Пространства имен. Перегрузки.

Введение объектно-ориентированное программирование

27.02.2025

Область видимости

• У любого имени есть область видимости (scope): совокупность всех мест в программе, откуда к нему можно обратиться.

```
int a = 2;
void foo() {
  int b = a + 3; // ok, we are in scope of a
  if (b > 5) {
    int c = (a + b) / 2; // ok we are in scope of a and b
  }
  b += c; // compilation fail
}
```

Время жизни

• У любой переменной есть время жизни (lifetime): совокупность всех **моментов времени** в программе, когда её состояние **валидно**.
• Первый такой момент случается после окончания инициализации.

```
int main() {
int a = a; // a declared, but lifetime of "a"not started
```

• Это довольно редкий пример, когда мы пытаемся использовать нечто до его рождения.
• Куда более часто мы будем пытаться использовать нечто после его смерти.

Провисшие указатели

• Указатель, ссылающийся на переменную с истекшим временем жизни называется провисшим (dangling)

```
int a = 2;
void foo() {
  int b = a + 3; int *pc;
  if (b > 5) {
    int c = (a + b) / 2; pc = &c;
  } // c scope end; c lifetime end; pc dangles
  b += *pc; // this is parrot no more
} // b scope end; b lifetime end;
```

Провисшие ссылки

- Сделать висячую ссылку чуть сложнее, чем указатель, но можно
- Классика: ссылка внутрь удалённой памяти

```
int *p = new int[5];
int &x = p[3];
delete [] p; // x dangles
```

- Сама по себе провисшая ссылка ничего не значит. Проблемы будут только если по ней куда-то обратятся

```
x += 1; // it ceased to be
```

Провисшие ссылки

- Сделать висячую ссылку чуть сложнее, чем указатель, но можно
- Ещё классика: вернуть ссылку на временное значение

```
int& foo() {
  int x = 42;
  return x;
}
int x = foo(); // it expired and gone
```

- Компиляторы довольно плохи в диагностике провисших ссылок и указателей

Продление жизни

• Константные (и только они) lvalue ссылки продлевают жизнь временных объектов

```
const int &lx = 0;
int x = lx; // ok
int foo();
const int &ly = 42 + foo();
int y = ly; // ok
```

• Но не стоит соблазняться. Ссылка связывается со значением, а не со ссылкой, так что константная ссылка тоже может провиснуть при возврате из функции

Жизнь временных объектов

• Временный объект живёт до конца полного выражения

```
struct S {
  int x;
  const int &y;
};
S x{1, 2}; // ok, lifetime extended
S *p = new S{1, 2}; // this is a late parrot
```

• На первой строчке у нас не временный, а постоянный объект

• На второй будет висячая ссылка потому что временный объект продлявший жизнь константе закончился в конце выражения

Иногда временный объект не создаётся

- Неконстантные левые ссылки не создают временных объектов и просто отказываются связываться с литералами

```
int foo(int &x);
foo(1); // ошибка компиляции
```

- И даже проще

```
int &x = 1; // ошибка компиляции
```

- И это одна из лучших новостей в этой части лекции
- Попробуйте догадаться отчего так сделано

Decaying

int foo(const int& t) { return t; }

- Ссылка на объект в выражениях ведёт себя как сам объект
- Мы это где-то встречали

Decaying

- Массив деградирует (decays) к указателю на свой первый элемент, когда он использован как rvalue

```
void foo(int *);
int arr[5];
int *t = arr + 3; // ok
foo(arr); // ok
arr = t; // fail
```

Lvalue & rvalue

- В языке C концепция lvalue означала "left-hand-side value"

y = x;

- Здесь y это lvalue, x это rvalue
- В языке C можно отделить синтаксически: вызов функции, имя массива, выражение сложения – всё это никогда не lvalue и технически не может встретиться в присваивании слева
- Так ли это в C++?

Lvalue & rvalue

- В языке C концепция lvalue означала “left-hand-side value“

```
y = x;
```

- Здесь y это lvalue, x это rvalue
- В языке C можно отделить синтаксически: вызов функции, имя массива, выражение сложения – всё это никогда не lvalue и технически не может встретиться в присваивании слева
- Увы, C++ усложняет вещи

```
int& foo();
foo() = x; // ok
```

Lvalue & rvalue

• В языке C++ lvalue это скорее "location value"– в смысле что-то у чего есть положение (location) в памяти
• В языке C++11 также есть более точный термин glvalue объединяющий положения с временными положениями, мы поговорим о нём на лекции по rvalue ссылкам
• Ссылки рассматриваемые здесь это lvalue ссылки
• Технически может существовать lvalue ссылка на массив. Это происходит именно потому, что, хотя массив и не может быть слева в присваивании, но он всегда lvalue в C++ потому что у него всегда есть локация (сам массив это локация по определению

Разрешение перегрузки

• Наличие перегрузки вносит некоторые сложности

```
float sqrtf(float x); // 1
double sqrt(double x); // 2
sqrtf(42); // вызовет 1, неявно преобразует int → float
```

• В языке C нет перегрузки и нет проблем, программист всегда явно указывает какую функцию нужно вызвать

Разрешение перегрузки

• Наличие перегрузки вносит некоторые сложности

```
float sqrt(float x); // 1
double sqrt(double x); // 2
sqrt(42); // неясно что вызвать, оба варианта подходят
```

• В языке C++ есть перегрузка и компилятор должен разрешить имя, то есть связать упомянутое в коде имя с обозначаемой им сущностью

• В коде выше как по вашему будет сделан вызов и почему?

Разрешение перегрузки

- Наличие перегрузки вносит некоторые сложности

```
float sqrt(float x); // 1
double sqrt(double x); // 2
sqrt(42); // неясно что вызвать, оба варианта подходят
```

- В языке C++ есть перегрузка и компилятор должен разрешить имя, то есть связать упомянутое в коде имя с обозначаемой им сущностью
- В коде выше как по вашему будет сделан вызов и почему?
- Разумеется будет ошибка компиляции. Оба варианта одинаково хороши

Правила разрешения перегрузки

• Первое приближение (здесь много чего не хватает)
1. Точное совпадение (int $\rightarrow$ int, int $\rightarrow$ const int&, etc) Обратите внимание: nullptr точно совпадает с любым указателем.
2. Точное совпадение с шаблоном (int $\rightarrow$ T)
3. Стандартные преобразования (int $\rightarrow$ char, float $\rightarrow$ unsigned short, etc)
4. Переменное число аргументов
5. Неправильно связанные ссылки (literal $\rightarrow$ int&, etc)
• Мы вернёмся к перегрузке когда подробнее поговорим о шаблонах функций

Обсуждение: nullptr

- Теперь мы можем дополнительно аргументировать почему наш выбор nullptr.

```
int foo(int *);
// call for null pointer foo(0);
```

- В чём хрупкость этой конструкции?
- Как nullptr это исправляет?

Коротко о пространствах имён

• Любое имя принадлежит к какому-то пространству имён

```
// no namespace here
int x;
int foo() {
return ::x;
}
```

• Здесь кажется, что x не принадлежит ни к какому пространству имён

• Но на самом деле x принадлежит к глобальному пространству имён

Пространство имён std

- Вся стандартная библиотека принадлежит к пространству имён std
std::vector, std::string, std::sort, ...
- Исключение это старые хедера наследованные от C, такие, как
<stdlib.h>
- Чтобы завернуть atoi в std, сделаны новые хедера вида <cstdlib>
- Вы не имеете права добавлять в стандартное пространство имён
свои имена
- Точно по той же причине по какой вы не можете начинать свои
имена с подчёркивания и большой буквы

Ваши пространства имён

- Вы можете вводить свои пространства имён и неограниченно вкладывать их друг в друга
- При том структуры тоже вводят пространства имён

```
namespace Containers {
  struct List {
    struct Node {
      // .... whatever ....
    };
  };
}
Containers::List::Node n;
```

Переоткрытие пространств имён

• В отличии от структур, пространства имён могут быть переоткрыты

```
namespace X {
int foo();
}
// теперь переоткроем и добавим туда bar
namespace X {
int bar();
}
```

• Структура вводит тип данных. Тип не должен существовать если в программе не будет его объектов

• Для пространств имён куда удобнее (сюрприз) пространства имён

Директива using

• Мы можем вводить отдельные имена и даже целые пространства имён

```
namespace X {
int foo();
}
using std::vector;
using namespace X;
vector<int> v; v.push_back(foo());
```

• Использовать эти механизмы следует осторожно так как пространства имён придуманы не просто так

Анонимные пространства имён

• Это распространённый механизм для замены статических функций

```
namespace {
int foo() {
return 42;
}
}
int bar() { return foo(); } // ok!
```

• Означает сделать пространство имён со сложным уникальным именем и тут же сделать его using namespace

Анонимные пространства имён

- Это распространённый механизм для замены статических функций

```
namespace IdFgghbjhbklbkuU6 {
int foo() {
return 42;
}
}
using namespace IdFgghbjhbklbkuU6;
int bar() { return foo(); } // ok!
```

- Поскольку имена из него не видны снаружи они как бы статические

Правила хорошего тона

- Не засорять глобальное пространство имён
- Никогда не писать using namespace в заголовочных файлах
- Использовать анонимные пространства имён вместо статических функций
- Не использовать анонимные пространства имён в заголовочных файлах

Ваши типы как встроенные

• Собственный класс кватернионов.

```
struct Quat {
int x, y, z, w;
};
```

• У нас уже есть бесплатное копирование и присваивание. Хотелось бы чтобы работало всё остальное: сложение, умножение на число и так далее.

• Начнём с чего-нибудь простого.

```
Quat q {1, 2, 3, 4};
Quat p = -q; // унарный минус: {-1, -2, -3, -4}
```

Общий синтаксис операторов

- Обычно используется запись operator и далее какой это оператор.

```
struct Quat {
int x, y, z, w;
};
Quat operator-(Quat arg) {
return Quat{-arg.x, -arg.y, -arg.z, -arg.w};
}
```

- Теперь всё как надо.

```
Quat q {1, 2, 3, 4};
Quat p = -q; // унарный минус: {-1, -2, -3, -4}
```

Общий синтаксис операторов

- Альтернатива: метод в классе.

```
struct Quat {
int x, y, z, w; Quat operator-() {
return Quat{-x, -y, -z, -w};
}
};
```

- Теперь всё как надо.

```
Quat q {1, 2, 3, 4};
Quat p = -q; // унарный минус: {-1, -2, -3, -4}
```

Обсуждение

- Обычно есть два варианта (исключение: присваивание и пара-тройка других).
- -a означает a.operator-()
- -a означает operator-(a)
- Как вы думаете, что будет если определить оба?

Обсуждение

- Как вы думаете чем закончится попытка:
- перегрузить operator- для int.

```
int operator-(int x) {
std::cout « "MINUS!«< std::endl;
return x;
}
```

Обсуждение

- Унарный минус всё-таки немного сомнительный оператор для перегрузки.
- Давайте, прежде чем двигаться дальше, мотивируем перегрузку операторов, то есть покажем, как она даёт нам производительность и возможности.

Функторы: постановка проблемы

• Эффективность std::sort резко проседает если для его объектов нет operator< и нужен кастомный предикат.

```
bool gtf(int x, int y) { return x > y; }
// неэффективно: вызовы по указателю
std::sort(myarr.begin(), myarr.end(), &gtf);
```

• Можно ли с этим что-то сделать?

Функторы: перегрузка ()

• Более правильный способ сделать функтор это перегрузка вызова.

```
struct gt { bool operator() (int x, int y) { return x > y; } };
// всё так же хорошо
std::sort(myarr.begin(), myarr.end(), gt{});
```

• Почти всегда это лучше, чем указатель на функцию.

• Кроме того в классе можно хранить состояние.

• Функторы с состоянием получат второе дыхание когда мы дойдём до так называемых лямбда-функций.

Источник названия языка

- Язык C++ получил название от операции ++ (постинкремента).
- Бывает также преинкремент.

```
int x = 42, y, z;
y = ++x; // y = 43, x = 43
z = y++; // z = 43, y = 44
```

- Для их переопределения используется один и тот же operator++.

```
Quat& Quat::operator++(); // это пре или пост?
```

Источник названия языка

- Язык C++ получил название от операции ++ (постинкремента).
- Бывает также преинкремент.

```
int x = 42, y, z;
y = ++x; // y = 43, x = 43
z = y++; // z = 43, y = 44
```

- Для их переопределения используется один и тот же operator++.

```
Quat& Quat::operator++(); // это pre-increment
Quat Quat::operator++(int); // это post-increment
```

- Дополнительный аргумент в постинкременте липовый.

Источник названия языка

- Обычно постинкремент делается в терминах преинкремента.

```
struct Quat {
  int x,y,z,w;
  Quat& Quat::operator++() { x_ += 1; return *this; }
  Quat Quat::operator++(int) {
    Quat tmp {*this};
    ++(*this);
    return tmp;
  }
};
```

- Разумеется точно так же работает декремент и постдекремент.

Цепочные операторы

• Например для кватернионов:

```
struct Quat {
  int x, y, z, w;
  Quat& operator+=(const Quat& rhs) {
    x += rhs.x; y += rhs.y; z += rhs.z; w += rhs.w;
  return *this;
  }
};
```

• Здесь возврат ссылки на себя нужен чтобы организовать цепочку.

```
a += b *= c; // a.operator+=(b.operator*=(c));
```

Неявные преобразования

• Часто мы хотим чтобы работали неявные преобразования.

```
Quat::Quat(int x);
Quat Quat::operator+(const Quat& rhs);
Quat t = x + 2; // ok, int -> Quat
Quat t = 2 + x; // FAIL
```

• Увы, метод класса не преобразует свой неявный аргумент.

• Единственный вариант делать настоящие бинарные операторы это делать их вне класса.

Неявные преобразования

- Часто мы хотим чтобы работали неявные преобразования.

```
Quat::Quat(int x);
Quat operator+(const Quat& lhs, const Quat& rhs);
Quat t = x + 2; // ok, int -> Quat rhs
Quat t = 2 + x; // ok, int -> Quat lhs
```

- Увы, метод класса не преобразует свой неявный аргумент.
- Единственный вариант делать настоящие бинарные операторы это делать их вне класса.

Определение через цепочки

• Это не мешает использовать для определения бинарных операторов цепочечные с соответствующими аргументами.

```
Quat operator+(const Quat& x, const Quat& y) {
  Quat tmp {x};
  tmp += y;
  return tmp;
}
```

• Это логично и позволяет переиспользовать код.

• Кроме того такой оператор может не быть friend и действовать в терминах открытого интерфейса.

Интермедия: невезучий сдвиг

• Меньше всего повезло достойному бинарному оператору сдвига.

```
int x = 0x50;
int y = x « 4; // y = 0x500
x »= 4; // x = 0x5
```

• У него, как видите, даже есть цепочечный эквивалент.

• Но сейчас де-факто принято в языке использовать его для ввода и вывода на поток и именно в бинарной форме.

```
std::cout « x «  « y « std::endl;
std::cin » z;
```

Интермедия: невезучий сдвиг

• Обычно сдвиг делают всё-таки вне класса используя внутренний дамп.

```
struct Quat {
  int x, y, z, w;
  void dump(std::ostream& os) const {
    os « x «  « y «  « z «  « w;
  }
};
```

• И далее собственно оператор (тут не лучшая его версия).

```
std::ostream& operator«(std::ostream& os, const Quat& q) {
q.dump(os); return os;
}
```

Сравнения как бинарные операторы

- В чём отличие следующих двух способов сравнить кватернионы?

```
// 1
bool operator== (const Quat& lhs, const Quat& rhs) {
return (&lhs == &rhs);
}
bool operator== (const Quat& lhs, const Quat& rhs) {
return (lhs.x == rhs.x) && (lhs.y == rhs.y) &&
(lhs.z == rhs.z) && (lhs.w == rhs.w);
}
```

Равенство и эквивалентность

• Считается, что хороший оператор равенства удовлетворяет трём основным соотношениям.

```
assert(a == a);
assert((a == b) == (b == a));
assert((a != b) || ((a == b) && (b == c)) == (a == c));
```

• Первое это рефлексивность, второе симметричность, третье транзитивность.

• Говорят что обладающие такими свойствами отношения являются отношениями эквивалентности

Дву и три валентные сравнения

- В языке C приняты тривалентные сравнения. strcmp(p, q); // returns -1, 0, 1
- В языке C++ приняты двувалентные сравнения.

if (p > q) // if (strcmp(p, q) == 1)
if (p >= q) // if (strcmp(p, q) != -1)

- Кажется из одного тривалентного сравнения <=> можно соорудить все двухвалентные.

Spaceship operator

• В 2020 году в C++ появился перегружаемый “оператор летающая тарелка“

```
struct MyInt {
  int x_;
  MyInt(int x = 0) : x_(x) {}
  std::strong_ordering operator<=>(const MyInt &rhs) {
    return x_ <=> rhs.x_;
}
};
```

• Такое определение MyInt сгенерирует все сравнения кроме равенства и неравенства (потому что он не сможет решить какое вы хотите равенство).