СОЗДАНИЕ И ПРЕОБРАЗОВАНИЕ

Перегрузка имён. Конструкторы и деструкторы. Копирование, присваивание и приведение типов.

К. Владимиров, Intel, 2021
mail-to: konstantin.vladimirov@gmail.com

- Имена и сущности
- Сбалансированные деревья
- Конструкторы и деструкторы
- Специальные конструкторы

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

mangle.c, 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
- В языке C есть строгие гарантии по именам. Нельзя сказать, что в C нет манглирования. Оно может и быть. 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

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

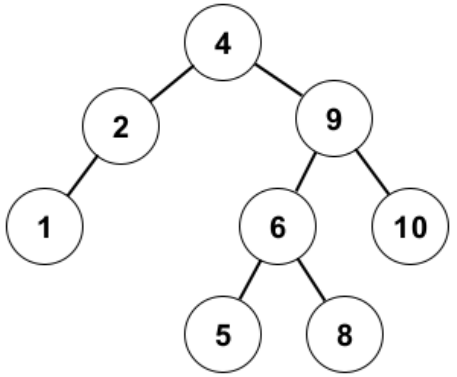
```
#include <iostream>

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

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

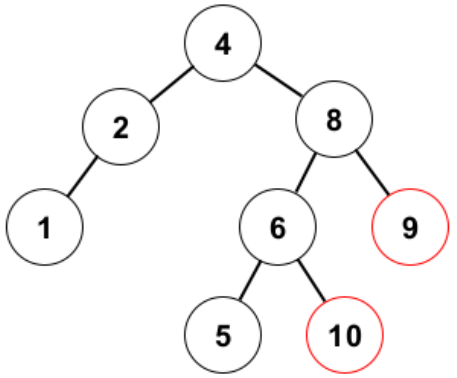
- Обратите внимание: функция main обязана быть в глобальном пространстве имён

- Перегрузка функций и методов
  - Сбалансированные деревья
- Конструкторы и деструкторы
- Специальные конструкторы

Поисковые деревья

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





Range queries

- К данным, хранящимся к дереве удобно применять range queries
- Пусть на вход поступают ключи (каждый ключ это целое число, **все ключи разные**) и запросы (каждый запрос это пара из двух целых чисел, **второе больше первого**)
- Нужно для каждого запроса подсчитать в дереве количество ключей, таких, что все они лежат строго между его левой и правой границами включительно
- Вход: `k 10 k 20 q 8 31 q 6 9 k 30 k 40 q 15 40`.
- Результат: `2 0 3`

Решение через std::set

```
template <typename C, typename T>
int range_query(const C& s, T fst, T snd) {
  using itt = typename C::iterator;
  itt start = s.lower_bound(fst);   // first not less then fst
  itt fin = s.upper_bound(snd);     // first greater then snd
  return mydistance(s, start, fin); // std::distance для set
}
```

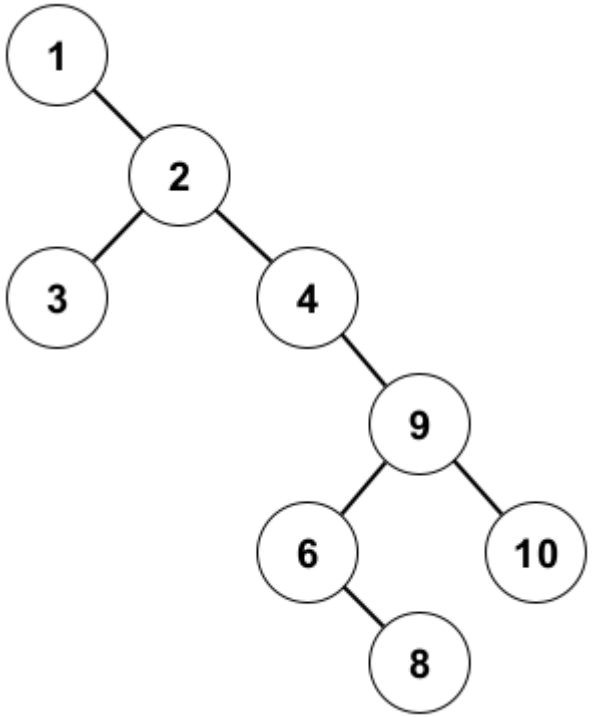
- Мы хотим, чтобы наше поисковое дерево поддерживало тот же интерфейс (кроме distance т. к. там нужны переопределённые операторы)
- Кроме того нужен метод insert для вставки ключа

Проектирование поискового дерева

```
namespace Trees {
  template <typename KeyT, typename Comp>
  class SearchTree {
    struct Node;                // внутренний узел
    using iterator = Node *; // положение внутри дерева
    Node *top_;

  public: // селекторы
    iterator lower_bound(KeyT key) const;
    iterator upper_bound(KeyT key) const;
    int distance(iterator fst, iterator snd) const;

  public: // модификаторы
    void insert(KeyT key);
```

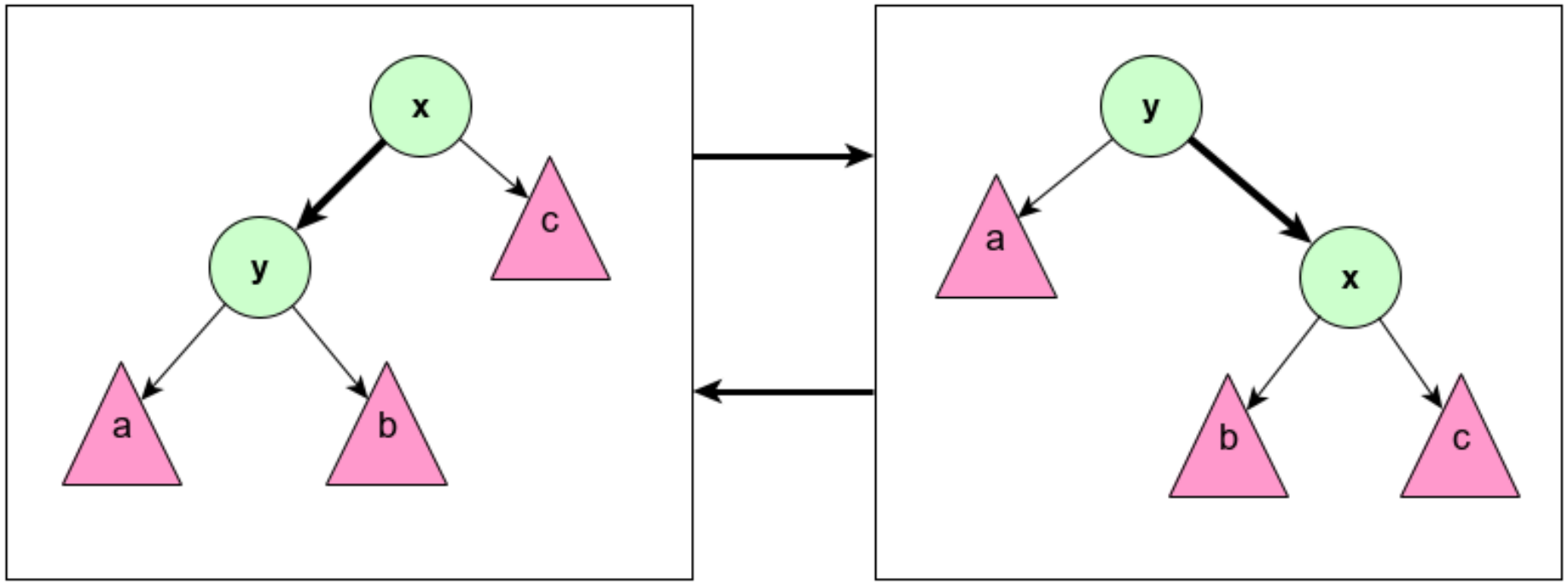
Проблема дисбаланса

- В лучшем случае поисковое дерево из N элементов будет иметь высоту lgN
- Но дерево может быть поисковым и при этом довольно бесполезным
- В худшем случае оно вырождается в список, что делает RBQ довольно неэффективными
- Но мы видим, что std::set работает довольно быстро, то есть как-то решает эту проблему



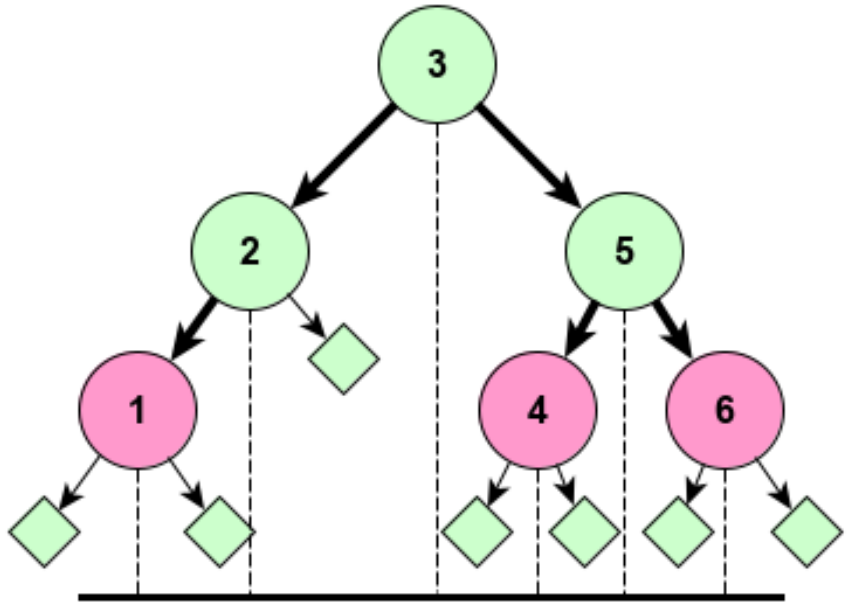
Балансировка поворотами

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



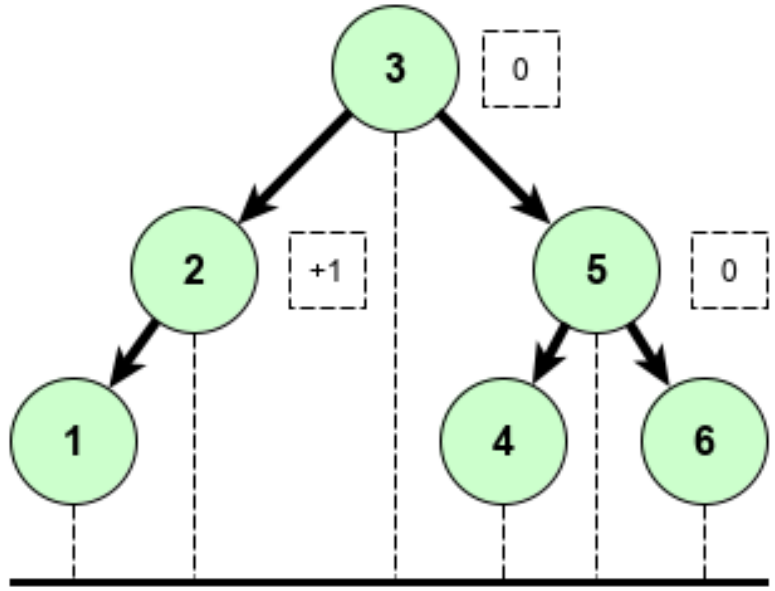
Хранение инварианта в узле

- 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



Хранение инварианта в узле

- 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



Проектирование узла

```
struct Node {
  KeyT key_;
  Node *parent_, *left_, *right_;
  int height_; // AVL инвариант
};
```

- Чем плох так спроектированный узел?

Проектирование узла

```
struct Node {
  KeyT key_;
  Node *parent_, *left_, *right_;
  int height_; // AVL инвариант
};
```

- Он может быть инициализирован только старой **агрегатной** инициализацией

```
Node n = { key, nullptr, nullptr, nullptr, 0 };
Node n = { key }; // остальные нули

Node n { key }; // остальные нули, новшество в C++11
```

Проектирование узла

```
struct Node {
  KeyT key_;
  Node *parent_, *left_, *right_;
  int balance_factor() const;

private:
  int height_; // AVL инвариант
};
```

- Агрегатная инициализация ломается при появлении приватного состояния

```
Node n { key }; // ошибка, это не агрегат
```

- Кроме того она не даёт **уверенности**, что поле key инциализировано

- Перегрузка функций и методов
- Сбалансированные деревья
- Конструкторы и деструкторы
- Специальные конструкторы

Проектирование узла

```
struct Node {
  KeyT key_;
  Node *parent_ = nullptr, *left_ = nullptr, *right_ = nullptr;
  int height_ = 0;

  Node(KeyT key) { key_ = key; } // конструктор
};
```

- Он может быть инициализирован либо **direct** либо **copy** инициализацией

```
Node n(key);  // прямая инициализация, старый синтаксис
Node n{key};  // прямая инициализация, новый синтаксис
Node k = key; // копирующая инициализация
```

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

ctors-ambig.cc

Двойная инициализация

- Присваивая в теле конструктора, мы инициализируем дважды (второй раз временный объект для присваивания

```
struct S {
  S() { std::cout << "default" << std::endl; }
  S(KeyT key) { std::cout << "direct" << std::endl; }
};

struct Node {
  S key_; int val_;

  Node(KeyT key, int val) { key_ = key; val_ = val; }
```


https://godbolt.org/z/7ovsoobzE

Списки инициализации

- Чтобы уйти от двойной инцииализации, до тела конструктора предусмотрены списки инициализации

```
struct S {
  S() { std::cout << "default" << std::endl; }
  S(KeyT key) { std::cout << "direct" << std::endl; }
};

struct Node {
  S key_; int val_;

  Node(KeyT key, int val) : key_(key), val_(val) {}
```


https://godbolt.org/z/oEWEhPvYh

Два правила для инициализации

- Список инициализации выполняется строго в том порядке, в каком поля определены в классе (не в том, в каком они записаны в списке)

```
struct Node {
  S key_; T key2_;
  Node(KeyT key) : key2_(key), key_ (key) {} // S, T
```

- Инициализация в теле класса незримо входит в список инициализации

```
struct Node {
  S key_ = 1; T key2_;
  Node(KeyT key) : key2_(key) {}  // S, T
};
```


https://godbolt.org/z/cGsbhzKEd

Параметры по умолчанию

- Если что-то уже есть в списке инициализации, то инициализатор в теле класса игнорируется

```
struct Node {
  S key_ = 1;
  Node() {}  // key_(1)
  Node(KeyT key) : key_(key) {}  // key_(key)
```

- Такое лучше переписать с **параметром по умолчанию**

```
struct Node {
  S key_;
  Node(KeyT key = 1) : key_(key) {}  // key_(key)
```

Обсуждение: делегация конструкторов

- Если конструктор делает нетривиальные вещи, его можно делегировать

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

  class_c(int my_max) : max(my_max > 0 ? my_max : DEFAULT_MAX) {}

  class_c(int my_max, int my_min) : class_c(my_max),
    min(my_min > 0 && my_min < max ? my_min : DEFAULT_MIN) {}
```

- Место делегированного конструктора первое в списке инициализации
- Далее делегирующий конструктор можно тоже делегировать и т.д.

Проектирование узла

- Кроме создания нам нужно освобождать память

```
struct Node {
  KeyT key_;
  Node *parent_ = nullptr, *left_ = nullptr, *right_ = nullptr;
  int height_ = 0;

  Node(KeyT key) : key_(key) {} // конструктор
  ~Node() { delete left_; delete right_; }
};
```

- Здесь деструктор через delete рекурсивно вызывает деструкторы подузлов
- Чем это решение плохо?

Мнимые и реальные проблемы

```
template <typename KeyT, typename Comp>
SearchTree::~SearchTree() { delete top_; }

template <typename KeyT, typename Comp>
SearchTree::Node::~Node() { delete left_; delete right_; }
```

- Пример некачественной критики: нет проверки на nullptr
- Пример качественной критики: возможно переполнение стека
- Как бы вы сделали без рекурсии?

Частые ненужные приседания

- Люди часто пытаются делать в деструкторе лишние обнуления состояния

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

- После того как деструктор отработал, время жизни окончено
- Технически компилятор имеет право **выбросить** выделенные строчки

unnecessary-2.cc

Ассимметрия инициализации

- Для класса с конструктором без аргументов, нет разницы между:

```
SearchTree s;   // default-init, SearchTree()
SearchTree t{}; // default-init, SearchTree()
```

- Но для примитивных типов и агрегатов разница гигантская

```
int n;   // default-init, n = garbage
int m{}; // value-init, m = 0

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

- То же самое для полей классов и т.д. рекурсивно

- Перегрузка функций и методов
- Сбалансированные деревья
- Конструкторы и деструкторы
- Специальные конструкторы

Волшебные очки

- Что вы видите здесь?

```
class Empty {



};
```

Волшебные очки

- Что вы видите здесь?

```
class Empty {




};
```

- Программист видит возможность скопировать и присвоить:

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

Отличия копирования от присваивания

- Копирование это в основном способ инициализации

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

- Присваивание это переписывание готового объекта

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

- Ergo: копирование похоже на конструктор. Присваивание совсем не похоже.

Волшебные очки

- Посмотрим на пустой класс через волшебные очки

```
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 для встроенных типов и агрегатов
  - вызов конструктора копирования, если есть

```
template <typename T> struct Point2D {
  T x_, y_;
  Point2D() : default-init x_, default-init 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;
  }
};
```

Обсуждение

- Должны ли мы делать неявное явным?

```
template <typename T, typename KeyT> class Cache {
  std::list<T> cache_;
  std::unordered_map<KeyT, T> locations_
```

- Здесь не нужны конструктор копирования и оператор присваивания

```
Cache c1 {c2}; // или Cache c1 = c2;
c2 = c1;
```

- По умолчанию копирование и присваивание тут отлично работают
- В таких случаях мы не должны определять копирование/присваивание

Случай когда умолчание опасно

- Казалось бы всё просто

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

- Что может пойти не так?

Случай когда умолчание опасно

- 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
```

Default и delete

- Мы можем явно попросить дефолтное поведение прописав default и явно его заблокировать, написав delete

```
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;
};

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

Обсуждение

- Хорошая ли идея иметь некопируемый буфер?

Реализуем копирование

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

  // думайте о "Buffer rhs; Buffer b{rhs};"
  Buffer(const Buffer& rhs) : n_(rhs.n_), p_(new int[n_]), {
    std::copy(p_, p_ + n_, rhs.p_);
  }

  Buffer& operator= (const Buffer& rhs);
};
```

Реализуем присваивание

```
Buffer& Buffer::operator= (const Buffer& rhs) {
  n_ = rhs.n_;
  delete [] p_;
  p_ = new int[n_];
  std::copy(p_, p_ + n_, rhs.p_);
  return *this;
}
```

- Тут можно визуализировать это как:

```
Buffer a, b; a = b;
```

- Видите ли вы ошибку в коде?

Не забываем о себе

```
Buffer& operator= (const Buffer& rhs) {
  if (this == &rhs) return *this;
  n_ = rhs.n_;
  delete [] p_;
  p_ = new int[n_];
  std::copy(p_, p_ + n_, rhs.p_);
  return *this;
}
```

- Первая проблема это присваивание вида `a = a`. Её довольно просто решить.
- Вторая проблема сложнее. Её мы пока отложим и поговорим о специальной семантике копирования и присваивания.

Спецсемантика копирования: RVO

```
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?

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. Для оператора присваивания также 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

coercion.cc

Спецсемантика инициализации

- Обычные конструкторы определяют неявное преобразование типа

```
struct MyString {
  char *buf_; size_t len_;
  MyString(size_t len) : buf_{new char[len]{}}, len_{len} {}
};

void foo(MyString);

foo(42); // ok, MyString implicitly constructed
```

- Почти всегда это очень полезно
- Но это не всегда хорошо, например в ситуации со строкой, мы ничего такого не имели в виду

implicit.cc

Требуем ясности

- Ключевое слово `explicit` указывается когда мы хотим заблокировать пользовательское преобразование

```
struct MyString {
  char *buf_; size_t len_;
  explicit MyString(size_t len) :
    buf_{new char[len]{}}, len_{len} {}
};
```

- Теперь здесь будет ошибка компиляции

```
void foo(MyString);

foo(42); // error: could not convert '42' from 'int' to 'MyString'
```

Снова direct vs copy

- Важно понимать, что explicit конструкторы рассматриваются для прямой инициализации

```
struct Foo {
  explicit Foo(int x) {} // блокирует неявные преобразования
};

Foo f{2}; // прямая инициализация

Foo f = 2; // инициализация копированием, FAIL
```

- В этом смысле инициализация копированием похоже на вызов функции

Пользовательские преобразования

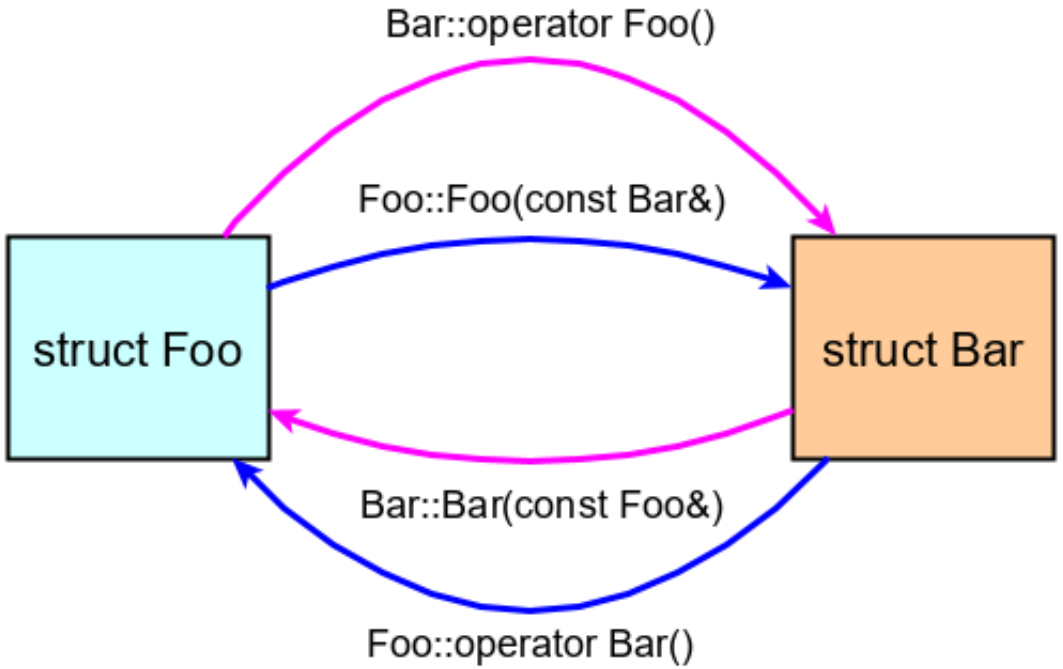
- В некоторых случаях мы не можем сделать конструктор. Скажем что если мы хотим неявно преобразовывать `Quat<int>` в `int`?
- Тогда мы пишем operator type

```
struct MyString {
  char *buf_; size_t len_;

  /* explicit? */ operator const char*() { return buf_; }
```

- Можно `operator int`, `operator double`, `operator S` и так далее
- На такие операторы можно навешивать `explicit` тогда возможно только явное преобразование

Пользовательские преобразования

- Таким образом есть некая избыточность: два способа перегнать туда и два способа перегнать обратно
- Конечно хороший тон это использовать конструкторы где возможно
- Как вы думаете что будет при конфликте?



Перегрузка

- Пользовательские преобразования участвуют в перегрузке
- Они проигрывают стандартным, но выигрывают у троеточий

```
struct Foo { Foo(long x = 0) {} };

void foo(int x);
void foo(Foo x);

void bar(Foo x);
void bar(...);

long l; foo(l); // вызовет foo(int)

bar(1); // вызовет bar(Foo)
```

Такие разные операторы

- Перегрузка операторов присвавания и приведения выглядит непохоже.

```
struct Point2D {
  int x_, y_;

  Point2D& operator=(const Point2D& rhs) = default;

  operator int() { return x; }
};
```

- В мире конструкторов спецсемантика есть только у копирования и приведения.
- В мире переопределенных операторов она есть везде и она нас ждёт уже на следующей лекции

Домашняя работа HWT

- Со стандартного ввода приходят ключи (каждый ключ это целое число, **все ключи разные**) и запросы двух видов.
- Запрос (m) на поиск k-го наименьшего элемента.
- Запрос (n) на поиск количества элементов, меньших, чем заданный.
- Вход: `k 8 k 2 k -1 m 1 m 2 n 3`
- Результат: `-1 2 2`
- Ключи могут быть как угодно перемешаны с запросами. Чтобы успешно пройти тесты, вы должны продумать такую балансировку дерева, чтобы оба вида запросов работали с логарифимической сложностью.

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition), 2013
- [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
- [CB] Charles Bay – Instruction Re-ordering Everywhere: The C++ 'As-If' Rule and the Role of Sequence, CppCon, 2016