

Наследование

Введение в объектно-ориентированное программирование

03.04.2025

- Базовый синтаксис: арифметика, while, if, print, ?, объявления переменных

```
fst = 0; // тип не требуется, все типы int
```

```
snd = 1;
```

```
iters = ?; // считать со stdin число, определить переменную
```

```
while (iters > 0) { // синтаксис для if такой же
```

```
tmp = fst;
```

```
fst = snd;
```

```
snd = snd + tmp;
```

```
iters = iters - 1;
```

```
}
```

```
print snd;
```

- Грамматика представляется синтаксическим деревом

```
while (iters > 0) {  
  iters = iters - 1;  
}
```

- И тут есть проблема: допустим мы хотим представить узел такого дерева
- Но все узлы очень разные

```
struct Node {  
    Node *parent_  
    Node_t type_; // enum Node_t  
    union Data {  
        struct Decl { std::string declname_; } decl_  
        struct Binop { BinOp_t op_; } binop_; // enum BinOp_t  
        // .... все остальные варианты ....  
    } u;  
    std::vector<Node *> childs_  
};
```

- Покритикуйте этот подход

Вторая попытка: void pointers

- Что если мы заведём структуру которая знает свой тип?

```
struct Node {  
    Node *parent_;  
    Node_t type_;  
    void *data_;  
};
```

- Конкретные узлы хранят базовую часть

```
struct BinOp {  
    Node basepart_;  
    BinOp_t op_;  
    Node *lhs_, *rhs_;  
};
```

Вторая попытка: void pointers

- Теперь можно написать функцию-конструктор для бинарной операции

```
Node* create_binop(Node *parent, BinOp_t opcode) {  
    Node base = {parent, Node_t::BINOP, nullptr};  
    BinOp *pbop = new BinOp {base, opcode, nullptr, nullptr};  
    pbop->basepart_.data_ = static_cast<void *>(pbop);  
    return &pbop->basepart_;  
}
```

- Мне кажется даже не надо просить это покритиковать
- Этот код является худшей критикой самого себя

Лучшее решение: поддержка в языке

- Кажется для идеи “B является A” (также называется “отношение is-a”) в языке нужна непосредственная поддержка
- Это называется **наследование** и его открытая форма записывается через двоеточие и ключевое слово `public`
`class A {};`
`class B : public A {};` // B is also A
- Это отношение открытого наследования позволяет нам переписать отношения более явно

- Мы сэкономили сколько-то данных

```
struct Node {  
    Node *parent_;  
    Node_t type_;  
};
```

- Но главное мы получили куда лучшую запись

```
struct BinOp : public Node {  
    BinOp_t op_;  
    Node *lhs_, *rhs_;  
};
```


- Теперь функция-конструктор станет и впрямь конструктором

```
struct Node {  
    Node *parent_;  
    Node_t type_;  
};  
  
struct BinOp : public Node {  
    BinOp_t op_;  
    Node *lhs_ = nullptr, *rhs_ = nullptr;  
    BinOp(Node *parent, BinOp_t opcode) :  
    Node{parent, Node_t::BINOP}, op_(opcode) {}  
};
```

- Поскольку объект производного класса является объектом базового класса, указатели и ссылки приводятся неявным приведением
- Обратно можно привести через `static_cast`

```
struct Node;  
struct BinOp : public Node;  
void foo(const Node &pn);  
BinOp *b = new BinOp(p, op);  
foo(*b); // ok  
Node *pn = b; // ok  
b = static_cast<BinOp*>(pn); // ok
```

Проблема срезки: первое приближение

```
struct A {  
    int a_;  
    A(int a) : a_(a) {}  
};  
struct B : public A {  
    int b_;  
    B(int b) : A(b / 2), b_(b) {}  
};  
B b1(10);  
B b2(8);  
A& a_ref = b2;  
a_ref = b1; // b2 == ???
```

- Базовая срезка возникает из-за того, что присваивание не полиморфно

```
struct A {
```

```
int a_;
```

```
A(int a) : a_(a) {}
```

```
A& operator=(const A& rhs) { a_ = rhs.a_; }
```

```
};
```

```
a_ref = b1; // a_ref.operator=(b1); b1 приводится к const A&
```

- Было бы здорово если бы функция во время выполнения вела себя по разному в зависимости от настоящего типа своего первого аргумента.
- Увы, для конструкторов копирования это недостижимо на практике.

- Когда мы работаем с классическим ООП и наследованием, мы работаем с указателями и ссылками.

- Мы можем спроектировать классы `Triangle` и `Polygon` так, чтобы они имели общий метод `square()`, вычисляющий их площадь.
- Можем ли мы сохранить массив из неважно каких объектов лишь бы они имели этот метод?
- Ответ да: для этого мы должны сделать для них общий интерфейс от которого они оба наследуют.

```
struct ISquare { void square(); };  
struct Triangle : public ISquare; // реализует square()  
struct Polygon : public ISquare; // реализует square()  
std::vector<ISquare*> v; // хранит и Triangle* и Polygon*
```

- Мы можем спроектировать классы Triangle и Polygon так, чтобы они имели общий метод square(), вычисляющий их площадь.
- Можем ли мы сохранить массив из неважно каких объектов лишь бы они имели этот метод?
- Ответ да: для этого мы должны сделать для них общий интерфейс от которого они оба наследуют.

```
struct ISquare {  
    void square();
```

```
};
```

• Проблемы возникают с тем как здесь **реализовать** этот метод в ISquare.

```
struct ISquare {  
    virtual double square() const;  
};  
struct Triangle : public ISquare {  
    Point x, y, z;  
    double square() const;  
}
```

- Это всё ещё очень плохой код (здесь три ошибки в семи строчках), мы скоро его улучшим.
- Но он иллюстрирует концепцию. Простое совпадение имени означает переопределение (overriding) виртуальной функции

Таблица виртуальных функций

- При создании класса с хотя бы одним виртуальным методом, в него добавляется `vptr`.
- Конструктор базового класса динамически выделяет память для таблицы виртуальных функций.
- Конструктор каждого потомка производит инициализацию её своими методами. В итоге там всегда оказываются нужные указатели.

- Сначала конструируется подобъект базового класса, который невидимо конструирует себе таблицу виртуальных функций.
- Потом конструктор подобъекта производного класса невидимо заполняет её адресами своих методов.

Проблемы с overloading

- Здесь допущена обычная человеческая ошибка с типами int vs long.

```
struct Matrix {  
    virtual void pow(int x); // возведение в степень любой матрицы  
};  
struct SparseMatrix : Matrix{  
    void pow(long x); // возведение в степень разреженной матрицы  
    // крайне эффективный алгоритм  
};  
Matrix *m = new SparseMatrix;  
m->pow(3); // увы, вызовется Matrix::pow
```

```
struct ISquare {  
    virtual double square() const;  
};  
struct Triangle : public ISquare {  
    Point x, y, z;  
    double square() override const;  
}
```

- Это всё ещё очень плохой код, мы скоро его улучшим.
- Следующая проблема это как нам написать тело самой общей функции? Тела наследников понятны. Но что должно быть в самой ISquare::square? Может быть аборт?

```
struct ISquare {  
    virtual double square() const = 0;  
};  
struct Triangle : public ISquare {  
    Point x, y, z;  
    double square() override const;  
}
```

- Это всё ещё очень плохой код, мы скоро его улучшим.
- Проблема решается чисто виртуальными методами которые не требуют определения и только делегируют наследникам.
- Объект класса с чисто виртуальными методами не может быть создан.

```
struct ISquare {  
    virtual double square() const = 0;  
};  
struct Triangle : public ISquare {  
    Point x, y, z;  
    double square() override const;  
}
```

- Это всё ещё очень плохой код, мы скоро его улучшим.
- Следующая проблема: удаление по указателю на базовый класс.
`ISquare *sq = new Triangle<int>; delete sq; // утечка`

- Мы хотим, чтобы удаление по указателю на базовый класс вызывало правильный деструктор производного класса.
- Это означает, что нам нужен **виртуальный** деструктор.

```
struct ISquare {  
    virtual double square() const = 0;  
    virtual ~ISquare()  
};  
struct Triangle : public ISquare {}  
ISquare *sq = new Triangle<int>;  
delete sq; // Ok, вызван Triangle::~~Triangle()
```

- Класс в котором все методы чисто виртуальные служит своего рода общим интерфейсом.

```
struct ISquare {  
    virtual double square() const = 0;  
    virtual ~ISquare() {}  
};
```

- Такой класс называется абстрактным базовым классом.
- К сожалению виртуальный конструктор (в том числе копирующий) невозможен.
- Тогда непонятно как нам скопировать по базовому классу.

- Обычно используется виртуальный метод clone.

```
struct ISquare {  
    // всё остальное  
    virtual ISquare *clone() const = 0;  
};  
  
struct Triangle : public ISquare {  
    std::array<Point, 3> pts_  
    Triangle *clone() const override { return new Trianglepts_; }  
}
```

- Обратите внимание: override здесь законный поскольку Triangle* открыто наследует и значит является ISquare*

Срезка возвращается

- Из-за невозможности виртуальных конструкторов, срезка возможна при передаче по значению.

```
void foo(A a) std::cout << a << std::endl;  
B b(10); foo(b1); // на экране "5"
```

- Поэтому никогда не передавайте объекты базовых классов по значению

- Используйте указатель или ссылку.

```
void foo(A& a) { std::cout << a << std::endl; }  
B b(10); foo(b1); // на экране "5 10"
```

```
struct ISquare {  
    virtual double square() const = 0;  
    virtual ~ISquare() {}  
};  
struct Triangle : public ISquare {  
    Point x, y, z;  
    double square() override const;  
}
```

- Вот это уже неплохо.
- Но хотя этот код стал неплохим, концептуально у нас проблем

- Допустим мы написали некий класс Foo.
- Писать ли у него виртуальный деструктор?
- Если мы хотим чтобы от него наследовались то да писать.
- Если не хотим и не хотим оверхеда на vtable, то можно объявить его final.

```
struct Foo final { // content };
```

- Теперь наследование будет ошибкой компиляции.

Пишем правильно: четыре способа

- Класс в C++ написан правильно если и только если любое из условий выполнено:
 1. Класс содержит виртуальный деструктор.
 2. Класс объявлен как `final`.
 3. Класс является `stateless` и подвержен EBCO.
 4. Класс не может быть уничтожен извне, но может быть уничтожен потомком.
- Первые два варианта мы уже обсудили.
- Давайте поговорим о третьем и четвёртом.

Empty Base Class Optimizations

- Оптимизации пустого базового класса (EBCO) применяются когда базовый класс хм... пустой.

```
class A{};
```

```
class B : public A{};
```

```
A a; assert(sizeof(a) == 1);
```

```
B b; assert(sizeof(b) == 1);
```

- Заметьте, класс с хотя бы одним виртуальным методом точно не пустой.
- Пока неясно зачем нам вообще такие ребята. Они сыграют позже, так как нужны для так называемых **МИКСИНОВ**.

- Модификатор `protected` служит для защиты от всех, кроме наследников.
- Он позволяет писать чисто-базовые классы.

```
class PureBase {  
    // что угодно  
protected:  
    ~PureBase() {}  
};
```

- Теперь объект класса-наследника просто нельзя удалить по указателю на базовый класс и проблема снимается.
- Если не удалять изнутри класса и тогда всё по прежнему.

Пишем правильно: два способа

- Класс в C++ написан правильно если и только если любое из условий выполнено:
 1. Класс содержит виртуальный деструктор
 2. Класс объявлен как `final`
 3. Класс является `stateless` и подвержен EBCO
 4. Класс не может быть уничтожен извне, но может быть уничтожен потомком
- Первые два варианта мы уже обсудили.
- Третий и четвёртый скорее культурно приемлимы, чем надежны.
- Кроме того ключевое слово `final` помогает **девиртуализации**

- Говорят, что виртуальные функции связываются **динамически** (так называется процесс разрешения адреса функции через `vtbl` во время выполнения).
- Обычные функции связываются **статически**.
- Даже если физически они приходят из динамических библиотек или являются позиционно независимыми и адресуются через PLT, это неважно.
- На уровне модели языка они считаются связываемыми **статически**.
- Увы, но многие другие вещи имеют статическое связывание, например аргументы по умолчанию.

- Как уже было сказано, они связываются статически, то есть **зависят только от статического типа**.

```
struct Base {  
    virtual int foo(int a = 14) { return a; }  
};  
struct Derived : public Base {  
    int foo(int a = 42) override { return a; }  
};  
Base *pb = new Derived{};  
std::cout << pb->foo() << std::endl; // на экране 14
```

Выход из положения: NVI

- Если хочется интерфейс с аргументами по умолчанию, его можно сделать не виртуальным, чтобы никто не смог их переопределить.

```
struct BaseNVI {  
    int foo(int x = 14) { return foo_impl(x); }  
private:  
    virtual int foo_impl(int a) { return a; }  
};  
struct Derived : public Base {  
    int foo_impl(int a) override { return a; }  
};
```

- Закрытая виртуальная функция открыто переопределена. Это нормально.

Два полиморфизма

- Полиморфной (по данному аргументу) называется функция, которая ведёт себя по разному в зависимости от **типа** этого аргумента.
- **Полиморфизм** бывает **статический**, когда функция управляется известными на этапе компиляции типами и **динамический**, когда тип известен только на этапе выполнения.
- Примеры:
 - Множество перегрузки можно рассматривать как одну статически полиморфную функцию (по перегруженному аргументу).
 - Шаблон функции это статически полиморфная функция (по шаблонному аргументу).
 - Виртуальная функция это динамически полиморфная функция (по первому неявному аргументу this).

- К этому времени мы знаем три модификатора доступа
- `public` – доступно всем
- `protected` – доступно только потомкам
- `private` – доступно только самому себе

Но мы также знаем, что `public` означает открытое наследование и вводит отношение is-a

```
class Derived : public Base { // Derived is a Base
```

- Можем ли мы представить себе иные отношения общее-частное?

Разновидности наследования

- При любом наследовании `private` поля недоступны классам наследникам
- Остальные поля изменяют в наследниках уровень доступа в соответствии с типом наследования
- Приватное наследование эквивалентно композиции в закрытой части
- Говорят что оно моделирует отношение `part-of`
- Неявного приведения типа при этом не происходит

	public inheritance	protected inheritance	private inheritance
public becomes:	public	protected	private
protected becomes:	protected	protected	private

Наследование по умолчанию

- Второе отличие class от struct: у class по умолчанию private, у struct public

```
struct S : public D {
```

```
public:
```

```
int n;
```

```
};
```

```
class S : private D {
```

```
private:
```

```
int n;
```

```
}
```

- Разумеется крайне хороший тон это писать явные модификаторы, если их больше одного

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 : public 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

```
class MyArray : public Array {
```

```
protected:
```

```
MyBuffer mbuf_;
```

- Чтобы здесь в порядке инициализации MyBuffer шёл раньше, чем Array, он должен быть включён в список базовых классов
- Но это означает, что нам нужно унаследоваться сразу от двух классов
- Разве это возможно?

```
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 защищённое потому что:
- мы не хотим прятать защищённую 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: сама идея сомнительна

- Множественное наследование интерфейса не вызывает вопросов
- ```
class Man: public ITwoLegs, public INoFeather {
public:
 // методы для двуногих
public:
 // методы для лишённых перьев
 // всё остальное
};
```
- Но в довольно большом количестве языков запрещено множественное наследование реализации. И сделано это неспроста

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

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

```
IOFile f11;
```

```
int x = f.a; // ошибка
```

```
int y = f.InputFile::a; // ok, но это боль
```

- Кроме того в принципе `f.InputFile::a` и `f.OutputFile::a` могут и разойтись в процессе работы
- В качестве решения хотелось бы иметь один экземпляр базового класса сколькими бы путями они ни пришёл в производный
- Такие базовые классы называются **виртуальными**

- Виртуальное наследование это поддержка в языке

```
struct File { };
```

```
struct InputFile : virtual public File { };
```

```
struct OutputFile : virtual public File { };
```

```
struct IOFile : public InputFile, public OutputFile { };
```

```
struct IOFile f{11};
```

```
int x = f.a; // ok
```

```
int y = f.InputFile::a; // ok, тоже работает
```

- Конечно тут сразу возникает масса вопросов....



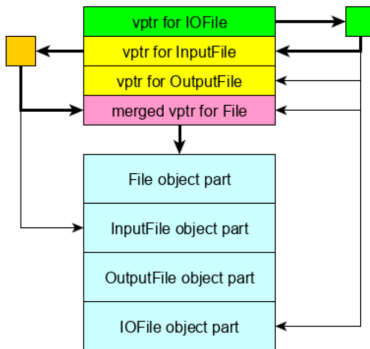
- Что если базовый класс виртуальный не по всем путям?
- Что если базовый класс виртуальный но в нижнем подобъекте всего один?
- В каком порядке и когда конструируются обычные и виртуальные подобъекты?

- Что если базовый класс виртуальный не по всем путям?  
никаких проблем, вниз попадёт один со всех виртуальных путей и по одному с каждого не виртуального
- 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

\*только хуже

# Виртуальные базовые классы

- Вызов виртуальной функции при множественном наследовании должен пройти через дополнительный уровень диспетчеризации
- А при виртуальном наследовании через **ещё один дополнительный уровень** из-за того, что таблицы для виртуальных подобъектов должны быть отдельно смержены



- Виртуальный базовый класс обязан появиться в списке инициализации самого нижнего подобъекта

```
struct InputFile : virtual public File {
 InputFile() : File(smths1) {} // этот File() не вызовется для IOFile
};
struct OutputFile : virtual public File {
 OutputFile() : File(smths2) {} // этот File() не вызовется для IOFile
};
struct IOFile : public InputFile, public OutputFile {
 IOFile() : File(smths3), InputFile(), OutputFile() {}
};
IOFile f; // вызовет File(smth3)
```

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

```
struct Base {};
```

```
struct Derived : public 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?

- Как ни странно всё 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, всё магическим образом работает и вниз (см. пример)

- Самым распространённым (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 типу

- `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` ведёт себя как `static_cast` и это наиболее безумное его использование
- `dynamic_cast` работает только для указателей и для ссылок
- Причём 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"