Initialization types

Агрегатная инициализация

```
struct Data x = {1, nullptr, "hi};
struct Numbers y = {1} // {1, 0, 0, 0, ...}
```

Агрегат невозможно реализовать, если в классе будет хотя бы одно приватное поле.

Default init

```
Data x;
Data x{};
```

Прямая инициализация

```
Data x(data); // old syntax
Data y{data}; // new sytax
```

Приоритет инициализации `{}`:

- Агрегат?
- Инициализирующий лист
- Конструктор?

> Это не всегда так:
>
> ```
>     std::vector<int> v{1, 2} // constructor(1, 2)
>     std::vector<int> v(1, 2) // v: [ 1 2 ]
> ```

Копирующая инициализация

```
Data x = arg; // like Data x(arg) w/ some difference
```

Зачем добавили новый синтаксис в прямую инициализацию?

```
// in ctors-ambig.cc
MyC m(list_t(), list_t()); // декларация функции, m.field вызовет ошибку

MyC m{list_t(), list_t()}; // вызов конструктора
// В старом синтаксисе можно было написать так
MyC m((list_t()), (list_t()));
```

Правильная инициализация конструктора

В момент захода в тело конструктора все поля уже проинициализированы. Если мы инициализируем в теле конструктора, то мы будем выполнять лишнюю работу. Его нужно инициализировать раньше. А если поле - константный указатель? Его невозможно проинициализировать в теле.

```
class A {
int m_a;
public:
    A::A() {/* ... */};
}
```

после волшебных очков оказывается:

```
class A {
int m_a;
public:
    A::A() : m_a("rubbish") {/* ... */}; // прямая инициализация списка в том порядке, в котором указаны поля в классе - порядок НЕ ЗАВИСИТ ОТ ТОГО, КАК ВЫ ЕГО УКАЗАЛИ после :
}
```

При этом `m_a{0}` на самом деле неявно вставляется в `A::A() : m_a(0) ...`.
Более того:

```
class A {
int m_a{1};
public:
    A::A() : m_a(2) {/* ... */}; // победит иниц. двойкой!
}
```

> Ненужно после `delete ptr;` чистить указатель `ptr`. У него закончилось время жизни! Если потом еще и встретится `}` то при оптимизациях компилятор строки типа `ptr = nullptr;` просто выкинет. Более того, `delete nullptr` законно.

Немного об инициализации

```
int a; // просто сдвиг стек-пойнтера --> просто мусор в переменной
int b{}; // zero-init (value-init)

int *pc = new init[5]{} // calloc
int *pm = new init[5] // malloc

// this is all zero-init for int!
int a = 0;
int a(0);
int a{0};
```

> По умолчанию коснструктор копирования если втроенные типы/агрегаты, либо вызов конструкторов копирования у частей.

В случае перегрузки опреатора присваивания и использовании в теле дин. указателей, то проверка на самоприсваивание обязательна!

RVO (very important frontend return value optimization)

...

CV-classification (const volatile classification)

`explicit` блокирует неявные преобразования:

```
class A {
public:
    A(int n) { /* ... */ }
//  explicit A(int n) { /* ... */ }
}

foo(A obj);

int main() {
    foo(42); // ok: foo(A obj(42)); // can be prohibited by explicit
}
```

```
struct A {
    explicit A(int x) {}
}

int main() {
    A obj{2} // direct init
    A obj = 2 // copy-init - FAIL
}
```