Взаимозаменяемые AoS и SoA контейнеры

Павел Крюков

Преподаватель-исследователь

Московский физико-технический институт

Лаборатория MIPT-ILab: https://mipt-ilab.github.io/

Содержание

- Коротко о пространственной локальности
- AoS и SoA организации массивов
- Качественное сравнение пространственной локальности AoS и SoA
- Результаты количественного сравнения
- Путь к взаимозаменяемым интерфейсам AoS и SoA контейнеров
- Заключение и дальнейшие шаги

Пространственная локальность

«За обращением к данным чаще всего следует обращение к соседним данным»

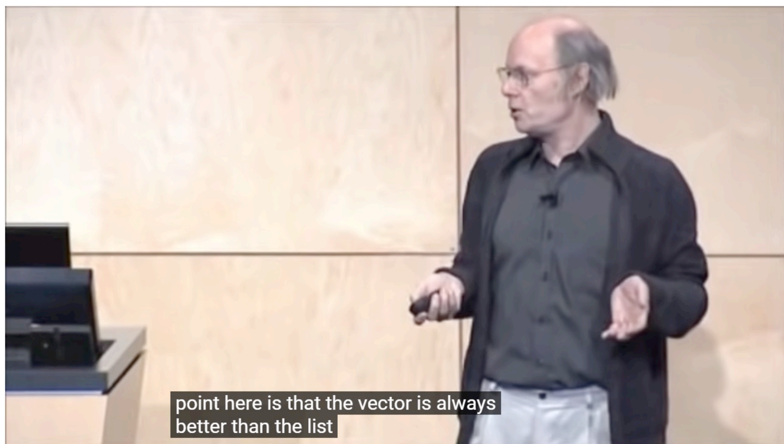
- Win-win для разработчиков как SW, так и HW
  - Программы становятся быстрее с увеличением локальности
  - CPU и DRAM ещё более оптимизируются под такие программы

- Программные примеры
  - Массивы
  - Секция кода
  - Стек вызова

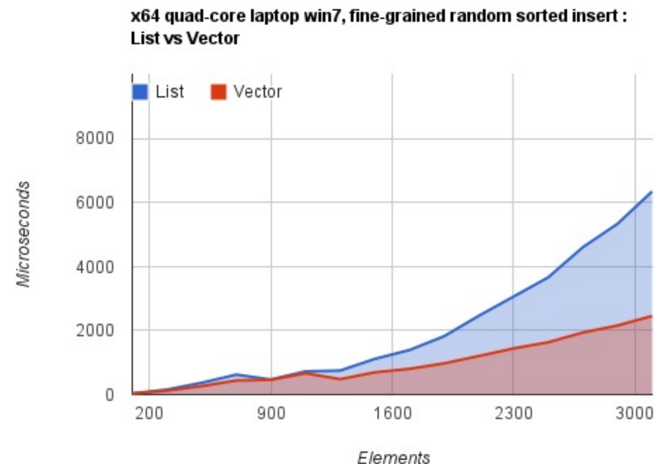
- Аппаратные оптимизации
  - Кэш-память
  - Векторные команды
  - Блочная организация
  - Предварительная подкачка данных (prefetch)

Пример: связные списки

- Локальность связных списков меньше, чем у массивов
  - меньше производительность
  - меньше используются в современном программировании
- Авторитетные и аргументированные мнения:



Bjarne Stroustrup: Why you should avoid Linked Lists
https://www.youtube.com/watch?v=YQs6IC-vgmo



Number crunching: Why you should never, ever, EVER use linked-list in your code again
https://www.codeproject.com/Articles/340797/Number-crunching-Why-you-should-never-ever-EVER-us

Оптимизации массивов

- Массив можно организовать разными способами:

Массив структур

- *англ. Array of Structures, AoS*
- Встроен в C++ и большинство других языков

```
struct Point {
    double x, y, z, n;
}; std::vector<Point> s;
for (auto& e : s)
    e.n = calc(e.x, e.y, e.z);
```

Структура массивов

- *англ. Structure of Arrays, SoA*
- «Собирается» из контейнеров C++

```
struct SoA {
   std::vector<double> x, y, z, n;
} s;
for (int i = 0; i < s.n.size(); ++i)
    s.n[i] = calc(s.x[i], s.y[i], s.z[i]);
```

- Что имеет бо́льшую локальность?

Пространственная локальность: AoS

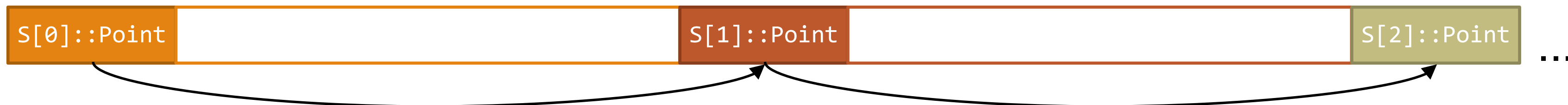
- Если используются все поля, то данные читаются непрерывно
  - Пример: копирование и инициализация структуры
  - Идеально для кэш-памяти, предварительной подкачки данных и т. д.

```
for (Point& e : s) // ptr += 32 bytes
    e.n = calc(e.x, e.y, e.z);
```

Как испортить локальность в AoS

- Добавлением полей в структуру
  - В кэш загружаются 64-байтные блоки, но не все данные в них используются
  - Требуются более сложные алгоритмы аппаратной предподкачки с предсказанием шага

```
struct NamedPoint : Point { std::array<char, 96> name; };
for (NamedPoint& e : s) // ptr += 128 bytes
    e.n = calc(e.x, e.y, e.z);
```



Пространственная локальность: SoA

- SoA подходит для «неравномерных» структур
  - используемые данные упакованы вновь локально
  - «лишние» данные не кэшируются

```
for (int i = 0; i < s.n.size(); ++i) // ptr += 8 bytes
    s.n[i] = calc(s.x[i], s.y[i], s.z[i]);
```

s[0].x	s[1].x	s[2].x	…
s[0].y	s[1].y	s[2].y	…
s[0].z	s[1].z	s[2].z	…
s[0].n	s[1].n	s[2].n	…

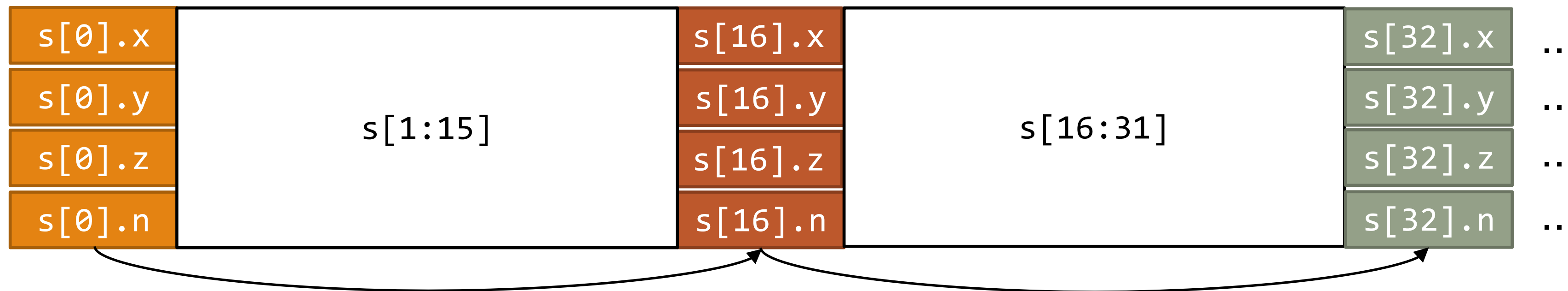
s[0].name	s[1].name	s[2].name	…

Как испортить локальность в SoA

Увеличением шага итерации

- В кэш загружаются 64-байтные блоки, но не все данные в них используются
- Требуются более сложные алгоритмы аппаратной предподкачки с предсказанием шага

```
for (int i = 0; i < s.n.size(); i += 16) // ptr += 128 bytes
    s.n[i] = calc(s.x[i], s.y[i], s.z[i]);
```



Что же выбрать?

- Выбор между AoS и SoA обусловлен параметрами всех уровней системы:

- **Алгоритм:**
  - *Размер контейнера*
  - *Размер элемента*
  - *Активная область элемента*
  - *Последовательность обхода контейнера*
  - Алгоритм обработки элемента
  - …

- **Система команд:**
  - Векторизация
  - Выравнивание
  - ...

- **Аппаратура – кэши:**
  - количество уровней
  - объём
  - aliasing
  - время доступа
  - пропускная способность
  - объём таблиц трансляции
  - …

- Трудно построить формулу, принимающую все факторы
- Вспомним пример связных списков? Уже простая модель дала уверенный ответ

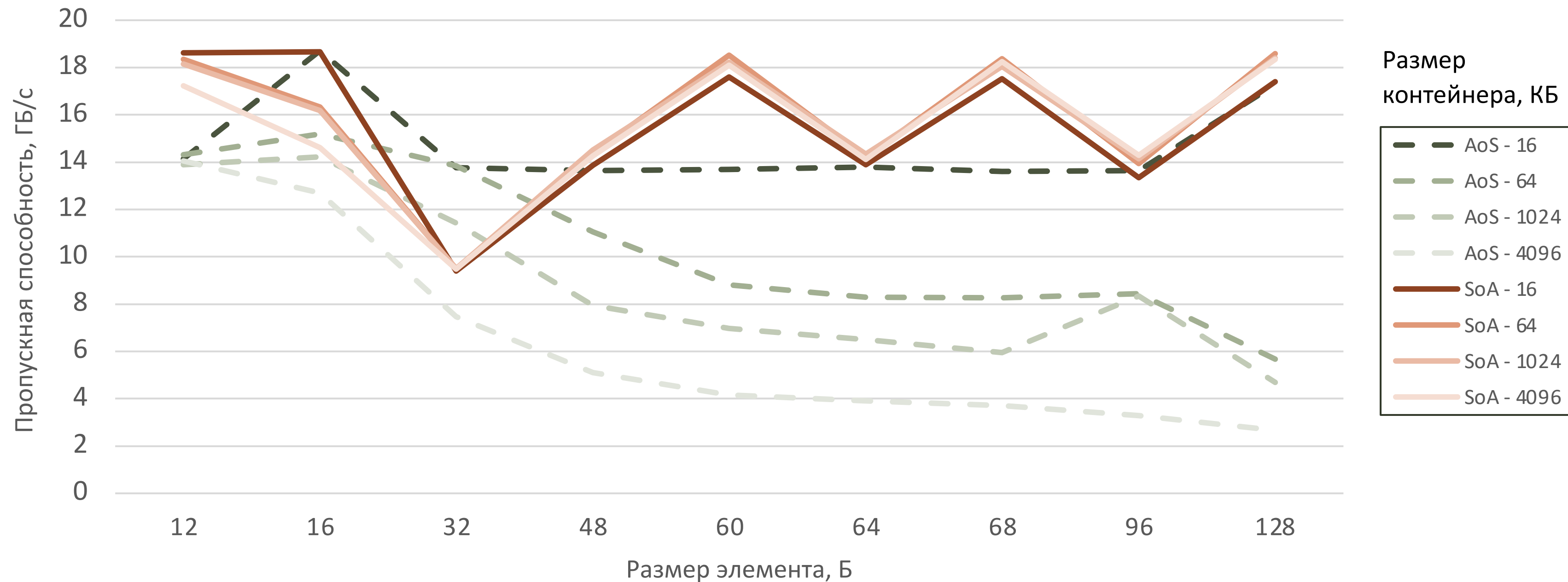
Упрощение модели

- Зафиксируем аппаратуру и систему команд
- Тактовая частота: 2095 МГц
- Размеры кэшей:
  - L1 – 32 КБ
  - L2 – 1 МБ
- Отключена векторизация
- Выравнивание массивов на 4 КБ

Методология измерений

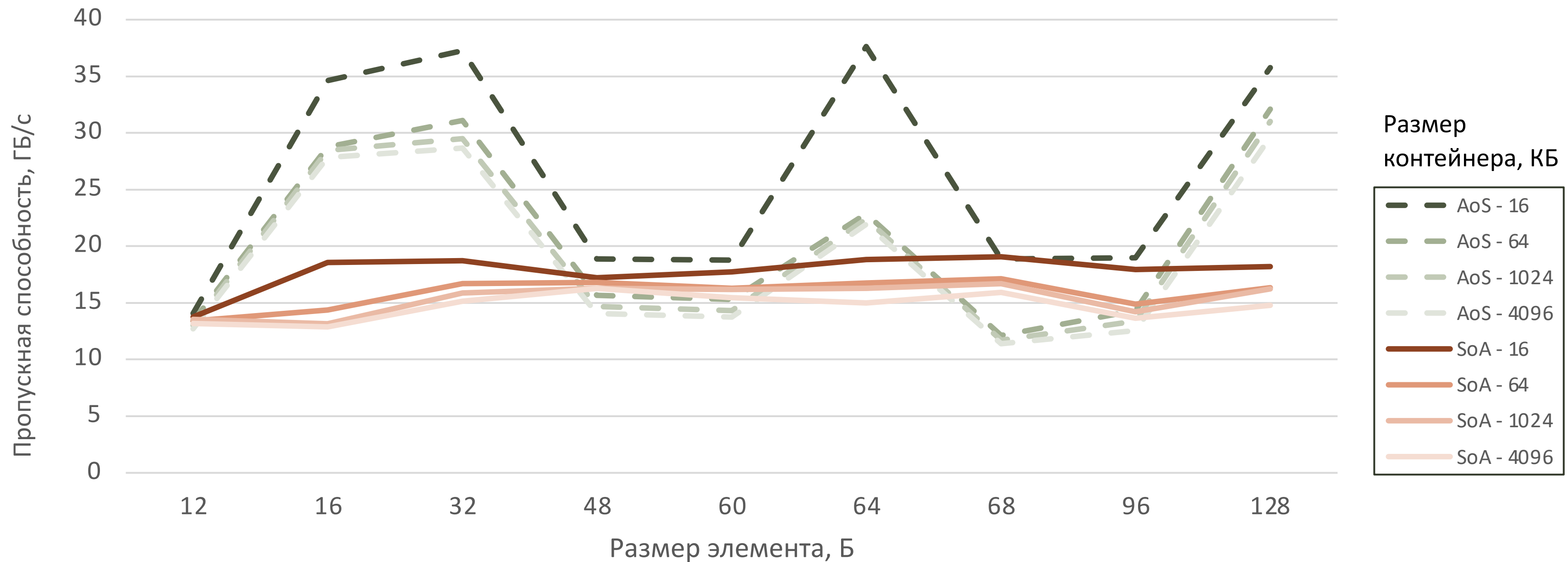
- Вариация размера контейнера:
  - 16 КБ, 64 КБ, 1 МБ, 4 МБ – разные отношения с объёмом L1 и L2 кэшей
- Вариация размера элемента
  - 12, 16, 32, 48, 60, 64, 68, 96, 128 Б – целые и дробные доли кэш-линии
- Вариация активной части элемента:
  - 12 Б, то есть остальная часть будет «лишней»
  - Полное использование
- Вариация шага: каждый элемент или каждый 16-й элемент
- Измерим ГБ/с в поисках тривиальной зависимости

Бенчмарк 1:
12 активных байтов, остальные «лишние»



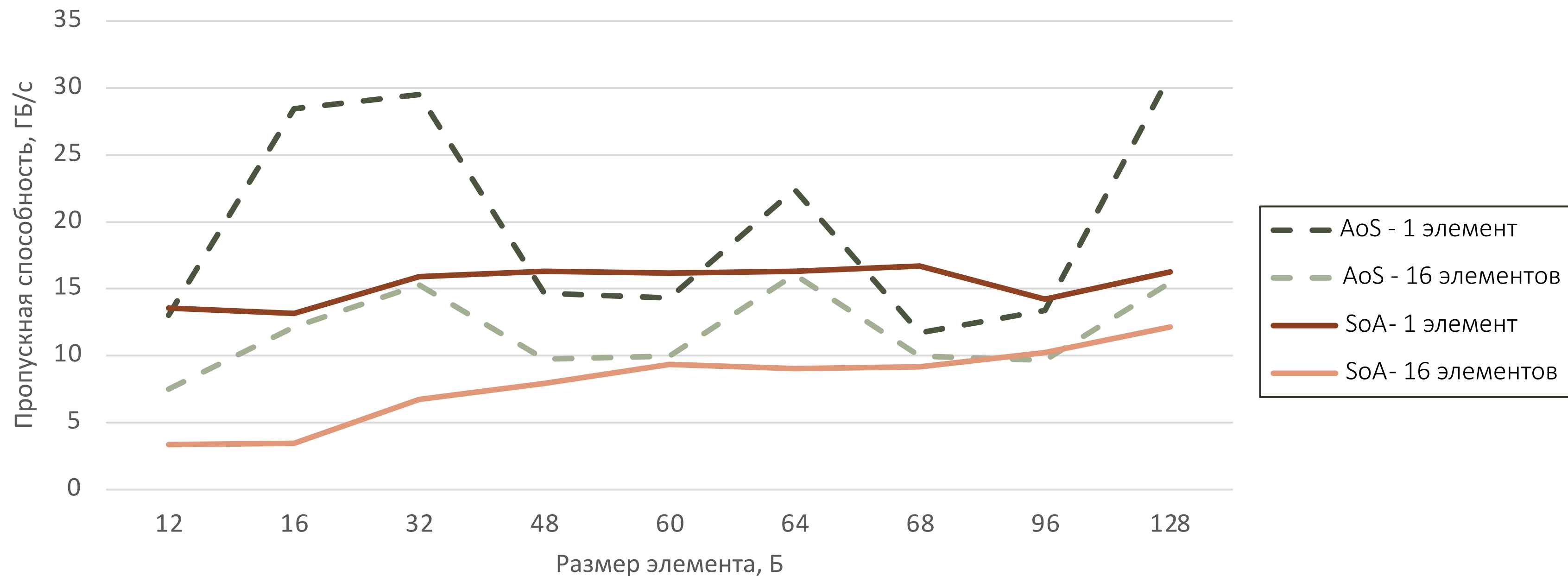
- SoA упаковывает данные без «дыр», поэтому не загрязнет кэш и быстрее в целом
- При отсутствии «лишних» данных AoS эффективнее (точка 12 Б)
- Производительность AoS заметно зависит от размера контейнера

Бенчмарк 2: работа со всем элементом



- SoA стабилен по ГБ/с
- AoS выигрывает от упаковки структур в линии кэша (32Б, 64Б, 128Б)

Бенчмарк 3:
работа с каждым 16-ым элементом (1 МБ)



- SoA проигрывает при «разреженном» доступе
- AoS упаковывает данные плотнее => производительность больше

Выводы

- Модель на 3 параметрах не позволяет отдать предпочтение AoS или SoA.
  - Для реальных задач нужно учесть все параметры!
- Надёжный способ: сравнение в «полевых» условиях
- «Наивная» замена AoS на SoA-код затратна и небезопасна
- Замкнутый круг
  - Для измерения Return of Investment нужен Investment!

Взаимозаменяемые AoS и SoA

- Благодаря STL, сравнивать списки и массивы просто
  - 90% работы по замене выглядит так:

```
- using Container = std::list<Entry>;
+ using Container = std::vector<Entry>;
```

- Можно ли так же легко исследовать AoS и SoA контейнеры?
  - Да, если мы абстрагируем алгоритмы от расположения данных
  - Аналогия с `list/vector`: взаимозаменяемые интерфейсы

```
- using Container = AoSVector<Entry>;
+ using Container = SoAVector<Entry>;
```

Задачи взаимозаменяемости

1. Вывести класс SoA из типов AoS:

```
struct T { T0 t0; T1 t1; ... };

Container<T>; => std::tuple<Container<T0>, Container<T1>...>;
```

2. Создать интерфейс доступа к полям:

```
storage[i].t0 = value;

value = storage[i].t1;
```

3. Создать интерфейс работы с целыми структурами

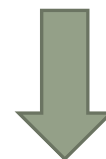
```
storage[i] = value;

value = storage[i];
```

Задача 1: Вывод SoA из AoS

1. Определить количество полей
   - Определить соответствие `class T => size_t N`
2. Изобрести ассоциативный контейнер типов
   - Хранить соответствие `(class T, size_t I) => class U`
3. Заполнить его типами полей
   - `Map<Tag<T, 0>>::type => T0`
   - `Map<Tag<T, 1>>::type => T1`
4. Свернуть поля в список типов/кортеж контейнеров

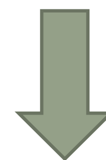
❖ Просто для языков с рефлексией
   - … но C++ не из таких!
   - Выручает Type Loophole

```
struct T {
    T0 t0;
    T1 t1;
    …
};
```



Type Loophole

```
type_list<T0, T1…>;
```



```
std::tuple<Container<T0>,
           Container<T1>…
          >;
```

Долго о Type Loophole

- *Alexandr Poltavsky:* The C++ Type Loophole
https://alexpolt.github.io/type-loophole.html
- *Antony Polukhin:* Boost.PFR (ex. Magic-get)
https://www.boost.org/doc/libs/1_77_0/doc/html/boost_pfr.html
- *Richard Smith:* Defect 2118. Stateful metaprogramming via friend injection
http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#2118

- *Антон Кватковский:* Type loopholes in C++: Убербаг уровня стандарта
https://www.youtube.com/watch?v=e41KdZ3RTJo

Коротко о Type Loophole

- # полей даёт {}-инициализация
  - `A{Cast<A, 0>(), Cast<A, 1>(), Cast<A, 2>()};`
    => не компилируется
  - `A{Cast<A, 0>(), Cast<A, 1>()}`
    => компилируется
    => количество полей равно 2
- Ассоциативный контейнер типов
  - Инъекция шаблонами friend-функций в глобальное пространство имён
  - Ядро Type Loophole (и «недоработка» C++)
- Контейнер заполняется компилятором при успешной подстановке типов в Cast

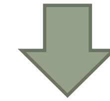
```
template<class T, class I>
struct Tag {
    friend constexpr auto get(Tag);
};

template<class Key, class Value>
struct Map {
    friend constexpr auto get(Key) {
        return Value{};
    }
};

template<class T, size_t I>
struct Cast {
   template<
        class U,
        size_t = sizeof(Map<Tag<T, I>, U>)
   >
   operator U();
};

struct A { int a; char b; };
```

Задача 2: доступ к полям

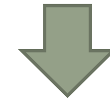
- **`operator.`** нельзя переопределить
- Есть **`operator->*`**
  - Похожий и перегружаемый
  - Тип правой части – pointer-to-member
- Вводим Proxy – обёртку над указателем на контейнер и индекс
  - AoS-версия тривиальна
  - SoA-версия «меняет местами» j и i

```
R x = storage[i].field
```

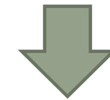
(строка зачёркнута)



```
R x = storage[i]->*(&T::field);
```



```
Proxy SoA::operator[](size_t i);
R Proxy::operator->*(R T::* j);
```



```
storage.tup.get<mem_to_idx(j)>()[i];
```

- Как определить mem_to_idx, используя скудную рефлексию C++?

Преобразование pointer-to-member

- C++ и TypeLoophole не приводят pointer-to-member к индексу поля
- Доработка TypeLoophole даёт `sizeof` первых X полей:
  - Это смещение X-го поля относительно начала структуры
  - Использование `offsetof` для pointer-to-member даст тот же результат
- Линейным поиском подберём нужное значение X
- Ограничение: в структуре не должно быть padding bytes

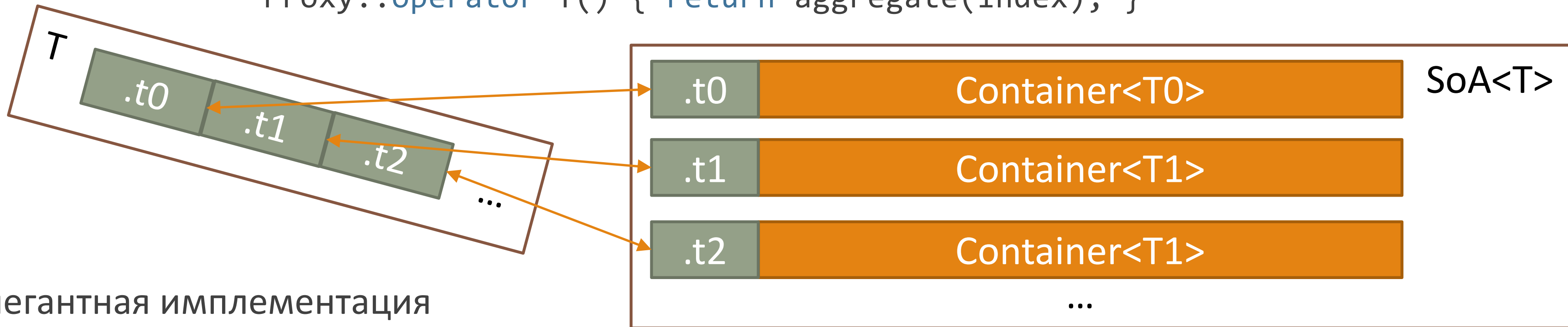
Задача 3: обращение к структурам в целом

- Для размещения объекта в контейнере его необходимо разложить на части

```
void Proxy::operator=(const T& src) { dissipate(index, src); }
```

- Обратно, при извлечении производится агрегация

```
Proxy::operator T() { return aggregate(index); }
```



- Элегантная имплементация через Boost.PFR и оператор «,»

```
template<typename T, size_t ... N>
void dissipate(size_t i, T&& src, std::index_sequence<N...>) {
    ((void)(std::get<N>(tup)[i] = std::forward<T>(boost::pfr::get<N>(src)), ...));
}
```

Функции и методы

- Неприменимы к разложенному объекту
- Read-Modify-Write => агрегация-функция-разложение
  - Реализована через `operator->*(R (T::* fun)(Args ...))`
  - Может оказаться небыстрой операцией
- Можно ли оставить Read для `const`-методов?
  - Нет, потому что бывают `mutable` поля
  - Trait’а на отсутствие таких полей в C++ пока нет

Деструкторы

- Для C++ разложенный объект уничтожен:
  - Деструктор вызывается в момент разложения
  - При удалении SoA контейнера деструктор не вызывается
- Деструктор по умолчанию не вызывает проблем
- Нельзя построить trait:
  - `struct Point { int x, y; std::unique_ptr<int> z; };`
  - `is_trivially_destructible<Point>::value == false;`
  - Однако, нет проблем разложить поля в SoA
- Те же проблемы с копированием и перемещением

https://github.com/pavelkryukov/AoAoAoTT

- Имплементация аналогов `std::vector` и `std::array` по вышеизложенным принципам
- Что не поддерживается:
  - Non-POD, пустые, или наследованные структуры – *ограничение Type Loophole*
  - Структуры с C-массивами – *`std::array` работает, используйте его*
  - Упакованные структуры – *ограничение Boost.PFR, возможно необязательное*
  - Структуры с padding bytes – *если вы задумались о SoA, паддинга уже быть не должно*
  - Структуры с булевыми полями для вектора – *да, виноват `std::vector<bool>`*
- Код распространяется по лицензии MIT
- Ваш вклад приветствуется

Заключение

- SoA выглядит необходимым контейнером для C++:
  - C++ заточен на производительность
  - Организация данных в SoA ускоряет многие приложения
  - Движение в сторону DoD (data-oriented design)
- Желание не осуществляется в полном объёме:
  - «Встраивание» SoA пойдёт вразрез принципу волатильности синтаксиса языка
  - Слабая рефлексия C++ делает STL-подобное решение сложным и ограниченным
- В любом случае C++ требует улучшений

Дальнейшие шаги: улучшение C++

- https://github.com/cpp-ru/ideas/issues/482:
  - преобразование pointer-to-member в индекс поля и обратно компилятором
  - «легализация» Boost.PFR как следствие
  - на данный момент отсутствует в Reflection TS
- https://github.com/cpp-ru/ideas/issues/484:
  - Type trait на деструкторы/копирование/перемещение по умолчанию
  - покрывается Reflection TS
- Trait’ы is_mutable или has_mutable
  - предлагаются в некоторых расширениях Reflection TS

Спасибо за внимание!

ВЗАИМОЗАМЕНЯЕМЫЕ AOS И SOA КОНТЕЙНЕРЫ
ПАВЕЛ КРЮКОВ
C++ RUSSIA 2021

Запасные слайды

Другие имплементации SoA

- CoPA Cabana: https://github.com/ECP-copa/Cabana
- SoAvsAoS: https://github.com/crosetto/SoAvsAoS
- EASTL: https://github.com/electronicarts/EASTL/blob/master/test/source/TestTupleVector.cpp
- Работают с «псевдо-структурами» (кортеж или массив)
  - теряются имена полей, что грозит ошибками в разработке
- Python: Pandas

Векторизация

- Аналогично разделяется на
  - горизонтальную, AoS-подобную
  - вертикальную, SoA-подобную – рекомендуемую обычно
- Качественный анализ совпадает с AoS/SoA
- Через gather/scatter доступны «диагональные» системы