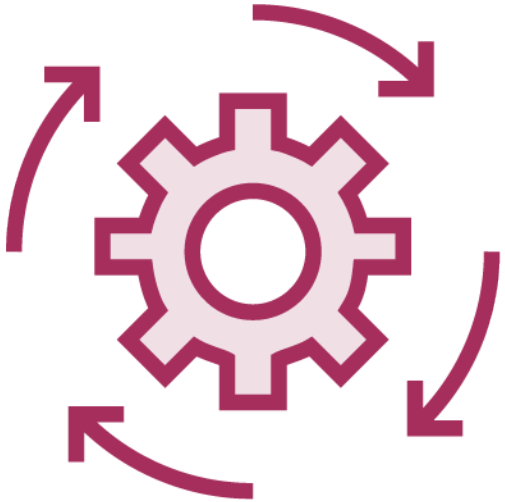
C++ Standard Library: Sequential Containers

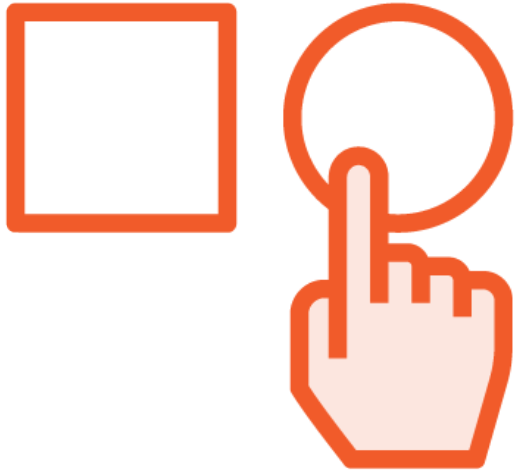
STORING SEQUENCES OF ELEMENTS WITH THE STANDARD STD::VECTOR CONTAINER

Giovanni Dicanio
AUTHOR, SOFTWARE ENGINEER

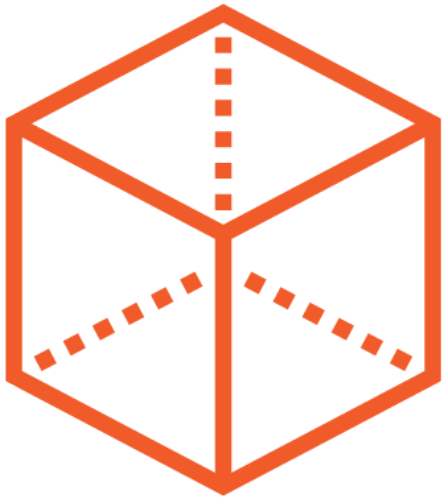
https://blogs.msmvps.com/gdicanio

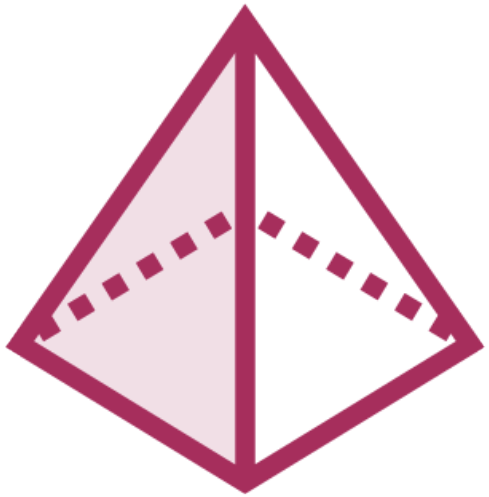
Navigate the C++ Standard Containers

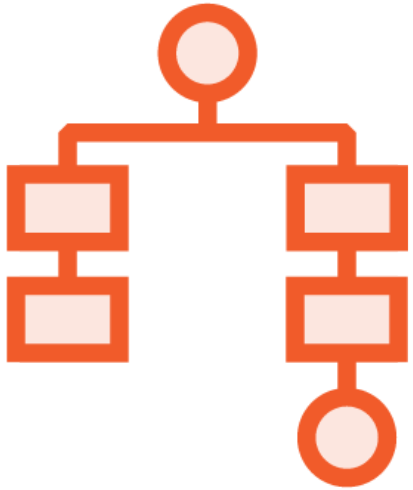




Different High-quality Standard Containers







C++ Standard Containers

Highly optimized

Well tested

C++ Standard **Library** (Not Framework)

Prefer using *standard* containers

Overview

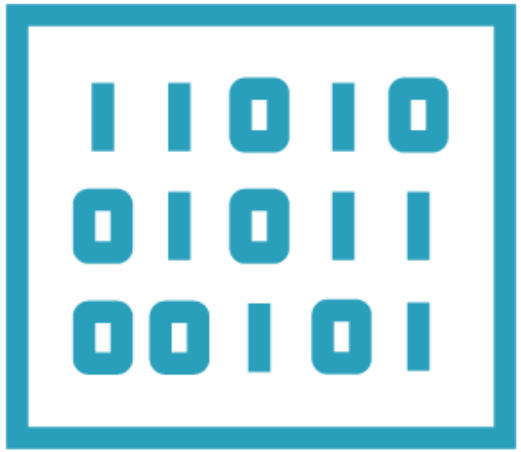
Introducing std::vector

Important properties and operations

Performance tips

Sample Scenario: IoT Application

IoT application



Data stream

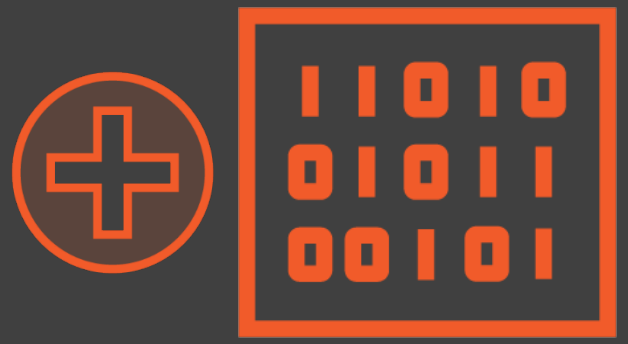
Where to store data?

```
// Data read from sensor
float temperatures[100];
```

Raw C-style Fixed-size Arrays

***Don't* do that!**

```
// Data read from sensor
float temperatures[100];
```



Raw C-style Fixed-size Arrays

***Don't* do that!**

```
// Data read from sensor
float* temperatures = new float[100];
```

Dynamic Heap-allocated Raw Arrays

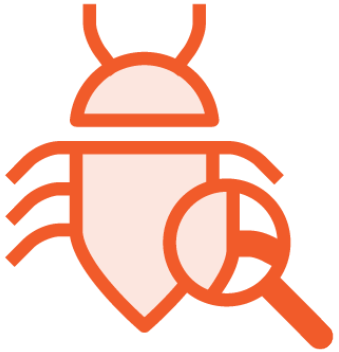
You can do *much better* than that...

Resizing Arrays

Allocate a new larger memory block

Copy data from previous block to new block

Release previous block



Storing a Sequence of Elements

Use *modern* C++ (and *libraries*)

std::*vector*

(C++ Standard Library)

Vector: A Dynamic Flexible Array

std::*vector*
(C++ Standard Library)

Easily resizable
at run-time

```
#include <vector>     // For std::vector
```

Using std::vector in Your C++ Code

Include the standard <vector> header

ELEMENT TYPE

```
std::vector<float> temperatures{};
```

Using std::vector in Your C++ Code

Create an empty vector

```
std::vector<float> temperatures{};

std::vector<float> temperatures; // Created empty, as well
```

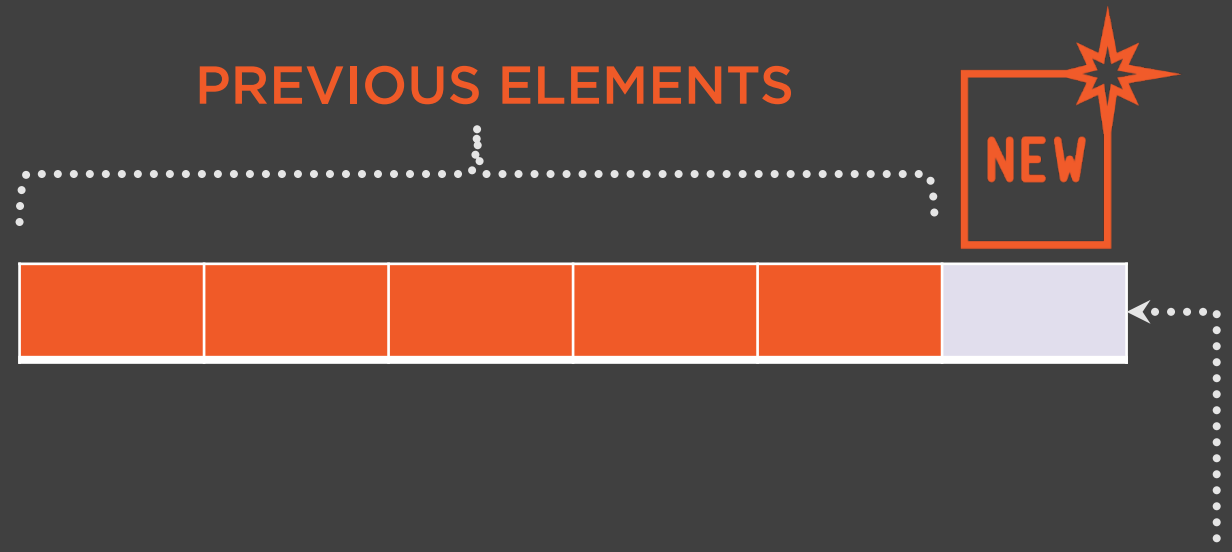
Using std::vector in Your C++ Code

Create an empty vector

```
temperatures.push_back(newTemperatureValue);
```

Using std::**vector** in Your C++ Code

Adding new elements with vector::*push_back*




```
temperatures.push_back(newTemperatureValue);
```

Using std::vector in Your C++ Code

Adding new elements with vector::*push_back*

std::vector Guarantees Efficient Growth

vector::push_back

O(1)+

Amortized Constant Time

Great performance!

To Learn More...

Introducing the Big O Notation and Asymptotic Runtime Complexity

Efficiently Searching

"Introduction to Data Structures and Algorithms in C++"

Using vector::push_back

Simple

Safe

Efficient

```
std::vector<int> v{};

std::vector<int> v; // Created empty, as well
```

Basic Operations with std::vector

Create an empty vector

INITIAL VALUES

```
std::vector<int> v{67, 79, 78, 78, 73, 69};
```

Basic Operations with std::vector

Create a vector with some initial values

COUNT ELEMENT

```
std::vector<string> v{3, "Connie"};
```

Connie	Connie	Connie

Basic Operations with std::vector

Create a vector containing ‘count’ copies of a given element

To Learn More...

Fixing a Couple of Subtle Bugs

Module on vector

“C++11 from Scratch”

```
// std::vector<int> v{67, 79, 78, 78, 73, 69};
```

```
cout << "vector v contains " << v.size() << " elements.";
```

Basic Operations with std::vector

size **returns the number of elements**

```
if (v.empty()) {
    cout << "vector is empty.";
}
```

Basic Operations with std::vector

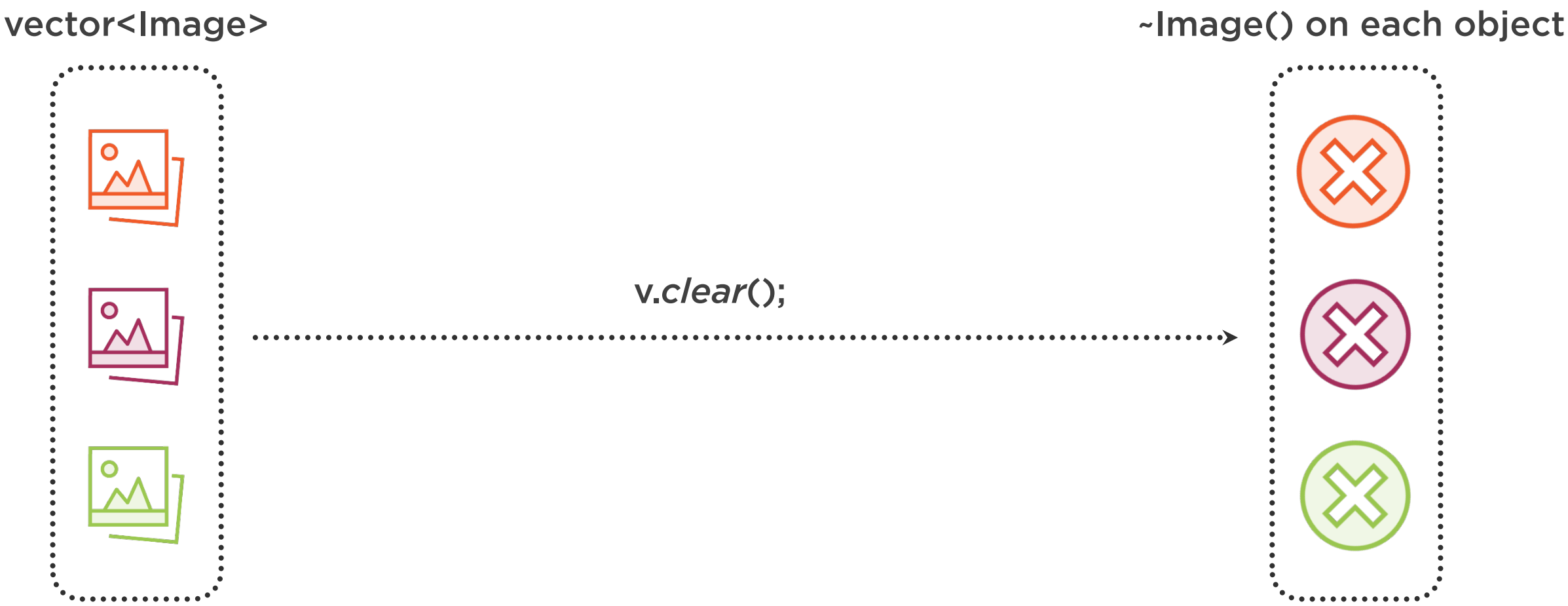
empty **checks if the vector has no elements**

```
v.clear();
// Vector is now empty
```

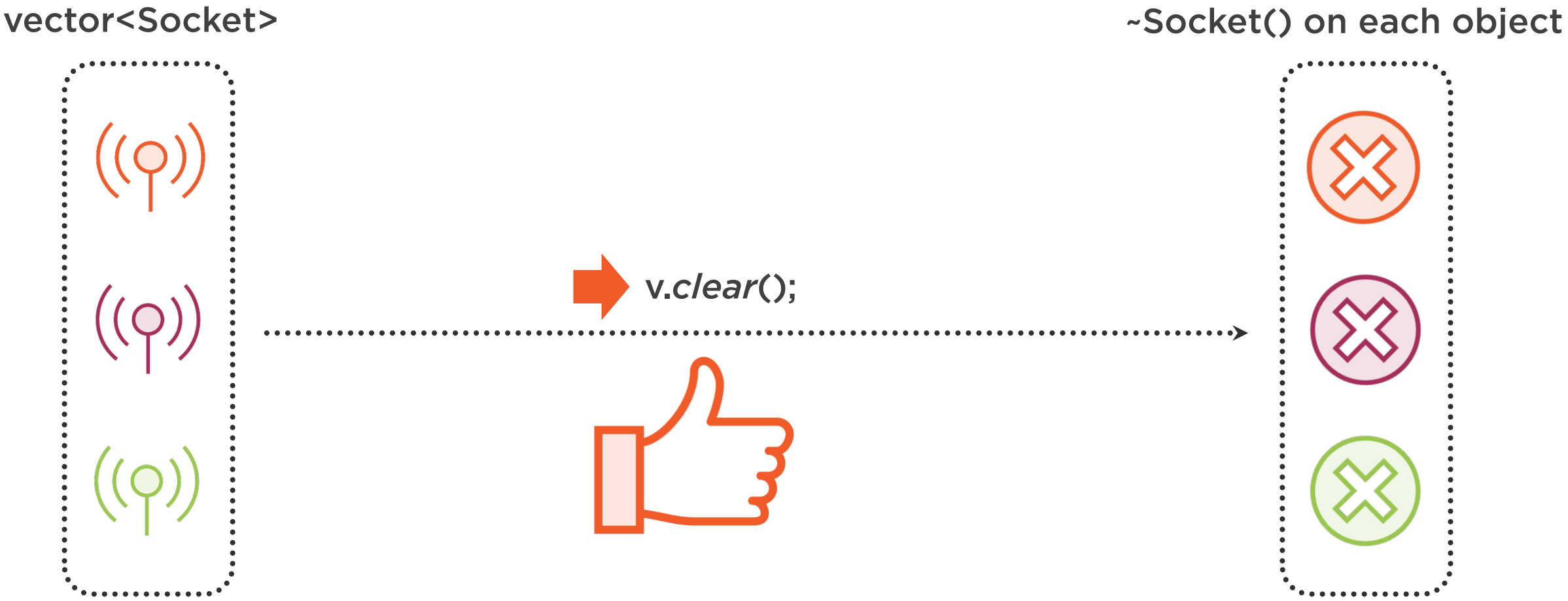
Basic Operations with std::vector

***clear* erases all elements from the vector**

std::vector of RAII Objects



std::vector of RAII Objects



```
    std::vector<Something> v;

    // . . . Some code . . .

}
```

VECTOR'S DESTRUCTOR AUTOMATICALLY INVOKED

Automatic Clean-up

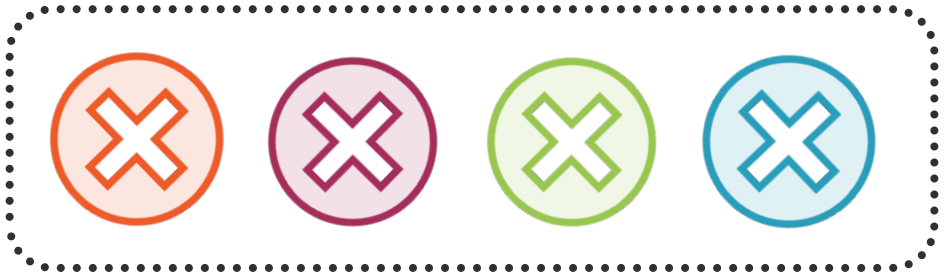
```
std::vector<Something> v;

// . . . Some code . . .

}
```

VECTOR'S DESTRUCTOR AUTOMATICALLY INVOKED

Automatic Clean-up



Destructor *automatically* invoked on each object

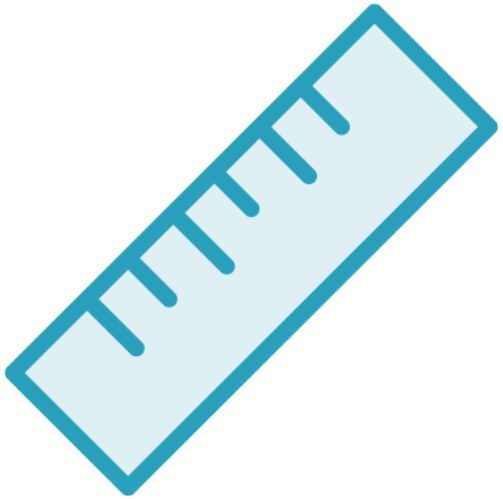
C++ RAII, Resource Managers, and Destructors

Automatic Resource Cleanup with Destructors

Defining Custom Types

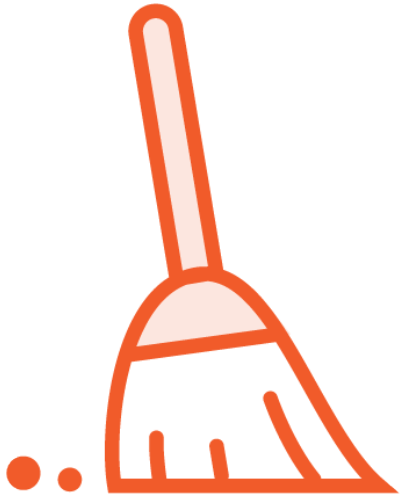
"C++11 from Scratch"

Common Method Names



size()

empty()

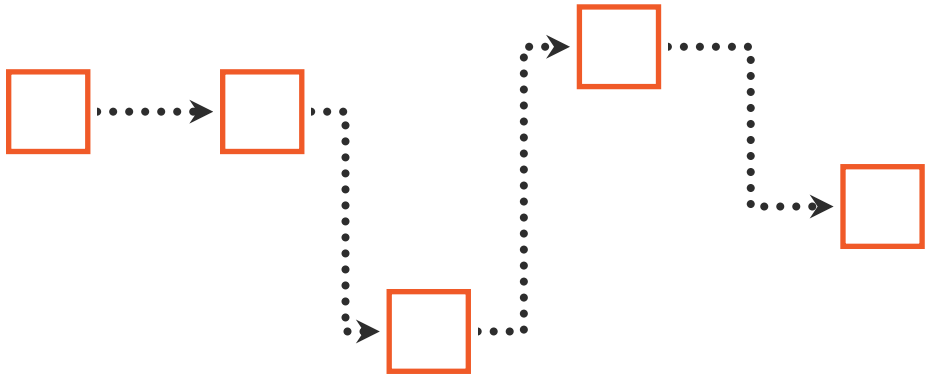


clear()

Common Method Names

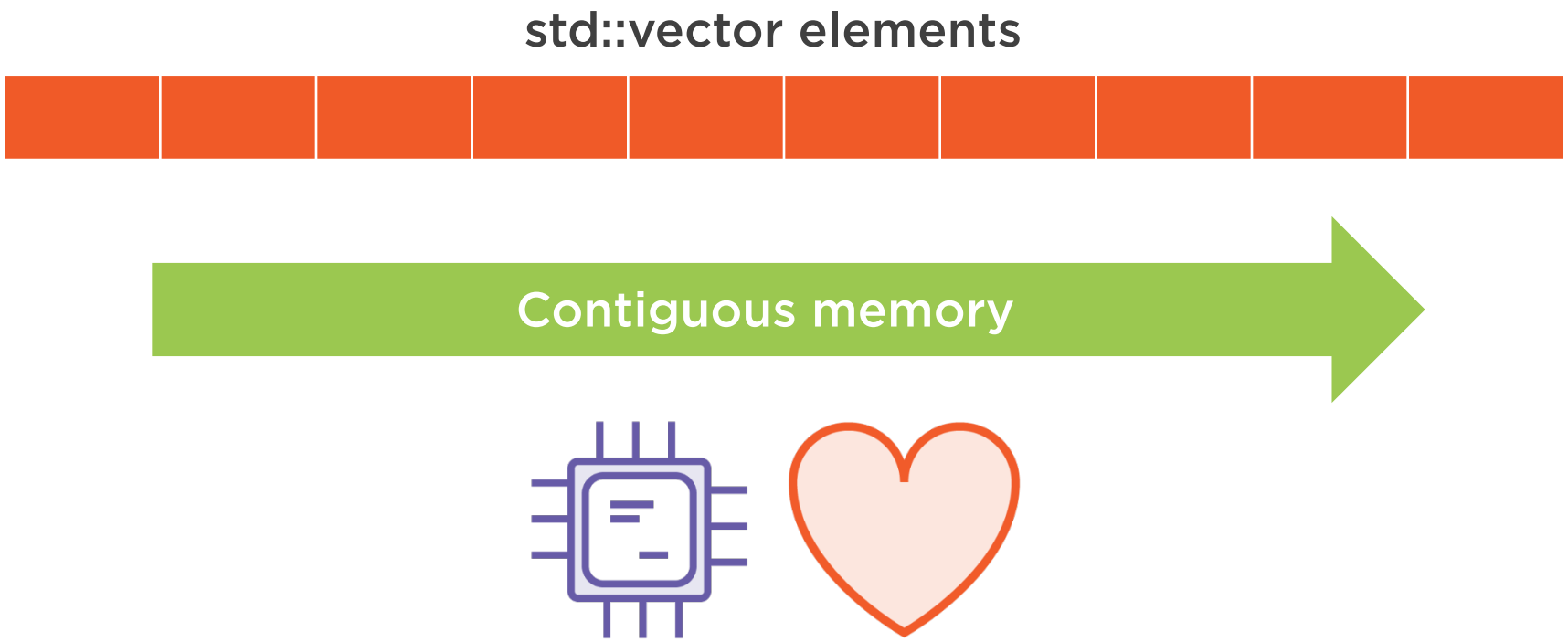
std::vector

std::list



size() ⟷ **size()**

std::vector Element Access



std::vector Element Access

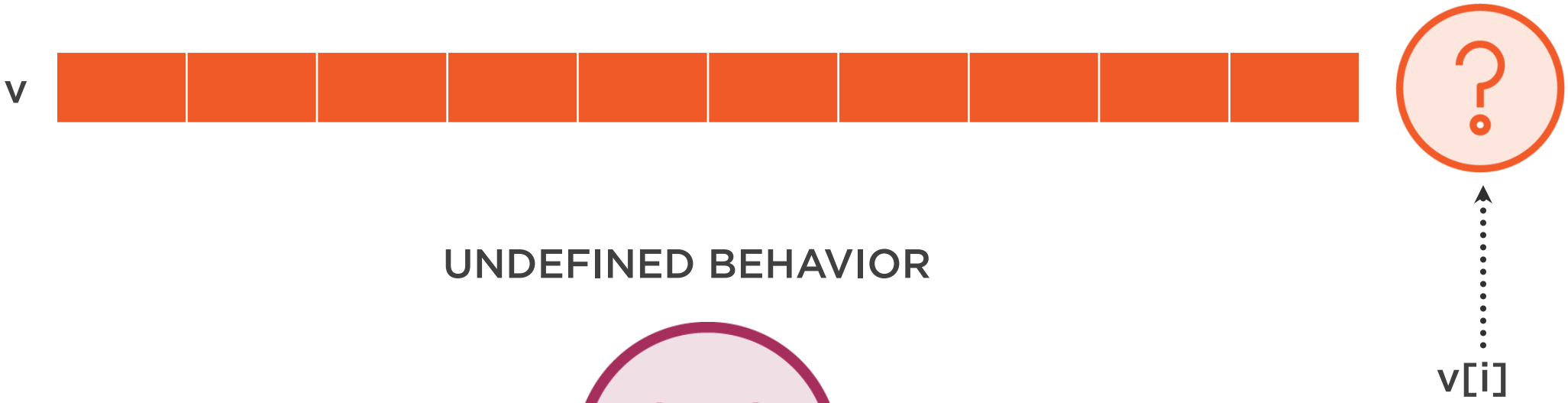
std::vector Element Access

Element indexes are 0-based

std::vector Element Access

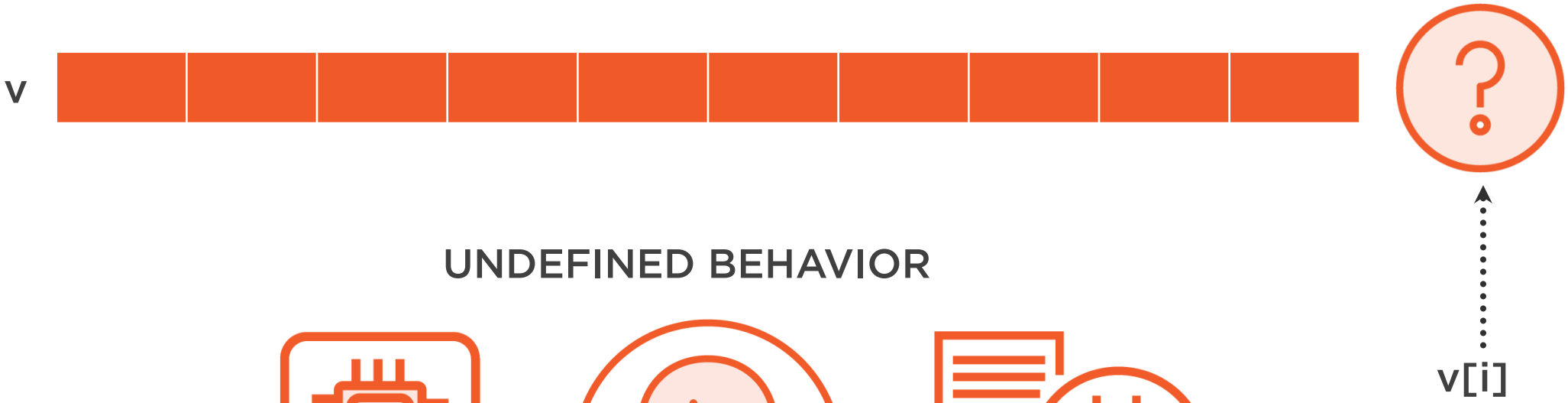
OVERLOADED OPERATOR[]

Index Out of Bounds



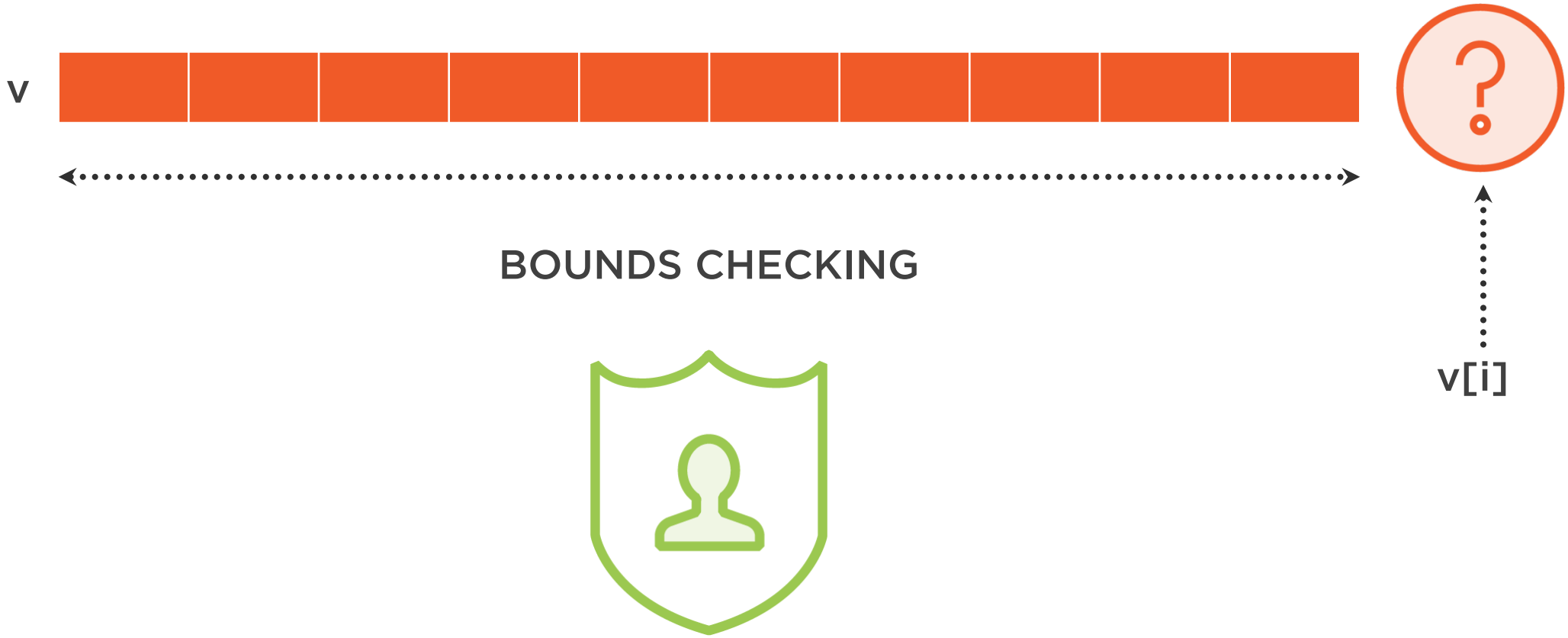
UNDEFINED BEHAVIOR

Index Out of Bounds

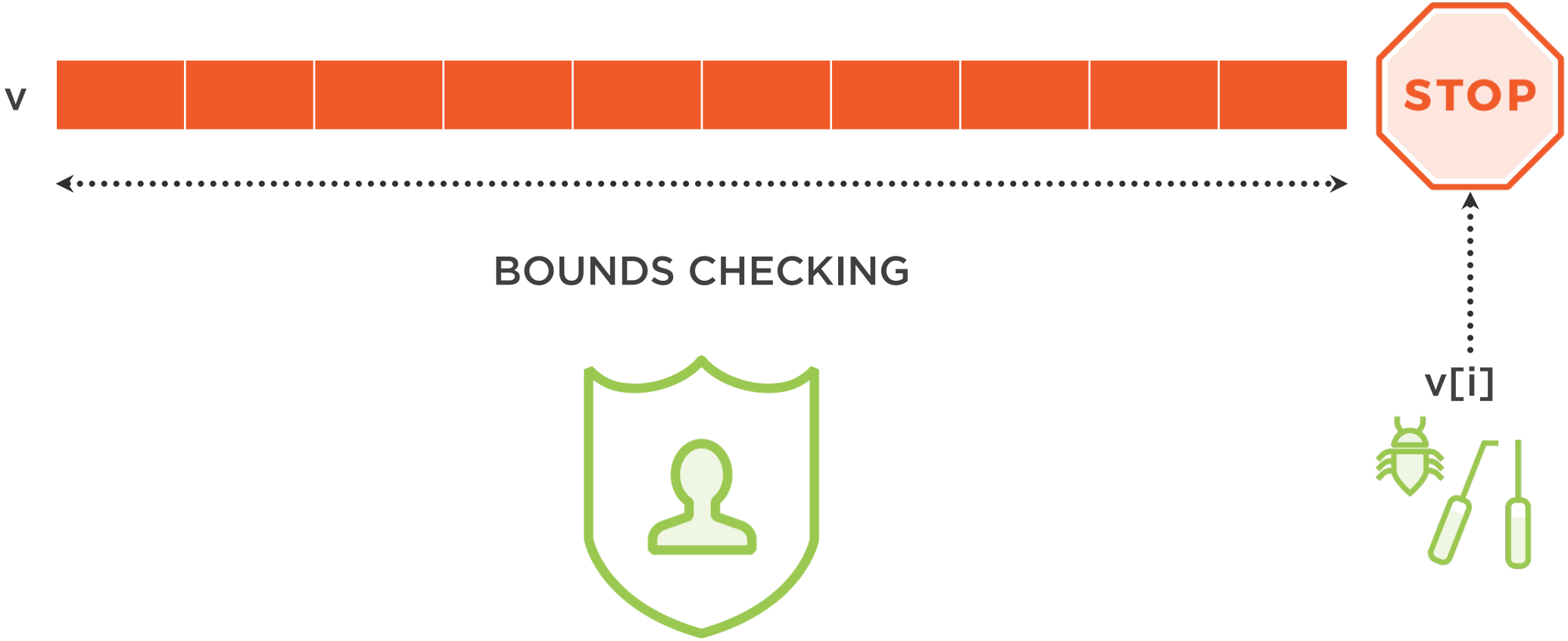


UNDEFINED BEHAVIOR

Index Out of Bounds



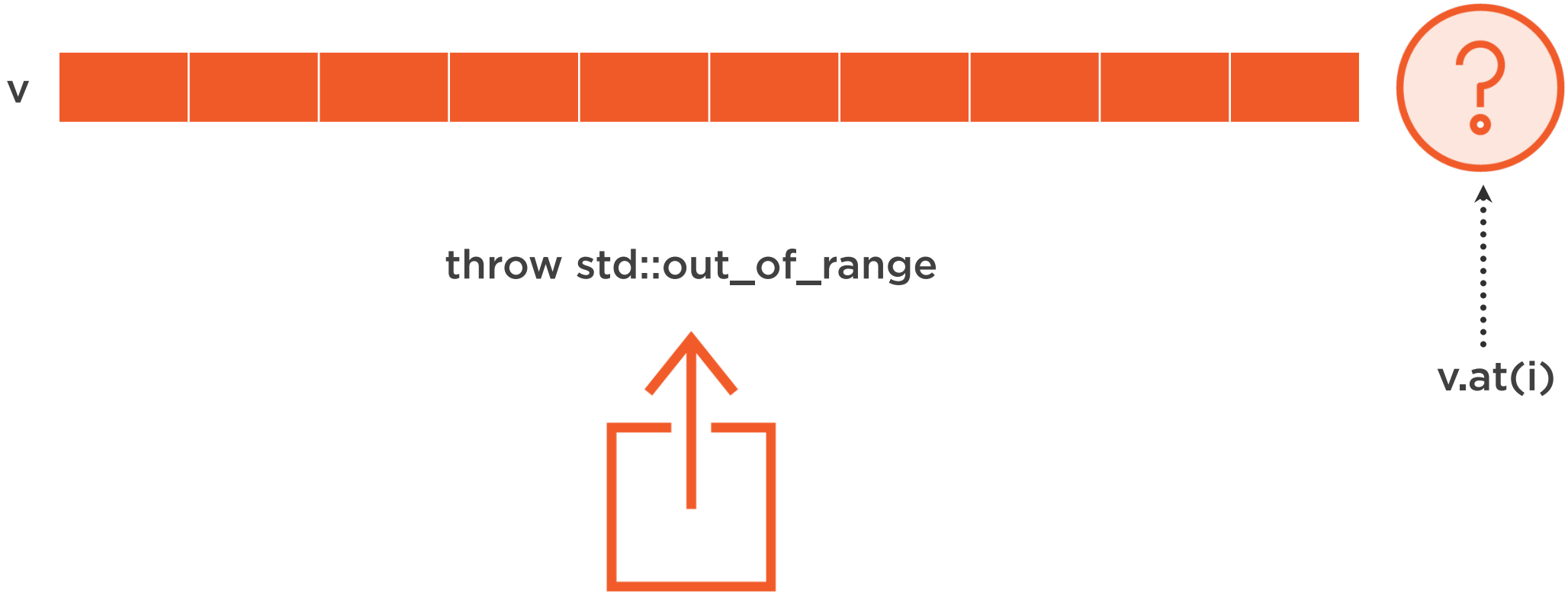
Index Out of Bounds



Element Access with vector::at

at : Index *always* bounds-checked

Index Out of Bounds

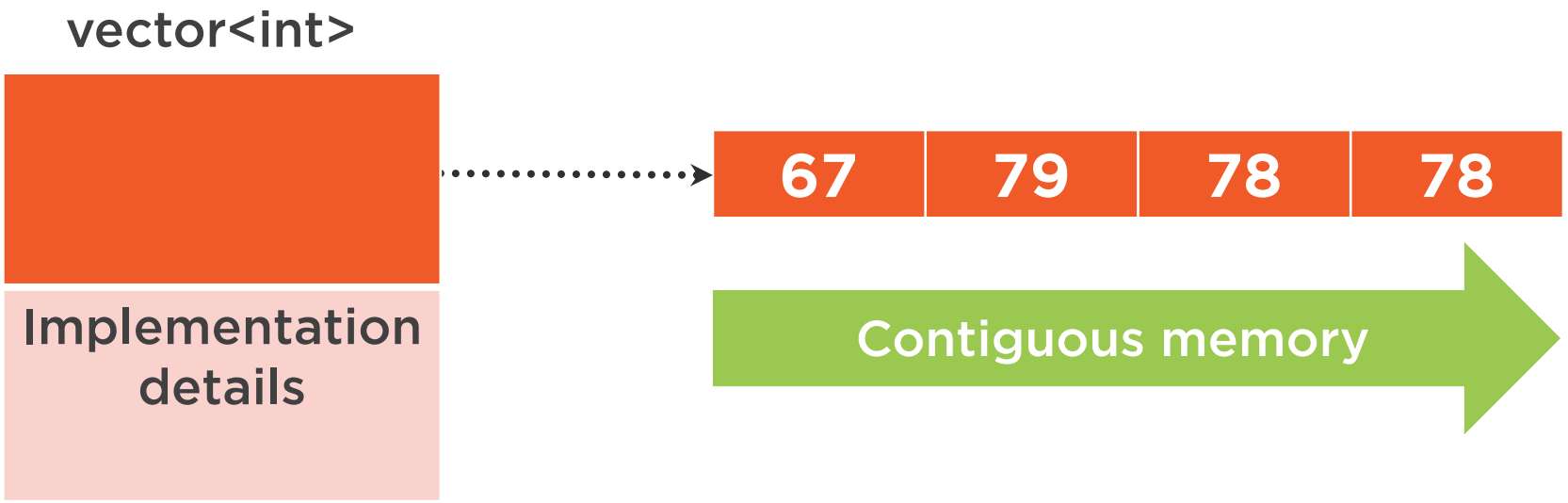


vector::at vs. vector::operator[]

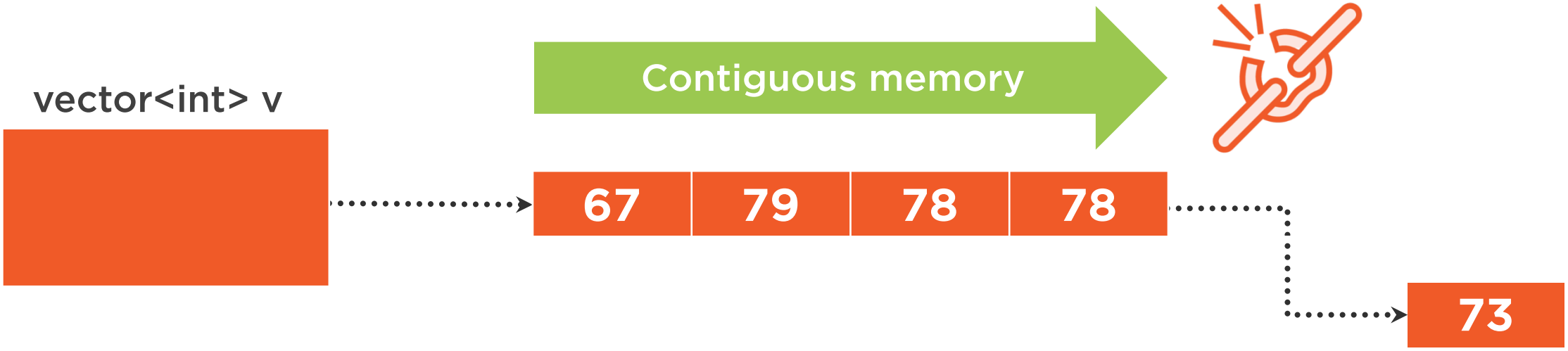
Bounds-checking

Slower element access

Simplified Model for std::vector

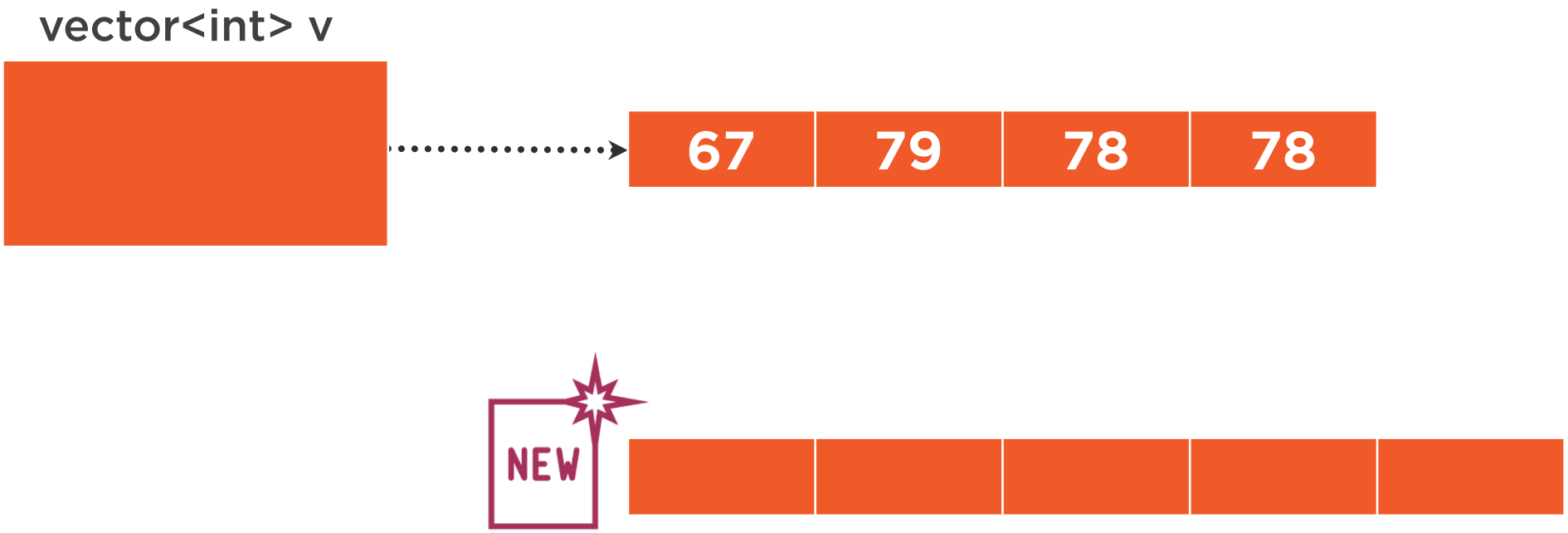


Adding a New Element with push_back



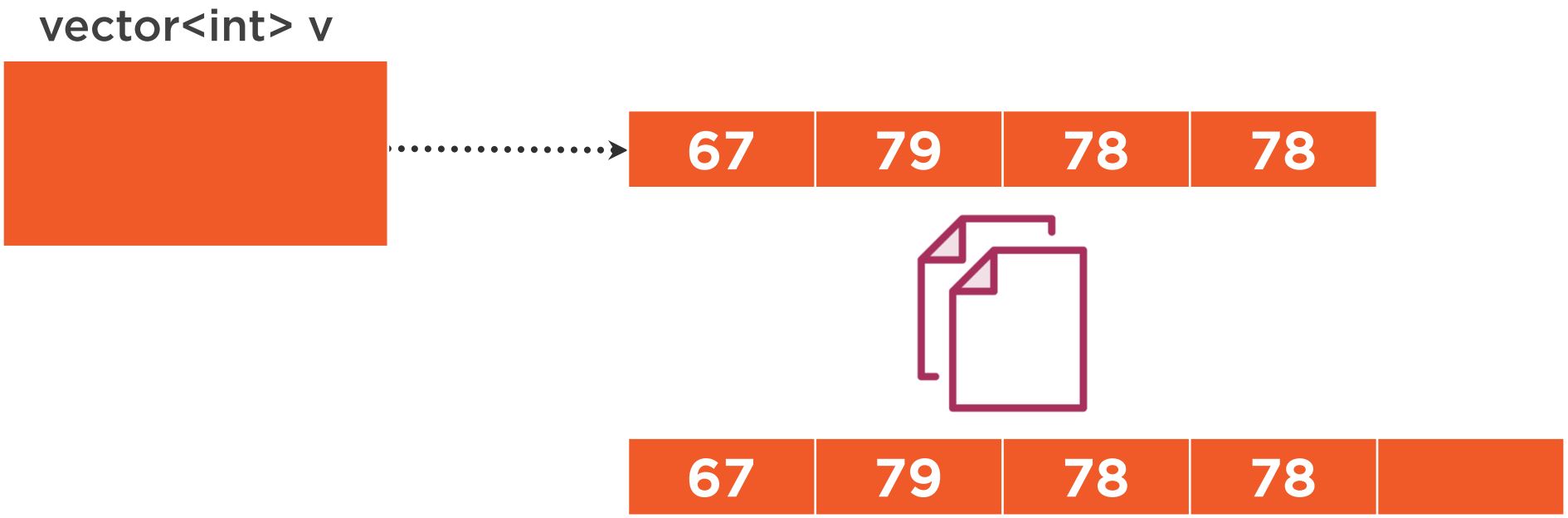

```
v.push_back(73);
```

Adding a New Element with push_back



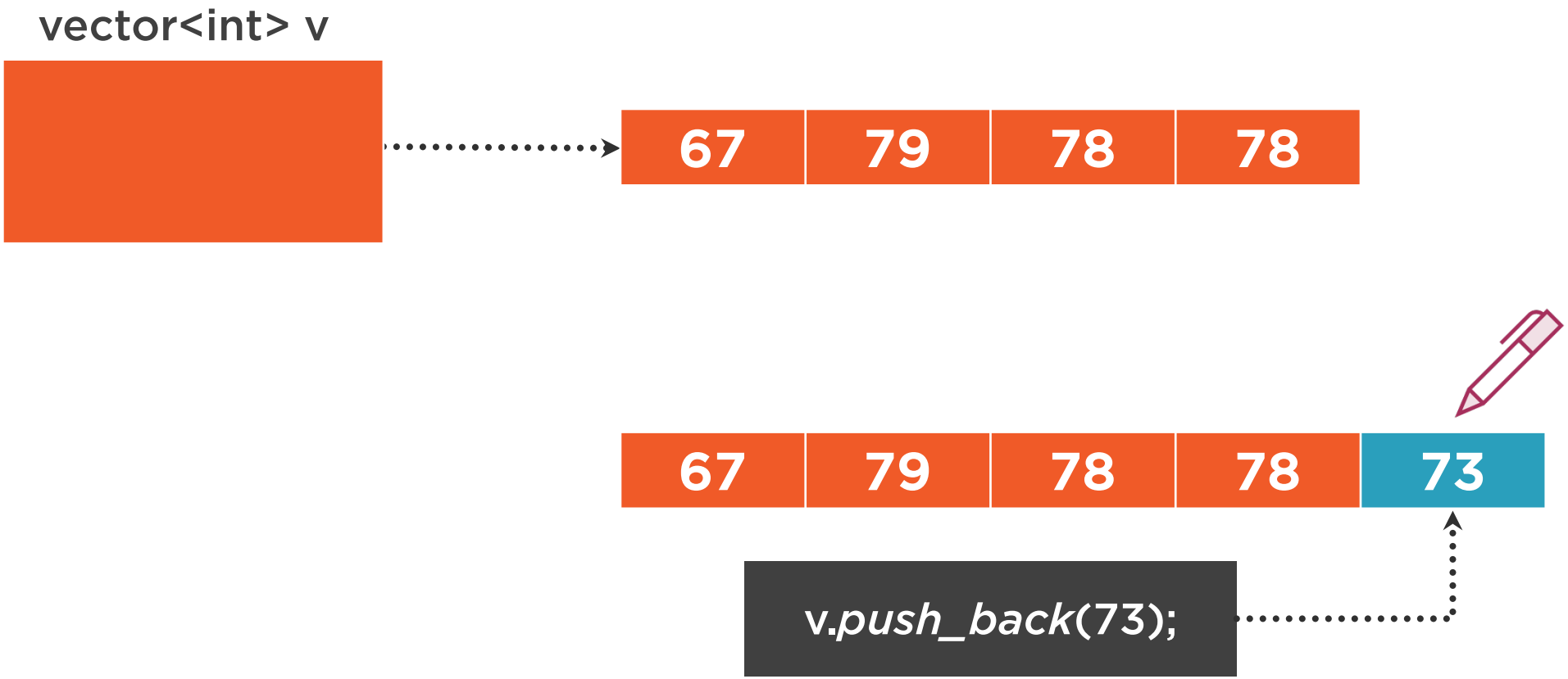

```
v.push_back(73);
```

Adding a New Element with push_back

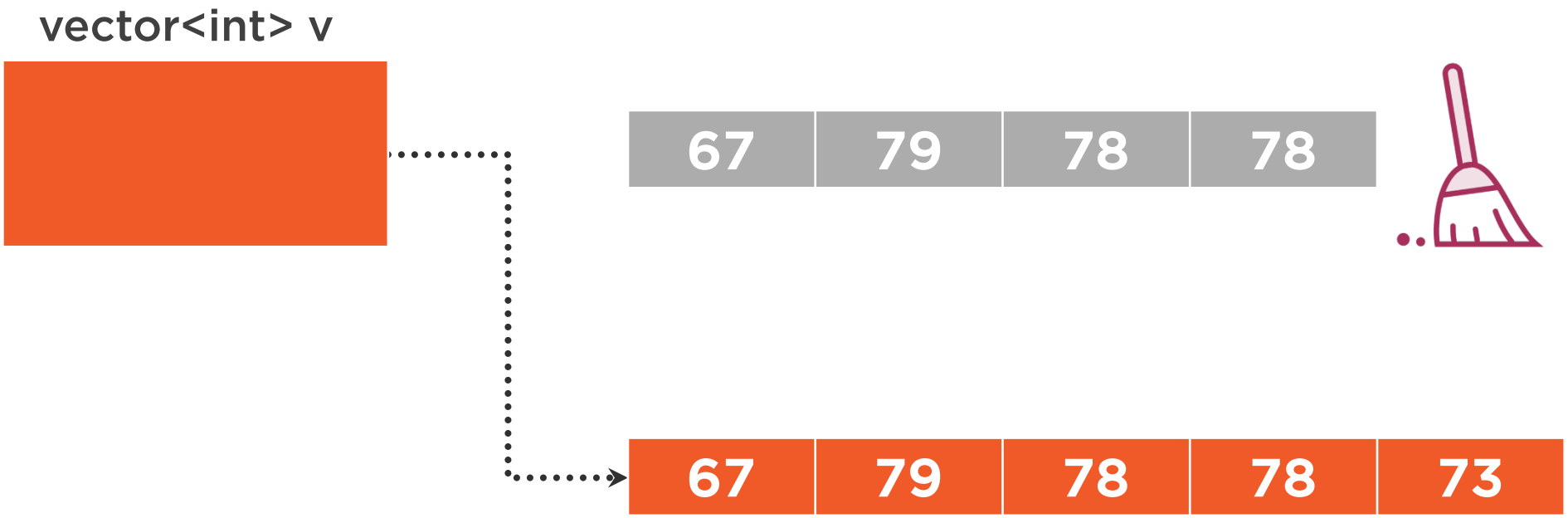



```
v.push_back(73);
```

Adding a New Element with push_back

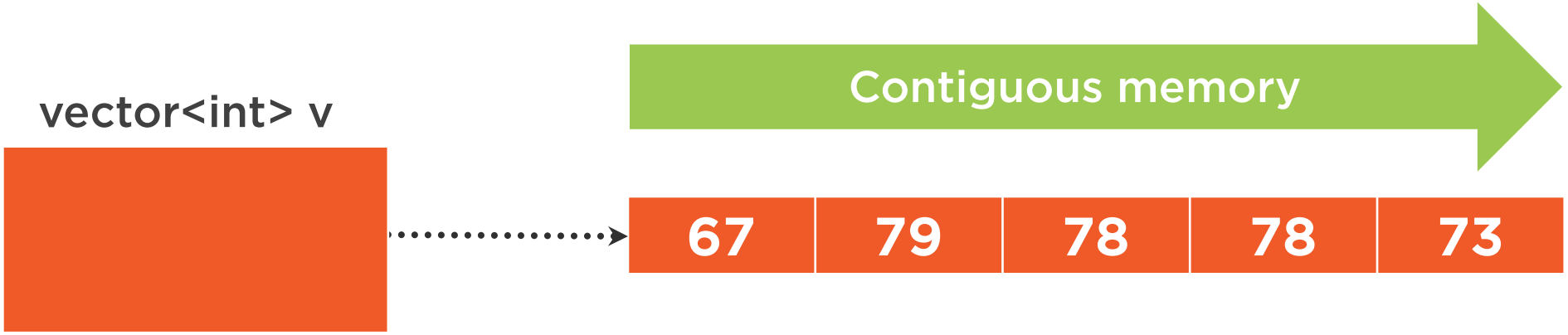


Adding a New Element with push_back



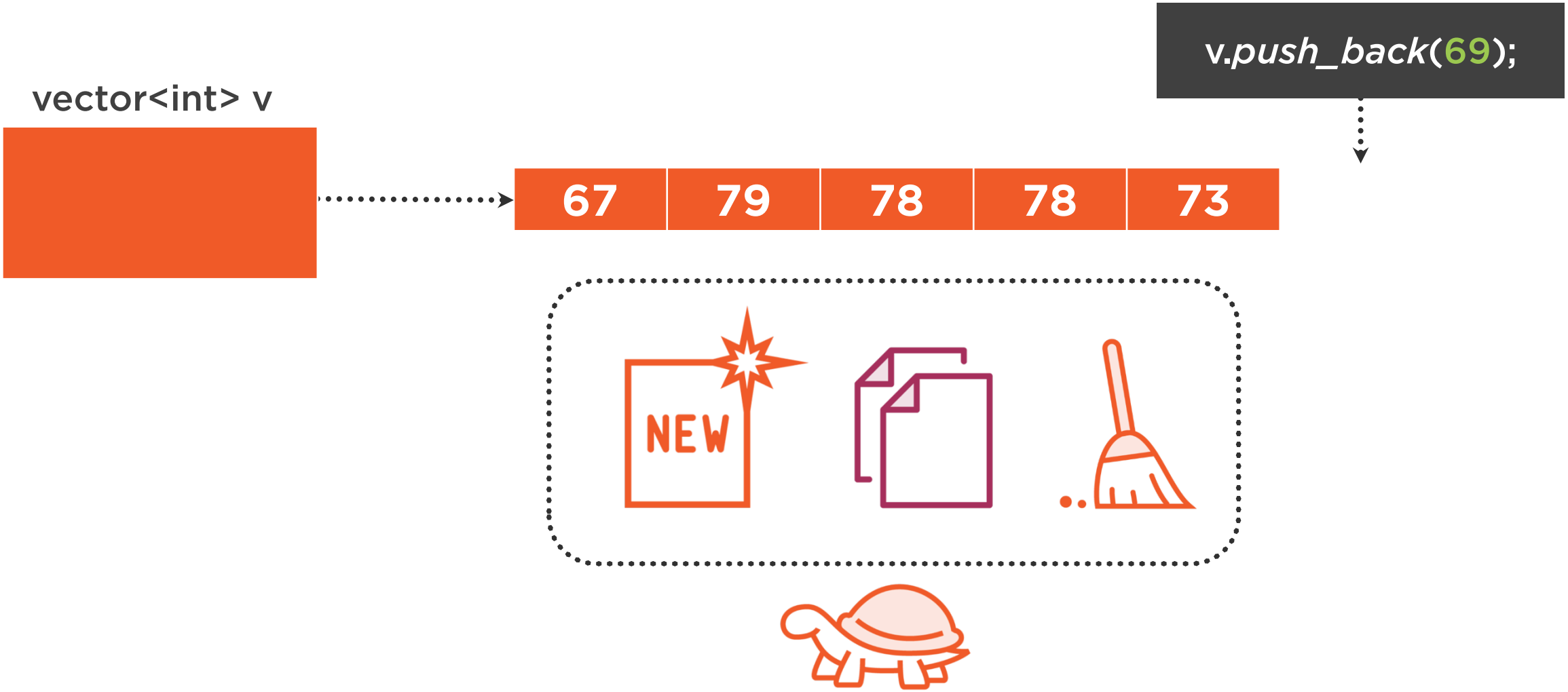
v.*push_back*(73);

Adding a New Element with push_back

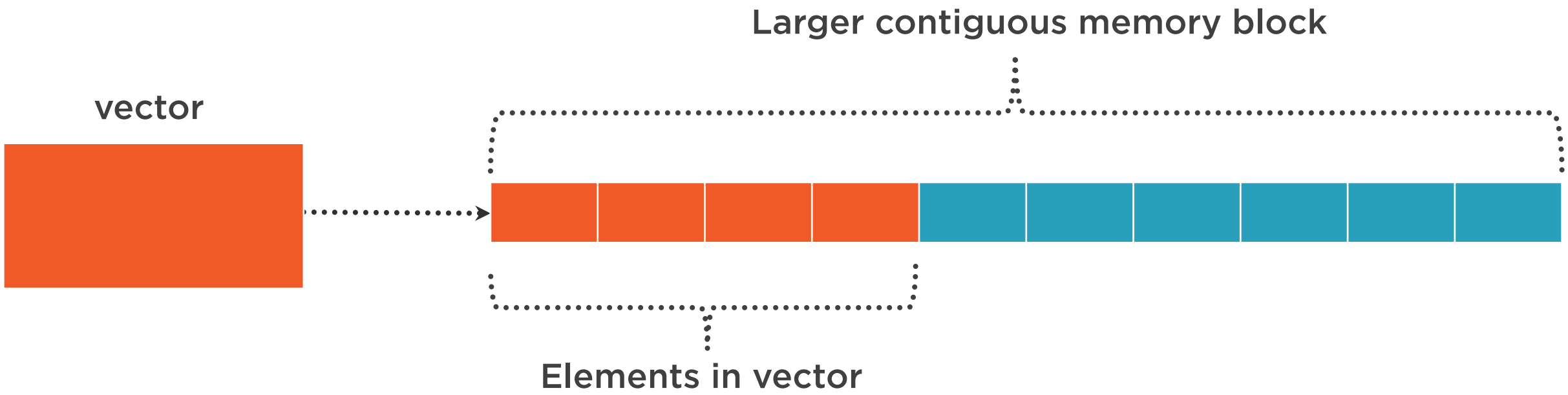



```
v.push_back(73);
```

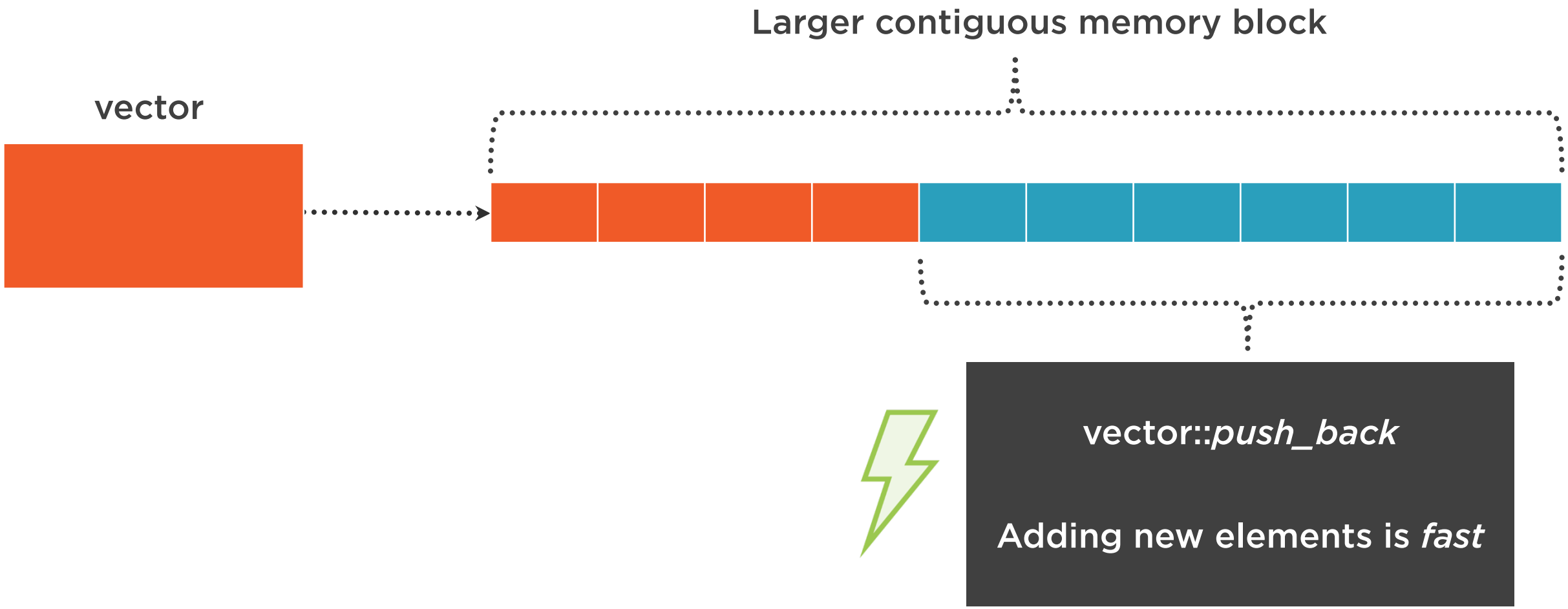
Adding a New Element with push_back



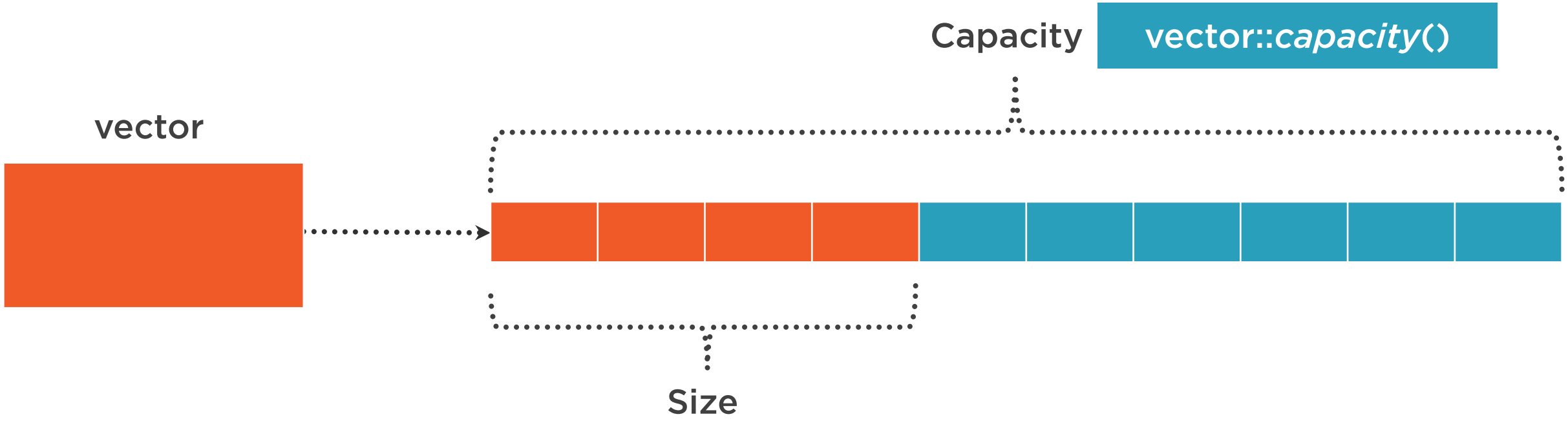
std::vector Pre-allocates a Larger Block



std::vector Pre-allocates a Larger Block

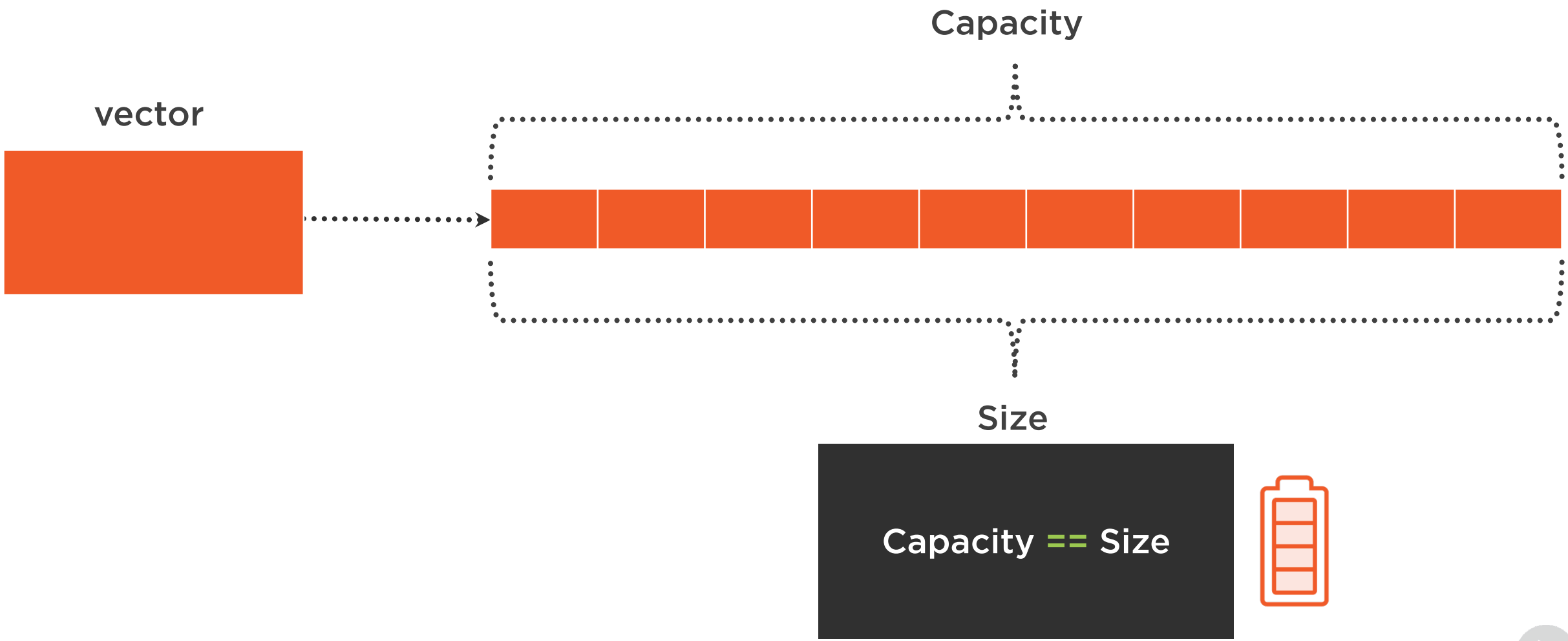


std::vector's Capacity vs. Size

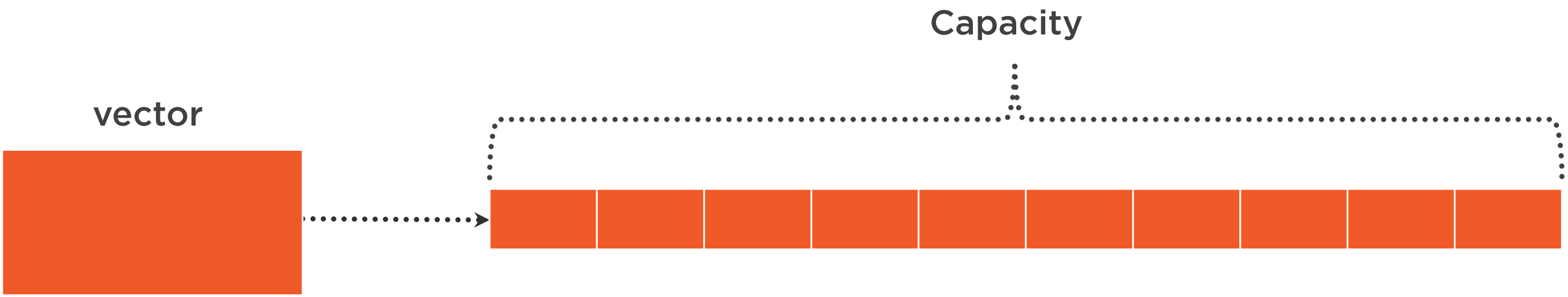


Capacity >= Size

std::vector's Capacity vs. Size

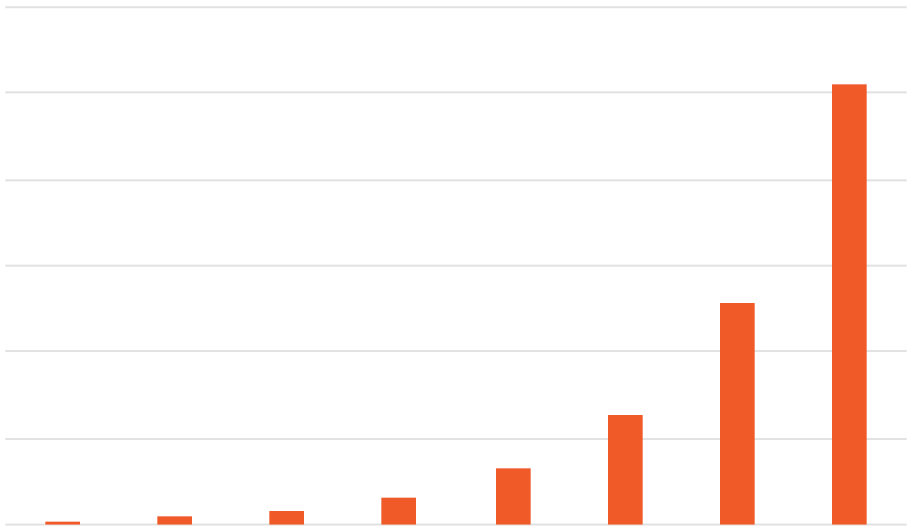


std::vector's Capacity vs. Size



vector::*push_back*

Smart Capacity Growth Policy



Algorithm for calculating new capacity makes re-allocations *rare*

Smart Capacity Growth Policy

Initial vector

N

After reallocation

k*N

(k>1)

k = 2
k = 1.5

Algorithm for calculating new capacity makes re-allocations *rare*

Smart Capacity Growth Policy

Initial vector

N

After reallocation

k*N

(k>1)

push_back

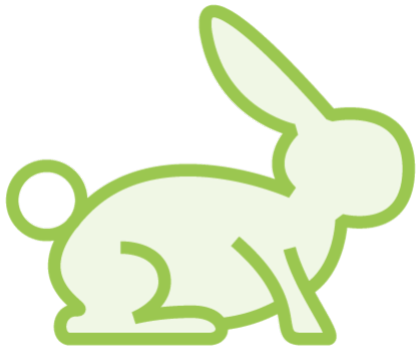
Amortized Constant

O(1)+

Algorithm for calculating new capacity makes re-allocations *rare*

Comparing Different Growth Algorithms

Good:
k*N



O(1)+

Poor:
k+N

N = 1,000

N^2
→ 1,000,000

O(N^2)

Performance Tip

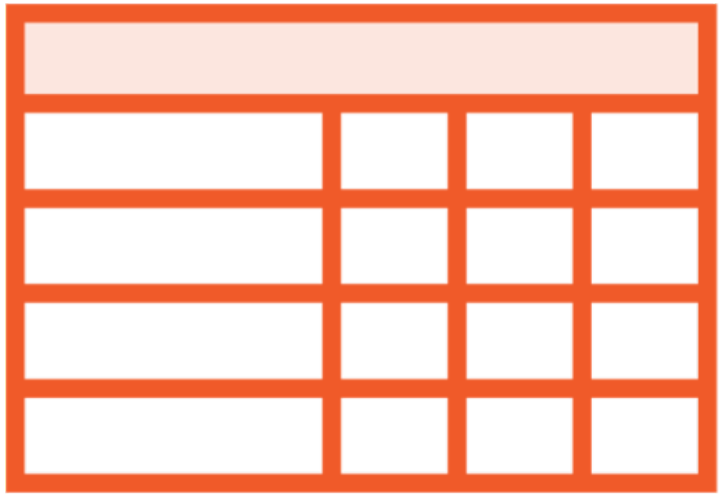
Estimate of
element count

Preallocate
storage

```
vector::reserve( capacity )
```

Example: Optimizing 3D Model Loading

3D model



3D coordinates

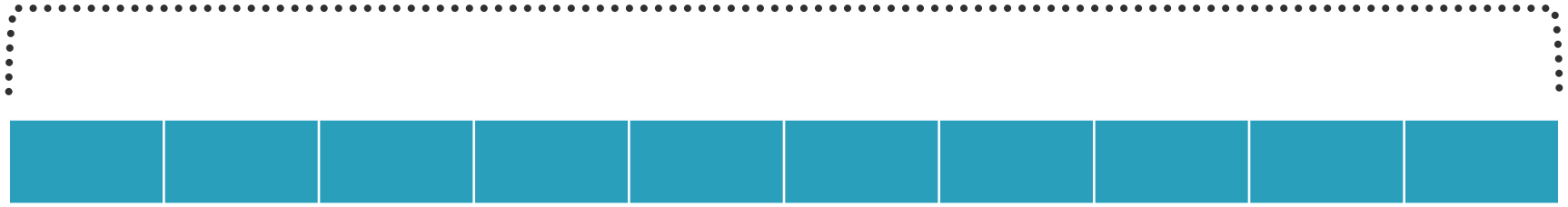
Example: Optimizing 3D Model Loading

Estimate of vertex count

3D Model File Format
Header Section
TotalVertexCount

Perf Tip: Reserve Enough Capacity

Pre-allocated storage



vector::reserve(*capacity*)

Speed up *push_back* time

Summary

std::vector: a powerful flexible array

Safe and automatic memory/resource management

Contiguous elements with safe and fast integer-index access

Important vector operations

Growth policy: size vs. capacity

- Efficient push_back