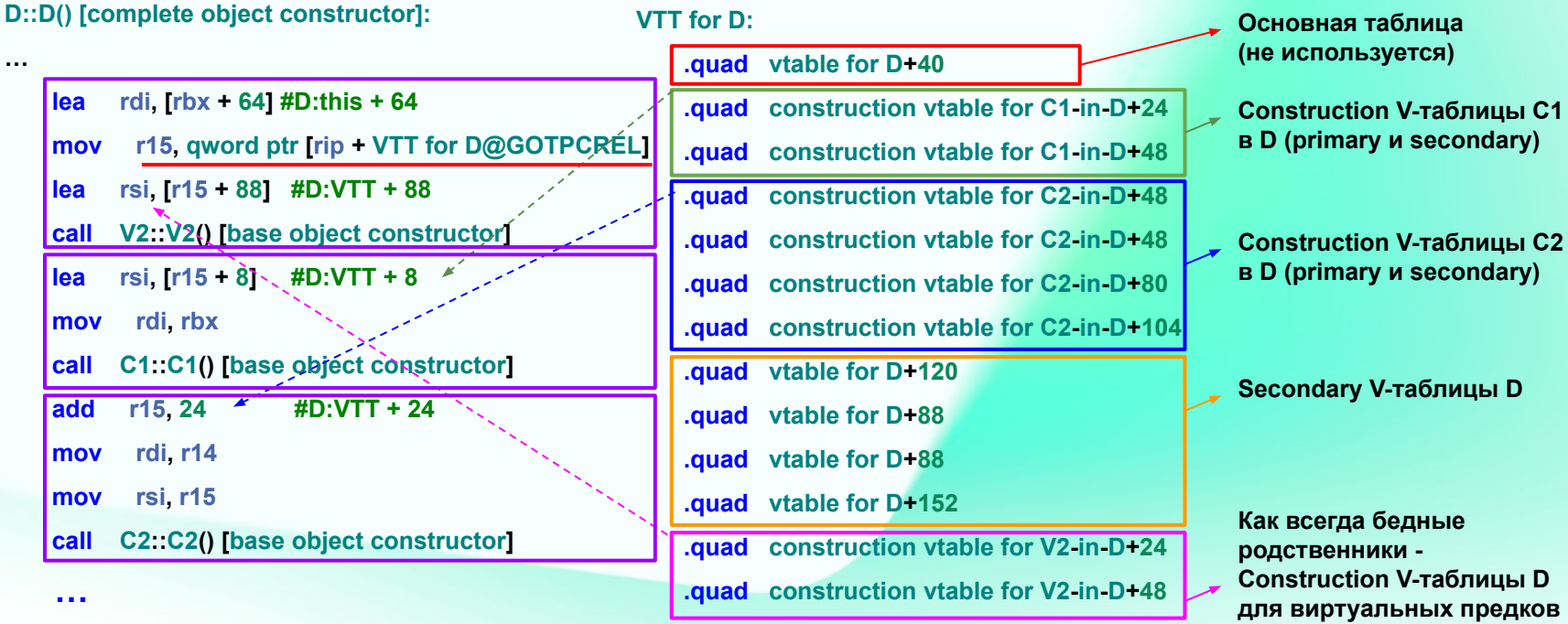
Secondary V-таблицы D

Как всегда бедные родственники - Construction V-таблицы D для виртуальных предков

https://godbolt.org/z/Tca34a1z1

Конструируем с VTT

Одна таблица, чтобы контролировать все V-таблицы.



https://godbolt.org/z/Tca34a1z1

Конструируем с VTT

Одна таблица, чтобы контролировать все V-таблицы.

D::D() [complete object constructor]:

```
...
    lea    rax, [rip + vtable for D+40]
    mov    qword ptr [rbx], rax          #D:this
    lea    rax, [rip + vtable for D+120]
    mov    qword ptr [rbx + 40], rax     #D:this + 40
    lea    rax, [rip + vtable for D+88]
    mov    qword ptr [rbx + 16], rax     #D:this + 16
    lea    rax, [rip + vtable for D+152]
    mov    qword ptr [rbx + 64], rax     #D:this + 64
    pop    rbx
    pop    r14
    pop    r15
    ret
```

VTT for D:

```
.quad   vtable for D+40
.quad   construction vtable for C1-in-D+24
.quad   construction vtable for C1-in-D+48
.quad   construction vtable for C2-in-D+48
.quad   construction vtable for C2-in-D+48
.quad   construction vtable for C2-in-D+80
.quad   construction vtable for C2-in-D+104
.quad   vtable for D+120
.quad   vtable for D+88
.quad   vtable for D+88
.quad   vtable for D+152
.quad   construction vtable for V2-in-D+24
.quad   construction vtable for V2-in-D+48
```

Основная таблица (не используется)

Construction V-таблицы C1 в D (primary и secondary)

Construction V-таблицы C2 в D (primary и secondary)

Secondary V-таблицы D

Как всегда бедные родственники - Construction V-таблицы D для виртуальных предков

Выставляем **Vptr**'ы при этом **VTT не используется**.

https://godbolt.org/z/Tca34a1z1

Деструкция

Почему 3 деструктора?

- **base object destructor** - деструкция без виртуальных предков
- **complete object destructor** - деструкция + деструкция виртуальных предков
- **deleting destructor** - деструкция + деструкция виртуальных предков + delete()

http://web.archive.org/web/20190930143149/https://ww2.ii.uj.edu.pl/~kapela/pn/cpp_vtable.html

2 слота деструктора

- **complete object destructor** - **не** вызывает **delete** в конце.
- **deleting destructor** - вызывает **delete** после деструкции.

Complete object destructor

Не вызывает **delete(void*)** на память лежащую под объектом.

```
char buf[1024];
class A {
public:
  int a;
  virtual void prnt();
  virtual ~A();
};
class C : public A {
public:
  int c;
  virtual ~C();
};
void create(void* buf) {
  new (buf) C;
}
void del(A* a) {
  a->~A();
}
```

```
vtable for C:
    .quad   0
    .quad   typeinfo for C
    .quad   A::prnt()
    .quad   C::~C() [base object destructor]
    .quad   C::~C() [deleting destructor]
```

```
del(A*):
# %bb.0:
    #DEBUG_VALUE: del:a <- $rdi
    mov     rax, qword ptr [rdi]
    jmp     qword ptr [rax + 8]   # TAILCALL
```

https://godbolt.org/z/hdY1MTxr8

Deleting destructor

```
class A {
public:
  int a;
  virtual void prnt();
  virtual ~A();
};
class C : public A {
public:
  int c;
  virtual ~C();
};
C* create() {
  return new C;
}
void del(A* a) {
  delete a;
}
```

Вызывает **delete(void*)** на память лежащую под объектом.

```
C::~C() [deleting destructor]:
        push    rbx
        mov     rbx, rdi
        call    C::~C() [base object destructor]
        mov     rdi, rbx
        pop     rbx
        jmp     operator delete(void*)@PLT    # TAILCALL
```

```
del(A*):
        test    rdi, rdi
        je      .LBB2_1
# %bb.2:
        mov     rax, qword ptr [rdi]
        jmp     qword ptr [rax + 16]    # TAILCALL
.LBB2_1:
        ret
```

https://godbolt.org/z/4GzrWq7bb

Общие правила деструкции

1. Вызывается **complete** или **deleting** деструктор.
2. Деструктор вызывает **base** деструкторы (возможно с **VTT**) непосредственных предков.
3. Деструктор вызывает **base** деструкторы (возможно с **VTT**) всех виртуальных предков.
4. Для **deleting** деструктора вызываем **delete(void*)**.

Указатели

Data Member Pointer

```
class A
{
public:
  int a;
  int b;
  int A::*selector;
};
void fn(int cond, A& a)
{
  if (cond > 10)
    a.selector = &A::a;
  else if (cond > 5)
    a.selector = &A::b;
  else
    a.selector = nullptr;
}
```

Мы можем делать указатель на член класса - такой указатель хранит **смещение** внутри класса. И внезапно **nullptr == -1**, т.к. 0 - валидное смещение.

```
fn(int, A&):
    cmp     edi, 11
    jl      .LBB0_2
    mov     qword ptr [rsi + 8], 0
    ret
.LBB0_2:
    cmp     edi, 6
    jl      .LBB0_4
    mov     qword ptr [rsi + 8], 4
    ret
.LBB0_4:
    mov     qword ptr [rsi + 8], -1
    ret
```

Этот ABI признан авторами неудачным!

Синтаксис использования к делу не относится, просто ломает голову =).

```
void set(A& a, int val)
{
   a.*(a.selector) = val;
}
```

https://godbolt.org/z/WTd4WxrKE

Member Function Pointer

Указатель на метод класса - это структура. Поле ptr - может быть в 2 ипостасях: указатель на виртуальную функцию и не-виртуальную.
На большинстве платформ, если младший бит 1, то в пртр смещение в V-таблице - 1, иначе указатель на адрес функции.

```
struct {
    fnptr_t ptr;
    ptrdiff_t adj;
};
```

младший бит 0: <type> **(ptr)(this + adj,** <args>**)** (вызов не-виртуальной функции)
младший бит 1: <type> **((this + adj)->vptr + ptr - 1)(this + adj,** <args>**)**

На некоторых ABI, указатель может иметь 1 в младшем разряде (ARM32), и схема чуть меняется.

https://www.codeproject.com/Articles/7150/Member-Function-Pointers-and-the-Fastest-Possible

Member Function Pointer: пример

```
struct {
    fnptr_t ptr;
    ptrdiff_t adj;
};
```

младший бит 0: <type> (ptr)(this + adj, <args>) (вызов не-виртуальной функции)
младший бит 1: <type> ((this + adj)->vptr + ptr - 1)(this + adj, <args>)

```
class A {
public:
  int fn1();
  virtual int fnV1();
  virtual int fnV2();
};
int (A::*ptr)() = &A::fnV2;

void call(A& a, int (A::*fn)()) {
  (a.*fn)();
}
```

```
call(A&, int (A::*)()): # a:rdi, fn:rsi,rdx
        add     rdi, rdx    # this + adj
        test    sil, 1
        je      .LBB0_2
        mov     rax, qword ptr [rdi]
        mov     rsi, qword ptr [rsi + rax - 1]
.LBB0_2:
        jmp     rsi   # TAILCALL не виртуальный
ptr:
        .quad   9
        .quad   0
```

https://godbolt.org/z/56Wz76116

Девиртуализация

Поможем компилятору

Для девиртуализации основные источники:

- Известен динамический тип
- Нет переопределений

Ключевое слово final

Большая часть девиртуализации происходит во фронтэнде компилятора.

```
class V1 final
{
};

class X
{
public:
  virtual void fnX1() final;
};
```

Ключевое слово говорит компилятору что виртуальные функции класса или конкретная функция не будет переопределена. Если тип соответствует - можем подставить не-виртуальный вызов.

Anonymous namespace

Большая часть девиртуализации происходит во фронтэнде компилятора.

```
namespace
{
  class A
  {
  };
}
```

Компилятору очевидно что он видит все возможные вариации класса внутри единицы трансляции, и снаружи никто переопределить не может.

Заэмитим таблицы

По умолчанию LLVM эмитит V-таблицу только с ключевой функцией. V-таблица позволяет мидл-энду найти информацию девиртуализации.

-fforce-emit-vtables - заставляет создавать V-таблицу во всех юнитах в COMDAT.

Строгий vptr

-fstrict-vtable-pointers - говорит компилятору, что **vptr** не будет меняться в течении жизни объекта.

Это дает мидл-энду дополнительные гарантии для оптимизаций.

[EXPERIMENTAL]

На самом деле опций больше - их можно найти в документации LLVM.

Как оптимизируем

Основное правило при оптимизациях -

дайте больше информации компилятору!

Заключение

Заключение

С этим докладом я НЕ хочу чтобы вы подумали что:

- **C++ ABI** переусложнен
- Неэффективен

О чем стоит подумать

- Этот ABI эффективно решает сложные задачи
- Но необходимо правильно моделировать решаемую задачу
- Не пренебрегайте паттернами проектирования для снижения сложности, такими как: Dependency Injection, …
- Изучайте принципы Объектно ориентированного проектирования и анализа

Спасибо!

Евгений Ерохин

Telegram: @the_gabber

kaspersky