Наследование

Полиморфизм подтипов. Динамическое связывание.

Постановка задачи

Даны типы геометрических фигур (Circle, Triangle, Rectangle).

Задача:

- Реализовать унифицированную обработку их экземпляров:
  - Хранение
  - Общие операции

Фигуры

```
struct Triangle { Point a_, b_, c_; };

struct Circle {
  Point center_;
  double radius_;
};

struct Rectangle {
  Point top_left_;
  Point bottom_right_;
};
```

Хотим создать общий массив фигур

Попытка 1: Union

```
union Shape {
  Circle circle_;
  Triangle triangle_;
  Rectangle rectangle_;
};
```

Создание объектов

```
Geometry create_circle(Point center, double radius) {
  Geometry g = {.circle_ = {center, radius}};
  return g;
}

Geometry geometries[]
  = {{.circle_ = {{0, 0}, 1}},
     {.triangle_ = {{0, 0}, {0, 1}, {1, 0}}},
     {.rectangle_ = {{0, 1}, {1, 0}}}};
```

Попытка 1: Union

```
union Shape {
  Circle circle_;
  Triangle triangle_;
  Rectangle rectangle_;
};
```

Как узнать, какой элемент активен?

Попытка 1.2: Tagged Union

```
struct Geometry {
  union Data {
    Circle circle_;
    Triangle triangle_;
    Rectangle rectangle_;
  } data_;
  ShapeType shape_type_;
};
```

Попытка 1.3: Tagged Union (union-like class)

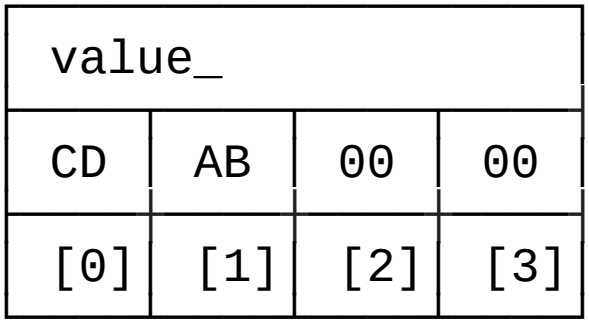
```
struct Geometry {
  union {
    Circle circle_;
    Triangle triangle_;
    Rectangle rectangle_;
  };
  ShapeType shape_type_;
};
```

Попытка 1.3: реализация полиморфизма

```
void print_geometry(Geometry g) {
  switch (g.shape_type_) {
  case STCircle:
    return print_circle(&g.circle_);
    ...
  }
}
```

Union в языке C

```
union Number {
  int value_;
  uint8_t bytes_[sizeof(int)];
};
```



https://godbolt.org/z/Kxo6aTd4e

Union в языке C++, n кругов ада

```
union U {
  std::vector<int> v_;
  std::string s_;
};
```

Какой конструктор должен быть вызван?

Union в языке C++, n кругов ада

```
union U {
  std::vector<int> v_;
  std::string s_;
};
```

Хорошая новость: это ошибка компиляции

Плохая новость: текст ошибки

https://godbolt.org/z/WEEY699YE

Union в языке C++, n кругов ада

```
union U {
  U(std::vector<int> v) {
    new (&v_) std::vector<int>(std::move(v));
  }

  U(std::string s) {
    new (&s_) std::string(std::move(s));
  }
  ...
};
```

Union в языке C++, n кругов ада

```
union U {
  U(std::vector<int> v) {
    new (&v_) std::vector<int>(std::move(v));
  }

  U(std::string s) {
    new (&s_) std::string(std::move(s));
  }
  ...
};
```

Какой деструктор будет вызван?

Union в языке C++, n кругов ада

```
union U {
  ~U() {
    // TODO: check active element
    s_.~basic_string();
  }
};
```

Без user-provided деструктора — ошибка компиляции

https://godbolt.org/z/G1fTqoTTK

Резюме

1. Union прост в C, но очень сложен в C++
2. Объединение типов в enum ограничивает создание пользовательских подтипов

Union окажется полезен нам гораздо позже

Попытка 2: композиция

```
struct Geometry {
  ShapeType shape_type_;
};

struct Circle {
  Geometry base_;
  Point center_;
  double radius_;
};
```

Создание окружности

```
Geometry *create_circle(Point center, double radius) {
  auto *c = new Circle{{STCircle}, center, radius};
  return &c->base_;
}
```

Диспетчеризация

```
void print_geometry(Geometry *g) {
  switch (g->shape_type_) {
  case STCircle:
    return print_circle(
      reinterpret_cast<Circle *>(g));

  ...
```

Наследование в C++

```
struct Geometry {};

struct Circle : public Geometry {
  Circle(Point center, double radius)
    : center_(center), radius_(radius) {}

  Point center_;
  double radius_;
};
```

Попытка диспетчеризации функции

```
struct Geometry {
  fun_ptr print_ptr_;
  void print() { print_ptr_(); }
};

struct Circle : public Geometry {
  Circle(Point center, double radius)
    : Geometry{&Circle::print},
      center_(center), radius_(radius) {}

  void print();
};
```

Виртуальный метод

```
struct Geometry {
  virtual void print() const;
};

struct Circle : public Geometry {
  void print() const;
  ...
};
```


https://godbolt.org/z/Yh1Ebnn7K

Виртуальный метод

```
struct Geometry {
  virtual void print() const;
};

struct Circle : public Geometry {
  void print() const;
  ...
};
```

Код делает вид, что работает.

Найдите 3 ошибки.

https://godbolt.org/z/Yh1Ebnn7K

overloading vs overriding

```
struct A {
  virtual void f(int) {
    std::cout << "A::f\n";
  }
};

struct B : public A {
  virtual void f(long) {
    std::cout << "B::f\n";
  }
};
```

```
void f(A& obj) {
  long x = 42;
  obj.f(x);
}

B obj;
f(obj);
```


https://godbolt.org/z/djxdzc3c7

overloading vs overriding

```
struct A {
  virtual void f(int) {
    std::cout << "A::f\n";
  }
};

struct B : public A {
  void f(long) override {
    std::cout << "B::f\n";
  }
};
```

```
error:
'void B::f(long int)'
marked 'override',
but does not override
```


https://godbolt.org/z/ordcPfsGj

Виртуальный метод

```
struct Geometry {
  virtual void print() const;
};

struct Circle : public Geometry {
  void print() const override;
  ...
};
```

Осталось еще 2 ошибки

Что должен делать Geometry::print?

Виртуальный метод

```
struct Geometry {
  virtual void print() const = 0;
};

struct Circle : public Geometry {
  void print() const override;
  ...
};
```

Geometry::print – чистый виртуальный метод

Абстрактные классы и интерфейсы

- Класс с хотя бы одним чистым виртуальным методом называется **абстрактным**.
- Экземпляр абстрактного класса нельзя создать.
- Классы, состоящие только из публичных чистых виртуальных методов часто называют **интерфейсами**.

Статический и динамический тип

```
void f(const Geometry& g) {
  g.print();
}

int main() {
  Circle c({0, 0}, 1);
  f(c);
}
```

Статический и динамический тип

```
Geometry* g = new Circle({0, 0}, 1);

delete g;  // Что будет вызвано здесь?
```


https://godbolt.org/z/nEsPqoKxf

Виртуальный деструктор

```
struct Geometry {
  virtual ~Geometry() = default;
  virtual void print() const;
};

struct Circle : public Geometry {
  void print() const override;
  ...
};
```

Виртуальный деструктор

Нужен для корректного уничтожения экземпляра дочернего класса через указатель на базовый класс

Что вы думаете о правиле 5 в этом случае?

Когда создавать виртуальный деструктор?

- Пусть мы пишем класс Widget.
- Нужен ли ему виртуальный деструктор?

Когда создавать виртуальный деструктор?

- Пусть мы пишем класс Widget.
- Нужен ли ему виртуальный деструктор?

Конфликт:

- Создание вирт деструктора в абсолютно каждом классе расточительно.
- Отсутствие деструктора позволяет некорректное полиморфное использование.

Ограничение наследования

```
class Widget final {};
```

Упрощенное правило. Класс должен:

- Либо содержать виртуальный деструктор
- Либо быть final

Есть еще пара правил, но о них в другой раз.

Перебегаем на красный свет

```
template <typename T>
struct ExtendedVector : std::vector<T> {
  using std::vector<T>::vector;

  T& operator[](size_t i) {
    return i < this->size() ? this->at(i) : this->back();
  }

  T const& operator[](size_t i) const {
    return i < this->size() ? this->at(i) : this->back();
  }
};
```

Корректность наследование

Наследование A <-- B несет одновременно два смысла:

- B является частным случаем A (**is a**)
- B расширяет A (**extends**)

Circle-ellipse problem

Пусть в нашей иерархии есть классы:

- Circle
- Ellipse

Какой должен быть базовым?

Circle <-- Ellipse

```
struct Circle {
  virtual double area();
  Point center_;
  double radius_;
};

struct Ellipse
   : public Circle {
  double area() override;
  double ry_;
};
```

```
void f(const Circle& c) {
  c.area(); // Oops…
}

f(Ellipse{{0, 0}, 1, 2});
```

Ellipse <-- Circle

```
struct Ellipse {
  void stretch_x(double k) { rx_ *= k; }
  Point center_;
  double rx_, ry_;
};

struct Circle : public Ellipse {
  Circle(Point center, double radius)
    : Ellipse{center, radius, radius} {}
};

Circle c({0, 0}, 1);
c.stretch_x(2); // Oops...
```

Circle-ellipse problem

- Circle является частным случаем Ellipse
- Ellipse расширяет Circle

Вывод: они не должны быть объединены наследованием.

Калькулятор

Предложите иерархию классов для представления арифметических выражений.

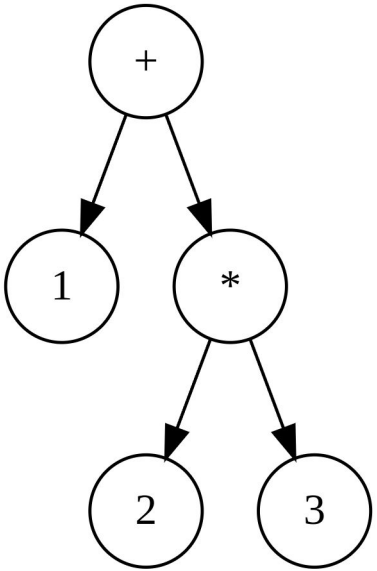
Выражения состоят из:

- Чисел
- Бинарных операторов +, −, *, /

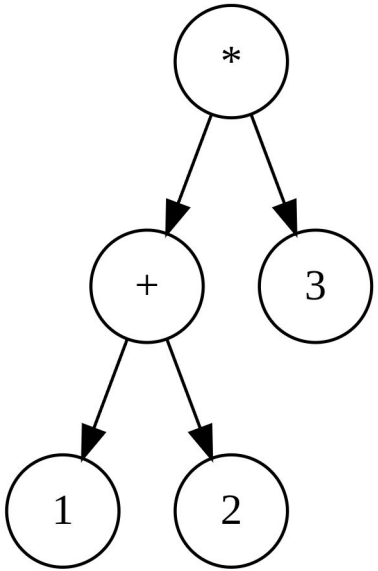
Метод **eval** вычисляет значение выражения.

Выражение — это дерево

1 + 2 * 3



(1 + 2) * 3



Калькулятор

```
class Expr {
 public:
  virtual ~Expr() = default;
  virtual double eval() const = 0;
};
```

Сложение

```
class Add : public Expr {
 public:
  Add(const Expr* left, const Expr* right);

  double eval() const override {
    return left_->eval() + right_->eval();
  }

 private:
  const Expr* left_;
  const Expr* right_;
};
```

Покритикуйте это решение

Сложение

```
class Add : public Expr {
 public:
  Add(const Expr* left, const Expr* right);

  double eval() const override {
    return left_->eval() + right_->eval();
  }

 private:
  const Expr* left_;
  const Expr* right_;
};
```

Подвыражения будут дублироваться в Sub, Mul и Div

Общий класс для бинарных операций

```
class BinExpr : public Expr {
 public:
  BinExpr(const Expr* left, const Expr* right);

  const Expr* left() const { return left_; }
  const Expr* right() const { return right_; }

  …
};
```

Класс все еще абстрактный

Сложение

```
class Add : public BinExpr {
 public:
  using BinExpr::BinExpr;

  double eval() const override {
    return left()->eval() + right()->eval();
  }
};
```

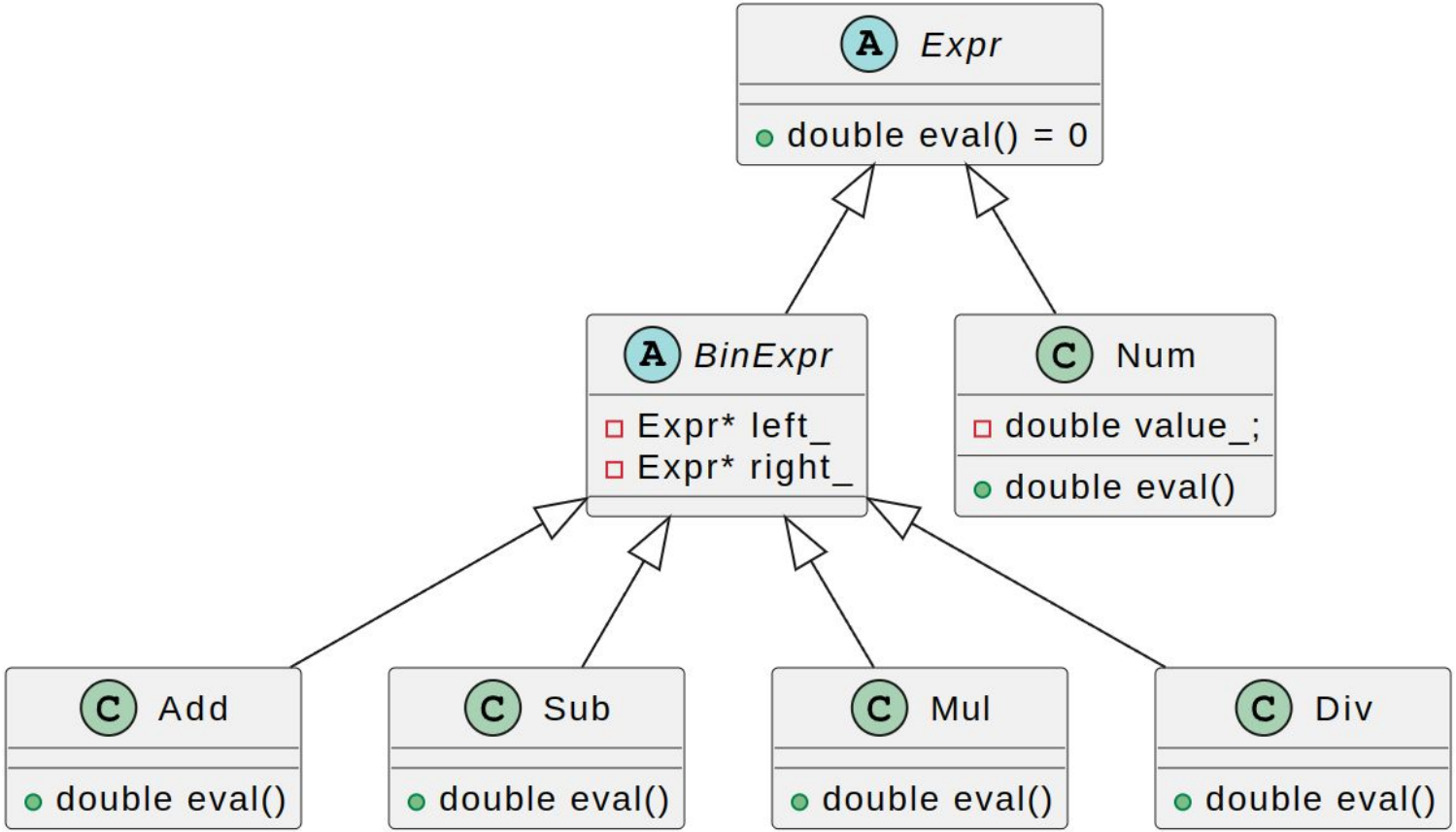
Остальные операции — по аналогии

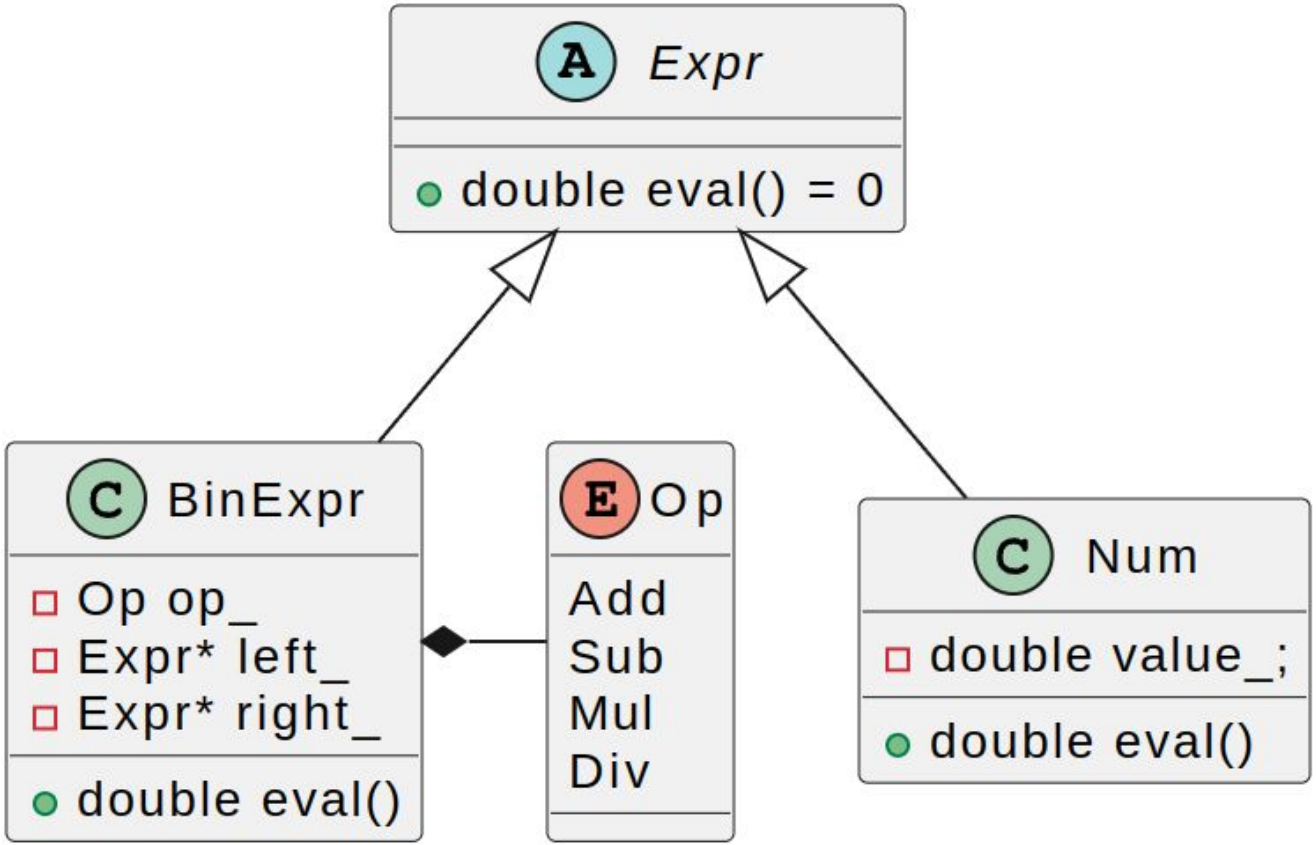
Число

```
class Num : public Expr {
 public:
  explicit Num(double val) : val_(val) {}

  double eval() const override { return val_; }

 private:
  double val_;
};
```

Промежуточный итог

- Код: https://godbolt.org/z/hP1PhnjzE
- Можно уменьшить количество классов
- Решение страдает от утечек памяти


A Expr
double eval() = 0
A BinExpr
Expr* left_
Expr* right_
C Num
double value_;
double eval()
C Add
double eval()
C Sub
double eval()
C Mul
double eval()
C Div
double eval()


A Expr
double eval() = 0
C BinExpr
Op op_
Expr* left_
Expr* right_
double eval()
E Op
Add
Sub
Mul
Div
C Num
double value_;
double eval()

BinExpr::Eval

```
double eval() const {
  switch (op_) {
    case Op::Add:
      return left_->eval() + right_->eval();
    …
  }
  throw std::runtime_error("Unreachable");
}
```

Конструирование дерева

```
Expr* expr
  = new Add(new Num(1),
            new Mul(new Num(2), new Num(3)));
```

Конструирование дерева

Оффтоп

```
Expr* expr
  = new Add(new Num(1),
            new Mul(new Num(2), new Num(3)));
```

Вам это что-нибудь напоминает?

Конструирование дерева

Оффтоп

```
Expr* expr
  = new Add(new Num(1),
            new Mul(new Num(2), new Num(3)));

Expr* expr
  = Add(Num(1),
        Mul(Num(2), Num(3)));
```

Убирает new...

Конструирование дерева

Оффтоп

```
Expr* expr
  = new Add(new Num(1),
            new Mul(new Num(2), new Num(3)));

Expr* expr
  = +(1,
        *(2, 3));
```

Заменяем классы на операторы

Конструирование дерева

Оффтоп

```
Expr* expr
  = new Add(new Num(1),
            new Mul(new Num(2), new Num(3)));

Expr* expr
  = (+ 1
       (* 2 3));
```

Двигаем скобки,
убираем запятые

Конструирование дерева

```
Expr* expr
  = new Add(new Num(1),
            new Mul(new Num(2), new Num(3)));
```

Вернемся к проблеме утечки памяти

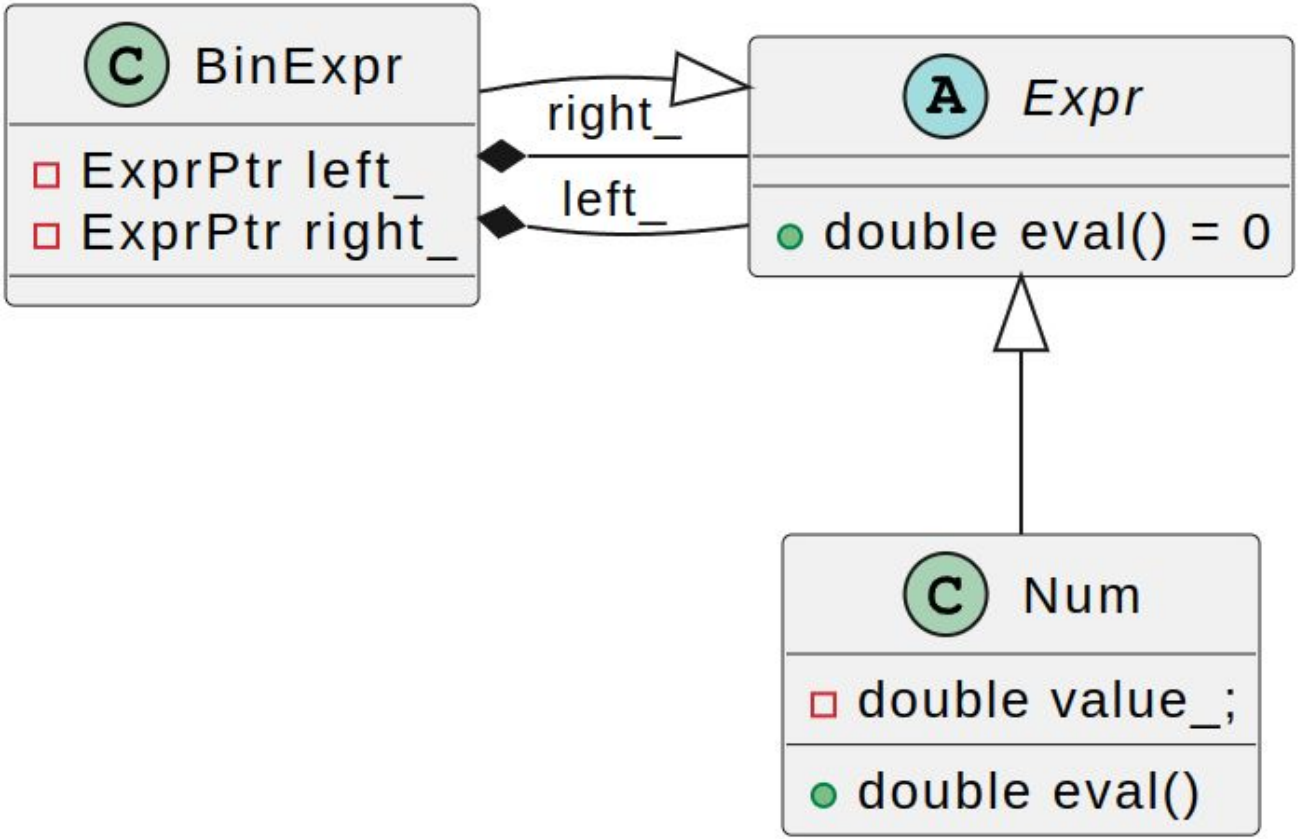
Использование умных указателей

```
using ExprPtr = std::unique_ptr<const Expr>;

class BinExpr : public Expr {
 public:
  BinExpr(ExprPtr left, ExprPtr right)
      : left_(std::move(left)),
        right_(std::move(right)) {}

  const Expr* left() const { return left_.get(); }
  const Expr* right() const { return right_.get(); }
};
```

Создание дерева

```
auto expr =
    std::make_unique<Add>(
        std::make_unique<Num>(1),
        std::make_unique<Mul>(
            std::make_unique<Num>(2),
            std::make_unique<Num>(3)));
```


https://godbolt.org/z/qveW4v4eM


C BinExpr
ExprPtr left_
ExprPtr right_
right_
left_
A Expr
double eval() = 0
C Num
double value_;
double eval()

Наследование в поиске

- SourceDocProvider
  - CsvProvider, SQLiteProvider, …
- IndexWriter
  - TextIndexWriter, BinaryIndexWriter, …
- IndexAccessor
  - TextIndexAccessor, BinaryIndexAccessor, …
- Ranker
  - TfIdfRaner, BM25Ranker, …
- ResultsPrinter
  - PlainTextPrinter, JsonPrinter, …