

C++

Лекция VII

Classes: Multiple Inheritance

Множественное наследование - I

The simplest case

- публичное наследование
- среди предков нет дубликатов
- нет виртуальности

```
struct A { /* A-data-members */ };  
struct B { /* B-data-members */ };  
struct C : public A, public B { /* C-data-members */ };
```

Layout

```
struct A { /* A-data-members */ };  
struct B { /* B-data-members */ };  
struct C : public A, public B { /* C-data-members */ };
```

```
+-----+ +-+ +-+  
| A-data-members | |A| | |  
+-----+ +-+ +-+ | |  
| B-data-members | |B| |C|  
+-----+ +-+ | |  
| C-data-members | | |  
+-----+ +-+
```

(*) Важно: Layout является деталью реализации компилятора и не фиксирован. В текущем примере и всех последующих приведены примеры Layout-ов, используемых в популярных компиляторах.

Layout

```
// main.cpp
struct A { int a; };
struct B { int b; };
struct C : public A, public B { int c; };
int main() { C c; }
```

```
clang++ -Xclang -fdump-record-layouts -c main.cpp
```

```
...
Dumping AST Record Layout
0 | struct C
0 |   struct A (base)
0 |     int a
4 |   struct B (base)
4 |     int b
8 |     int c
  | [sizeof=12, dsize=12, align=4,
  |   nvsize=12, nvalign=4]
...
```

Calling member function

```
struct A { int a; };
struct B { int b; void f() const { std::cout << this; } };
struct C : public A, public B { int c; };

int main() {
    C c;
    std::cout << &c << " ";
    c.f();
}
```

Possible output: 0x7ffee986bf80 0x7ffee986bf84

Внутри метода `this` должен указывать на начало объекта `B`.

`this` передается в метод неявно, и чтобы его получить компилятор преобразует `C*` к `B*`. Смещение известно в compile-time, преобразование указателя выполняется в runtime.

Calling member function

```
struct A { int a; };  
struct B { int b; void f() const { std::cout << this; } };  
struct C : public A, public B { int c; };  
int main() {  
    C c;  
    c.f();  
}
```

вызов метода примерно выглядит так:

```
char* m = reinterpret_cast<char*>(&c);  
B* b = reinterpret_cast<B*>(m + offset(B));  
_ZNK1B1fEv(b); // b передается как this
```


Calling member function

```
struct A { int a; };  
struct B { int b; void f() const { std::cout << this; }  
struct C : public A, public B { int c; };  
int main() {  
    C c;  
    c.f();  
}
```

```
lea    rdi, [rbp - 16]    // rdi хранит адрес c  
add    rdi, 4             // получение this, 4 -- of  
call   _ZNK1B1fEv        // вызов функции
```

godbolt: [Click me](#)

(*) при приведении типов, компилятор может вычислить offset, так как ему известен layout.

Ambiguity

Неоднозначность

```
struct A { int p; };  
struct B { int p; };  
struct C : public A, public B {};  
  
int main() {  
    C c;  
    c.p = 10; // error  
}
```

error: member 'p' found in multiple base classes of different types

Ambiguity

Неоднозначность

```
struct A { int p; };  
struct B { int p; };  
struct C : public A, public B {};  
  
int main() {  
    C c;  
    c.A::p = 10; // OK  
}
```

(*) Для функций было бы аналогично: например, `c.A::f();`

(**) Для указателей код бы выглядел так: `pc->A::p = 10;`

Casting

upcast

```
struct A { int p; };
struct B { int p; };
struct C : public A, public B {};

int main() {
    C* pc = new C;

    B* pb = pc; // implicit cast (upcast)

    std::cout << std::hex << pc << " " << pb << std::endl;
}
```

Possible output: 0x602000000090 0x602000000094

```
+-----+ <----- pc (0x..90)
| A-data-members |
+-----+ <----- pb (0x..94)
| B-data-members |
+-----+
| C-data-members |
+-----+
```

Casting

upcast

```
struct A { int p; };  
struct B { int p; };  
struct C : public A, public B {};  
  
int main() {  
    C* pc = new C;  
  
    B* pb = pc; // implicit cast (upcast)  
}
```

Грубо говоря, это приведение работает таким образом:

```
char* p = reinterpret_cast<char*>(pc) + offset(B);  
pb = reinterpret_cast<B*>(p);
```

Casting

upcast, равенство указателей

```
struct A { int p; };  
struct B { int p; };  
struct C : public A, public B {};  
  
int main() {  
    C* pc = new C;  
    B* pb = pc;  
  
    std::cout << std::boolalpha  
                << "(pc == pb) - " << (pc == pb); // Equality t  
}
```

Output: pc == pb: true

Casting

downcast

```
struct A { int p; };  
struct B { int p; };  
struct C : public A, public B {};  
  
int main() {  
    C* pc = new C;  
    B* pb = pc;  
  
    pc = pb; // Error: no implicit cast (downcast)  
}
```

Casting

downcast

```
struct A { int p; };  
struct B { int p; };  
struct C : public A, public B {};  
  
int main() {  
    C* pc = new C;  
    B* pb = pc;  
  
    pc = static_cast<C*>(pb); // OK  
}
```

Избегайте downcast-приведения типов:

```
B b;  
B* pb = &b;  
C* pc = static_cast<C*>(pb); // Oops
```


Casting

crosscast

```
struct A { int p; };  
struct B { int p; };  
struct C : public A, public B {};  
  
int main() {  
    C* pc = new C;  
    B* pb = pc;  
  
    A* pa = static_cast<A*>(pb); // Error  
}
```

crosscast не работает, так как в compile-time не всегда возможно определить тип объекта, на который указывает pb, например:

```
pb = rand() ? new B : new C;
```

Casting

crosscast

```
struct A { int p; };  
struct B { int p; };  
struct C : public A, public B {};  
  
int main() {  
    C* pc = new C;  
    B* pb = pc;  
  
    A* pa = static_cast<A*>(static_cast<C*>(pb)); // OK  
}
```

Null pointers

Особый случай с нулевыми указателями

```
int main() {  
    C* pc = nullptr;  
    B* pb = pc;  
}
```

Вспомним примерно каким образом это работало:

```
char* p = reinterpret_cast<char*>(pc) + offset(B);  
pb = reinterpret_cast<B*>(p);
```

Такой способ не работает с нулевыми указателями.

Null pointers

Особый случай с нулевыми указателями

В compile-time проверить указатель на null не всегда возможно, поэтому есть небольшой оверхед в runtime:

```
pb = (pc == nullptr) ? nullptr : /* applying offset
```

Runtime overheads

1. Расчет адреса подобъекта: например при вызова метода этого подобъекта
2. Проверка указателей на nullptr во время приведения

Множественное наследование - II

Повторяющийся предок

- публичное наследование
- нет виртуальности

```
struct A { int f(); /* A-data-members */ };  
struct B : A { /* B-data-members */ };  
struct C : A { /* C-data-members */ };  
struct D : B, C { /* D-data-members */ };
```

Layout

```
+-----+ +-+ +-+ +-+
| A-data-members | |A| | |
+-----+ +-+ |B| | |
| B-data-members | | |
+-----+ +-+ +-+ | |
| A-data-members | |A| |D|
+-----+ +-+ |C| | |
| C-data-members | | |
+-----+ +-+
| D-data-members | | |
+-----+ +-+
```

- Объект типа D содержит два подобъекта типа A

(*) Layout не фиксирован и может отличаться у разных компиляторов

Layout

```
// main.cpp
struct A { int f(); /* A-data-members */ };
struct B : A { /* B-data-members */ };
struct C : A { /* C-data-members */ };
struct D : B, C { /* D-data-members */ };

int A::f() {}

int main() {
    D d;
}
```

```
clang++ -Xclang -fdump-record-layouts -c main.cpp
```

```
...
Dumping AST Record Layout
0 | struct D
0 |   struct B (base)
0 |     struct A (base)
0 |       [A-data-members]
4 |       [B-data-members]
8 |   struct C (base)
8 |     struct A (base)
8 |       [A-data-members]
12 |     [C-data-members]
16 |   [D-data-members]
...
```

Access members

```
int main() {  
    D* d = new D;  
    d->f(); // Error  
}
```

error: non-static member 'f' found in multiple base-class subobjects of type 'A'

Access members

```
int main() {  
    D* d = new D;  
    d->C::f(); // OK  
}
```

(*) Access members

Что, если бы класс `C` имел функцию `f` с аналогичной сигнатурой?
Как тогда вызвать `f` для подобъекта `A`, который находится в подобъекте `C`?

```
d->C::A::f(); // Будет ли такое работать?
```

(*) Access members

Вызов `d->C::A::f()` не скомпилируется, так как компилятор не понимает на каком подобъекте вызвать метод.

Объяснение

```
d->C::A::f()
```

можно интерпретировать как

```
int(A::*method)() = &C::A::f;  
(d->*method)();
```

- Тип указателя на метод не хранит информацию о `C`, поэтому при вызове нельзя понять, какому подобъекту адресован вызов метода

(*) Access members

Продолжение объяснения

Случай

```
d->C::f()
```

можно интерпретировать как

```
int(C::*method)() = &C::f;  
(d->*method)();
```

Так как метод может быть вызван для объектов типа C, то тип указателя d будет преобразован к C* — это преобразование однозначно.

Casting

upcast

```
D* d = new D;  
A* a1 = d;           // Error  
A* a2 = static_cast<A*>(d); // Error
```

Неявное и явное преобразования типов не работают, так как неоднозначен выбор подобъекта

Casting

upcast

Нужно явно указать компилятору, как разрешить неоднозначность:

```
D* d = new D;  
A* a = static_cast<A*>(static_cast<C*>(d)); // OK
```


Casting

downcast

```
D* d = new D;  
A* a = static_cast<A*>(static_cast<C*>(d));  
  
d = static_cast<D*>(a); // Error  
d = static_cast<D*>(static_cast<C*>(a)); // OK
```

(*) Будьте осторожнее с downcast-ами

Множественное наследование - III

Пример с виртуальными функциями

- публичное наследование
- среди предков нет дубликатов

```
struct A {  
    virtual void f() {}  
    /* A-data-members */  
};  
struct B {  
    virtual void g() {}  
    /* B-data-members */  
};  
struct C : A, B {  
    void f() override {}  
    void g() override {}  
    /* C-data-members */  
};
```

Пример с виртуальными функциями

Вызов виртуальных функций

```
C* pc = new C;  
A* pa = pc;  
B* pb = pc;  
  
pa->f();  
// C::f() called because A::f() overridden  
  
pb->g();  
// C::g() called because B::g() overridden
```

(*) Слайд для того чтобы вспомнить, как работают виртуальные функции

Layout

+-----+	++	++
A vtable ptr		
A-data-members	A	
+-----+	++	++
B vtable ptr		
B-data-members	B	C
+-----+	++	
C-data-members		
+-----+		++

- overhead: несколько указателей на виртуальную таблицу

virtual function call under the hood

```
int main() {  
    C* pc = new C;  
    B* pb = pc;  
  
    pb->g(); // how does it work?  
}
```

1. Таблица виртуальных методов предоставит нам нужный адрес метода `g`
2. Нужно получить указатель, который будет использован как `this` в `g`. Один из способов это сделать — это хранить в таблице смещение. Тогда: `this = pb + offset(g)`

(*) В общем случае, вычисления, описанные выше, нельзя сделать в `compile-time`, так как неизвестно, на какой объект/подобъект указывает ``pb`` на самом деле

Множественное наследование - IV

Проблема дублирующихся подобъектов

```
struct A { /* A-data-members */ };
struct B : A { /* B-data-members */ };
struct C : A { /* C-data-members */ };
struct D : B, C { /* D-data-members */ };
```

```
+-----+ +-+
| A-data-members | |A| <--+
+-----+ +-+      |
| B-data-members |      +--- duplicates
+-----+ +-+      |
| A-data-members | |A| <--+
+-----+ +-+
| C-data-members |
+-----+
| D-data-members |
+-----+
```

Обычно при таком наследовании хотелось бы иметь один подобъект типа A.

Виртуальное наследование

```
struct A { /* A-data-members */ };  
struct B : virtual A { /* B-data-members */ };  
struct C : virtual A { /* C-data-members */ };  
struct D : B, C { /* D-data-members */ };
```

Virtual Inheritance Layouts

Object B	Object D
+-----+	+-----+
B vtable ptr	B vtable ptr
B-data-members	B-data-members
+-----+	+-----+
A-data-members	C vtable ptr
+-----+	C-data-members
	+-----+
	D-data-members
	+-----+
	A-data-members
	+-----+

- Подобъекты B и C разделяют подобъект типа A (в объекте D)
- Так как A общий, он не может быть жестко зафиксирован относительно B и C. Поэтому B и C используют указатель для доступа к A.

Casting

upcast

```
struct A { /* A-data-members */ };
struct B : virtual A { /* B-data-members */ };
struct C : virtual A { /* C-data-members */ };
struct D : B, C { /* D-data-members */ };

int main() {
    D* d = new D;
    B* b = d;    // OK
    A* a = d;    // OK
}
```

Чтобы преобразовать к типу A, нужно воспользоваться `vtable`.

Casting

downcast

```
struct A { /* A-data-members */ };
struct B : virtual A { /* B-data-members */ };
struct C : virtual A { /* C-data-members */ };
struct D : B, C { /* D-data-members */ };

int main() {
    D* d = new D;
    A* a = d;

    d = static_cast<D*>(a); // compile-time error
}
```

- Подобъект A не содержит "статическую" информацию об объемлющем объекте.
- Добраться до D можно другими способами: `virtual function` или `dynamic_cast`

Virtual functions

```
struct A { virtual void f() {} };
struct B : virtual A { int b; };
struct C : virtual A { int c; };
struct D : B, C { void f() override {} };

int main() {
    D* d = new D;
    A* a = d;
    a->f(); // Будет вызван D::f()
}
```

```
+-----+
| B vtable ptr |
| B-data-members |
+-----+
| C vtable ptr |
| C-data-members |
+-----+
| D-data-members |
+-----+
| A vtable ptr | <--- Появляется указатель на vtable
| A-data-members |
+-----+
```

Constructors & Destructors

- Конструкторы базовых классов вызываются до конструктора производного класса
- Деструкторы базовых классов вызываются после деструктора производного класса

Constructors & Destructors

порядок вызова

- Конструкторы виртуальных базовых классов вызываются в первую очередь в порядке объявления их классов в списке базовых классов
- Остальные конструкторы вызываются в порядке объявления их классов в списке базовых классов
- Деструкторы вызываются в порядке, обратном порядку вызовов конструкторов

Constructors & Destructors

example 1

```
struct A { };
struct B { };
struct C : B { };
struct D : A, C { };
int main() {
    D d; // A() -> B() -> C() -> D()
        // ~D() -> ~C() -> ~B() -> ~A()
}
```


Constructors & Destructors

example 2

```
struct E {};  
struct A {};  
struct B : virtual A {};  
struct C : virtual A {};  
struct D : E, B, C {};  
  
int main() {  
    D d;  
    // A() -> E() -> B() -> C() -> D()  
    // ~D() -> ~C() -> ~B() -> ~E() -> ~A()  
}
```

Полиморфный объект

Объект, тип которого объявляет или наследует хотя бы один виртуальный метод.

Такие объекты хранят дополнительную информацию: обычно в виде указателя.

Проверка типа на полиморфность

```
#include <iostream>
#include <type_traits>

struct T {};
struct P { virtual void f(); };
struct D : P { };

int main() {
    std::cout << std::boolalpha
        << "T: " << std::is_polymorphic<T>::value << " "
        << "P: " << std::is_polymorphic<P>::value << " "
        << "D: " << std::is_polymorphic<D>::value << " "
}
```

Output:

```
T: false
P: true
D: true
```

RTTI

Run-time type information

- Позволяет пользоваться информацией о типе **полиморфного** объекта во время исполнения
- Используется для реализации `dynamic_cast` и `typeid`
- Обычно у компиляторов есть возможность отключения этого механизма: `-fno-rtti`

dynamic_cast & polymorphic types

- Используется со ссылками и указателями на полиморфные типы
- Позволяет делать upcast, downcast, crosscast
- Если приведение невозможно, возвращает nullptr для указателей и выбрасывает исключение `std::bad_cast` для ссылок
- Работает при downcast-е от полиморфной виртуальной базы
- `dynamic_cast<void*>` возвращает указатель на начало объекта в памяти

The "dark side" of `dynamic_cast`

`dynamic_cast` не бесплатен и оверхед зависит от того, как устроен механизм RTTI в компиляторе

Нельзя определенно сказать насколько плох `dynamic_cast`, так как его реализации разнятся от компилятора к компилятору

typeid operator

Syntax:

```
typeid( type )      (1)  
typeid( expression ) (2)
```

- Выражение имеет тип, который преобразуется к `const std::type_info&`
- Если `expression` возвращает объект полиморфного типа, который имеет адрес ¹, то выражение будет вычислено и `typeid` вернет `type_info` для динамического типа выражения. Иначе выражение вычислено не будет.

(1) glvalue expression

std::type_info

- содержит имя типа (не фиксировано стандартом)
- содержит хэш-код для типа
- задает порядок на типах: `type_info::before(const type_info&)`
- позволяет определить равны ли типы: `operator==`

Пример 1

```
struct A { };\nstruct B { };\n\nint main() {\n    const std::type_info& a_info = typeid(A);\n    const std::type_info& b_info = typeid(B);\n\n    std::cout << a_info.name() << " " <<\n        << a_info.hash_code() << " " << a_info.before(b_info) << "\n";\n\n    assert(a_info == typeid(A));\n    assert(a_info != typeid(B));\n}
```

Пример 2

```
struct A { };  
int i = 0;  
A makeA() { i = 1; return A(); }  
  
int main() {  
    // The expression will not be evaluated  
    // because has non-polymorphic type  
    const std::type_info& info = typeid(makeA());  
    assert(i == 0);  
}
```

Пример 3

```
struct A { virtual ~A() = default; };
struct B: A {};

int main() {
    A* a = new B;
    assert(typeid(a) == typeid(A*));

    assert(typeid(a) != typeid(A));
    assert(typeid(a) != typeid(B));

    // The expression `*a` will be evaluated
    // 1) It has polymorphic type
    // 2) The result object has an address
    assert(typeid(*a) == typeid(B));
}
```

Пример 4

```
struct A { virtual ~A() = default; };
struct B: A {};

int i = 0;
B createB() { i = 1; return B{}; }

int main() {
    // The expression `createB()` will not be evaluated
    // 1) It has polymorphic type
    // 2) But the result object has no address
    assert(typeid(createB()) == typeid(B));
    assert(i == 0);
}
```

