ЛЕКЦИЯ 17
НАСЛЕДОВАНИЕ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

ПОДДЕРЖКА ОТНОШЕНИЯ IS-A В C++

Кажется, для идеи «B является A» (также это называется отношением is-a) в языке нужна непосредственная поддержка
Это называется наследованием 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 писать отношения более явно.

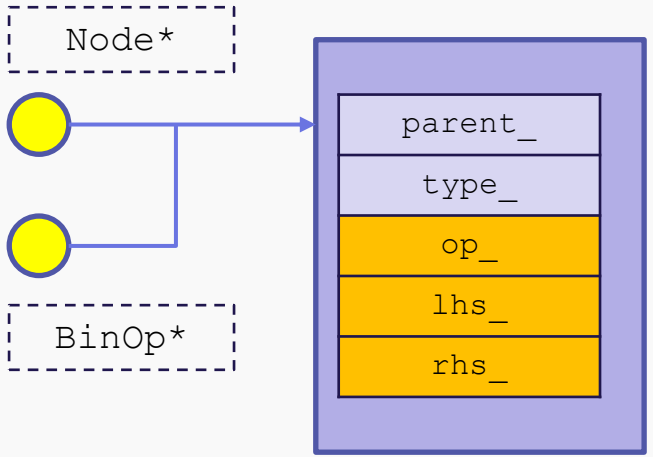
ОТКРЫТОЕ НАСЛЕДОВАНИЕ

Мы экономим сколько-то данных

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

Но главное, мы получаем отличную запись:

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



ОТКРЫТОЕ НАСЛЕДОВАНИЕ

Теперь функция-конструктор станет и впрямь конструктором

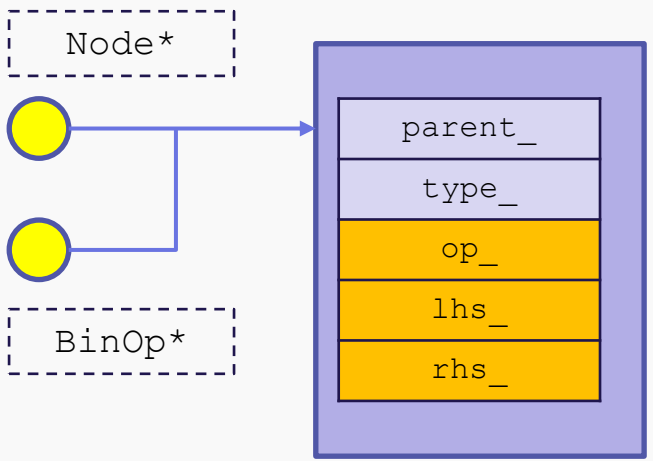
```
struct Node {
    Node* parent_;
    Node_t type_;
};
struct BinOp : public Node {
    BinOp_t op_;
    Node *lhs_ = nullptr, *rhs_ = nullptr;

    BinOp(Node* parent, BinOp_t opcode) :
      Node{parent, Node_t::BINOP}, op_(opcode) {}
};
```

ОТКРЫТОЕ НАСЛЕДОВАНИЕ

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
```



КВАДРАТ И ПРЯМОУГОЛЬНИК

У открытого наследования есть два несвязанных смысла:

- B **расширяет** A
- B **является частным случаем** A

```
struct Square {
    double x; // x * x square
    void double_square() { x *= std::sqrt(2.0);}
};
struct Rectangle: public Square {
    double y; // x * y rectangle
};
Rectangle r{2, 3}; r.double_square(); // ???
```

КВАДРАТ И ПРЯМОУГОЛЬНИК

У открытого наследования есть два несвязанных смысла:

- B **расширяет** A
- B **является частным случаем** A

```
struct Rectangle {
    double x, y; // x * y rectangle
    void double_square() { x *= 2.0;}
};
struct Square: public Rectangle {
    // кажется, теперь у нас лишнее поле
};
Square s{2}; s.double_square(); // всё еще хуже
```

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- Типы `Base` и `Derived` связаны отношением is-a (`Derived` является `Base`), если **любой истинный предикат** (интересующий нас) относительно `Base` остается истинным при подстановке `Derived`
- Именно этот принцип даёт нам возможность завести в языке неявное приведение из `Derived` в `Base`
- Для C++ этот принцип обычно выполняется с точностью до декорирования
- При правильном проектировании вы всегда можете подставить `Derived*` вместо `Base*` и `Derived&` вместо `Base&`
- Подстановка значений в C++ сопряжена с некоторыми проблемами

ПРОБЛЕМА СРЕЗКИ

```
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 == ???
```

ОБСУЖДЕНИЕ

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.
Мы теперь поговорим о **полиморфизме**

ОБЩИЙ ИНТЕРФЕЙС

- 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 наследуются

```
struct ISquare { void square(); };
struct Triangle : public ISquare; // реализуем square()
struct Polygon : public ISquare; // реализуем square()

std::vector<ISquare*>  v; // хранит и Triangle* и Polygon*
```

ОБЩИЙ ИНТЕРФЕЙС

- 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 наследуются

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

Проблемы возникают с тем, как здесь **реализовать** этот метод в `ISquare`

УКАЗАТЕЛЬ НА МЕТОД

```
class ISquare {
    sometype *sqptr_;
public:
    ISquare(sometype *sqptr) : sqptr_(sqptr) {}
    double square() const {return sqptr_->square();}
};
struct Triangle : public ISquare {
    Point x, y, z;
    Triangle() : ISquare(this) {}
    double square() const; //вычисление площади треугольника
};
```

Покритикуйте такой подход. Подумайте о том, чем может быть *sometype*

ПОДДЕЖКА В ЯЗЫКЕ: VIRTUAL

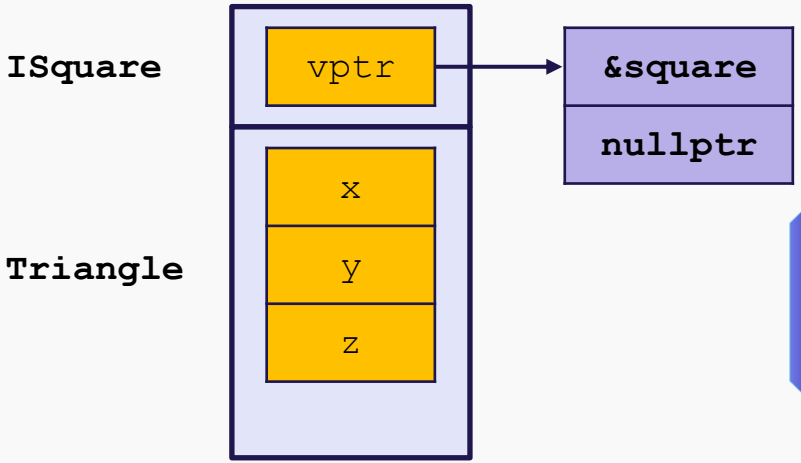
```
struct ISquare {
    virtual double square() const;
};
struct Triangle : public ISquare {
    Point x, y, z;
    double square() const; //вычисление площади треугольника
};
```

Это всё еще **очень плохой код (в семи строчках совершено три ошибки)**, мы скоро улучшим его

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 всегда там оказываются нужные указатели



ПОРЯДОК КОНСТРУИРОВАНИЯ

- При наследовании он имеет ключевое значение

```
struct Triangle : public ISquare {
    Point x, y, z;
    double square() const; //вычисление площади треугольника
    Triangle() : ISquare(), x{}, y{}, z{} {}
};
```

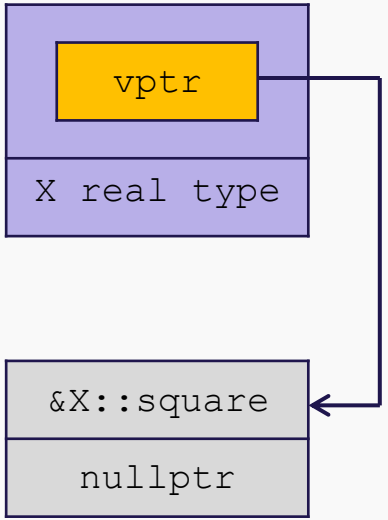
- Сначала конструируется подобъект базового класса, который невидимо конструирует себе таблицу виртуальных функций.
- Потом конструктор подобъекта производного класса невидимо заполняет её адресами своих методов.

СТАТИЧЕСКИЙ И ДИНАМИЧЕСКИЙ ТИП

- Рассмотрим функцию

```
double sum_square(const ISquare &lhs, const ISquare &rhs) {
    return lhs.square() + rhs.square();
}
Triangle t; Polygon p;
sum_square(t, p);
```

- **Статическим типом** для `lhs` и `rhs` является известный на этапе компиляции тип const `ISquare&`
- При этом в конкретном вызове у них могут быть разные динамичные типы



ПОДДЕЖКА В ЯЗЫКЕ: VIRTUAL

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

- Это всё еще **очень плохой код**, мы скоро улучшим его
- Но он иллюстрирует концепцию. Простое совпадение имени означает **переопределение (overriding)** *виртуальной* функции
- Увы, имена могут быть ещё и **перегружены (overloaded)**

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 SparceMatrix : public Matrix {
    void pow(long x); // возведение в степень разреженной матриц
                                  // крайне эффективный алгоритм
};
Matrix*m = new SparceMatrix;
m->pow(3); // увы, вызовется Matrix::pow()
```

OVERLOADING VS OVERRIDING

Переопределение функции (overriding) – это замещение в классе-наследнике виртуальной функции на функцию наследника
Перегрузка функции (overloading) – это введение того же имени с другими типами аргументов

```
struct Matrix {
    virtual void pow(int x);
};

struct SparceMatrix : public Matrix {
    void pow(int x) override; // никогда не overload
};
```

Аннотация override сообщает, что мы имели ввиду переопределение

ЯЗЫКОВАЯ ПОДДЕРЖКА: OVERRIDE

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

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

ЯЗЫКОВАЯ ПОДДЕРЖКА: PURE VIRTUAL

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

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

ВНЕЗАПНАЯ УТЕЧКА ПАМЯТИ

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

- Это всё еще **очень плохой код**, мы скоро ещё улучшим его
- Следующая проблема – удаление по указателю на базовый класс

```
ISquare *sq = new Triangle; delete sq; // memory leak
```

ОБСУЖДЕНИЕ

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;
delete sq; // OK, вызван Triangle::~Triangle()
```

ИНТЕРФЕЙСНЫЕ КЛАССЫ

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 не понятно, как нам скопировать по указателю на базовый класс

ВИРТУАЛЬНОЕ КОПИРОВАНИЕ

Обычно используется виртуальный метод `clone()`

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

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

СРЕЗКА

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

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

Поэтому никогда не передавайте объекты производных классов по значению

Используйте указатель или ссылку

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

НЕОБХОДИМОСТЬ: VIRTUAL DTOR

```
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 проблемы

КАК ТЕПЕРЬ ЖИТЬ?

Допустим, мы написали некий класс Bar
Писать ли у него виртуальный деструктор?

КАК ТЕПЕРЬ ЖИТЬ?

Допустим, мы написали некий класс Bar
Писать ли у него виртуальный деструктор?
Если мы хоти от него наследовать, то да, хотим.
Если мы не хотим наследовать и не хотим оверхеда на vtable, то можно объявить его `final`

```
struct Foo final {
    // something
}
```

Теперь наследование будет ошибкой компиляции

ПИШЕМ ПРАВИЛЬНО: ЧЕТЫТРЕ СПОСОБА

Класс в C++ написан правильно, если и только если выполнено любое из условий:

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 обсудили
Давайте поговорим о третьем и четвертом

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 они нужны для так называемых **миксинов**

EBCO И UNIQUE_PTR

Мы говорили, что `unique_ptr` выглядит как-то так:

```
template <typename T, typename Deleter = default_delete<T>>
class unique_ptr {
    T *ptr_; Deleter del_;
public:
    unique_ptr(T *ptr = nullptr, Deleter del = Deleter()) :
      ptr_(ptr), del_(del) {}
    ~unique_ptr() { del_(ptr_); }
    // и так далее
};
```

Но можем ли мы сэкономить, если `Deleter` – это stateless class?

EBCO И UNIQUE_PTR

Мы говорили, что unique_ptr выглядит как-то так:

```
template <typename T, typename Deleter = default_delete<T>>
class unique_ptr : public Deleter {
    T *ptr_; Deleter del_;
public:
    unique_ptr(T *ptr = nullptr, Deleter del = Deleter()) :
      Deleter(del), ptr_(ptr), del_(del) {}
    ~unique_ptr() { Deleter::operator()(ptr_); }
    // и так далее
};
```

Увы, это невозможно, если делетер – функция

Оставим за кулисами как же unique_ptr отличает класс от функции

ОБСУЖДЕНИЕ

Разумеется, при использовании таких миксинов никто не будет стирать класс по указателю на его делетер

```
struct CDeleterTy {
    void operator()(int* p) {delete[] p;}
};
CDeleterTy *pDel =
  new std::unique_ptr<int, CDeleterTy> {new int[SZ]()};
delete pDel; // к счастью, это CE
```

Писать виртуальный деструктор в миксин не хочется. Потому что он резко станет stateful.

ЯЗЫКОВАЯ ПОДДЕРЖКА: PROTECTED

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 все по-прежнему

ПИШЕМ ПРАВИЛЬНО: ДВА СПОСОБА

Класс в C++ написан правильно, если и только если выполнено любое из условий:

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 обсудили

Третий и четвертый скорее культурно приемлемы, чем надежны

Кроме того, ключевое слово `final` помогает **девиртуализации**

ОБСУЖДЕНИЕ

Как вы вообще считаете: как виртуальные функции влияют на производительность? А на стабильность?

ОБСУЖДЕНИЕ

Как вы вообще считаете: как виртуальные функции влияют на производительность? А на стабильность?

Сугубо мрачно. Виртуальная функция вызывается, как минимум по указателю (в случае множественного наследования всё ещё хуже)

Мало того, этот указатель должен быть правильно заполнен в конструкторе

На практике это значит целый новый класс ошибок

ОБСУЖДЕНИЕ: PVC

Распространенной ошибкой является вызов чисто виртуального метода

```
struct Base {
    Base() { doIt(); } // PVC invocation
    virtual void doIt() = 0;
};
struct Derived : public Base { void doIt() override; };

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

Заметьте, вызов чисто виртуальной функции это ошибка не только в ctor/dtor, но и в любой функции, которая из них вызывается

ВИРТУАЛЬНЫЕ ФУНКЦИИ В КОНСТРУКТОРАХ

Даже если они не приводят к PVC, они работают как невиртуальные

```
struct Base {
    Base() { doIt(); }
    virtual void doIt();
};
struct Derived : public Base { void doIt() override; };

int main() {
    Derived d; // Base::doIt()
}
```

Поэтому многие вообще скептически относятся к вызовам функций в ctor/dtor

СТАТИЧЕСКОЕ И ДИНАМИЧЕСКОЕ СВЯЗЫВАНИЕ

- 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 связывающимися статически
- Увы, но многие другие вещи имеют статическое связывание, например, аргументы по умолчанию

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 невиртуальным, чтобы никто не смог их переопределить

```
struct BaseNVI {
    int foo(int a = 14) { return foo_impl(a); }
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 ведет 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 статическую полиморфную функцию (по любому аргументу)
- Шаблон функции – это статически полиморфная функция (по любому аргументу)
- Виртуальная функция – это динамически полиморфная функция (по первому неявному аргументу `this`)

ОГРАНИЧЕНИЯ

Давайте посмотрим насколько можно смешивать динамический и статический полиморфизм

Два вопроса:

1. Как вы думаете, может ли существовать шаблон виртуального метода?

2. Как вы думаете, можно ли перегружать виртуальные функции?

ОГРАНИЧЕНИЯ

Давайте посмотрим насколько можно смешивать динамический и статический полиморфизм

Два вопроса:

1. Как вы думаете, может ли существовать шаблон виртуального метода?

К счастью не может (какие последствия это вызвало бы для таблиц виртуальных функций?)

2. Как вы думаете, можно ли перегружать виртуальные функции?

К сожалению можно и это вызывает крайне мрачные последствия из-за скрытия имён

ПЕРЕГРУЗКА ВИРТУАЛЬНЫХ ФУНКЦИЙ

Предположим, что мы умеем эффективно возводить разреженные матрицы в целые степени и хотим просто переиспользовать возведение в дробные

```
struct Matrix {
    virtual void pow(double x); // обычный алгоритм
    virtual void pow(int x); // эффективный алгоритм
}
struct SparseMatrix : public Matrix {
    void pow(int x) override; // крайне эффективный алгоритм
};
SparseMatrix d;
d.pow(1.5); // Какой метод будет вызван?
```

СОКРЫТИЕ ИМЁН

Увы, в коде ниже будет вызван метод `SparseMatrix::pow`

```
struct Matrix {
    virtual void pow(double x); // обычный алгоритм
    virtual void pow(int x); // эффективный алгоритм
}
struct SparseMatrix : public Matrix {
    void pow(int x) override; // имя pow скрывает
Matrix::pow
};
SparseMatrix d;
d.pow(1.5); // SparseMatrix::pow(1)
```

ВВЕДЕНИЕ ИМЁН В ОБЛАСТЬ ВИДИМОСТИ

Для введения имён в область видимости, используем `using`

```
struct Matrix {
    virtual void pow(double x); // обычный алгоритм
    virtual void pow(int x); // эффективный алгоритм
}
struct SparseMatrix : public Matrix {
    using Matrix::pow;
    void pow(int x) override; // крайне эффективный алгоритм
};
SparseMatrix d;
d.pow(1.5); // Matrix::pow(1.5)
```

КОНТРОЛЬ ДОСТУПА

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

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

- Приватное наследование эквивалентно композиции закрытой части
- Говорят, что оно моделирует отношение part-of
- Неявного приведения типа при этом не происходит

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**

ОТНОШЕНИЕ PART-OF

Закрытое наследование

```
class Whole : private Part
{
    // everything else
};
```

Композиция

```
class Whole {
    // everything else
private: Part p_;
};
```

Ключевое отличие наследование это:

- возможность переопределять виртуальные функции из базового класса
- доступ к защищенным (`protected`) полям базового класса
- возможность использовать `using` и вводить имена из базового класса в свой scope

Композиция должна быть выбором по умолчанию

EBCO И UNIQUE_PTR: PRIVATE INH

Логично, что мы хотим private, на него EBCO также работает:

```
template <typename T, typename Deleter = default_delete<T>>
class unique_ptr : private Deleter {
    T *ptr_; Deleter del_;
public:
    unique_ptr(T *ptr = nullptr, Deleter del = Deleter()) :
      Deleter(del), ptr_(ptr), del_(del) {}
    ~unique_ptr() { Deleter::operator()(ptr_); }
    // и так далее
};
```

Теперь нет опасности приведения к базовому классу

```
DeleterTy *pd = new unique_ptr<int, DeleterTy>{} // FAIL
```

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

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