

VectorV1: destruktor, move i jeden właściciel

od wycieku i płytkiej kopii do obserwowalnego RAI

Karta	K03 / 16	Czas	45 minut
Platforma	Hazard3 / RV32I	Język	freestanding C++17
Allocator	FreeRTOS heap_4	Kernel	V11.3.0, bez zmian
Wersja	v00.01	UUID	de1a2818-...
		karty	

Punkt startowy: celowo niebezpieczny Vector V0

```
class VectorV0 {
public:
    explicit VectorV0(size_t n)
        : elements_{n ? new double[n] : nullptr}, size_{n} {}
private:
    double *elements_;
    size_t size_;
}; // brak destruktor; kopiowanie byłoby płytke
```

Inwariant K03. Każdy żywy bufor ma dokładnie jednego właściciela. Obiekt po przeniesieniu jest pusty: `data()==nullptr` i `size()==0`.

Najpierw diagnoza

Operacja na V0	Co może pójść źle?
wyjście z zakresu	
domyślna kopia wskaźnika	
dodanie destruktor bez za-	
kazu kopii	
przypisanie do obiektu,	
który już ma bufor	

Plan lekcji

Czas	Tryb	Dowód
0–5	V0	wyciek, płytka kopia, brak reguły właściciela
5–11	destruktor	<code>delete[]</code> prowadzi do <code>vPortFree()</code>
11–17	copy / move	copy usunięte; źródło po move jest puste
17–25	move assignment	release -> take -> clear source
25–38	Hazard3/GDB	ten sam adres A, zwolnienie B, heap wraca do baseline
38–45	ćwiczenie i wyjście	kolejność operacji oraz granica RAI

1 Co naprawdę posiada obiekt?

VectorV1 object: [elements_=A | size_=4] → heap_4: [A: 4 * double]

Obiekt ma dwa pola i może leżeć na stosie. Dane są osobnym blokiem w ucHeap. Destruktor zwalnia *bufor*, nie pamięć samego obiektu.

Pięć operacji właściciela

Operacja	K03	Powód
destruktor	implementowany	zwalnia jedyny posiadany bufor
copy constructor	= delete	plytka kopia stworzyłaby dwóch właścicieli
copy assignment	= delete	ten sam problem oraz możliwa utrata starego celu
move constructor	noexcept	przejmuje adres, czyści źródło
move assignment	noexcept	najpierw zwalnia stary bufor celu

Destruktor i stan pusty

```

~VectorV1() noexcept { release(); }

void release() noexcept {
    delete[] elements_;    // delete[] nullptr jest bezpieczne
    elements_ = nullptr;
    size_ = 0;
}

```

Move constructor: nowy obiekt nie ma starego zasobu

```

VectorV1(VectorV1&& other) noexcept
: elements_{other.elements_}, size_{other.size_} {
    other.elements_ = nullptr;
    other.size_ = 0;
}

```

Predykcja. Przed move: source.elements_=A. Po move:

new.elements_=_____ source.elements_=_____ liczba delete[] podczas samego move: _____

Wniosek: move przenosi prawo do późniejszego zwolnienia. Nie kopiuje bufora i nie alokuje drugiego.

2 Move assignment do zajętego celu

source -> A destination -> B $A \neq B$

Cel już posiada B. Wpisz numery 1–5 w jedynej bezpiecznej kolejności:

Kolejność	Operacja
_____	other.elements_ = nullptr; other.size_ = 0;
_____	elements_ = other.elements_;
_____	sprawdź this != &other
_____	release();
_____	size_ = other.size_;

Sprawdź cztery błędne warianty

1. Najpierw nadpisz elements_ adresem A. Co stało się z B?
2. Po przejściu A wykonaj release(). Który bufor zwolnisz?
3. Nie wyzeruj źródła. Ile obiektów uważa, że posiada A?
4. Pomiń kontrolę self-move. Co zrobi `x = move(x)`?

Odsłonięcie po predykcji

```
if (this != &other) {
    release();           // free B
    elements_ = other.elements_; // take A
    size_ = other.size_;
    other.elements_ = nullptr; // source no longer owns A
    other.size_ = 0;
}
```

Most C++ → FreeRTOS C

VectorV1 -> delete[] -> operator delete[] -> vPortFree

Most alokacji definiuje komplet: `new`, `new[]`, zwykłe i tablicowe `delete`, każde w wariancie `sized` oraz `unsized`. Zwykle `new` w tym profilu nie może zwrócić `nullptr`; błąd alokacji kończy się kontrolowanym fail-fast.

3 Hazard3/GDB: śledź A i B, nie nazwy zmiennych

```
make check
riscv64-unknown-elf-gdb build/task01_vector_raii/prog.elf
b vector_raii_debug_checkpoint
```

STOP	Stan	Obowiązkowa obserwacja
1	baseline	<code>g_initial_free</code> zapisane
2	source ma A	allocation count = 1; A leży w <code>ucHeap</code>
3	source ma A, destination ma B	dwa różne adresy; free spadło
4	assignment wykonany	pierwszy free to B; destination ma A; source puste
5	move construction	final owner ma nadal A; destination puste
6	final owner poza zakresem	drugi free to A; free wróciło do baseline
7	wynik	allocations=2, frees=2, live=0, pass=1

Minimalny zestaw poleceń

```
p g_last_checkpoint
p/x g_source_buffer_before_move
p/x g_destination_old_buffer
p/x g_destination_buffer_after_assignment
p/x g_final_buffer_after_construction
p g_cpp_allocation_count
p g_cpp_deallocation_count
p g_cpp_live_allocations
p g_initial_free
p g_after_two_owners_free
p g_after_scope_free
p g_minimum_ever_free
```

Relacje do wpisania

Warunek	Odczyt / dowód
<code>after_two_owners < initial</code>	_____
<code>minimum_ever <= after_two_owners</code>	_____
<code>after_scope == initial</code>	_____
adres po assignment = A	_____
adres po move construction = A	_____
kolejność freed addresses = B, A	_____

Wyjście — odpowiedz bez uruchamiania programu

1. Dlaczego samo dodanie destruktora do płytko kopiowalnego V0 pogarsza błąd z wycieku do możliwego double free?
2. Dlaczego move assignment musi zwolnić B przed przejęciem A?
3. Dlaczego `other=nullptr` jest zmianą własności, a nie tylko kosmetycznym „czyszczeniem”?
4. Kiedy destruktor C++ nie pomoże: normalny koniec zakresu czy wymuszone usunięcie taska bez unwindingu?

Zaliczenie

- ☐ formułuję inwariant „jeden bufor — jeden właściciel”;
- ☐ uzasadniam `copy = delete i move noexcept`;
- ☐ układam `release -> take -> clear source`;
- ☐ wskazuję A po obu move oraz B jako pierwszy zwolniony adres;
- ☐ pokazuję `allocations=2 frees=2 live=0`;
- ☐ pokazuję `after_scope == initial i pass=1`;
- ☐ łączę `delete[] -> operator delete[] -> vPortFree()`.

Granica RAI

RAI działa wtedy, gdy kończy się czas życia obiektu zgodnie z regułami C++. Nie obiecuje automatycznego sprzątnięcia po zaniku zasilania, zatrzymaniu systemu ani arbitralnym `vTaskDelete()` bez odwinienia ramek C++. Dlatego późniejszy wrapper taska nie będzie ukrywał wymuszonego usunięcia w destruktorze.

Następna karta K04: stany schedulera, priorytety, time slicing i typowane ticki. **VectorV1** pozostaje małym właścicielem pamięci; nie staje się jeszcze kontenerem o zmiennym rozmiarze ani allocator-aware.