КОНКРЕТНЫЕ КЛАССЫ

Конструкторы и деструкторы. Копирование и присваивание.

К. Владимиров, Intel, 2020
mail-to: konstantin.vladimirov@gmail.com

- Имена и сущности
- Конструкторы и деструкторы
- Копирование и присваивание
- Сбалансированные деревья

Одна забавная странность в языке C

- Функция `strstr(haystack, needle)` ищет подстроку needle в строке haystack
- Она определена мягко скажем странно

```
char *strstr(const char* str, const char* substr);
```

- Почему аргументы const?

- Почему если оба аргумента const, результат non-const?

https://en.cppreference.com/w/c/string/byte/strstr

Одна забавная странность в языке C

- Функция `strstr(haystack, needle)` ищет подстроку needle в строке haystack
- Она определена мягко скажем странно

```
char *strstr(const char* str, const char* substr);
```

- Почему аргументы const?
  - Потому что иначе не будет работать передача const строк
- Почему если оба аргумента const, результат non-const?
  - Потому что иначе не будет работать возврат non-const строк
- При этом мы сознательно жертвуем возвратом const строк. Омерзительно.

https://en.cppreference.com/w/c/string/byte/strstr

Обсуждение

- Как решить эту проблему?

Обсуждение

- Как решить эту проблему?
- Пункт первый: разрешить в языке перегрузку функций
- Пункт второй: перегрузить функции

```
char const * strstr(char const * str, char const * target);

char * strstr(char * str, char const * target);
```

- Теперь константность первого аргумента правильно согласована с константностью результата
- Так и сделано в C++. Увы, этого нельзя сделать в C

https://en.cppreference.com/w/cpp/string/byte/strstr

Гарантии по именам

- Язык C предоставляет строгие гарантии по именам

```
double sqrt(double); // метка не будет зависеть от сигнатуры
```

- Язык C++ не даёт гарантий по именам

```
double sqrt(double); // метка может зависеть от сигнатуры
```

- Кроме случая extern "C"

```
extern "C" double sqrt(double); // то же что и в C
```

- Последний случай введён чтобы согласовать API
- Процесс искажения имён называется манглированием

03-concrete/mangle.c
03-concrete/mangle.cc

Обсуждение

- Догадайтесь можно ли делать вот так:

```
extern "C" template <typename T> void foo(T x); // 1
struct S { extern "C" void foo(); };            // 2
```

- Обоснуйте свои догадки

Обсуждение

- Догадайтесь можно ли делать вот так:

```
extern "C" template <typename T> void foo(T x); // 1
struct S { extern "C" void foo(); };            // 2
```

- Обоснуйте свои догадки
- Оба ответа: нельзя
- Оба механизма с первой лекции: (1) обобщение данных и (2) объединение данных с методами невозможны без манглирования имён
- Точно так же перегрузка функций невозможна без манглирования имён

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
- Именно в этом и только в этом смысле в языке C есть строгие гарантии по именам. Нельзя сказать, что в C нет манглирования. Оно может и быть (stdcall добавляет подчёркивание и т.п.). Чего в C нет так это манглирования типом.

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 → int, int → const int&, etc)
2. Точное совпадение с шаблоном (int → T)
3. Стандартные преобразования (int → char, float → unsigned short, etc)
4. Переменное число аргументов
5. Неправильно связанные ссылки (literal → int&, etc)

- Мы вернёмся к перегрузке когда подробнее поговорим о шаблонах функций

03-concrete/overload.cc

Перегрузка конструкторов

- Методы класса, разумеется, тоже можно перегружать и наиболее полезно это для конструкторов

```
class line_t {
  float a_ = -1.0f, b_ = 1.0f, c_ = 0.0f;

public:
  // по умолчанию
  line_t() {}

  // из двух точек
  line_t(const point_t &p1, const point_t &p2);

  // явные параметры линии
  line_t(float a, float b, float c);
```

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, второй смысл

- 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

Правильный hello world

```
#include <iostream>

namespace {
  const char * const helloworld = "Hello, world";
}

int main() {
  std::cout << helloworld << std::endl;
}
```

- Обратите внимание: функция main обязана быть в глобальном пространстве имён

- Перегрузка функций и методов
  - Конструкторы и деструкторы
- Копирование и присваивание
- Сбалансированные деревья

Конструкторы

- Как уже было сказано конструктор используется для инициализации

```
line_t gline{1.0f, 1.0f, 0.0f};                    // global

int foo() {
  line_t sline{1.0f, 1.0f, 0.0f};                  // stack
  line_t *pline = new line_t{1.0f, 1.0f, 0.0f}; // heap

  line_t {1.0f, 1.0f, 0.0f}; // anonymous object
```

- Объект может быть сконструирован и в глобальной памяти и на стеке и в динамической памяти
- Анонимный объект, как и временный, живёт до конца полного выражения

Отступление: старая инициализация

- До 2011 года вызов конструктора предполагал круглые скобки

```
Triangle2D<double> t(p1, p2, p3); // вызов конструктора
Triangle2D<double> t{p1, p2, p3}; // вызов конструктора
```

- До сих пор это означает одно и то же. Но есть одно но.

```
myclass_t m(list_t(), list_t()); // вызов конструктора?
myclass_t m{list_t(), list_t()}; // вызов конструктора?
```

- Одна из этих строчек значит не то, что вы думаете

03-concrete/ctors-ambig.cc

Отступление: старая инициализация

- До 2011 года вызов конструктора предполагал круглые скобки

```
Triangle2D<double> t(p1, p2, p3); // вызов конструктора
Triangle2D<double> t{p1, p2, p3}; // вызов конструктора
```

- До сих пор это означает одно и то же. Но есть одно но.

```
myclass_t m(list_t(), list_t()); // объявление функции
myclass_t m{list_t(), list_t()}; // вызов конструктора
```

- По стандарту компилятор засчитывает за объявление функции всё, что выглядит как объявление функции
- Всегда для корректности старайтесь использовать фигурные скобки

Списки инициализации

- Полезны при наличии аргументов

```
template <typename T> struct Point2D {
  T x_, y_;

  Point2D(T x, T y): x_(x), y_(y) {}
```

- Это гораздо лучше, чем делать тривиальную реализацию в теле конструктора

```
  Point2D(T x, T y) { x_ = x; y_ = y; }
```

- Преимущество первого подхода очевидно?

Списки инициализации

- Полезны при наличии аргументов

```
template <typename T> struct Point2D {
  T x_, y_;

  Point2D(T x, T y): x_(x), y_(y) {}
```

- Это гораздо лучше, чем делать тривиальную реализацию в теле конструктора

```
  Point2D(T x, T y) : x_(), y_() { x_ = x; y_ = y; }
```

- Преимущество первого подхода очевидно?
- Коричневым курсивом выделен неявно вставленный компилятором код инициализации по умолчанию. Она тут точно не нужна а то и невозможна

Списки инициализации

- До 2011 года использовались даже для тривиальной инициализации

```
template <typename T> class list_t {
  node_t *top_, *back_;

public:
  list_t(): top_(nullptr), back_(nullptr) {}
```

- Список инициализации выполнял ту же роль что и прямое присвоение

```
template <typename T> class list_t {
  node_t *top_ = nullptr, *back_ = nullptr;
```

- Вы можете использовать любой подход, но не смешивайте их

Обсуждение

- Что насчёт членов ссылок и констант?

```
class Weirdo {
  int &x_;
  const int y_;

public:
  // здесь хочется написать конструктор
};
```

- Как вы думаете что здесь делать?

Обсуждение

- Что насчёт членов ссылок и констант?

```
class Weirdo {
  int &x_;
  const int y_;

public:
  Weirdo(int &x) : x_(x), y(42) {}
};
```

- Лучше всего не использовать таких членов
- Но если уж использовали то подходит только список инициализации

Делегация конструкторов

- Если конструктор делает нетривиальные вещи, его можно делегировать

```
struct class_c {
  int max = 0, min = 0, middle = 0;

  class_c(int my_max) { max = my_max > 0 ? my_max : 10; }

  class_c(int my_max, int my_min) : class_c(my_max) {
    min = my_min > 0 && my_min < max ? my_min : 1;
  }

  class_c(int my_max, int my_min, int my_middle) :
    class_c (my_max, my_min) {
    middle = my_middle < max && my_middle > min ? my_middle : 5;
  }
};
```

Деструкторы

- Кроме вещей которые объект класса делает при создании, нужно предусмотреть что он делает при окончании времени жизни

```
{
  MyVector<int> v{10}; // выделяет память на 10 целых
  // что-то делает
} // тут память нужно освободить
```

- Для этих целей в язык введён механизм деструкторов: функций, автоматически вызываемых в конце жизни объекта

Деструкторы

```
template <typename T> class MyVector { // так себе MyVector
  int size_, int capacity_;
  T *buf_;

public:
  MyVector(int cap = 0) :
    size_(0), capacity_(cap), buf_(new T[cap]) {}

  ~MyVector() {
    delete [] buf_;
  }
};
```

- Синтаксически деструктор помечается тильдой ~MyVector

Частые ненужные приседания

- Люди часто пытаются делать в деструкторе лишние обнуления состояния

```
public:
  ~MyVector() {
    delete [] buf_;
    buf_ = nullptr;
    size_ = 0;
    capacity_ = 0;
  }
};
```

- Но после того как деструктор отработал, время жизни окончено и все поля принимают неопределённое состояние, так что технически компилятор имеет право выбросить выделенные строчки

03-concrete/unnecessary-2.cc

Семантика new и delete

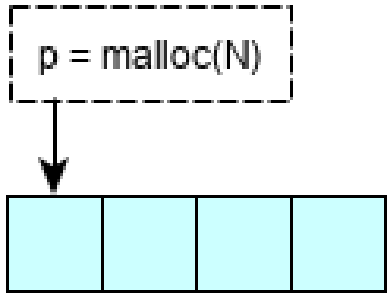
- Уже упоминалось, что new и delete важны, поскольку они вызывают конструкторы и деструкторы

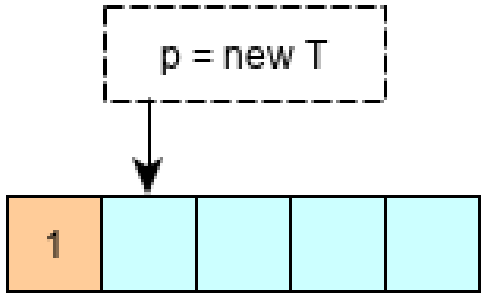
```
using mvi = MyVector<int>;
mvi *pv = new mvi{};                  // ctor
mvi *pvs = new MyVector<int>[5]; // 5 ctors
mvi *vpv = static_cast<mvi *> malloc(sizeof(mvi));
delete pv;                            // dtor
delete [] pvs;                        // 5 dtors
free(vpv); // no dtors
```

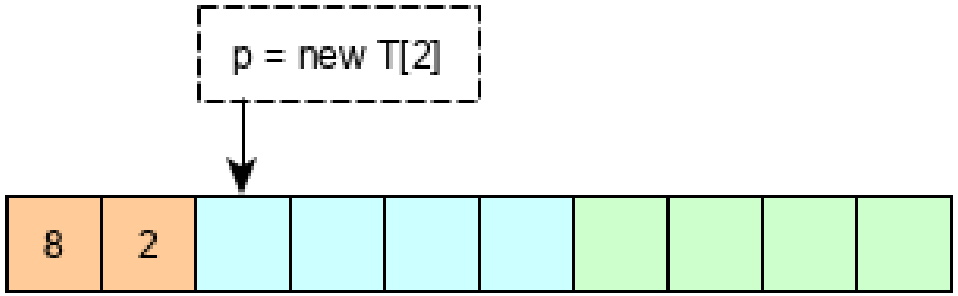
- По типу pv и pvs очень похожи. Как в точке удаления по pvs понять что нужно пять деструкторов?

Семантика new и delete

- Здесь схематично обозначено возможное выделение памяти
- Конечно не факт, что всё будет именно так, но основная идея в том, что для new и для new[] нужна дополнительная служебная информация
- Именно поэтому нельзя смешивать создание и удаление разными способами







Дополнение: value-initialization

- Встроенного типа может быть value-инициализирован если он упоминается в синтаксисе конструктора

```
{ // local scope
  int a; // a undefined
  int b{}; // b value-initialized, hence zero
```

- Для new такое тоже возможно, причём даже для массивов

```
int *pvs = new int[5]{} // calloc
```

- Увы для C++ классов нет альтернативы realloc
- Прежде чем мы поймём почему, давайте поговорим о копировании

- Перегрузка функций и методов
- Конструкторы и деструкторы
- Копирование и присваивание
- Сбалансированные деревья

Волшебные очки

- Что вы видите здесь?

```
class Empty {



};
```

Волшебные очки

- Что вы видите здесь?

```
class Empty {



};
```

- Программист видит возможность вот такого кода:

```
{
  Empty x; Empty y(x); x = y;
} // x, y destroyed
```

Волшебные очки

- Такое видение даёт волшебные очки, через которые кое что проявляется

```
class Empty {
  Empty() {} // ctor
  ~Empty() {} // dtor
  Empty(const Empty&) {} // copy ctor
  Empty& operator=(const Empty&) {} // assignment
};
```

- Все эти (и пару других) методов для вас сгенерировал компилятор

```
{
  Empty x; Empty y(x); x = y;
} // x, y destroyed
```

Семантика копирования

- По умолчанию конструктор копирования и оператор присваивания реализуют побитовое копирование и присваивание

```
template <typename T> struct Point2D {
  T x_, y_;
  Point2D() : x_(), y_() {}
  ~Point2D() {}
  Point2D(const Point2D& rhs): x_(rhs.x_), y_(rhs.y_) {}
  Point2D& operator=(const Point2D& rhs) {
    x_= rhs.x_; y = rhs.y_; return *this;
  }
};
```

RVO: return value optimizations

```
struct foo {
  foo () { cout << "foo::foo()" << endl; }
  foo (const foo&) { cout << "foo::foo( const foo& )" << endl; }
  ~foo () { cout << "foo::~foo()" << endl; }
};

foo bar() { foo local_foo; return local_foo;}

int main() {
  foo f = bar();
  use(f); // void use(foo &);
}
```

- Что здесь должно быть на экране? А что реально будет?

03-concrete/rvo.cc

Допустимые формы

- Поскольку конструктор копирования подвержен RVO, это не просто функция. У неё есть специальное значение, которое компилятор должен соблюдать
- Но чтобы он распознал конструктор копирования, у него должна быть одна из форм, предусмотренных стандартом. Основная форма это константная ссылка

```
struct Copyable {
  Copyable(const Copyable &c);
};
```

- Допустимо также принимать неконстантную ссылку, как угодно cv-квалифицированную ссылку или значение

Отступление: cv-квалификация

- В языке C++ есть два очень специальных квалификатора const и volatile
- Что означает const для объекта?

```
const int c = 34;
```

- Что означает volatile для объекта?

```
volatile int v;
```

- Что означает const volatile для объекта?

```
const volatile int cv = 42;
```

Отступление: cv-квалификация

- В языке C++ есть два очень специальных квалификатора const и volatile
- Что означает const для метода?

```
int S::foo() const { return 42; }
```

- Что означает volatile для метода?

```
int S::bar() volatile { return 42; }
```

- Что означает const volatile для метода?

```
int S::buz() const volatile { return 42; }
```

Исторический анекдот

- Что вы сможете сделать с `volatile` объектом `std::vector`?

```
volatile std::vector v;
```

- Посмотрите в предусмотренную стандартом реализацию
- Потом поэкспериментируйте

Недопустимые формы

- Шаблонный конструктор это никогда не конструктор копирования

```
template <typename T> struct Copyable {
  Copyable(const Copyable &c) {
    std::cout << "Hello!" << std::endl;
  }
};

Copyable<void> a;
Copyable<void> b{a}; // на экране Hello
```

- Здесь всё нормально, класс шаблонный, конструктор не шаблонный

Недопустимые формы

- Шаблонный конструктор это никогда не конструктор копирования

```
template <typename T> struct Coercible {
  template <typename U> Coercible(const Coercible<U> &c) {
    std::cout << "Hello!" << std::endl;
  }
};

Coercible<void> a;
Coercible<void> b{a}; // на экране ничего

Coercible<int> c{a};
```

- Здесь компилятор сгенерирует копирующий конструктор по умолчанию

03-concrete/coercion.cc

Отличия копирования от присваивания

- Копирование это в основном способ инициализации

```
Copyable a;
Copyable b(a), c{a}; // копирующее конструирование
Copyable d = a; // копирующее конструирование
```

- Присваивание это переписывание готового объекта

```
a = b; // присваивание
d = c = a = b; // присваивание цепочкой (правоассоциативно)
```

Присваивание

- Почему эта реализация присваивания крайне плоха?

```
struct File {
  FILE *f_ = nullptr;

  File &operator=(const File &rhs) {
    if (f_) close(f_);
    f_ = rhs.f_;
    return *this;
  }
};
```

Присваивание: самому себе?

- Что будет если rhs совпадает с текущим объектом?

```
struct File {
  FILE *f_ = nullptr;

  File &operator=(const File &rhs) {
    if (f_) close(f_);
    f_ = rhs.f_;
    return *this;
  }
};

File a; a = a;
```

Присваивание: самому себе?

- Основной выход: проверка `this`

```
struct File {
  FILE *f_ = nullptr;

  File &operator=(const File &rhs) {
    if (this != &rhs) {
      if (f_) close(f_);
      f_ = rhs.f_;
    }
    return *this;
  }
};
```

Поведение по умолчанию

- Можно сделать поведение по умолчанию очевидным, используя ключевое слово `default`
- Каждый раз когда вы его пишете, вы скорее всего неправы

```
template <typename T> struct Point {
  T a = 0, T b = 0;
  Point() = default;
};
```

- И зачем? Компилятор сгенерирует его правильно

Поведение по умолчанию

- Можно сделать поведение по умолчанию очевидным, используя ключевое слово `default`
- Каждый раз когда вы его пишете, вы скорее всего неправы

```
template <typename T> struct Point {
  T a_ = 0, b_ = 0;
  Point() = default;
  Point(T a, T b) : a_(a), b_(b) {}
};
```

- Осмысленно, но можно лучше

Поведение по умолчанию

- Можно сделать поведение по умолчанию очевидным, используя ключевое слово `default`
- Каждый раз когда вы его пишете, вы скорее всего неправы

```
template <typename T> struct Point {
  T a_, b_;
  Point(T a = 0, T b = 0) : a_(a), b_(b) {}
};
```

- Тут ещё кое-что не учтено, к сожалению
- Но об этом чуть позже

Владение ресурсом

- И конструктор копирования и оператор присваивания предназначены для передачи владения ресурсом
- Это очень важная концепция

```
int *arr = new int[ASZ];

foo(arr, ASZ); // foo(int *, int);

delete [] arr;
```

- Что может пойти не так в этом коде?

Владение ресурсом

- И конструктор копирования и оператор присваивания предназначены для передачи владения ресурсом
- Это очень важная концепция

```
int *arr = new int[ASZ];

foo(arr, ASZ); // foo(int * x, int) { delete [] x; }

delete [] arr;
```

- Что может пойти не так в этом коде?
- Передача указателя передаёт владение памятью

Забавный пример

```
int foo (int n) {
  int *a = new int[n];
  // .... some code ....
  if (condition) {
    delete [] a;
    return FAILURE;
  }
  // .... some code ....
  delete [] a;
  return SUCCESS;
}
```

- Хотелось бы иметь одну точку освобождения чтобы избежать проблем

Страшное goto

```
int foo (int n) {
  int *a = new int[n]; int result = SUCCESS;
  // .... some code ....
  if (condition) {
    delete [] a;
    result = FAILURE;
    goto cleanup;
  }
  // .... some code ....
cleanup:
  delete [] a;
  return result;
}
```


https://www.kernel.org/doc/Documentation/CodingStyle

Социально-приемлимое goto

```
int foo (int n) {
  int *a = new int[n]; int result = SUCCESS;
  do {
    // .... some code ....
    if (condition) {
      delete [] a;
      result = FAILURE;
      break;
    }
    // .... some code ....
  } while(0);
  delete [] a;
  return result;
}
```

Отступление: goto considered harmful

- Что вы думаете о вот таком коде?

```
struct X {
  int smth = 42;
};

int foo(int cond) {
  switch(cond) {
  case 0: X x;
  case 1: return x.smth; // 42?
  }
}
```

Отступление: goto considered harmful

- Что вы думаете о вот таком коде?

```
struct X {
  int smth = 42;
};

int foo(int cond) {
  switch(cond) {
  case 0: X x;
  case 1: return x.smth; // FAIL
  }
}
```

- К счастью это ошибка компиляции

Обсуждение

- Какие мы знаем goto-маскирующие конструкции?

Обсуждение

- Какие мы знаем goto-маскирующие конструкции?

`switch-case`, `break`, `continue`, `return`, ещё?

- Будьте со всеми ними крайне осторожны при работе с конструкторами и деструкторами. Ваш выбор явные блоки

```
int foo(int cond) {
  switch(cond) {
  case 0: { X x; }
  case 1: return x.smth; // очевидная ошибка, x не виден
  }
}
```

RAII: resource acquisition is initialization

- Чтобы не писать goto можно спроектировать класс, в котором конструктор захватывает владение, а деструктор освобождает ресурс

```
int foo (int n) {
  Buffer a(n); // new called in ctor
  // .... some code ....
  if (condition)
    return FAILURE; // dtor called: delete
  // .... some code ....
  return SUCCESS; // dtor called: delete
}
```

- Как этот класс мог бы выглядеть?

Проектирование буфера: попытка

- Казалось бы всё просто

```
class Buffer {
  int *p;
public:
  Buffer(int n) : p(new int[n]) {}
  ~Buffer() { delete [] p; }
};
```

- Что может пойти не так?

Проектирование буфера: попытка

- Казалось бы всё просто

```
class Buffer {
  int *p;
public:
  Buffer(int n) : p(new int[n]) {}
  ~Buffer() { delete [] p; }
  Buffer(const Buffer& rhs) : p(rhs.p) {}
  Buffer& operator= (const Buffer& rhs) { p = rhs.p; }
};
```

- Увы, в волшебных очках мы видим проблему

```
{ Buffer x; Buffer y{x}; } // double deletion
```

Простейшее решение: запретить

- Казалось бы всё просто

```
class Buffer {
  int *p;
public:
  Buffer(int n) : p(new int[n]) {}
  ~Buffer() { delete [] p; }
  Buffer(const Buffer& rhs) = delete;
  Buffer& operator= (const Buffer& rhs) = delete;
};
```

- Теперь компилятор может нам подсказать, что буффер не делится ресурсами

```
{ Buffer x; Buffer y{x}; } // compilation error
```

Обсуждение: renew?

- Допустим мы хотели бы ввести в язык C++ ключевое слово renew которое будет аналогом realloc
- Напомню: new выделяет память и вызывает конструкторы
- Кроме того нам хотелось бы сохранить C-like поведение: всё что может выделить malloc может реаллоцировать realloc
- С какой фундаментальной проблемой мы столкнёмся в C++?

Обсуждение: renew?

- Допустим мы хотели бы ввести в язык C++ ключевое слово renew которое будет аналогом realloc
- Напомню: new выделяет память и вызывает конструкторы
- Кроме того нам хотелось бы сохранить C-like поведение: всё что может выделить malloc может реаллоцировать realloc
- С какой фундаментальной проблемой мы столкнёмся в C++?
- Разумеется: копирующие конструкторы могут быть запрещены при этом обычные разрешены. То есть new отработает а renew не сможет
- Это первая причина по которой его нет

Обсуждение

- Что является инвариантом RAII класса?

```
class Buffer {
  int *p;
public:
  Buffer(int n) : p(new int[n]) {}
  ~Buffer() { delete [] p; }
  Buffer(const Buffer& rhs) = delete;
  Buffer& operator= (const Buffer& rhs) = delete;
};
```

Обсуждение

- Что является инвариантом RAII класса?

```
class Buffer {
  int *p;
public:
  Buffer(int n) : p(new int[n]) {}
  ~Buffer() { delete [] p; }
  Buffer(const Buffer& rhs) = delete;
  Buffer& operator= (const Buffer& rhs) = delete;
};
```

- Что никто, кроме методов класса, не имеет владеющего доступа к p
- А что, кто-то разве может?

У кого есть доступ к членам?

- Кроме методов класса, доступ есть у статических и дружественных функций

```
class S {
  int x = 0;
public:
  int get_x() const { return x; }
  static int s_get_x(const S *s) { return s->x; }
  friend int f_get_x(const S *s);
};

int f_get_x(const S *s) { return s->x; }
```

Диаграмма возможностей

	методы	статические функции	друзья
получает неявный указатель на this	да	нет	нет
находится в пространстве имён класса	да	да	нет
имеет доступ к закрытому состоянию класса	да	да	да

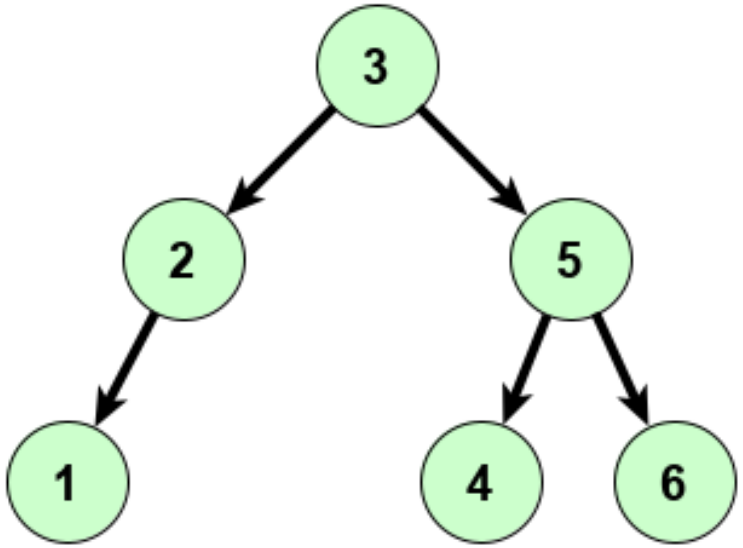
Обсуждение

- Статические функции более безопасны. Они являются частью интерфейса класса и их пишет разработчик, который заботится о сохранении инвариантов
- Функции-друзья обычно пишет кто-то другой и они могут нарушать инварианты как хотят
- Особенно опасно дружить с целыми классами
- В целом дружба часто бывает связана с магией и заводя себе друзей вы почти всегда ошибаетесь

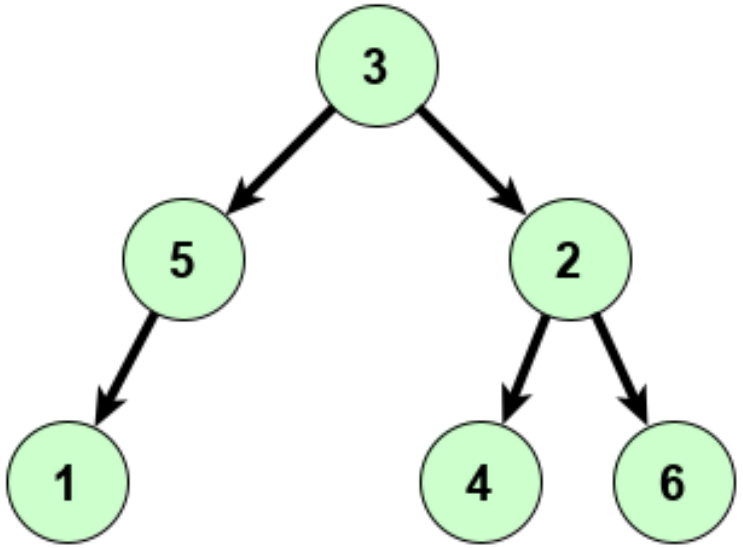
- Перегрузка функций и методов
- Конструкторы и деструкторы
- Копирование и присваивание
- Сбалансированные деревья

Поисковые деревья

- Поисковость это свойство дерева, заключающееся в том, что любой элемент в правом поддереве больше любого элемента в левом



поисковое дерево



не поисковое дерево

Обсуждение

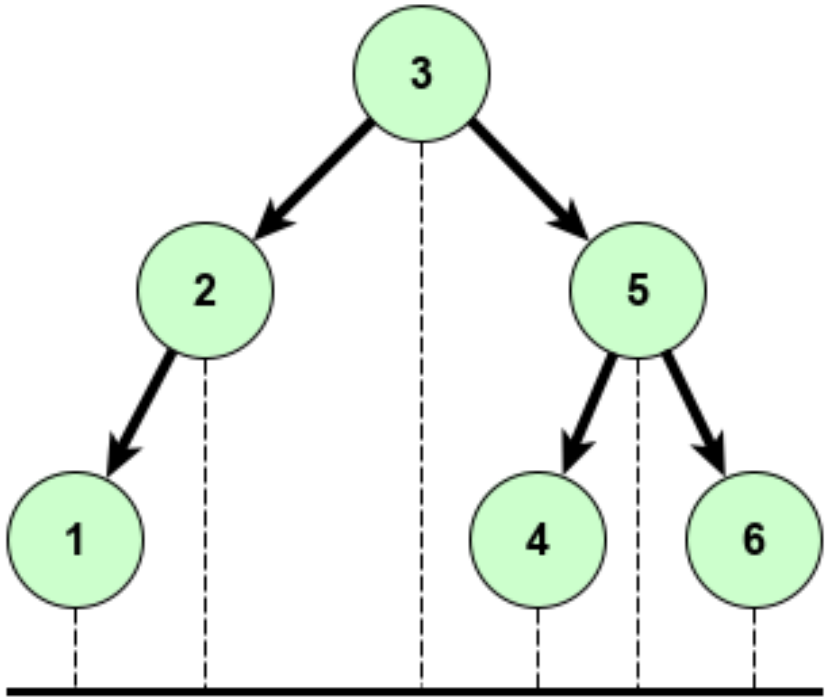
- Чем нам ценно свойство поисковости?

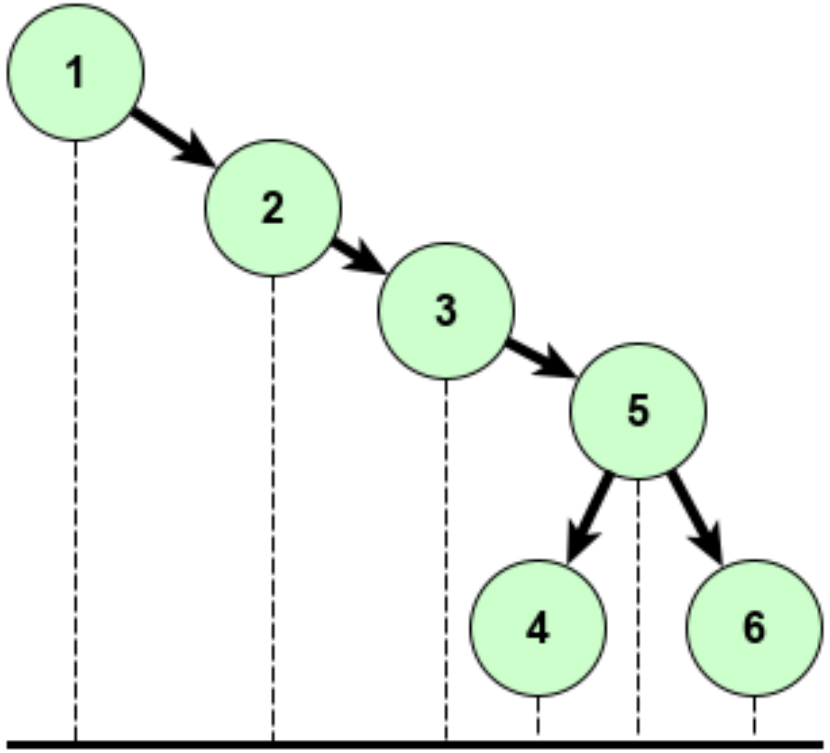
Обсуждение

- Чем нам ценно свойство поисковости?
- Тем, что любой ключ может быть найден начиная от верхушки дерева за время пропорциональное высоте дерева
- В лучшем случае у нас дерево из N элементов будет иметь высоту lgN
- Важное наблюдение: над одним и тем же множеством элементов все возможные поисковые деревья сохраняют его inorder обход сортированным
- Вопросы для математически настроенной аудитории
  - Сколько всевозможных поисковых деревьев над множеством N элементов?
  - Какова их максимальная, минимальная и средняя высота?
  - Назовём полезными деревья минимальной высоты. Сколько полезных деревьев?

Поисковые деревья: проблема

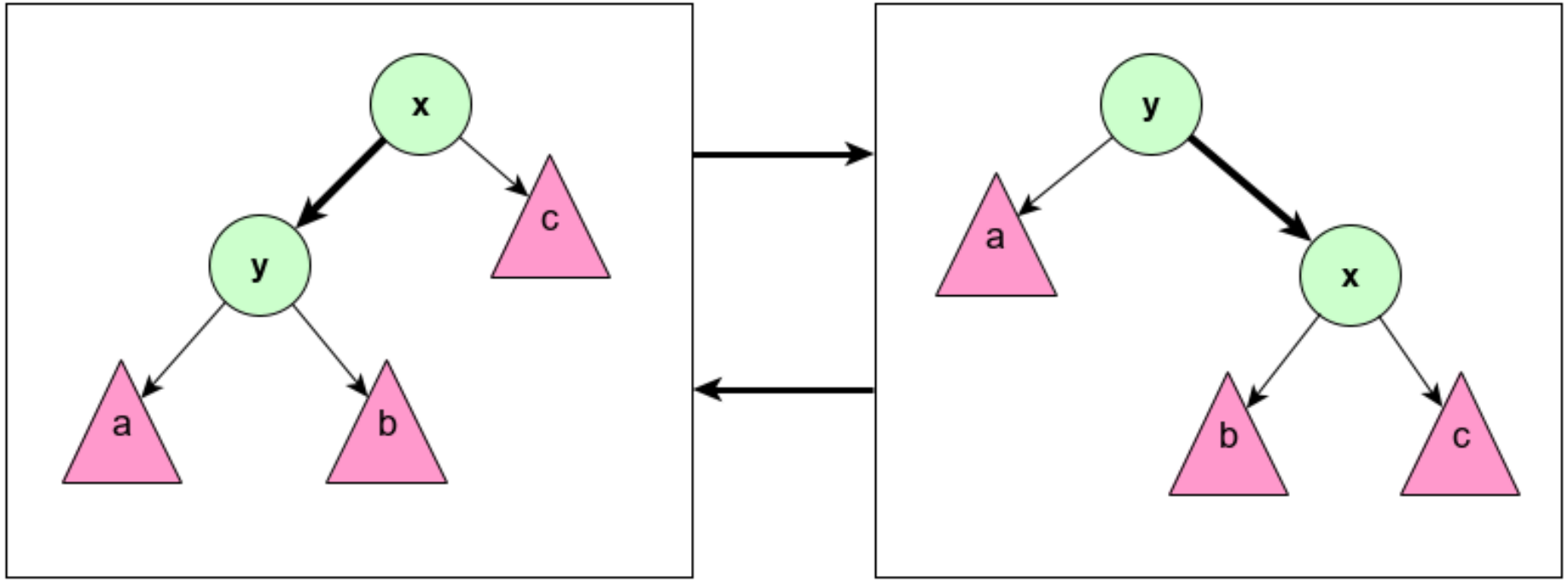
- Разные процессы построения ведут как к полезным так и к бесполезным деревьям из одних и тех же элементов





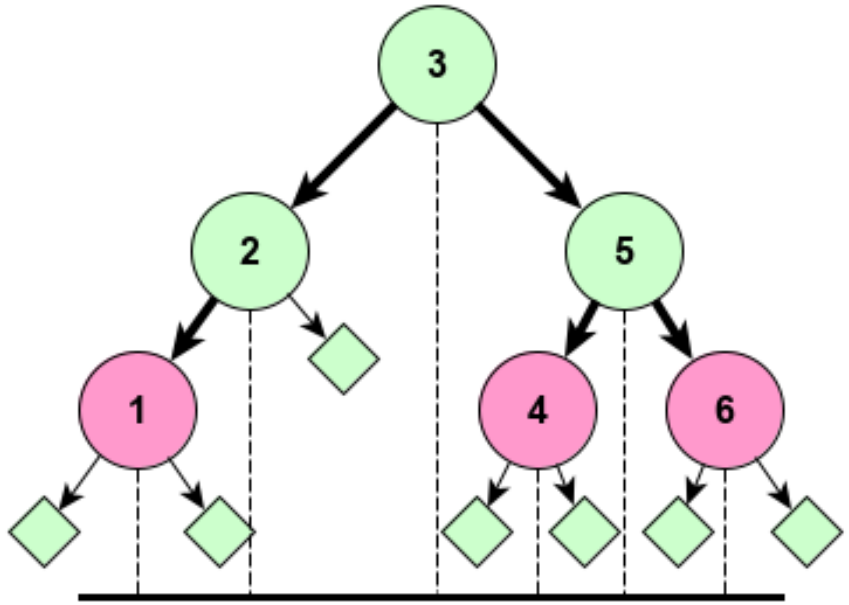
Преобразования поисковых деревьев

- Два базовых преобразования, сохраняющих инвариант поисковости это левый и правый поворот



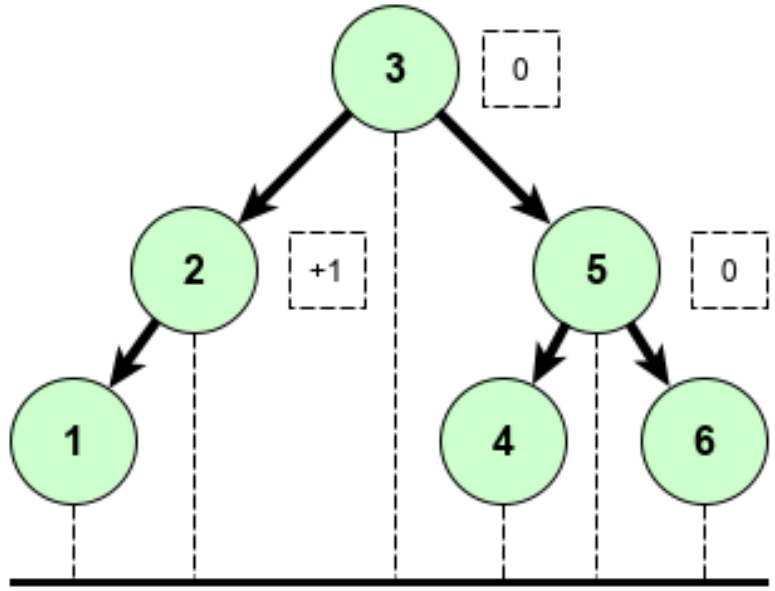
Преобразования поворотами

- Надлежащим количество поворотов можно сделать любое дерево полезным, но это нетривиальная задача
- Гораздо проще при каждой вставке поддерживать поворотами какой-нибудь инвариант, который гарантирует нам полезность дерева
- **Красно-черный** инвариант:
  - Корень черный
  - Все нулевые потомки черные
  - У каждого красного узла все потомки чёрные
  - На любом пути от данного узла до каждого из нижних листьев одинаковое количество чёрных узлов



Преобразования поворотами

- Надлежащим количество поворотов можно сделать любое дерево полезным, но это нетривиальная задача
- Гораздо проще при каждой вставке поддерживать поворотами какой-нибудь инвариант, который гарантирует нам полезность дерева
- Инвариант **AVL**:
  - Высота пустого узла нулевая
  - Высота дерева это длина наибольшего пути от корня до пустого узла
  - Для каждой вершины высота обоих поддеревьев различается не более чем на 1



Домашняя работа

- К данным, хранящимся к дереве удобно применять range queries
- К вам на вход поступает количество ключей N и далее все эти ключи (каждый ключ это целое число)
- Далее к вам на вход поступает количество запросов M и далее все запросы (каждый запрос это пара из двух целых чисел)
- Вам нужно для каждого запроса подсчитать в дереве количество ключей, таких что каждые из них лежит строго между его левой и правой границами включительно
- Вы должны сделать это быстрее чем с использованием `std::set`
- Пример. Вход: `4 10 20 30 40 2 8 31 6 9`. Результат: `3 0`

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition), 2013
- [ED] Edsger W. Dijkstra – Go To Statement Considered Harmful, 1968
- [EDH] Edsger W. Dijkstra – The Humble Programmer, ACM Turing Lecture, 1972
- [GB] Grady Booch – Object-Oriented Analysis and Design with Applications, 2007
- [$Cormen$] Thomas H. Cormen – Introduction to Algorithms, 2009
- [$TAOCP$] Donald E. Knuth – The Art of Computer Programming, 2011