Rule of Zero. STL. Smart Pointers.

STL контейнери и алгоритми. Smart pointers. Rule of Zero. Примери.

д-р Тодор Цонков
todort@uni-sofia.bg

Какво е STL?

STL (Standard Template Library) е част от стандартната библиотека на C++ и предоставя: контейнери, итератори, Алгоритми и Функционални обекти.

Ключовата идея е, че алгоритмите не знаят нищо за конкретния контейнер.

Основната философия е: разделяне на структура от данни и алгоритъм чрез абстракция на итератори.

STL контейнерите и помощните класове:

1)управляват динамична памет
2)спазват RAII
3)имат правилни copy/move семантики
4)освобождават ресурсите автоматично
5)няма нужда от delete и new

STL в детайли

Примери за STL типове:

std::string
std::vector
std::array
std::map
std::unique_ptr
std::shared_ptr

std::array има фиксиран размер (compile-time), а std::vector има динамичен размер (run-time).

std::vector – Непрекъсната памет
Предлага следните характеристики: 1)достъп по индекс без допълнителна сложност и време

2) Добавяне с константна сложност в края

3)Реалокация при изчерпване на капацитета

Вътрешно пази:

T* begin_; T* end_; T* capacity_end_;

Има размер, който е текущият му размер и капацитет - максималния възможен

std::vector

```
struct Student {
    std::string name;
    double grade;
};

int main() {
    std::vector<Student> students;
    students.push_back({"Ivan", 5.50});
    students.push_back({"Maria", 6.00});
    students.push_back({"Georgi", 4.80});

    for (const auto& s : students) {
        std::cout << s.name << " -> " <<
s.grade << '\n';
    }
}
```

Кога да използваме std::vector?

✔ Default избор

✔ Когато е нужна бърза итерация

✔ Когато имаме предимно добавяне в края

Кога НЕ е подходящ?

✖ Чести вмъквания в началото

✖ Чести изтривания в средата

✖ Строги изисквания за стабилни адреси

std::vector

Достъп до елемент:
v[i]; // Без проверка
v.at(i); // С проверка и грешка
v.front(); v.back();

При изчерпване на капацитета:
Алокира се нов по-голям блок памет.
Елементите се: преместват (move), ако е възможно
Копират, ако няма move constructor и старият блок се освобождава

Елементите се добавят с push_back или се променят по индекс както в стандарния масив, стига индекс < capacity.

reserve(n)

Увеличава capacity без да променя size : v.reserve(100);

resize(n) - променя size

Конструира или унищожава елементи

v.resize(10);

std::array

std::array е контейнер с фиксиран размер, дефиниран в <array>, част от стандартната библиотека на C++.

Той представлява wrapper над C-style масив, но със STL интерфейс, интеграция с алгоритмите, value semantics.

Пример: std::array<int, 5> a;

което е различно от: std::array<int, 10>

Различно е от int array[5];

Проблеми при стандартния масив: Няма .size(), няма .at(), не може да се присвоява

При подаване във функция се ползва като указател

std::array<int,5> a1{1,2,3,4,5};
std::array<int,5> a2 = a1; // валидно копиране (value based semantics)

Методи: a.size();a.empty();

a.front();a.back();

a.data(); a.fill(value); a.swap(other);

STL - продължение

std::array - размерът се знае по време на компилация, паметта се намира в стека,

няма реалокация, няма push_back.

std::vector - размерът се знае по време на изпълнение, heap based, поддържа реалокация и push_back

std::list е последователен контейнер, реализиран като двусвързан списък (doubly linked list) в стандартната библиотека на C++.

Той се различава фундаментално от std::vector по паметен модел, сложност и performance характеристики.

```
#include <list>

int main() {
    std::list<int> l{1,2,3};
    l.push_front(0);
    l.push_back(4);
    for (int x : l) { std::print("{} ", x); }
}
```

std::list

std::list е последователен контейнер, реализиран като двусвързан списък (doubly linked list) в стандартната библиотека на C++.

Той се различава фундаментално от std::vector по паметен модел, сложност и performance характеристики.

Основни свойства:
Няма произволен достъп към всеки елемент в списъка, а трябва да се обхожда целият списък.

Няма operator[] list[2]; // Грешка

Може да се добавят елементи само отпред и отзад в списъка.

```
#include <list>
#include <print>

int main() {
  std::list<int> l{1,2,3};

  l.push_front(0);
  l.push_back(4);

  for (int x : l)
     std::print("{} ", x);
}
```

std::list

Обхождане на лист:

```
auto it = l.begin(); std::advance(it, 2);
```

Прехвърляне на елементи от списъци става чрез указатели:

```
std::list<int> l1{1,2,3};

std::list<int> l2{4,5};

l1.splice(l1.begin(), l2);
```

Триенето и добавянето на елементи става с константна сложност.

Да се използва когато имаме честни триенета в средата или мърджвания

Паметта при list е разпокъсана, а при vector е непрекъсната. Когато ни трябва произволен достъп до елементите на структурата да се ползва std::vector, в противен случай std::list.

Реализация като идея:

```
struct Node {
  Node* prev;
  Node* next;
  T value;
};
```

Какво е iterator?

Iterator е поведенчески (behavioral) design pattern, който предоставя начин за последователен достъп до елементите на агрегатен обект, без да се разкрива вътрешното му представяне.

С други думи:

- Контейнерът пази данните.
- Итераторът осигурява интерфейс за обхождане.
- Алгоритмите работят чрез итератори.

Ако имаме:

```
for (int i = 0; i < vector_size; ++i)
  process(v[i]);
```

Това работи само за структури с random access.

Но какво ако структурата е:
linked list, tree, hash table?

Iterator pattern абстрахира начина на обхождане.

Итератор в STL

Всеки контейнер дефинира begin() и end(). Те връщат итератори.
Алгоритмите приемат диапазон [first, last).

```
#include <vector>
#include <print>

int main() {
  std::vector<int> v{1,2,3};

  for (auto it = v.begin(); it != v.end(); ++it)
     std::print("{} ", *it);
}
```

STL разширява класическия pattern с категории:

Input еднопосочно четене
Output еднопосочно записване
Forward многократно преминаване
Bidirectional ++ и --
Random Access +, -, индекс

Предимства на итераторите:
Generic programming, Decoupling между алгоритъм и контейнер
Compile-time оптимизация, Zero-cost abstraction

Итератори в STL - продължение

В някои случаи:

std::vector<int>::iterator

е просто: int*, но при по-сложни структури - примерно set или list може да е по-сложно. Пример:

```
struct MyIterator {
  int* ptr;
  int& operator*() { return *ptr; }
  MyIterator& operator++() { ++ptr; return
*this; }
  bool operator!=(const MyIterator&
other) const {
      return ptr != other.ptr;
  }
```

В модерния C++ (C++20):

Iterator pattern е разширен с Ranges
Те са еволюция на STL алгоритмите, въведени в C++20, които работят върху диапазони (ranges), а не върху двойки итератори.

Те са част от стандартната библиотека на C++ и се намират в namespace std::ranges.

Пример:
std::ranges::sort(v); Подобрява:

std::sort(v.begin(), v.end());

Алгоритми в STL

Алгоритмите в STL (Standard Template Library) са функции, които работят върху диапазони от елементи чрез итератори.
Те са част от стандартната библиотека на C++ и основно се намират в:
<algorithm> <numeric> <ranges> (от C++20)
Алгоритмите:
Не знаят нищо за контейнера
Работят върху [first, last) диапазон
Използват iterator abstraction
Пример:

Алгоритъмът не знае дали v е:
std::vector, std::array, std::deque
Разделят се на различни видове:

Non-modifying алгоритми - примери:
std::find, std::count, std::all_of
std::any_of, std::none_of,
std::accumulate, std::inner_product

```
auto it = std::find(v.begin(), v.end(), 42);
int sum = std::accumulate(v.begin(),
v.end(), 0);
```

Алгоритми в STL - продължение

Modifying алгоритми:

std::copy, std::transform, std::fill,
std::remove

```
std::transform(v.begin(), v.end(),
        v.begin(),
        [](int x){ return x * 2; });
```

std::sort, std::stable_sort,
std::partial_sort, std::nth_element

```
std::partition(v.begin(), v.end(),
        [](int x){ return x % 2 == 0; });
```

STL алгоритмите следват принципа:

Разделяне на данните от операциите.

Контейнерът съхранява.
Алгоритъмът обработва.

```
for (auto& x : v)
  if (x == 5) ...
```

vs std::find(v.begin(), v.end(), 5);

По-четим код, по-малко грешки, по-добра оптимизация, Ясна семантика

Smart pointers

```
int* p = new int(42);
delete p;
```

Проблеми:

❌ Memory leaks (неосвободена памет)

❌ Double delete (триене на вече изтрит pointer)

❌ Dangling pointers (използване на)

❌ Липса на ясно ownership правило (кой е отговорен за живота на pointer-а?)

Решение:

Smart pointers

Smart pointer е клас, който:
енкапсулира суров указател, управлява неговия живот и дефинира ясна ownership семантика.

Те биват: std::unique_ptr, std::shared_ptr, std::weak_ptr

Всеки има различен ownership модел.

std::unique_ptr

```
#include <memory>
std::vector<std::unique_ptr<MyClass>>
v;
```

std::unique_ptr

std::unique_ptr е smart pointer в стандартната библиотека на C++, който реализира ексклузивна (exclusive) собственост върху динамичен ресурс.

В даден момент съществува точно един собственик на ресурса.
Това означава:
❌ Не може да се копира
✔ Може да се премества (move)
✔ Освобождава ресурса автоматично

```
{
  std::unique_ptr<int> p =
std::make_unique<int>(42);
} // автоматично delete

std::unique_ptr<int> p1 =
std::make_unique<int>(5);

// std::unique_ptr<int> p2 = p1; ❌
грешка
std::unique_ptr<int> p2 = std::move(p1);
```

След move:

p1 == nullptr

p2 притежава ресурса

std::unique_ptr

```
class unique_ptr {
  T* ptr;
public:
  ~unique_ptr() {
     delete ptr;
  }

  unique_ptr(unique_ptr&& other)
noexcept
     : ptr(other.ptr)
  {
     other.ptr = nullptr;
  }
};
```

```
auto deleter = [](FILE* f) {
  if (f) fclose(f);
};

std::unique_ptr<FILE, decltype(deleter)>
file(
  fopen("data.txt", "r"),
  deleter
);
std::unique_ptr<int[]> arr =
std::make_unique<int[]>(10);

void foo(std::unique_ptr<int> p);
```

std::shared_ptr

std::shared_ptr<T> е smart pointer от STL дефиниран в <memory>, който реализира споделена собственост (shared ownership) върху динамично заделен обект.

Обектът, към който сочи shared_ptr, се унищожава автоматично, когато последният притежател (последното копие на shared_ptr) освободи ресурса.
Пример: Student се използва от няколко структури
School, Course, External system

2. Основна идея: Reference Counting
std::shared_ptr използва механизъм за reference counting:
Всеки път, когато shared_ptr се копира → броячът се увеличава.
Когато shared_ptr се унищожи или ресетне → броячът се намалява.
Когато броячът стане 0 → обектът се унищожава.

std::shared_ptr

❌ Непрепоръчителен начин

```
std::shared_ptr<int> p(new int(42));
```

✅ Препоръчителен начин

```
auto p = std::make_shared<int>(42);
```

Пример:

```
auto p1 = std::make_shared<int>(10);
auto p2 = p1;   // споделят собственост
use_count() == 2
```

```
struct Resource {
    Resource() { std::print("Constructed\n"); }
    ~Resource() { std::print("Destroyed\n"); }
};
```

```
int main() {
auto p1 = std::make_shared<Resource>();
    std::print("Count: {}\n", p1.use_count());
    {
        auto p2 = p1;
        std::print("Count: {}\n", p1.use_count());
    }
    std::print("Count: {}\n", p1.use_count());
}
```

```
Constructed
Count: 1
Count: 2
Count: 1
Destroyed
```

unique_ptr vs shared_ptr

✔ Автоматично управление на ресурс
✔ Без memory leak (ако няма цикли)
✔ Подходящ за споделена собственост

❗ По-бавен от unique_ptr
❗ Допълнителна памет (control block)
❗ Риск от циклични зависимости
❗ Неясна собственост (кой притежава ресурса?)

Кога да използваме shared_ptr
✔ Когато няколко обекта трябва да споделят ресурс
✔ При graph структури

```
struct A {
    std::shared_ptr<B> b;
};

struct B {
    std::shared_ptr<A> a;
};
```

A и B никога няма да бъдат унищожени.
Reference count никога не става 0
Memory leak

Решение: std::weak_ptr

std::weak_ptr

std::weak_ptr<T> е non-owning smart pointer, който наблюдава обект, управляван от std::shared_ptr<T>, без да участва в неговата собственост.

Той не увеличава strong reference count.
Обектите не се унищожават

```
struct B;
struct A {
    std::shared_ptr<B> b;
};
struct B {
    std::weak_ptr<A> a;
};
```

```
struct A {
    std::shared_ptr<B> b;
};

struct B {
    std::shared_ptr<A> a;
};
```

A и B никога няма да бъдат унищожени.
Reference count никога не става 0
Memory leak

Решение: std::weak_ptr

std::weak_ptr

```
auto sp = std::make_shared<int>(42);
std::weak_ptr<int> wp = sp;
strong count = 1
weak count = 1
if (auto locked = wp.lock()) {
    // получаваме shared_ptr
}
```

weak_ptr моделира:
„Имам достъп, но не притежавам.“

Използва се:
Graph структури, Observer pattern, Cache механизми

```
#include <memory>
#include <print>
int main() {
std::weak_ptr<int> wp;

{
    auto sp = std::make_shared<int>(10);
    wp = sp;
    std::print("Expired? {}\n", wp.expired());
}

    std::print("Expired? {}\n", wp.expired());
}
```

observer design pattern

```
class Observer {
public:
void notify() {}
};

class Subject {
std::vector<std::weak_ptr<Observer>> observers;
public:
      void notify_all() {
            for (auto& w : observers) {
                  if (auto s = w.lock())
                  s->notify();
      }
   }
};
```

В модерния C++:

unique_ptr → един собственик

shared_ptr → много собственици

weak_ptr → наблюдател

Това формира завършен ownership модел.

Безопасен non-owning указател

Решение на циклични зависимости

Механизъм за наблюдение на lifetime

Добри практики

❗ Да се почва почти винаги с unique_ptr
❗ Да се ползва shared_ptr само при нужда
❗ Винаги да се прекъсваат циклите с weak_ptr

✔ Да се ползва make_unique / make_shared (exception safety)
✔ Да се мисли за ownership още при дизайна

std::observer_ptr е официалният стандартен не-притежаващ (non-owning) указателен wrapper. C++ 20
Той:
НЕ притежава паметта
НЕ освобождава (delete)
НЕ управлява жизнения цикъл (lifetime)

Rule Of Zero

Rule of Zero е идиом в C++, който гласи:

„Ако един клас не управлява ресурси директно (например динамична памет, файлови дескриптори, mutex-и), той не трябва да дефинира десктруктор, конструктор за копиране, move конструктор, copy/move assignment оператори. Вместо това трябва да разчита на RAII обекти и стандартни контейнери, които вече безопасно управляват ресурсите.“

При използване на класове, които правилно управляват ресурси (RAII) — std::string, std::vector, std::unique_ptr — няма нужда от специални функции: компилатора ги генерира → по-малко грешки, по-чист код.

Дизайн: предпочитаме Rule of Zero
Когато можем: използваме std::string и std::vector<double> за name и grades.
Минимален код, всички специални функции генерирани безопасно.

Rule Of Zero пример

```
class Student {
private:
    std::string name_;
    int faculty_number_;
    std::vector<double> grades_;

public:
    Student(std::string name, int fn,
std::vector<double> grades)
        : name_(std::move(name)),
          faculty_number_(fn),
          grades_(std::move(grades)) {}

    /
```

```
// Rule of Zero → нищо друго не пишем
    auto operator<=>(const Student&) const =
default;
    double average_grade() const {
        if (grades_.empty()) return 0.0;
        return std::accumulate(grades_.begin(),
grades_.end(), 0.0)
                / grades_.size();
    }
    void print() const {
        std::print("Student: {}\n", name_);
        std::print("FN: {}\n", faculty_number_);
        std::print("Average grade: {:.2f}\n",
average_grade());
    } };
```

Rule Of Zero Обобщение

Премахване на специални функции: компилаторът автоматично генерира правилния деструктор, copy/move конструктори и assignment оператори, ако няма нужда от ръчно управление на ресурси.

RAII и smart pointers: Rule of Zero се постига чрез smart pointers (std::unique_ptr, std::shared_ptr) и стандартни контейнери (std::vector, std::string).

Сравнение с Rule of Five:

Rule of Five: трябва да имплементираме 5 функции при управление на ресурс.

Rule of Zero: премахваме всички, когато е възможно.

Предимства:
По-малко код → по-малко грешки
Автоматично управление на ресурси
По-добра съвместимост с move семантика
Кодът става по-модулен и безопасен

Пример с класа Student

```java
import java.util.List;

public class Student implements Comparable<Student> {
    private final String name;
    private final int facultyNumber;
    private final List<Double> grades;

    // Конструктор
    public Student(String name, int facultyNumber, List<Double> grades) {
        this.name = name;
        this.facultyNumber = facultyNumber;
        this.grades = List.copyOf(grades); // иммутабилна версия на листа
    }

    // Comparable → "default comparison" по име, след това по фак. номер
    @Override
    public int compareTo(Student other) {
        int cmp = this.name.compareTo(other.name);
        if (cmp != 0) return cmp;
        return Integer.compare(this.facultyNumber, other.facultyNumber);
    }

    // Изчисляване на среден успех
    public double averageGrade() {
        if (grades.isEmpty()) return 0.0;
        return grades.stream()
                     .mapToDouble(Double::doubleValue)
                     .average()
                     .orElse(0.0);
    }

    // Печат
    public void print() {
        System.out.printf("Student: %s%n", name);
        System.out.printf("FN: %d%n", facultyNumber);
        System.out.printf("Average grade: %.2f%n", averageGrade());
    }

    // Getters, ако са нужни
    public String getName() { return name; }
    public int getFacultyNumber() { return facultyNumber; }
    public List<Double> getGrades() { return grades; }
}
```

Въпроси?

Благодаря за вниманието!