

MemoryResource i FreeRtosAllocator<T>

wybór storage w runtime bez vtable i bez przepełnienia

|           |                    |          |                     |
|-----------|--------------------|----------|---------------------|
| Karta     | K06 / 16           | Czas     | 30 minut            |
| Platforma | Hazard3 / RV32I    | Język    | freestanding C++17  |
| Zasoby    | heap + arena 256 B | Dispatch | context + 2 funkcje |
| Wersja    | v00.01             | UUID     | 1af48a2e-...        |
|           |                    | karty    |                     |

Jedna operacja typowana, dwie polityki

Ten sam FreeRtosAllocator<uint32\_t> rezerwuje 16 elementów z:

- **HeapResource**: pvPortMalloc/vPortFree, pojedyncze free;
- **StaticArenaResource**: bufor w .bss, liniowy offset, pojedyncze deallocate nic nie robi, odzyskanie przez reset().

**Kontrakt K06.** Przed count\*sizeof(T) sprawdzamy dzielenie. Porażka nie zmienia dostępnej pamięci ani offsetu. Widok zasobu nie posiada kontekstu; provider musi żyć dłużej niż allocator.

Plan 30 minut

| Czas  | Tryb       | Dowód                                        |
|-------|------------|----------------------------------------------|
| 0–5   | kontrakt   | trzy słowa: context, allocate, deallocate    |
| 5–10  | arytmetyka | guard przed mnożeniem count i sizeof(T)      |
| 10–16 | alokacja   | 16 uint32 z heapu i 16 z areny               |
| 16–21 | adresy     | ucHeap kontra globalny bufor .bss, alignment |
| 21–26 | failure    | overflow i capacity nie zmieniają storage    |
| 26–30 | release    | heap free kontra arena deallocate/reset      |

Predykcja

Rozmiar 16 elementów uint32\_t: \_\_\_\_\_. B. Po arena-deallocate used=\_\_\_\_\_; po reset used=\_\_\_\_\_.

# 1 Resource to wartość, nie hierarchia klas

```
class MemoryResource final {
    void* context_;
    void* (*allocate_)(void*, size_t, size_t) noexcept;
    void (*deallocate_)(void*, void*, size_t, size_t) noexcept;
};
```

allocator -> MemoryResource -> function(context, bytes, alignment)

| Element              | Właściciel                    | Co sprawdzamy               |
|----------------------|-------------------------------|-----------------------------|
| MemoryResource       | nikt; non-owning view         | 3 wskaźniki, bez vtable     |
| HeapResource         | liczniki providera            | alignment ≤ gwarancja portu |
| StaticArenaResource  | bufor przekazany przez caller | base, capacity, used        |
| FreeRtosAllocator<T> | nikt; kopiuje view            | count, overflow, alignof(T) |

Wpisz odczyty z checkpointu 1:

| Pole                    | Adres |
|-------------------------|-------|
| heap context            | _____ |
| arena context           | _____ |
| heap allocate function  | _____ |
| arena allocate function | _____ |

Funkcje allocate powinny być różne, ale kod wywołujący typed allocator jest ten sam. Koszt dispatch to jawne pośrednie wywołanie, nie wirtualny obiekt.

## 2 Guard mnożenia przed mnożeniem

Uzupełnij:

```
T* try_allocate(size_t count) const noexcept {
    if (count == 0 || count > _____) {
        return nullptr;
    }
    const size_t bytes = _____;
    return static_cast<T*>(resource.allocate_bytes(
        bytes, _____));
}
```

**Dlaczego test po mnożeniu jest błędny?** Gdy wynik zawinie się do małej liczby, allocator może zwrócić za mały, pozornie poprawny blok. W tej karcie overflow nie wywołuje nawet funkcji providera.

### Dwie klasy porażki

1. **Arithmetic rejection:** count nie daje się bezpiecznie przeliczyć; context nie jest dotykany.
2. **Capacity rejection:** liczba bajtów jest poprawna, ale provider nie ma miejsca; zwraca nullptr i nie przesuwą storage.

### 3 Ten sam bufor logiczny, dwa obszary adresowe

HeapResource -> [16 \* uint32] inside ucHeap

ArenaResource -> [16 \* uint32] inside g\_static\_arena\_storage

| Dowód                     | Heap  | Arena |
|---------------------------|-------|-------|
| adres                     | _____ | _____ |
| zakres pamięci            | _____ | _____ |
| address %                 | _____ | _____ |
| alignof(uint32_t)         | _____ | _____ |
| liczba elementów / bajtów | _____ | _____ |
| suma wpisanych wartości   | _____ | _____ |

### Porażka ma zostawić stan storage bez zmian

| Próba                              | Stan przed | Stan po    |
|------------------------------------|------------|------------|
| overflow count dla heap allocator  | free=_____ | free=_____ |
| overflow count dla arena allocator | used=_____ | used=_____ |
| za duży poprawny byte request      | free=_____ | free=_____ |
| heap                               |            |            |
| za duży poprawny byte request      | used=_____ | used=_____ |
| arena                              |            |            |

Przy overflow provider failure count też się nie zmienia, bo wywołanie zostaje odrzucone wcześniej. Przy capacity failure może wzrosnąć licznik nieudanych prób, ale ilość zajętej pamięci pozostaje taka sama.

### Dwie polityki zwalniania

|                                                                                  | po allocate | po deallocate       | po reset    |
|----------------------------------------------------------------------------------|-------------|---------------------|-------------|
| heap current used                                                                | 64+metadata | 0 względem baseline | nie dotyczy |
| arena used                                                                       | 64          | <b>64</b>           | <b>0</b>    |
| Dlaczego arena nie może zwolnić „środkowego” bloku przez samo cofnięcie offsetu? |             |                     |             |

Reset jest operacją zbiorczą. Wolno go wykonać dopiero, gdy żaden żywy obiekt nie używa adresów z areny.

## 4 Hazard3/GDB: siedem checkpointów

```
make check
riscv64-unknown-elf-gdb build/task01_allocator_resource/prog.elf
b allocator_resource_debug_checkpoint
```

| STOP | Stan               | Wymagany dowód                                |
|------|--------------------|-----------------------------------------------|
| 1    | dwa resource views | context i function pointers zapisane          |
| 2    | dwa bufory żywe    | heap w ucHeap, arena w .bss, alignment i sumy |
| 3    | overflow rejected  | oba nullptr; provider/storage bez zmian       |
| 4    | capacity rejected  | oba nullptr; available/used bez zmian         |
| 5    | heap deallocated   | heap current free wraca do baseline           |
| 6    | arena deallocated  | arena used nadal 64                           |
| 7    | arena reset        | arena used 0; liczniki i pass=1               |

### Odczyty

```
p/x g_heap_address
p/x g_arena_address
p/x g_arena_storage_begin
p/x g_arena_storage_end
p g_heap_after_capacity_failure
p g_arena_used_after_allocation
p g_arena_used_after_deallocate
p g_arena_used_after_reset
p g_heap_successful_resource_allocations
p g_heap_failed_resource_allocations
p g_arena_successful_resource_allocations
p g_arena_failed_resource_allocations
p g_allocator_resource_pass
```

### Zaliczenie

- ☐ rysuję 3-słowy resource i nie nazywam go vtable;
- ☐ stosuję `count > SIZE_MAX/sizeof(T)` przed mnożeniem;
- ☐ klasyfikuję heap address i arena address po zakresach;
- ☐ pokazuję niezmieniony storage po obu rodzajach failure;
- ☐ odróżniam heap free, arena deallocate i arena reset;
- ☐ wiem, że view nie posiada providera ani bufora.

### Wyjście

Dlaczego `reset()` nie może być ukryty w destruktorze dowolnego allocator'a?

Co musiałoby żyć dłużej: `FreeRtosAllocator<T>` czy jego provider?

**Następna karta K07:** posiadający obiekt taska, jawne `start()`, stabilny kontekst i statyczny trampoline `C → C++`.