Инициализация.

Введение объектно-ориентированное программирование

20.02.2025

Инициализация в языке C

• Язык C содержит массивы и структуры, инициализируемые как агрегаты.

scalar-type value; // local scope, value undefined

scalar-type value = initializer;

agregate-type value = initializers ;

• И это всё. Приятная простота.

int arr[5] = 1, 2, 3;

struct S int x, y; s = 1, 2;

Что добавил язык C++?

- У объектов появились конструкторы.

MyClass obj(1, 2); // конструктор это что-то вроде функции
int i(1); // имитируем то же самое для int
int k = int(1); // то же, но с временным объектом

- Синтаксис конструктора с прямым указанием значения это direct initialization.
- Уже это приводит к странным последствиям.

int i(1, 2); // ошибка
int i = (1, 2); // ok, comma operator

Скобки в объявлениях

Круглые скобки это обычный разделитель для грамматических конструкций.

```
int (v0); // int v0;
int (v1)[5]; // int v1[5];
int (&v2)[5] = v1; // тут уже обязательно ставить скобки
unsigned (*(*v3[4])(const (int *)[2]))(int); // и т. д.
```

- При объявлении они ничего кроме этого не значат.
- И становится очень странно, когда они начинают что-то значить в инициализациях.

Most vexing parse

•Частая проблема до 2011-го.

list<int> lst(istream_iterator<int>(cin), istream_iterator<int>());

• Все понимают что это такое?

resolution is to consider any construct that could possibly be a declaration a declaration [dcl.ambig.res]

• До C++11 решением были дополнительные скобки.

list<int> lst((istream_iterator<int>(cin)), (istream_iterator<int>()));

Что добавил C++11

• Возможность вызвать конструктор объекта с помощью фигурных скобок.

```
int i{ }; // точно не объявление функции
int j{5}; // тоже можно
list<int> lst{istream_iterator<int>{cin}, // всё хорошо
istream_iterator<int>{ } };
```

• Это работает для любых пользовательских объектов:

```
struct S{ S(int, int); }; S t{1, 2}; // ok
```

• Это называется uniform (unicorn) initialization.

Перегруженные значения {}

- Агрегатная инициализация если это агрегат (структура массив).
  Разрешены сужающие преобразования.
- Инициализация initializer-list ctor если он есть (отложим это пока).
  Запрещены сужающие преобразования.
- Просто вызов конструктора (или value-init если скобки пустые).
  Запрещены сужающие преобразования.

```
struct S { int x = 0, y = 0; };
S s{1, 2}; // агрегатная инициализация
struct T { int x = 0, y = 0; T(int a, int b) : x(a + b) { } };
T t{1, 2}; // вызов конструктора
```

Основные виды инициализации

• Для понимания правил инициализации, необходимо выучить совсем немного основных категорий инициализации.

```
int global; // zero-initialization
int foo() {
    std::vector<int> v; // default-initialization
    int j {}; // value-initialization
    int k = 7; // copy-initialization
    int i{7}; // direct-initialization
    std::vector<int> w {1, 2, 3}; // direct-list-initialization
    std::vector<int> z = {1, 2, 3}; // copy-list-initialization
```

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

```
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 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;
  int balance_factor() const;
private:
  Node *parent, *left, *right;
  int height;
};
```

• Агрегатная инициализация ломается при появлении приватного состояния.

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

• Кроме того она не даёт **уверенности**, что поле key инициализировано.

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
```

Списки инициализации

• Чтобы уйти от двойной инициализации, 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) {}
```

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) {}
};
```

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) {}
```

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

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

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?

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. This is a late parrot. Технически компилятор имеет право выбросить выделенные строчки.

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.

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 {
public: Empty(); // ctor
public: ~Empty(); // dtor
public: Empty(const Empty&); // copy ctor
public: 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.

```
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;
  }
};
```

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
```

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 brhs;"
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& 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?

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;

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.

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)
```