C++ Allocators

Why Allocators Exist

Imagine you're using:

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
```

When you create a container like this, it uses the default allocator, which internally calls the global heap allocator (`::operator new() / malloc()`).
That means every time the vector grows, it may request new memory from the OS heap — which can be slow, fragmented, or unpredictable. That's fine for normal programs...

But in low-latency or high-performance systems (e.g. trading, game engines, or networking frameworks), this is a big problem:

- Every `malloc/new` call can trigger mutex locks, heap fragmentation, and cache misses.
- Each allocation can cost hundreds of nanoseconds to microseconds — unacceptable when your SLA is <10 µs.

➡️ Allocators solve this by letting you control where and how memory is managed.

C++ Allocators

The Core Idea

➡️ An allocator is a pluggable memory management strategy for STL containers.

Allocators let you change that memory source.
For example:

```
std::pmr::monotonic_buffer_resource pool;
std::pmr::vector<int> v{ &pool };
```

Here's what happens:

- You first create a pool (a pre-allocated block of memory).
- Then you tell the vector to use that pool instead of the heap.

Now:

✅ Every allocation comes from the pool, not the global heap.
✅ Allocation is just a pointer bump inside that pre-allocated memory.
✅ No locks, no system calls, no fragmentation.
✅ When done, you can free the whole pool in one go — no per-object frees.

```
Container → Allocator → Memory Pool
(instead of Heap)
```

C++ Allocators

Allocators separate **data structure logic** from **memory management logic**.
This is why STL containers can use **any allocator** you give them:

```
std::vector<int, CustomAllocator<int>> vec;
```

or

```
std::pmr::vector<int> vec{&resource};
```

C++ provides two key building blocks for memory management:
`std::allocator`
`std::allocator_traits`

`std::allocator`

The default allocator in the C++ Standard Library; it obtains and releases memory from the global heap using operator new and operator delete.

`std::allocator_traits`

A wrapper that provides a uniform interface for all allocator types, letting STL containers interact generically with different allocator implementations.

C++ Allocators

Program 1 — Using `std::allocator` directly

```
#include <iostream>
#include <memory>

int main() {
   std::allocator<int> alloc;

   // allocate space for 3 ints
   int* p = alloc.allocate(3);

   for (int i = 0; i < 3; ++i) {
      alloc.construct(&p[i], i * 10);
   }

   for (int i = 0; i < 3; ++i) {
      std::cout << p[i] << " ";
   }

   std::cout << "\n";

   for (int i = 0; i < 3; ++i) {
      alloc.destroy(&p[i]);
   }

   alloc.deallocate(p, 3);
}
```

C++ Allocators

Program 2 — Using `std::allocator_traits` directly

```
#include <iostream>
#include <memory>

int main() {
   std::allocator<int> alloc;
   using Traits =
   std::allocator_traits<decltype(alloc)>;

   int* p = Traits::allocate(alloc, 3);
   for (int i = 0; i < 3; ++i) {
      Traits::construct(alloc, &p[i], i + 1);
   }

   for (int i = 0; i < 3; ++i) {
      std::cout << p[i] << " ";
   }

   std::cout << "\n";

   for (int i = 0; i < 3; ++i) {
      Traits::destroy(alloc, &p[i]);
   }

   Traits::deallocate(alloc, p, 3);
}
```