# Rule Of Zero. STL. Smart pointers.

титуляр на курса: д-р Тодор Цонков (ttsonkov@gmail.com)


# От какво ще се състои настоящата лекция?

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

## STL контейнери

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

## Пример

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).

## Пример за 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';
    }
}
```

## Пример за std::array

```
//Знае се броя по време на компилация!
#include <array>
#include <iostream>

int main() {
    std::array<double, 3> position = {1.5,
2.0, -0.5};

    std::cout << "x=" << position[0]
              << " y=" << position[1]
              << " z=" << position[2] <<
'\n';
}
```

## std::array

Тип контейнер: Фиксиран масив
Размер: Фиксиран (compile-time)
Памет: Вътре в обекта (няма heap)
Динамично разширяване: Не
Случаен достъп []: Да – O(1)
Вмъкване в края: Не
Вмъкване в средата: Не
Изтриване в средата: Не
Cache-friendly: Да (много)
Допълнителна памет: Не
Copy семантика: Пълно копие
Move семантика: Пълно преместване
Rule of Zero: Да
Кога се използва: Малък, фиксиран размер

## std::vector

Тип контейнер: Динамичен масив
Размер: Динамичен (run-time)
Памет: Heap (непрекъсната)
Динамично разширяване: Да
Случаен достъп []: Да – O(1)
Вмъкване в края: Да – amort. O(1)
Вмъкване в средата: O(n)
Изтриване в средата: O(n)
Cache-friendly: Да (много)
Допълнителна памет: Малко (capacity)
Copy семантика: Пълно копие
Move семантика: Евтино
Rule of Zero: Да

Кога се използва: Най-често използваният контейнер

# Пример

## Пример кога да ползваме std::list

Симулация на плейлист, където:

1)често трябва да се махат  и добавят песни в средата

2)пазят се валидни итератори

3)няма нужда от бърз достъп по индекс

✔ Защо list?

1)O(1) insert/erase навсякъде

2)Итераторите не се инвалидират

```cpp
#include <list>

int main() {
    std::list<int> playlist = {1, 2, 3, 4};

    auto it = playlist.begin();
    std::advance(it, 2); // позиция преди 3

    playlist.insert(it, 99);  // добавяме в средата
    playlist.remove(2);       // махаме песен

    for (int song : playlist) {
        std::cout << song << ' ';
    }
}
```

# Проблем - Масиви от указатели

```
MyClass* arr[3];
arr[0] = new MyClass(1);
arr[1] = new MyClass(2);
arr[2] = new MyClass(3);

Move семантиката не решава ownership при raw
Move на указател = копиране на адрес.

Няма информация кой е отговорен да извика delete.

MyClass* p = new MyClass(42);
MyClass* q = p; // копие на указател

// кой трябва да delete-не p/q?
delete p;
delete q;
```

## Решение на проблема с ресурсите

Smart pointers

**Пример**:

```
std::vector<std::unique_ptr<MyClass>> v;
v.push_back(std::make_unique<MyClass>(1));

std::vector<std::unique_ptr<MyClass>> v2 =
std::move(v);
// ownership е коректно прехвърлен
```

## Обобщение

Масиви от raw pointer + move = проблеми!

Move работи добре с типове, които моделират ownership

В модерния C++: raw pointers ≠ ownership

## Какво са smart pointers?

Проблемът с new / delete

Ръчно управление на памет → лесно води до:

1)memory leaks

2)double delete

3)dangling pointers

Особено проблемно при exceptions и сложна логика

Smart pointer = обект, който управлява lifetime-а на друг обект и автоматично освобождава ресурса в деструктора си

Част от STL (<memory>)

## Видове

Основни видове (C++11):

std::unique_ptr, std::shared_ptr, std::weak_ptr

std::unique_ptr (единствен собственик)

Само един обект притежава ресурса

Не може да се копира, само да се премества

Най-ефективният smart pointer и избор по подразбиране

Кога се използва

Когато School е единственият собственик на Student

Ясна архитектура и ownership

# Пример за unique_ptr

Пример

```
class Student {
public:
    std::string name;
    Student(const std::string& n) : name(n) {}
};
class School {
    std::vector<std::unique_ptr<Student>> students;
public:
    void addStudent(std::unique_ptr<Student> s) {
        students.push_back(std::move(s));
    }
};
```

## Видове

```
int main() {

    School school;
school.addStudent(std::make_unique<Student>("
Ivan"));

}
```

Имплементира се като:
Забранено копиране, позволено преместване

Copy constructor и copy assignment са изтрити

=delete

Move constructor и move assignment са позволени

## shared_ptr

Един обект може да има няколко собственика

Използва reference counting

Обектът се изтрива, когато броячът стане 0

Кога се използва?

Когато Student се използва от няколко структури

School, Course, External system (например дневник)

## Пример

```
class School {
    std::vector<std::shared_ptr<Student>> students;
public:
    void addStudent(const std::shared_ptr<Student>&
s) {
        students.push_back(s);
    }
};
int main() {
    auto st = std::make_shared<Student>("Maria");
    School school;
    school.addStudent(st);
}
```

## Проблем със shared_ptr

School държи shared_ptr<Student>

Student държи shared_ptr<School>

```
class Student {

    std::shared_ptr<School> school; // ❌

};
```

### Резултат

Reference count никога не става 0

Memory leak

Обектите не се унищожават

## Решение

std::weak_ptr

weak_ptr не увеличава reference count

Използва се за:

back-reference,observer, избягване на цикли

```
class School; // forward declaration

class Student {
    std::weak_ptr<School> school; // ✔
public:
    void setSchool(const std::shared_ptr<School>& s) {
        school = s;
    }
};
```

# 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)

## Какво е std::string?

В C++ std::string е стандартен клас за работа с низове (strings), който се намира в стандартната библиотека (<string>). Той представлява динамично разширяем масив от символи, който опростява работата с текст спрямо стария стил с char[].

Няколко ключови точки за std::string:

Ключови функции:
Дължина: s.size() или s.length()
Достъп до символ: s[i] или s.at(i),
Конкатенация s1 + s2 или s1 += s2,
Сравнение: s1 == s2, s1 < s2
Изтриване: s.clear(),
Подниз: s.substr(pos, len)
Търсене: s.find("abc")

## Пример

```cpp
#include <iostream>
#include <string>

int main() {
    // 1 Създаване на низове
    std::string name = "Todor";
    std::string greeting = "Hello, " + name + "!";
    std::string stars(5, '*'); // "*****"

    std::cout << greeting << "\n"; // Hello, Todor!
    std::cout << "Stars: " << stars << "\n";

    // 2 Достъп до символи
    greeting[0] = 'h'; // промяна на първия символ
    std::cout << "Modified greeting: " << greeting << "\n";

    // 3 Дължина на низа
    std::cout << "Length of greeting: " << greeting.size() << "\n";

    // 4 Поднизове
    std::string firstWord = greeting.substr(0, 5);
    std::cout << "First word: " << firstWord << "\n";

    // 5 Търсене
    size_t pos = greeting.find("Todor");
    if (pos != std::string::npos) {
        std::cout << "'Todor' found at index " << pos << "\n";
    }

    // 6 Конкатенация
    greeting += " How are you?";
    std::cout << "Extended greeting: " << greeting << "\n";

    // 7 Сравнение
    std::string a = "abc";
    std::string b = "abd";
    if (a < b) {
        std::cout << a << " < " << b << "\n";
    }

    // 8 Почистване
    greeting.clear();
    if (greeting.empty()) {
        std::cout << "Greeting string is now empty\n";
    }

    return 0;
}
```

## Rule of Zero

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.

## Пример

```cpp
#include <string>
#include <vector>
#include <iostream>

class Student {
    std::string name;        // manages its own memory
    std::vector<int> grades; // manages its own memory

public:
    Student(std::string n, std::vector<int> g)
        : name(std::move(n)), grades(std::move(g)) {}

    double average() const {
        if (grades.empty()) return 0.0;
        int sum = 0;
        for (int g : grades) sum += g;
        return static_cast<double>(sum) / grades.size();
    }

    void print() const {
        std::cout << name << " | avg = " << average() <<
'\n'}
};
```

# Пример за клас School на Java

```java
import java.util.ArrayList;
import java.util.List;

public class School {
    private List<Student> students = new ArrayList<>
();
    // Добавяне чрез референция (НЕ копира обекта!)
    public void addStudent(Student s) {
        students.add(s);
    }

    // Явно копиране (по желание)
    public void addStudentCopy(Student s) {
        students.add(new Student(s));
    }

    public void print() {
        for (Student s : students)
            s.print();
    }
}
```

# Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com Глава 22