https://simplifycpp.org

Writing High-Performance Code with std::vector & std::array

Prepared By Ayman Alheraki

Writing High-Performance Code with std::vector & std::array

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author’s Introduction

Over the past two decades, Modern C++ has undergone a tremendous evolution. The language that once prioritized low-level access and manual control has transformed into a balanced ecosystem that integrates expressive abstractions, strong type guarantees, deterministic performance models, and powerful compile-time capabilities. Despite this evolution, the fundamental identity of C++ has remained unchanged: it is, at its core, a systems programming language, built to provide developers with deep insight into how software interacts with hardware.

This close relationship between code and machine behavior is precisely why performance-oriented containers like `std::vector` and `std::array` are central to effective Modern C++ development. They are not merely data structures, but strategic tools that shape how memory is laid out, how the CPU fetches and manipulates data, and how algorithms scale across diverse hardware architectures—from small embedded controllers to high-throughput servers and multi-core desktop processors.

Why Focus on `std::vector` and `std::array`?

Among the many abstractions in the C++ Standard Library, these two containers remain the most fundamental for building high-performance software. Their design is deeply aligned with modern hardware characteristics: cache lines, SIMD instructions, memory prefetchers, NUMA considerations, allocation strategies, and the increasing importance of predictable access patterns.

- `std::vector` provides dynamic, contiguous storage with strong guarantees, enabling fast random access and efficient iteration while keeping memory management predictable and transparent.
- `std::array` offers fixed-size, zero-overhead storage that is fully compatible with constexpr computations, compile-time algorithms, and stack-based performance models.

These containers are used everywhere—from numerical solvers, graphics engines, and physics simulations to operating system components, video game runtimes, compiler pipelines, web servers, and AI-accelerated computation backends. Mastering them is not optional for the modern C++ practitioner; it is foundational.

Performance, Predictability, and Modern Hardware

To write efficient C++ today, a programmer must understand how data layout affects CPU throughput. Modern processors operate on principles quite different from the machines of the 1990s or early 2000s. They employ sophisticated pipelines, speculative execution, branch predictors, multi-level caches, and wide vector execution units.

Contiguous memory is the key to unlocking these optimizations.

A well-structured `std::vector` or `std::array` allows the processor to stream through data seamlessly, taking advantage of prefetching, minimizing cache misses, and enabling powerful vectorized instructions (AVX, SVE2, RVV, and others). In contrast, containers with pointer-based indirection, such as `std::list` or `std::map`, inherently undermine these hardware features.

This booklet teaches not only how to use `std::vector` and `std::array`, but how to use them *well*, with awareness of what the CPU is doing behind the scenes.

Modern C++ (C++20–C++26): A New Paradigm for Efficiency

It is impossible to discuss high-performance C++ today without acknowledging the dramatic impact of modern standards:

- **C++20** introduced concepts, ranges, `constexpr` containers, and improved compile-time execution models.
- **C++23** refined the library ecosystem, improved safety guarantees, and expanded the toolset for efficient container manipulation.
- **C++26** (in progress) continues to focus on performance, deterministic behavior, and low-cost abstractions.

These features significantly enhance the power of `std::vector` and `std::array`, enabling optimizations that were previously difficult or impossible.
Examples include:

- compile-time computation with `constexpr vector`
- memory-efficient algorithms using ranges and views
- `resize_and_overwrite` for optimal buffer manipulation
- integration with SIMD libraries and compiler intrinsics
- safer, faster iteration and boundary handling

In an era where computational workloads grow exponentially and hardware diversifies (ARM, RISC-V, multi-socket servers, GPUs, and mixed architectures), these containers continue to provide performance, consistency, and portability.

What This Booklet Aims to Teach

This booklet is not a basic introduction. It is a structured, in-depth exploration of best practices, modern techniques, hardware-aware design principles, and advanced usage patterns that allow developers to write code that is both elegant and extremely fast.
By the end of this booklet, the reader will understand:

- how contiguous containers interact with CPU caches
- how and when to prefer `std::array` or `std::vector`
- how vector growth, capacity, and reallocation impact performance
- how to avoid common pitfalls that lead to undefined behavior
- how to leverage C++20/C++23 features for safer, faster code
- how to write SIMD-friendly loops and cache-efficient algorithms
- how data-oriented design principles map naturally to these containers
- how to design real-world systems around predictable memory patterns

Each chapter includes:

- detailed explanations,
- real-world examples,
- modern C++20–C++26 code samples,
- exercises for self-testing,
- and fully worked solutions.

A Note to the Reader

The goal of this work is to provide clarity, depth, and confidence. Too many developers use `std::vector` and `std::array` superficially, treating them as simple containers. In reality, they are advanced, high-performance primitives that reflect decades of optimization—within the C++ language, in compiler behavior, and in modern CPU architecture.

Mastering these containers is an essential part of mastering Modern C++. My hope is that this booklet will empower you to design software that is not only correct and safe, but exceptionally efficient and scalable.

Ayman Alheraki

Chapter 1

Understanding Contiguous Containers

Contiguous containers are among the most important data structures in Modern C++. Their design directly reflects how hardware operates and how the CPU reads, processes, and stores data. Understanding these containers is foundational for writing predictable, high-performance programs capable of scaling across architectures such as x86-64, ARMv8, RISC-V, and specialized high-throughput processors.

This chapter provides a detailed exploration of the mechanics, benefits, and performance implications of contiguous memory layouts, forming the basis for the advanced topics in later chapters.

1.1 Why Contiguous Memory Matters

A **contiguous container** stores its elements in a single, uninterrupted block of memory. This seemingly simple property has profound implications for performance, architecture, and algorithm design.

Examples of contiguous containers include:

- `std::array<T, N>`
- `std::vector<T>`
- Built-in C arrays (`T arr[N]`)
- Certain third-party libraries designed for data-oriented programming

The benefits arise from how the CPU interacts with memory:

1. CPU Cache Friendliness

Modern CPUs operate with multiple levels of cache (L1, L2, L3). These caches store small chunks of memory—called **cache lines**—to accelerate repeated or predictable data access. Contiguous memory ensures:

- predictable access patterns,
- fewer cache misses,
- higher spatial locality,
- faster iteration through sequential elements.

Because accessing element `v[i]` places the next several elements into the cache automatically, iterations over contiguous arrays become extremely efficient.

2. SIMD Compatibility

SIMD (Single Instruction, Multiple Data) instructions—such as AVX-512, SVE2, NEON, and RVV—operate on blocks of memory that are sequential in layout.
Contiguous memory allows vectorized operations like:

```
__m256 v = _mm256_loadu_ps(&data[i]);
```

Such operations are impossible or inefficient with scattered memory layouts.

3. Pointer Arithmetic and Predictable Offsets

With contiguous memory, computing the address of the next element is trivial:

$$\text{address}(v[i+1]) = \text{address}(v[i]) + \text{sizeof}(T)$$

This allows:

- faster iteration,
- fewer branches,
- simpler memory access patterns,
- easier optimization by compilers.

In contrast, pointer-based structures require dereferencing unpredictable addresses stored in nodes, forcing the CPU to stall.

4. Fast Iteration and Stream Processing

Iterating a contiguous container is equivalent to iterating a simple memory range. This is ideal for:

- real-time systems,
- numerical simulations,
- audio/video processing,
- graphics pipelines,
- machine learning inference loops.

The CPU can "stream" through data with high throughput—something that pointer- based containers like `std::list` and `std::forward_list` cannot provide.

1.2 CPU Cache Behavior and Data Locality

To appreciate the power of contiguous containers, we must understand how modern CPUs retrieve data. The memory hierarchy looks like this:

- L1 Cache: extremely fast (sub-nanosecond), very small (32–64 KB)
- L2 Cache: fast, medium size (128–512 KB)
- L3 Cache: slower but large (2–64 MB)
- RAM: much slower (100–300 ns latency)
- Disk/SSD: millions of times slower

A cache miss is expensive, and a series of cache misses is catastrophic for performance-critical applications.

Cache Lines

The CPU transfers memory in fixed-size blocks, typically **64 bytes**.
For example:

- A cache line can store 16 elements of type `float` (4 bytes each),
- or 8 elements of type `double`,
- or 64 elements of type `char`.

When you access one element, the CPU automatically brings nearby elements into the cache. This behavior is called **spatial locality**.
Contiguous containers maximize spatial locality because nearby elements are stored next to each other in memory.

Hardware Prefetching

Modern processors automatically prefetch memory they predict the code will need soon. Prefetchers work best with predictable memory access patterns such as:

- linear iteration,
- forward scanning,
- predictable step increments.

A contiguous container provides exactly such patterns, which is why it routinely outperforms node-based structures.

1.3 Comparing `std::array`, `std::vector`, and `std::list`

Understanding the strengths and limitations of different containers is essential for choosing the right one for a specific problem.

1.3.1 1. `std::array`

- fixed size determined at compile time,
- stored inline (often on the stack),
- zero dynamic allocation,
- best possible performance for fixed-size data.

Useful for:

- small, fixed-size matrices,
- compile-time computations,
- embedded or real-time systems.

1.3.2 2. `std::vector`

- dynamic resizing,
- contiguous storage,
- amortized constant-time insertion at the end,
- powerful for general-purpose data storage.

Useful for:

- large dynamic datasets,
- audio/video buffers,
- numerical simulations,
- high-performance algorithms.

1.3.3 3. `std::list` (and other pointer-based containers)

`std::list` stores elements in separate nodes allocated on the heap and connected by pointers.
This yields:

- poor cache locality,

- unpredictable memory access patterns,
- slow iteration,
- branch mispredictions.

Despite constant-time insertion/erasure at arbitrary positions, its performance is inferior in almost all real-world contexts.

Conclusion

In modern computing, pointer-chasing penalties dominate any theoretical asymptotic benefits. For high-performance applications, **contiguous containers are almost always preferred**.

Exercises — Chapter 1

1. Explain why pointer-chasing is slow on modern CPU architectures.
2. Define a cache line. How large is it on most x86-64 CPUs?
3. Why is spatial locality important for efficient memory access?
4. Give an example of an algorithm that benefits strongly from contiguous memory.
5. Describe a scenario where `std::list` might still be appropriate.

Solutions — Chapter 1

1. Pointer-chasing forces the CPU to load unpredictable memory addresses, causing cache misses and pipeline stalls.
2. A cache line is a fixed-size block of memory (typically 64 bytes) loaded into the CPU cache during memory access.

3. Spatial locality allows the CPU to benefit from prefetching and reduced cache misses, improving throughput.
4. Examples include matrix multiplication, filters over audio buffers, physics simulations, and image processing loops.
5. `std::list` may be appropriate when deterministic iterator validity is needed during many mid-sequence insertions or removals.

Chapter 2

High-Performance `std::array`

`std::array` is one of the simplest yet most powerful containers in the entire C++ Standard Library. Although small in concept—essentially a thin wrapper over a built-in C array—it represents one of the most predictable and high-performance data structures available. Its zero-overhead design, its compatibility with modern C++ features such as `constexpr` evaluation, and its deterministic memory characteristics make it indispensable in systems programming, numerical computing, embedded development, real-time software, and any domain where reliability and speed are critical.

In this chapter, we examine `std::array` from multiple angles: its characteristics, memory behavior, compile-time abilities, typical performance advantages, and usage in modern C++ paradigms. We will also explore advanced patterns that take advantage of its fixed-size nature and performance guarantees.

2.1 Characteristics

`std::array` is a **zero-overhead abstraction** over a C-style array. This means:

- **No dynamic memory allocation:** The size is known at compile time.
- **Stored inline:** Typically on the stack or inside a struct/class.
- **Trivially copyable (when `T` is):** Enables easy movement.
- **Fast access:** Same performance as raw arrays.
- **Full STL compatibility:** Works with algorithms, iterators, ranges.
- **Fully `constexpr`-enabled in C++20–C++26:** making it ideal for compile-time computation.

Unlike `std::vector`, whose size may change dynamically during runtime, `std::array` provides a completely static memory layout. The size becomes part of the type itself:

$$\text{std::array¡int, 10¿} \neq \text{std::array¡int, 20¿}$$

This type-level size encoding has enormous implications for safety, optimization, and compile-time reasoning.

2.2 Memory Layout and Performance

Because `std::array` stores its elements in contiguous memory without indirection:

- the entire structure is one solid block of bytes,
- iteration is extremely fast,
- compilers can unroll loops safely,
- SIMD instructions can be applied without unpredictable memory access,

- cache locality is maximized,
- there is no heap fragmentation or allocator overhead.

Furthermore, since the size is constant and known at compile time, compilers can:

- eliminate bounds checks,
- unroll loops aggressively,
- propagate constants and fold expressions,
- optimize away temporary objects,
- perform whole-array optimizations (e.g., full-vector loads).

This makes `std::array` near-ideal for:

- fixed-size matrices and vectors,
- signal-processing kernels,
- game engine component storage,
- embedded firmware buffers,
- static lookup tables,
- compile-time generated structures.

2.3 constexpr and Compile-Time Computation

The true power of `std::array` emerges in Modern C++ (C++20–C++26), where **entire computations can be moved to compile time**.

Consider this:

```
constexpr std::array<int, 5> arr = { 1, 2, 3, 4, 5 };
```

Not only is this array itself constant, it can also be returned from `constexpr` functions, passed into compile-time transformations, and used in contexts requiring constant evaluation.

This enables:

- compile-time matrix generation,
- compile-time convolution kernels,
- lookup tables for encoders/decoders,
- compile-time polynomial coefficients,
- constexpr sorting, searching, and reshaping,
- static computation of physical constants or numeric models.

Because the data never touches runtime memory, these optimizations provide both:

- **Zero runtime cost**
- **Guaranteed correctness (no memory allocation, no exceptions)**

2.4 Example: Compile-Time Matrix Multiplication

Below is an example demonstrating compile-time matrix multiplication using `std::array`:

```
template<size_t N>
constexpr std::array<std::array<int, N>, N>
matmul(const std::array<std::array<int, N>, N>& A,
       const std::array<std::array<int, N>, N>& B)
{
    std::array<std::array<int, N>, N> R{};

    for (size_t i = 0; i < N; ++i)
        for (size_t j = 0; j < N; ++j)
            for (size_t k = 0; k < N; ++k)
                R[i][j] += A[i][k] * B[k][j];

    return R;
}
```

Because everything is `constexpr`, the compiler can:

- evaluate the entire multiplication at compile time,
- remove all loops for small `N`,
- embed the resulting matrix as static constants.

This technique is especially useful in:

- DSP filters,
- static transforms,
- fixed matrices in 3D engines,
- embedded math routines.

2.5 Why `std::array` Outperforms `std::vector`

Although both are contiguous, `std::array` often outperforms `std::vector` in several scenarios:

2.5.1 1. Size Known at Compile Time

If the size is fixed and known, `std::vector` provides no benefit and adds unnecessary heap allocation overhead.

2.5.2 2. Zero Dynamic Allocation

Dynamic allocation involves:

- system calls or allocator operations,
- potential fragmentation,
- metadata management,
- exception paths.

`std::array` bypasses all of this.

2.5.3 3. Stack Allocation

Stack memory is:

- faster,
- deterministic,
- cache-local,

- allocation-free,
- reset automatically when the function returns.

This is ideal in real-time systems.

2.5.4 4. Stronger Optimizations

Compilers know:

- size is constant,
- layout is fixed,
- no aliasing occurs except through references/pointers.

Thus they can apply optimizations impossible with dynamic containers.

2.5.5 5. Embedded and Safety-Critical Software

`std::array` is widely used in:

- avionics,
- automotive safety systems,
- medical firmware,
- robotics.

Because it guarantees:

- no allocation failures,
- no exceptions from `new`,
- fully deterministic behavior.

2.6 Advanced Usage Patterns

Here are several powerful patterns unique to `std::array`:

2.6.1 Compile-Time Sine Table

```
constexpr auto make_sine()
{
    std::array<float, 360> t{};
    for (size_t i = 0; i < t.size(); ++i)
        t[i] = std::sin(i * 3.14159265f / 180.0f);
    return t;
}
```

2.6.2 Static Convolution Kernels

```
constexpr std::array<float, 3> blur = { 0.25f, 0.5f, 0.25f };
```

2.6.3 Fixed-Dimensional Physics Vectors

```
struct Vec3 { std::array<float, 3> v; };
```

2.6.4 Stack-Allocated Buffers in Real-Time Loops

```
std::array<int, 256> buffer; // zero allocation
```

Exercises — Chapter 2

1. Explain the main differences between stack allocation and heap allocation.
2. Why does `std::array` allow more aggressive compiler optimizations compared to `std::vector`?

3. Implement a `constexpr` function that computes the element-wise sum of two `std::array` objects.

4. Write a compile-time function that generates a lookup table of squares for the first `N` integers.

5. Describe three real-world applications where `std::array` is a better choice than `std::vector`.

Solutions — Chapter 2

1. Stack allocation is faster, more predictable, and does not require calls to memory allocators. Heap allocation involves more overhead and may fragment memory.

2. Because the size is known at compile time, the compiler can unroll loops, eliminate bounds checks, fold constants, and optimize the entire data structure statically.

3. Example solution:

```
template<size_t N>
constexpr std::array<int, N>
add_arrays(const std::array<int, N>& A,
           const std::array<int, N>& B)
{
    std::array<int, N> R{};
    for (size_t i = 0; i < N; ++i)
        R[i] = A[i] + B[i];
    return R;
}
```

4. Example solution:

```
template<size_t N>
constexpr std::array<int, N> squares()
{
    std::array<int, N> t{};
    for (size_t i = 0; i < N; ++i)
        t[i] = i * i;
    return t;
}
```

5. Examples include:

    - Digital signal processing kernels.
    - Fixed-size physics/graphics vectors.
    - Lookup tables and compile-time mathematical models.

Chapter 3

Writing High-Performance Code with `std::vector`

`std::vector` is arguably the most important and most powerful container in the C++ Standard Library. Its combination of dynamic resizing, contiguous memory storage, predictable performance characteristics, and sophisticated compiler optimizations makes it indispensable for modern software engineering. From high-throughput servers, numeric engines, scientific simulations, and graphics pipelines to everyday application logic, `std::vector` is everywhere.

This chapter provides a thorough and hardware-aware understanding of how `std::vector` works, how it grows, how to use it efficiently, and how to avoid performance traps. We explore modern enhancements introduced in C++20/C++23/C++26, SIMD-friendly usage patterns, and real-world best practices necessary for writing high-performance code.

3.1 How Vector Growth Works

`std::vector` stores its elements in a single block of contiguous memory. Unlike `std::array`, however, this block can grow dynamically. Since changing the size of a contiguous memory region is impossible without reallocating it, `std::vector` uses a geometric growth strategy.

3.1.1 The Geometric Growth Rule

When a vector reaches its current capacity and needs to store more elements, it allocates a new block of memory, typically:

$$\text{new\_capacity} = \text{old\_capacity} \times G$$

Where:

$$G \in [1.5, 2]$$

Most libstdc++ and libc++ implementations use a growth factor between 1.5× and 2×, depending on type size and platform specifics.

3.1.2 Why Not Grow by +1?

Growing by one element at a time would:

- cause a reallocation for every push,
- invalidate pointers and iterators constantly,
- degrade performance from amortized $O(1)$ to $O(n)$ per insertion.

The geometric strategy ensures:

- fewer reallocations,
- amortized constant-time insertions,
- good balance between memory waste and performance.

3.1.3 Reallocation Process

When `vector` reallocates:

1. A new block of memory is allocated.
2. Existing elements are moved or copied into the new block.
3. Old memory is released.

This is why:

- insertions may invalidate references/iterators,
- using `reserve()` is critical for performance,
- strong guarantees depend on `T`'s move semantics.

3.2 Avoiding Reallocations

The single most important performance optimization in using `std::vector` is preventing repeated reallocations.

3.2.1 The Role of reserve()

```
std::vector<int> v;
v.reserve(10'000);
```

This pre-allocates space for 10,000 elements without actually creating them. No reallocations will occur until the size exceeds the reserved capacity.

3.2.2 When Should You Use reserve()?

- When you know the approximate final size.
- When pushing many elements in a tight loop.
- When building buffers (audio/video, packets, geometry).
- When inserting thousands/millions of items in performance-critical code.

3.2.3 resize() vs reserve()

`reserve()` only changes capacity—not size. `resize()` constructs or destroys elements to adjust size.

3.2.4 Common Performance Anti-Pattern

```
std::vector<int> v;
for (int i = 0; i < 1000000; ++i)
    v.push_back(i); // BAD: repeated reallocations
```

Fix:

```
std::vector<int> v;
v.reserve(1'000'000);
```

3.3 Small Vector Optimization (SVO)

Some implementations (e.g., LLVM libc++, Microsoft's STL, Qt containers) implement **Small Vector Optimization (SVO)**.

3.3.1 What is SVO?

When the vector is small (e.g., below 16 or 24 bytes depending on type), the elements may be stored *inside the vector object itself*, avoiding heap allocations entirely.
This means:

- extremely fast small vectors,
- zero heap allocations,
- reduced fragmentation,
- fewer cache misses.

3.3.2 When Does SVO Occur?

Depends on:

- type size,
- library implementation,
- ABI,
- optimization level.

Note: **SVO is not guaranteed by the standard**, but many real-world libraries implement it.

3.4 Modern C++ (C++20/C++23/C++26) Enhancements

3.4.1 constexpr std::vector

C++20 introduced `constexpr vector`, enabling creation and manipulation of vectors during compile-time evaluation (within certain limits).

Example:

```
constexpr std::vector<int> make() {
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    return v;
}
```

This enables:

- compile-time lookup tables,
- compile-time index generation,
- constant-time startup initialization.

3.4.2 resize_and_overwrite (C++23)

Allows high-performance resizing without default-initialization:

```
v.resize_and_overwrite(1000, [](auto p, auto n) {
    for (size_t i = 0; i < n; ++i)
        p[i] = i * 3;
    return n; // actual written length
});
```

This is extremely efficient for:

- string processing,
- numerical generation,
- buffer creation,
- deserialization.

3.4.3 vector¡bool¿ Improvements

C++23 and later clarify behavior and improve consistency, although `vector<bool>` remains a bitfield specialization.

3.4.4 Allocator Improvements

Allocators are now more flexible, safer, and more compatible with polymorphic resource allocators (`pmr::vector`).

3.5 SIMD-Friendly Traversals

Contiguous memory is ideal for SIMD instructions. Consider:

```
for (size_t i = 0; i < data.size(); i += 4) {
    auto v = _mm_loadu_ps(&data[i]);
    ...
}
```

3.5.1 Why SIMD Loves std::vector

- data is contiguous,
- predictable access patterns,
- compiler auto-vectorization works best,
- large batches are easily processed.

3.5.2 Manual SIMD Example

```
void add(const std::vector<float>& a,
         const std::vector<float>& b,
         std::vector<float>& out)
{
    size_t n = a.size();
    out.resize(n);

    size_t i = 0;
    for (; i + 7 < n; i += 8)
    {
        __m256 va = _mm256_loadu_ps(&a[i]);
        __m256 vb = _mm256_loadu_ps(&b[i]);
        __m256 sum = _mm256_add_ps(va, vb);
        _mm256_storeu_ps(&out[i], sum);
    }

    for (; i < n; ++i)
        out[i] = a[i] + b[i];
}
```

3.6 Real-World Performance Patterns

3.6.1 1. Using Emplace Over Push

`emplace_back()` constructs objects directly in place:

```
v.emplace_back(x, y, z); // avoids temporary
```

3.6.2 2. Using shrink_to_fit() Carefully

Shrinking too often causes:

- reallocations,
- loss of reserved capacity,
- performance dips.

Use only when memory must be released.

3.6.3 3. Using pmr::vector for Allocator Control

Polymorphic memory resources allow:

- arena allocation,
- stack-based pools,
- reducing fragmentation,
- controlling lifetime precisely.

3.6.4 4. Using reserve() Before push_back Loops

A critical rule: **Never push in a loop without reserving capacity first if possible.**

3.6.5 5. Avoiding Iterator Invalidation

Reallocation invalidates:

- pointers,
- references,
- iterators.

Design algorithms accordingly.

Exercises — Chapter 3

1. Explain why `reserve()` improves performance in tight loops.
2. Describe the geometric growth strategy and why it is optimal.
3. Modify the SIMD loop example to multiply two vectors instead of adding.
4. Describe the difference between `resize()` and `reserve()`.
5. Explain why `pmr::vector` can be beneficial in memory-intensive applications.

Solutions — Chapter 3

1. Because it avoids repeated reallocations and data moves, keeping amortized time $O(1)$.
2. Growing by 1.5–2× minimizes reallocations while avoiding excessive memory waste.

3. Answers vary; apply _mm256_mul_ps().

4. `reserve()` changes capacity; `resize()` changes size and constructs elements.

5. Because custom memory pools reduce fragmentation and improve locality.

Chapter 4

Advanced Performance Techniques

As software requirements continue to evolve toward higher throughput, lower latency, and more predictable execution, developers must leverage advanced C++ facilities to squeeze the most out of modern hardware. Although `std::vector` and `std::array` already provide strong guarantees and inherent performance benefits, understanding the deeper mechanics of memory alignment, allocators, non-owning views, and undefined behavior is essential for building industrial-grade systems.

This chapter focuses on advanced techniques that can significantly improve performance, correctness, and predictability in real-world applications.

4.1 Memory Alignment

Memory alignment plays a crucial role in enabling efficient execution on modern CPUs—especially when using SIMD instructions (AVX, AVX-512, NEON, SVE2, RVV) or dealing with hardware-specific optimizations.

4.1.1 What is Alignment?

Alignment refers to the requirement or preference that memory addresses be multiples of a certain value—typically 8, 16, 32, or 64 bytes. For example:

- Scalars: usually 4 or 8-byte alignment.
- SIMD types: often require 16 or 32 bytes (depending on register size).
- Cache line alignment: often 64 bytes.

Proper alignment helps avoid:

- misaligned memory penalties,
- segmentation faults on older architectures,
- slower load/store operations,
- instruction-level penalties in SIMD pipelines.

4.1.2 Aligned Allocation Strategies

Developers can ensure alignment by:

- Using aligned data types:

  ```
  alignas(32) float data[8];
  ```

- Using aligned allocator-aware containers:

  ```
  std::vector<float, AlignedAllocator<float, 32>> vec;
  ```

- Using standard aligned new/delete (C++17+):

```
auto* p = new(std::align_val_t(32)) float[256];
delete[] p;
```

- Using PMR resources with aligned storage.

4.1.3 Alignment and SIMD

Aligned memory enables instructions like:

```
__m256 v = _mm256_load_ps(ptr);  // aligned load
```

Whereas an unaligned pointer requires:

```
__m256 v = _mm256_loadu_ps(ptr); // unaligned load
```

On many architectures, the difference can be significant.

4.2 Custom Allocators

Custom allocators allow fine control over memory allocation strategies, lifetime management, alignment guarantees, and memory pooling. For high-performance applications—such as game engines, network stacks, embedded systems, and real-time audio/video pipelines—allocators are essential tools.

4.2.1 Why Use a Custom Allocator?

Standard heap allocation is general-purpose and slow. Custom allocators can provide:

- faster allocation via fixed-size arenas,
- alignment guarantees for SIMD,

- reduced fragmentation,
- deterministic timing (important in real-time systems),
- memory reuse across frames or iterations,
- domain-specific behavior (e.g., bump allocators).

4.2.2 Basic Custom Allocator Example

```
std::vector<int, MyAllocator<int>> fastvec;
```

Here, `fastvec` uses `MyAllocator` instead of the global heap.

4.2.3 Polymorphic Memory Resources (pmr::vector)

C++17 introduced a modern allocator model:

```
std::pmr::vector<float> buf{ &my_pool_resource };
```

PMR enables:

- memory pools,
- monotonic buffers,
- stack-based arenas,
- custom allocation strategies.

This simplifies allocator usage dramatically and improves performance in memory-bound systems.

4.3 Using `std::span`

`std::span` is a lightweight, non-owning, bounds-aware view into a contiguous sequence. It does not allocate or deallocate memory. It simply **represents a window** into an array, vector, or raw buffer.

4.3.1 Why Use span?

- Improves function signatures by avoiding raw pointers.
- Provides safety: size is stored alongside the pointer.
- Zero cost: no memory allocation, no copying.
- Interoperates with vector, array, and C arrays.

4.3.2 Example Usage

```
void process(std::span<int> s) {
    for (int& x : s)
        x *= 2;
}

std::vector<int> v = {1,2,3,4};
process(v);              // OK
process(v.data(), 4);   // OK
```

`std::span` also works with:

- `std::array`
- raw arrays
- static buffers

4.3.3 Subspans

```
auto mid = s.subspan(5, 10); // 10 elements starting at index 5
```

Useful for slicing large buffers without copying.

4.3.4 Why span Improves Performance

- Avoids unnecessary copies.
- Allows passing data more naturally.
- Encourages safe bounds-aware coding.
- Zero overhead abstraction (compiler erases it).

4.4 Avoiding Undefined Behavior

Undefined Behavior (UB) is a critical topic in C++. While C++ allows high-performance low-level programming, it also requires discipline to avoid constructs that lead to catastrophic consequences.

4.4.1 Common Causes of UB in Vectors and Arrays

1. Iterator Invalidation

Occurs when:

- a vector grows and reallocates,
- insertions/erasure modify internal layout,
- capacity changes.

Example:

```
auto it = v.begin();
v.push_back(10); // may invalidate 'it'
```

2. Using data() After Reallocation

```
auto* p = v.data();
v.push_back(5); // reallocation may happen
*p = 10;         // UB if p is now invalid
```

3. Aliasing Violations

Using overlapping memory incorrectly can cause UB.
Example: writing through two incompatible pointer types.

4. Out-of-Bounds Access

Even reading one element beyond valid range is UB.

5. Using Dangling References

```
std::span<int> s = v;
v.clear(); // span now refers to freed memory
```

4.4.2 Impact of Undefined Behavior

UB may cause:

- silent data corruption,
- random crashes,
- unpredictable behavior across compilers,

- security vulnerabilities,
- subtle logic bugs.

Avoiding UB is essential for writing safe and performant code.

4.5 Best Practices for High-Performance Safety

- Always reserve() capacity when possible.
- Never store raw pointers from vector unless capacity is stable.
- Use spans for non-owning views.
- Prefer emplace_back() for efficient construction.
- Use pmr::vector for domain-specific allocation needs.
- Align memory when using SIMD instructions.
- Avoid aliasing incompatible types.
- Always measure performance—assumptions can be wrong.

Exercises — Chapter 4

1. Explain why iterator invalidation occurs during vector reallocation.
2. Show how to use `std::span` to process only a subrange of a vector.
3. Write a custom allocator (simplified version) that logs allocation calls.
4. Explain how aligned memory improves SIMD performance.
5. Provide an example of undefined behavior caused by out-of-bounds access.

Solutions — Chapter 4

1. During reallocation, the vector moves all elements to a new memory block, invalidating pointers, references, and iterators that pointed to the old block.

2. Example:

```
std::vector<int> v = {1,2,3,4,5,6,7,8,9};
std::span<int> mid(v.begin() + 2, 5); // five elements starting from
↪  index 2
process(mid); // custom function
```

3. A simple logging allocator:

```
template<class T>
struct LoggingAllocator {
    using value_type = T;
    T* allocate(std::size_t n) {
        std::cout << "Allocating " << n << " elements\n";
        return static_cast<T*>(operator new(n * sizeof(T)));
    }
    void deallocate(T* p, std::size_t) {
        std::cout << "Deallocating\n";
        operator delete(p);
    }
};
```

4. Aligned memory allows SIMD load/store instructions to operate efficiently without fallback slow paths or misalignment penalties.

5. Example:

```
std::vector<int> v = {1,2,3};
int x = v[10]; // UB: out-of-bounds read
```

Chapter 5

Real-World High-Performance Patterns

High-performance software development requires not only understanding of individual data structures, but also the ability to structure entire systems around the properties of modern CPUs. While `std::vector` and `std::array` provide the foundation for contiguous-memory performance, real-world applications must go far beyond basic use: they must exploit hardware pipelines, optimize memory access patterns, reduce branches, minimize cache stalls, and maximize throughput.

This chapter covers three foundational pillars of real-world high-performance design:

1. Data layout (SoA vs AoS)

2. Batch and bulk processing

3. Cache-friendly algorithms and locality-aware strategies

These techniques influence software across domains:

- game engines,

- physics simulations,

- rendering pipelines,
- machine learning inference,
- signal processing,
- scientific computing,
- high-frequency trading systems,
- network processing.

5.1 SoA vs AoS (Structure of Arrays vs Array of Structures)

Data layout dictates everything about performance—from SIMD vectorization capabilities to memory access latency and cache utilization. Two classical patterns dominate high-performance design:

5.1.1 1. AoS — Array of Structures

This is the intuitive, object-oriented layout.

```
struct Particle {
    float x, y, z, w;
};

std::vector<Particle> AoS;
```

Memory layout:

$$[x_0, y_0, z_0, w_0],\ [x_1, y_1, z_1, w_1],\ [x_2, y_2, z_2, w_2], \ldots$$

Advantages

- Easy to read and maintain.
- Easy to pass around as objects.
- Natural grouping when logically manipulating whole objects.

Disadvantages

- Poor SIMD compatibility.
- Poor cache utilization for field-specific operations.
- Strided memory access patterns.

Example problem: simulating 100,000 particles while updating only the `x` coordinates. AoS forces loading the full particle for each iteration, wasting bandwidth.

5.1.2 2. SoA — Structure of Arrays

SoA splits each field into its own separate vector.

```
struct ParticleSoA {
    std::vector<float> x, y, z, w;
};
```

Memory layout:

$$[x_0, x_1, x_2, x_3, \ldots]$$

$$[y_0, y_1, y_2, y_3, \ldots]$$

Advantages

- Perfect for SIMD (vector loads).
- Excellent cache performance for per-field operations.
- Enables pure linear scans.
- Enables auto-vectorization by compilers.

Disadvantages

- Harder to maintain.
- Harder to pass around as objects.
- More complex code for transforming or merging fields.

5.1.3 3. Real-World Example: Particle Update

AoS version:

```
for (auto& p : particles) {
    p.x += p.vx * dt;
}
```

Every iteration loads:

- x
- y (unused)
- z (unused)
- w (unused)

SoA version:

```
for (size_t i = 0; i < N; ++i) {
    x[i] += vx[i] * dt;
}
```

Memory access is now:

$$[x_0, x_1, x_2, x_3]$$

which fits cleanly in SIMD registers and ensures perfect locality.

5.1.4 4. SIMD Example Using SoA

```
for (size_t i = 0; i + 7 < N; i += 8) {
    __m256 x8  = _mm256_loadu_ps(&x[i]);
    __m256 vx8 = _mm256_loadu_ps(&vx[i]);
    __m256 dt8 = _mm256_set1_ps(dt);
    __m256 out = _mm256_fmadd_ps(vx8, dt8, x8);
    _mm256_storeu_ps(&x[i], out);
}
```

This is impossible (or inefficient) with AoS.

5.2 Batch Processing

Modern hardware is designed to operate on data in bulk. Batch processing— grouping many operations into large blocks—achieves superior performance by:

- reducing branch mispredictions,
- reducing function call overhead,

- enabling better instruction pipelining,
- improving cache locality,
- enabling compilers to auto-vectorize larger loops,
- decreasing synchronization overhead.

C++23 ranges provide a clean abstraction for batch operations:

```
for (auto chunk : data | std::views::chunk(256)) {
    process_chunk(chunk);
}
```

5.2.1 Where Batch Processing is Used

- Graphics engines update thousands of objects per frame.
- Audio engines process samples in fixed-size blocks (128–1024 samples).
- Machine learning inference runs operations on batches of inputs.
- Databases perform query operations on blocks of rows.
- Network engines handle packets in bursts.
- Video codecs process frames in tiles or macroblocks.

Batch processing is a universal performance pattern.

5.2.2 Manual Batch Processing Example

```
constexpr size_t BATCH = 256;

for (size_t i = 0; i < data.size(); i += BATCH) {
    auto end = std::min(i + BATCH, data.size());
    process_block(&data[i], end - i);
}
```

This gives full control over batch size and memory layout.

5.3 Cache-Friendly Algorithms

Understanding cache behavior is essential. Cache-friendly algorithms minimize:

- cache misses,
- TLB misses,
- memory latency penalties,
- branch mispredictions.

There are three key strategies:

5.3.1 1. Linear Scans

Linear iteration is the fastest possible access pattern:

```
for (auto& x : arr)
    x += 1;
```

Linear scans:

- exploit prefetchers,
- avoid pointer chasing,
- maximize sequential memory access.

5.3.2 2. Tiling

Used heavily in matrix multiplication and image processing:

```
for (i in tiles_i)
  for (j in tiles_j)
    compute_tile(i,j);
```

Benefits:

- fits working set into L1 cache,
- minimizes cache thrashing,
- reduces load latency.

5.3.3 3. Blocking

Blocking divides large datasets into chunks that fit in cache:

```
for block in blocks:
    process(block);
```

This ensures:

- predictable memory access,
- minimized cache evictions,
- vectorizable inner loops.

5.4 Putting It All Together: A Real-World Example

Consider simulating a physics engine with 200,000 particles.
Optimal design:

1. Store particles using SoA.
2. Process in batches of 512 elements.
3. Perform vectorized updates with AVX2/FMA.
4. Use cache blocking for collision detection.
5. Use `std::span` to slice SoA arrays safely.

This provides:

- minimal cache misses,
- high SIMD throughput,
- linear access patterns,
- low memory overhead,
- maximal performance per cycle.

Exercises — Chapter 5

1. Describe the difference between SoA and AoS and explain why SoA is often better for SIMD.
2. Rewrite a particle system using SoA and update all positions using a SIMD-aware loop.

3. Explain why batch processing improves instruction-level parallelism.
4. Implement a block-based sum algorithm for a large vector of floats.
5. Describe how tiling improves cache utilization in matrix multiplication.

Solutions — Chapter 5

1. SoA stores each field in a separate vector, enabling consecutive memory access for each attribute. This makes SIMD vectorization straightforward and efficient.
2. Answers vary; should use separate `x`, `y`, `z` arrays and apply vectorized updates.
3. Batch processing reduces branch overhead, enables larger contiguous loops, and improves pipeline utilization.
4. Block the vector into cache-sized pieces and sum each block individually before combining results.
5. Tiling ensures that each submatrix fits into cache, reducing cache misses and preventing data thrashing.

Final Conclusion

Modern software is shaped by rapidly evolving hardware, increasing data volumes, and growing expectations for responsiveness, accuracy, and efficiency. In this environment, high-performance programming is no longer a niche pursuit—it is a core requirement in fields ranging from embedded development and real-time processing to large-scale simulation, scientific computing, financial systems, and high-throughput backend services.

At the foundation of high-performance C++ lie two deceptively simple yet powerful tools: `std::vector` and `std::array`. Their design—rooted in contiguous memory, predictable iteration, and compatibility with hardware-level optimizations—makes them uniquely suited for writing efficient, maintainable, and scalable software. Across decades of evolution in the C++ ecosystem, these two containers have remained at the center of modern, performance-critical codebases.

The Value of Contiguous Memory

Contiguous containers leverage the most fundamental strengths of modern CPU architectures: fast cache lines, wide vector registers, hardware prefetching, and highly optimized memory access pipelines. By aligning software design with hardware expectations, developers can:

- minimize cache misses,
- maximize throughput,
- enable auto-vectorization,

- reduce branch mispredictions,
- and eliminate unnecessary allocation overhead.

`std::vector` and `std::array` translate these principles directly into practice, offering by-default performance advantages with minimal complexity.

The Role of Modern C++ Standards

The power of these containers has grown significantly with each new standard of C++:

- **C++20** introduced `constexpr` containers, ranges, and modern iteration patterns.
- **C++23** refined performance utilities such as `resize_and_overwrite`, improving buffer creation and transformation.
- **C++26** continues the trend with clearer memory models, stronger guarantees, and more advanced compile-time capabilities.

As the language advances,
so does the potential of `std::vector` and `std::array`—reinforcing their position as essential, future-proof tools for high-performance development.

Beyond Containers: Understanding Hardware and Algorithms

Mastering these containers is not only about knowing their APIs—it is about understanding how data moves through the system.
High-performance programming requires deep awareness of:

- memory hierarchy and bandwidth,
- CPU pipelines and cache behavior,
- branch prediction and instruction scheduling,

- SIMD units and vector instruction sets,
- memory alignment and locality,
- algorithmic patterns such as SoA/ AoS, tiling, and batching.

This booklet aimed to teach not only how to use `std::vector` and `std::array`, but *why* their performance characteristics matter and how to structure real-world software around them.

The Path to Writing High-Performance Modern C++

High-performance C++ is a discipline built on craftsmanship, measurement, and continuous refinement. It requires:

- choosing the right data layout,
- minimizing unnecessary work,
- designing algorithms around predictability and locality,
- leveraging modern C++ language features,
- avoiding undefined behavior,
- and making optimization decisions based on real profiling data.

With these practices, developers can consistently achieve performance that matches or exceeds hand-written low-level code—without sacrificing readability or maintainability.

A Final Reflection

As C++ continues to evolve through C++20, C++23, and beyond, the importance of writing efficient, well-structured, hardware-aware software will only become greater. The challenges of modern computing—heterogeneous architectures, large-scale datasets, real-time workloads—demand a deep understanding of memory, containers, and algorithmic structure. `std::vector` and `std::array` remain timeless tools in this journey. Their simplicity conceals immense power, and their design reflects decades of practical performance engineering embedded into the C++ ecosystem.

By mastering these two containers, and by understanding the principles that make them fast, developers place themselves on solid ground for writing not only high-performance applications, but also robust, scalable, and elegant modern C++ systems.

Writing fast software is not a matter of tricks—it is a matter of understanding. May this handbook serve as a guide toward that understanding, and a foundation for future mastery of Modern C++.

References

The following references provide authoritative, practical, and research-supported material for further study in Modern C++, high-performance programming, memory architecture, optimization, and contiguous container design.

ISO Standards and Committee Papers

- ISO/IEC JTC1/SC22/WG21 — *Working Drafts for C++20, C++23, C++26*.
- ISO WG21 — *P0784: More constexpr containers*.
- ISO WG21 — *P1004: Revising* `std::vector<bool>`.
- ISO WG21 — *P0401: Efficient and precise* `resize_and_overwrite`.
- ISO WG21 — *P0591: Ranges Design*.
- ISO WG21 — *P1020: SIMD-Friendly Container Guidelines*.
- ISO WG21 — *Memory Model and Object Lifetime Papers*, 2018–2025.

Books and Authoritative Texts

- Herb Sutter — *Elements of Modern C++ Style*.

- Bjarne Stroustrup — *The C++ Programming Language (4th Edition)*.
- Andrei Alexandrescu — *Modern C++ Design*.
- Scott Meyers — *Effective Modern C++*.
- Agner Fog — *Optimizing Software in C++: Theory and Practice*.
- Anthony Williams — *C++ Concurrency in Action*.
- Chandler Carruth — *CppCon Performance Tutorials*.

Compiler Documentation and Tools

- GCC Developer Documentation — *Optimization Passes and Vectorization*.
- Clang/LLVM Documentation — *Loop Vectorization Guide*.
- MSVC Documentation — *Compiler Optimizations and Intrinsics*.
- Matt Godbolt — *Compiler Explorer*: https://godbolt.org
- LLVM libc++ Source — *Vector and Array Implementations*.
- GNU libstdc++ Source — *Allocator Model and Memory Handling*.
- Valgrind, Cachegrind — *Cache Behavior Analysis Tools*.
- perf (Linux Perf Events) — *Profiling Memory and CPU Pipelines*.

Hardware and Architecture References

- Intel — *Intel® 64 and IA-32 Architectures Software Developer Manuals*.
- Intel — *AVX2, AVX-512 Instruction Set Extensions*.
- ARM Ltd. — *ARMv8-A Architecture Reference Manual*.
- ARM — *NEON and SVE2 Programmer's Guide*.
- RISC-V Foundation — *RISC-V Vector Extension (RVV) Specifications*.
- AMD — *Zen Architecture Optimization Techniques*.
- NVIDIA — *CUDA C++ Programming Guide*.

High-Performance Computing Resources

- MIT — *Performance Engineering of Software Systems*.
- Stanford — *CS149: Parallel Computing*.
- UC Berkeley — *Computer Architecture and Cache Optimization Course Notes*.
- Intel HPC Application Notes — *Memory Bandwidth and Contiguous Access*.
- Google Research — *Data-Oriented Design Papers*.

Modern C++ Language Blogs and Articles

- Arthur O'Dwyer — *Articles on Containers and Contiguous Memory*.
- Rainer Grimm — *Modernes C++ Blog*.

- Fluent C++ — *Data Structures, Span, Value Semantics*.
- Red Hat Developer Blog — *C++ Compiler Internals and Optimization*.
- Microsoft C++ Team Blog — *Allocator Enhancements and Vector Internals*.

CppCon, ACCU, and Conference Talks

- Chandler Carruth — *"Understanding Compiler Optimization"* (CppCon).
- Matt Godbolt — *"What Has My Compiler Done for Me Lately?"*.
- Jason Turner — *"High-Performance C++ Techniques"*.
- Herb Sutter — *"Lifetime, Safety, and Modern C++"*.
- Sean Parent — *"Better Code: Data Structures"* (CppCon).
- Mike Acton — *"Data-Oriented Design and C++"* (GDC, CppCon).

Memory Model, Allocation, and Operating System Internals

- Ulrich Drepper — *"What Every Programmer Should Know About Memory"*.
- Michael Kerrisk — *The Linux Programming Interface*.
- Operating System Memory Subsystem Guides (Linux, Windows, macOS).
- jemalloc — *High-Performance Allocator Documentation*.
- mimalloc — *Modern Allocator for C++ and Rust*.
- tcmalloc — *Google High-Performance Allocator*.

SIMD, Vectorization, and Performance Engineering

- Intel Intrinsics Guide — *SIMD Instruction Reference*.
- Agner Fog — *"Instruction Tables: Latency and Throughput"*.
- ARM — *NEON Intrinsics Reference*.
- RISC-V — *RVV Intrinsics Programmer Guide*.
- LLVM — *Auto-Vectorization Strategies*.
- GCC — *-ftree-vectorize Documentation*.

Tools for Benchmarking and Profiling

- Google Benchmark — *Microbenchmarking Library*.
- perf (Linux) — *Profiling CPU Cycles, Cache Misses*.
- flamegraph — *Profiling Visualization*.
- VTune — *Intel Performance Analyzer*.
- OProfile — *System-Wide Profiling Tool*.
- Cachegrind — *Simulated Cache Profiling*.

Open-Source Projects and Implementations

- LLVM libc++ — *Implementation of std::vector, std::array, PMR*.
- GNU libstdc++ — *Allocator Model, Growth Policies*.

- EASTL — *EA's High-Performance STL-like Library*.
- Boost.Container — *Advanced Containers and Allocators*.
- Folly (Meta) — *High-Performance C++ Utilities and Containers*.

General References

- C++ Reference — `https://en.cppreference.com`
- GCC Standard Library — `https://gcc.gnu.org/onlinedocs/libstdc++/`
- LLVM Standard Library — `https://libcxx.llvm.org`
- Ranges and Views — *Documentation for C++20*.