How to Choose the Right Standard Library Container

And Why You Should Want Some More

CppCon 2019

Alan Talbot
cpp@alantalbot.com

September 17, 2019

A Tale of Letters

A Tale of Letters

1980's

Minicomputer – 64 KB RAM

A Tale of Letters

1980's

Minicomputer – 64 KB RAM

1990's

Mac SE – 4 MB RAM

A Tale of Letters

1980's

Minicomputer – 64 KB RAM

1990's

Mac SE – 4 MB RAM

2000's

32 bit PC – 4 GB RAM

A Tale of Letters

1980's

Minicomputer – 64 KB RAM

1990's

Mac SE – 4 MB RAM

2000's

32 bit PC – 4 GB RAM

2010's

64 bit 8 core – 32 GB RAM

Program Runs Slow

- Simulation took 7.25 hours on my fast 2019 workstation
  - It was a lot slower on the customer's laptop

Container Variety

- Each container has different tradeoffs between speed and space and convenience
- There is a specific set of containers in the Standard Library which have withstood the test of time
  - Excepting perhaps vector<bool>
- New containers are regularly proposed which offer different speed/space/convenience tradeoffs

Container Choice

- **Vector:** contiguous, fast iteration, random access, growth invalidates everything, geometric growth behavior, reasonable overhead
- **Deque:** sort of contiguous, pretty fast iteration, random access, growth invalidates iterators only, chunk linear growth behavior, large overhead for small sizes
- **List:** noncontiguous, slow iteration, linear access, elements never move, fine linear growth, low overhead for large elements

Obvious Choice?

- **vector**
  - Has **push_back** and **pop_back** because they are (usually) *O*(1)
  - Doesn't have **push_front** and **pop_front** because they are *O*(n)
  - And **insert** is *O*(n)
  - Contiguous memory – fast iteration
- **deque**
  - Has **push_back** and **pop_back** because they are (usually) *O*(1)
  - Has **push_front** and **pop_front** because they are (usually) *O*(1)
  - But **insert** is *O*(n)
  - More or less contiguous memory – pretty fast iteration

Obvious Choice?

- **list**
  - Has **push_back** and **pop_back** because they are *O*(1)
  - Has **push_front** and **pop_front** because they are *O*(1)
  - And **insert** is *O*(1)
  - Noncontiguous memory – slow iteration

Obvious Choice?

- Need to insert and erase fast anywhere in a container?
  - **list** offers this ability
  - **vector** is faster if the container size is fairly small and it uses far less memory
- Need to store millions of records and access them fast?
  - **vector** offers this ability
  - **deque** is also fast to iterate and wastes far less memory if the container is large
  - **vector** can't be used at all if its size is a significant percentage of total memory
- Need to sort objects which can't be moved or copied?
  - **list** offers this ability
  - **list** is quite fast to iterate if elements are in cache

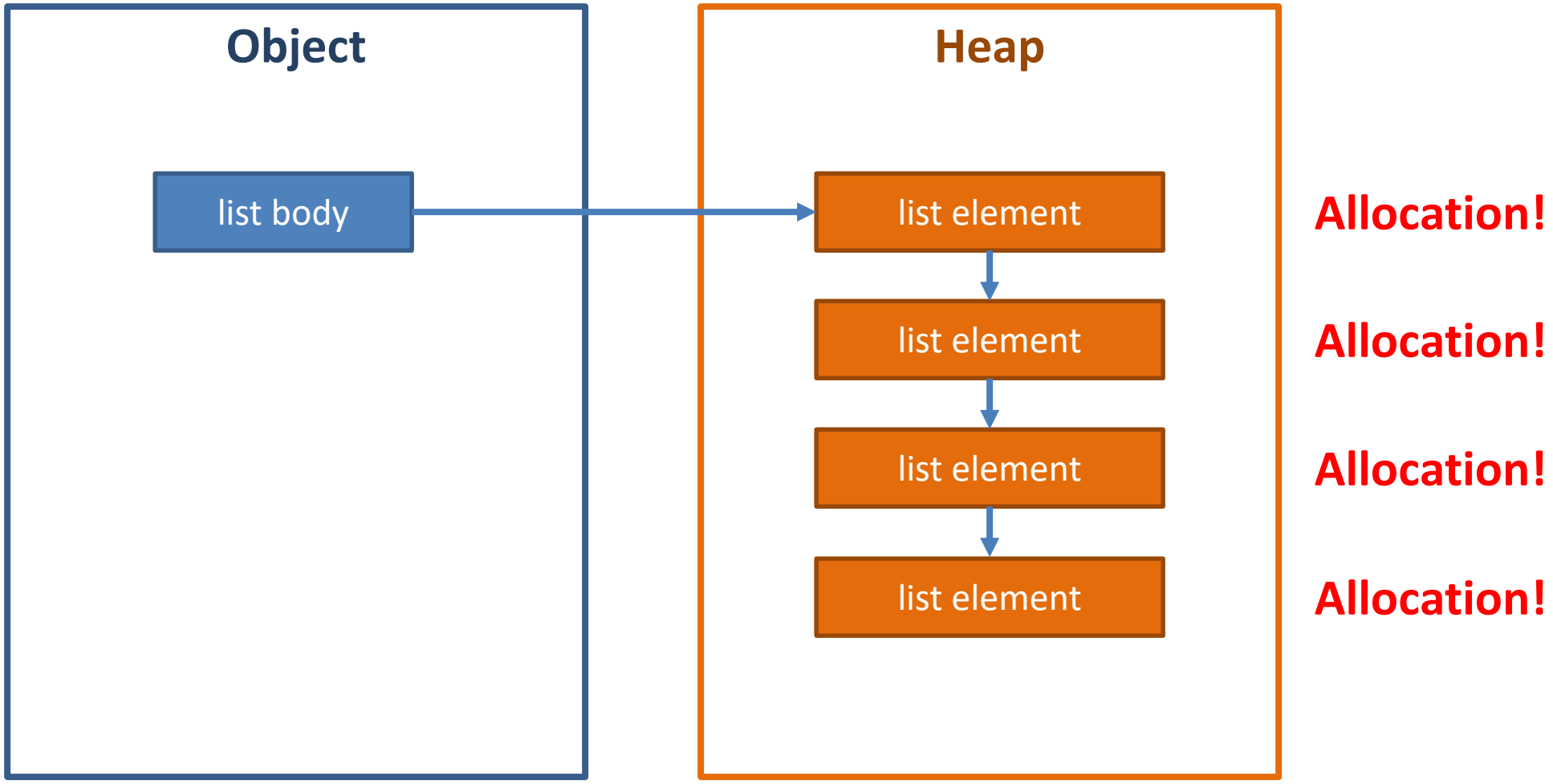
The Queue

- Suppose we want a number of objects, each with a small FIFO buffer
  - Assume that the buffer is typically small (around 10)
  - But it can't have a fixed max size because it can occasionally be much larger
  - What is the best way to implement this?
- The **queue** container adaptor is the Standard Library FIFO solution
  - Requires **push_back** and **pop_front**
  - So works (only) on **list** and **deque**

The Queue

```
vector<widget> widgets(count);
for (int i = 0; i < reps; ++i)
{
      for (auto& widget : widgets)
            for (int n = 0; n < adds; ++n) // 1 < adds < size
                  widget.push(n);
      for (auto& widget : widgets)
            for (int n = 0; n < subs; ++n) // 1 < subs < size
                  if (!widget.empty())
                        widget.pop();
}
```

The Queue – List



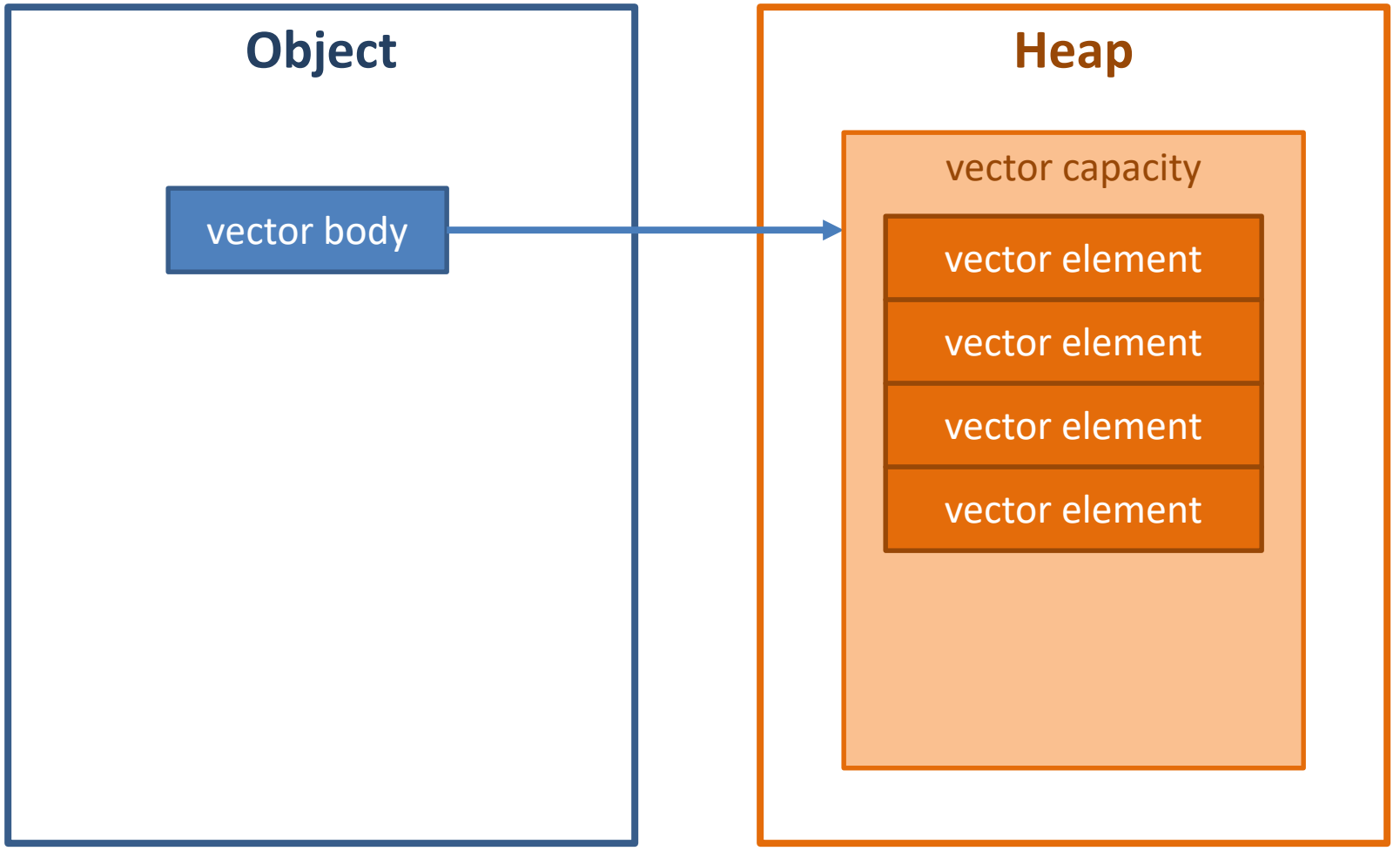
The Queue – List

```
struct widget {
    void push(int i) { fifo.push_back(i); }
    int pop() { int i = fifo.front(); fifo.pop_front(); return i; }
    bool empty() const { return fifo.empty(); }
    list<int> fifo;
};
```

The Queue – List

Container	Count	Reps	Size	Time (s)	Memory (MB)
List	10000	10000	10	35.4	95.7

The Queue – Vector



1 Allocation

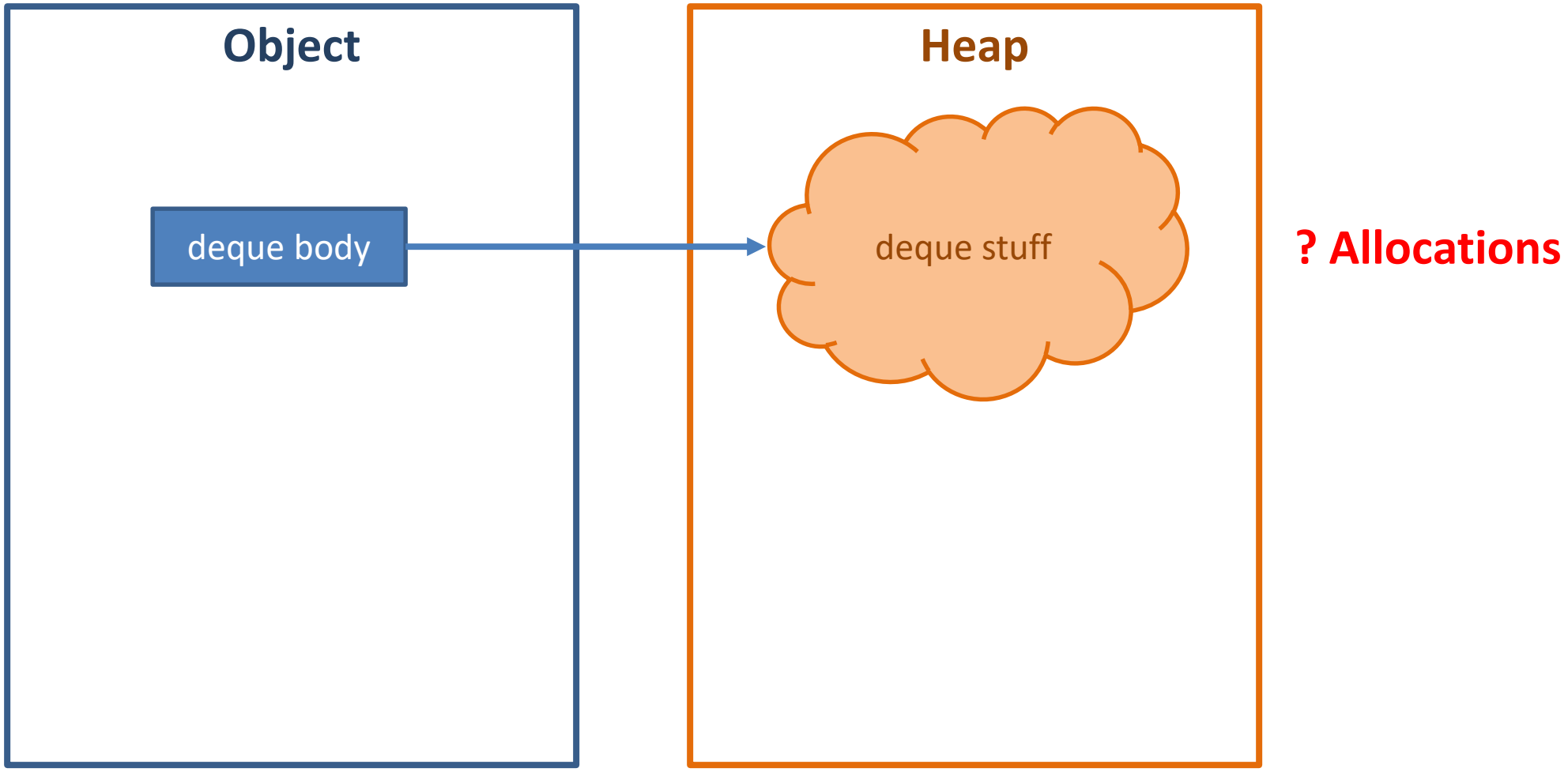
The Queue – Vector

```
struct widget {
    widget() { fifo.reserve(size); }
    void push(int i) { fifo.push_back(i); }
    int pop() {
        int i = fifo.front();
        fifo.erase(fifo.begin());
        return i;
    }
    bool empty() const { return fifo.empty(); }
    vector<int> fifo;
};
```

The Queue – Vector

Container	Count	Reps	Size	Time (s)	Memory (MB)
List	10000	10000	10	35.4	95.7
Vector	10000	10000	10	10.3	19.6

The Queue – Deque



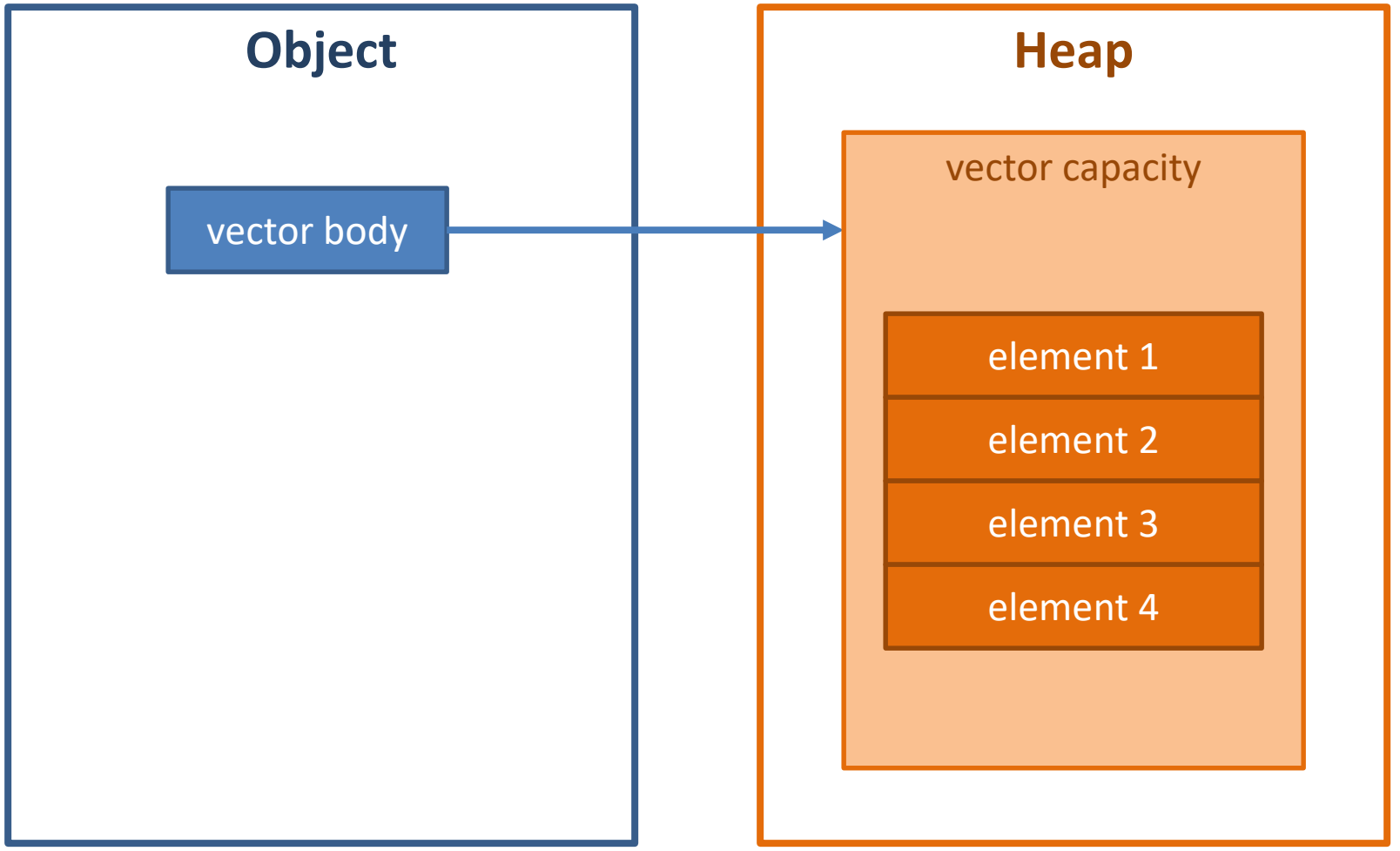
The Queue – Deque

```
struct widget {
    void push(int i) { fifo.push_back(i); }
    int pop() { int i = fifo.front(); fifo.pop_front(); return i; }
    bool empty() const { return fifo.empty(); }
    deque<int> fifo;
};
```

The Queue – Deque

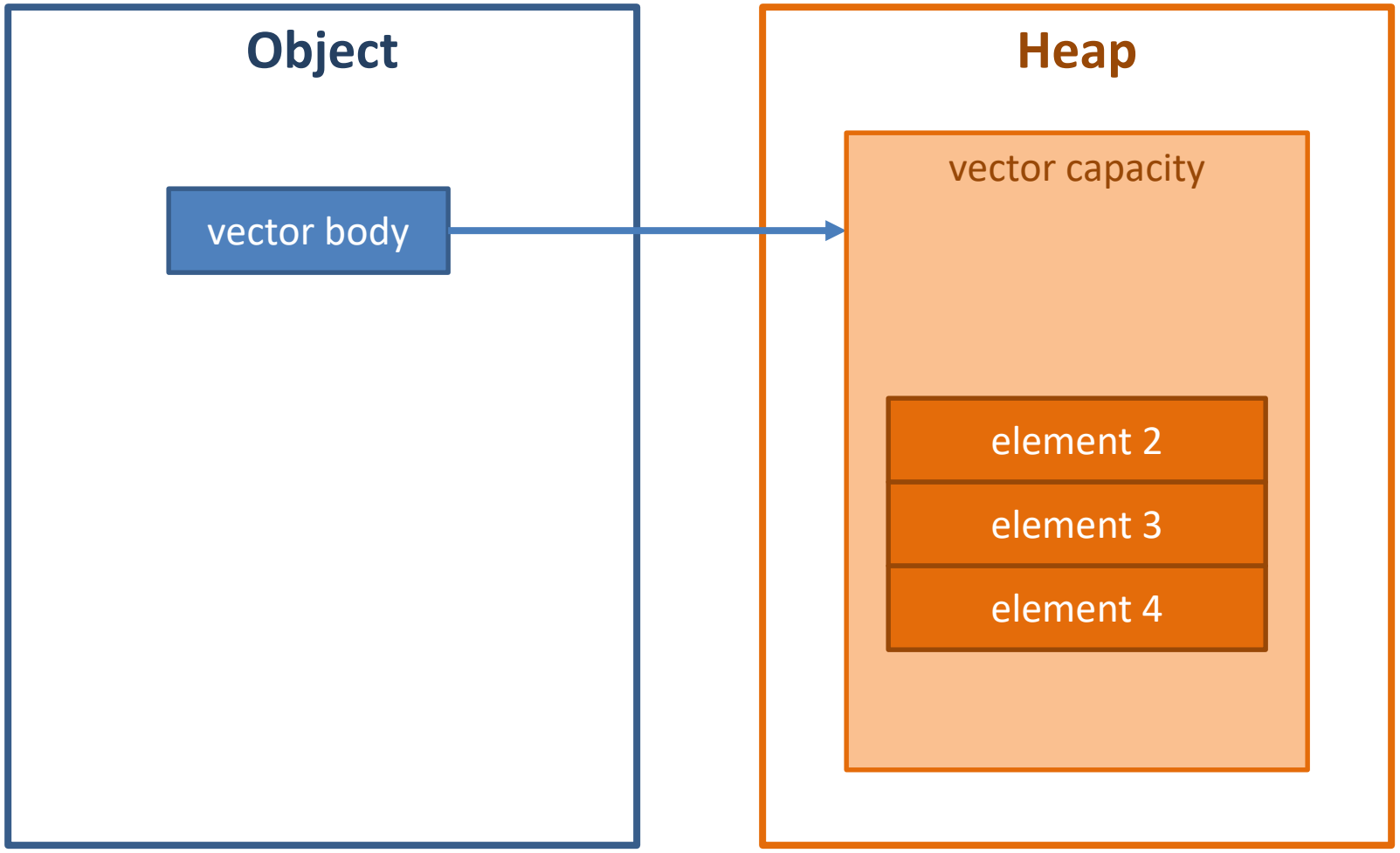
Container	Count	Reps	Size	Time (s)	Memory (MB)
List	10000	10000	10	35.4	95.7
Vector	10000	10000	10	10.3	19.6
Deque	10000	10000	10	5.7	55.9

The Queue – Shifty Vector

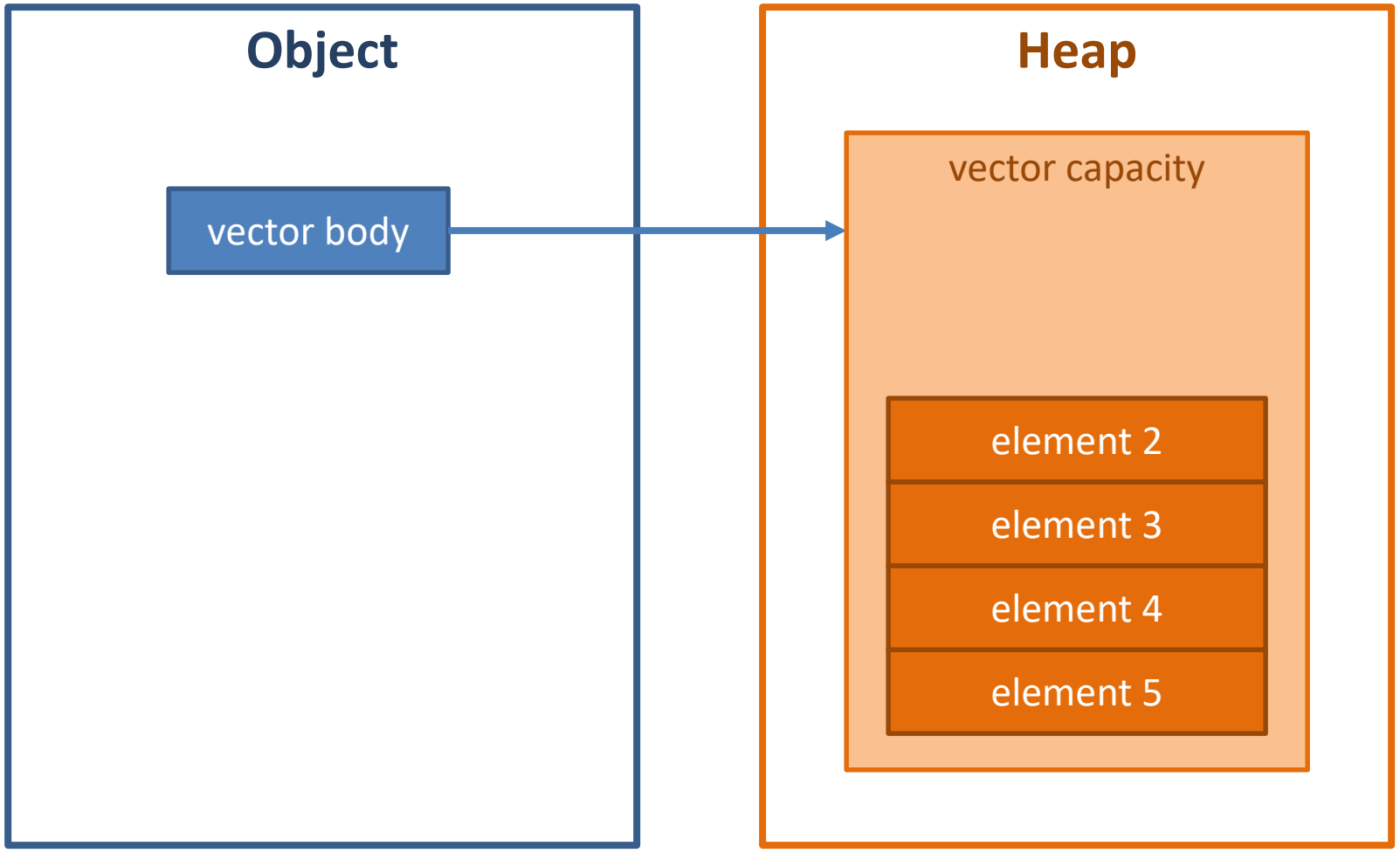


1 Allocation

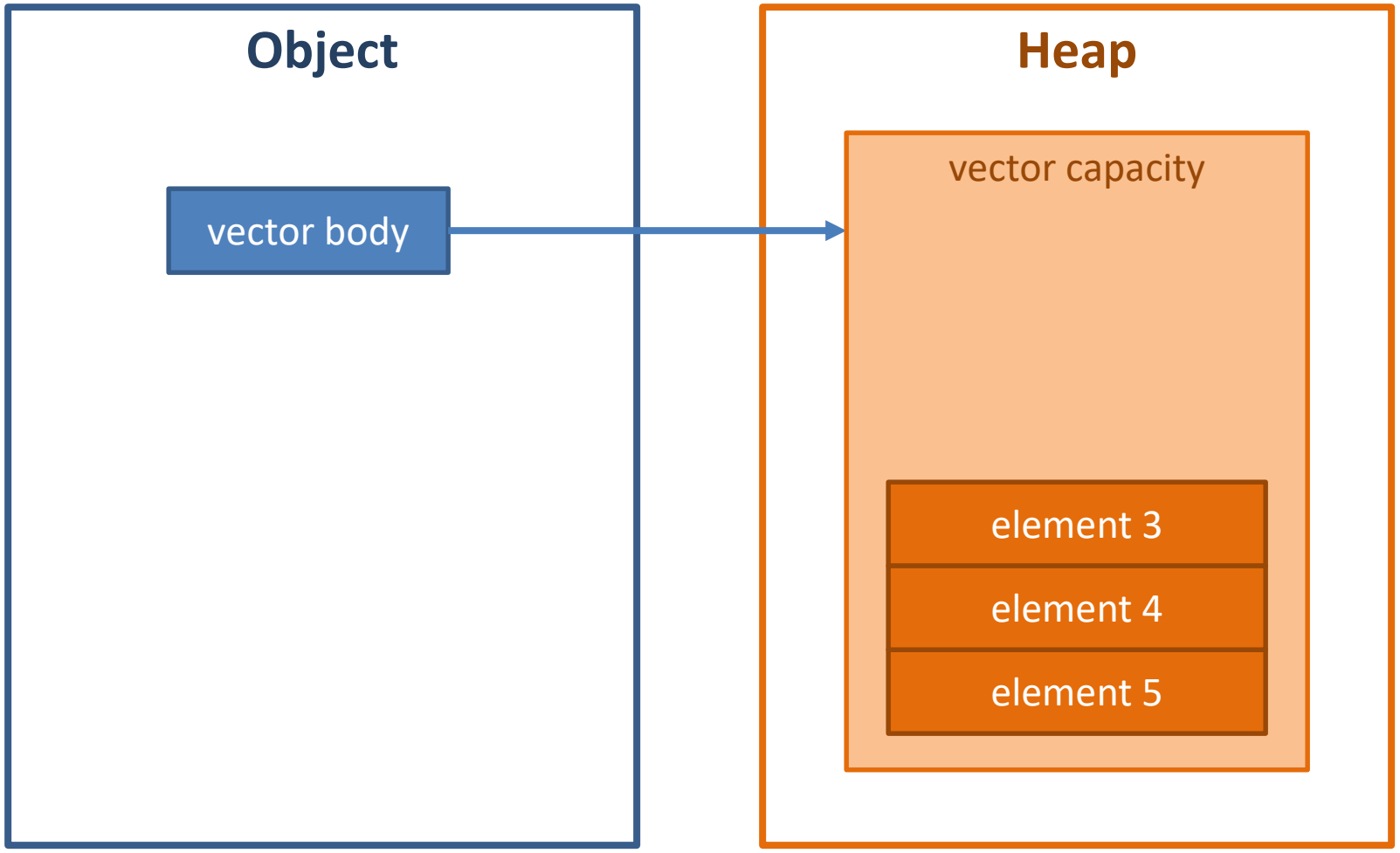
The Queue – Shifty Vector



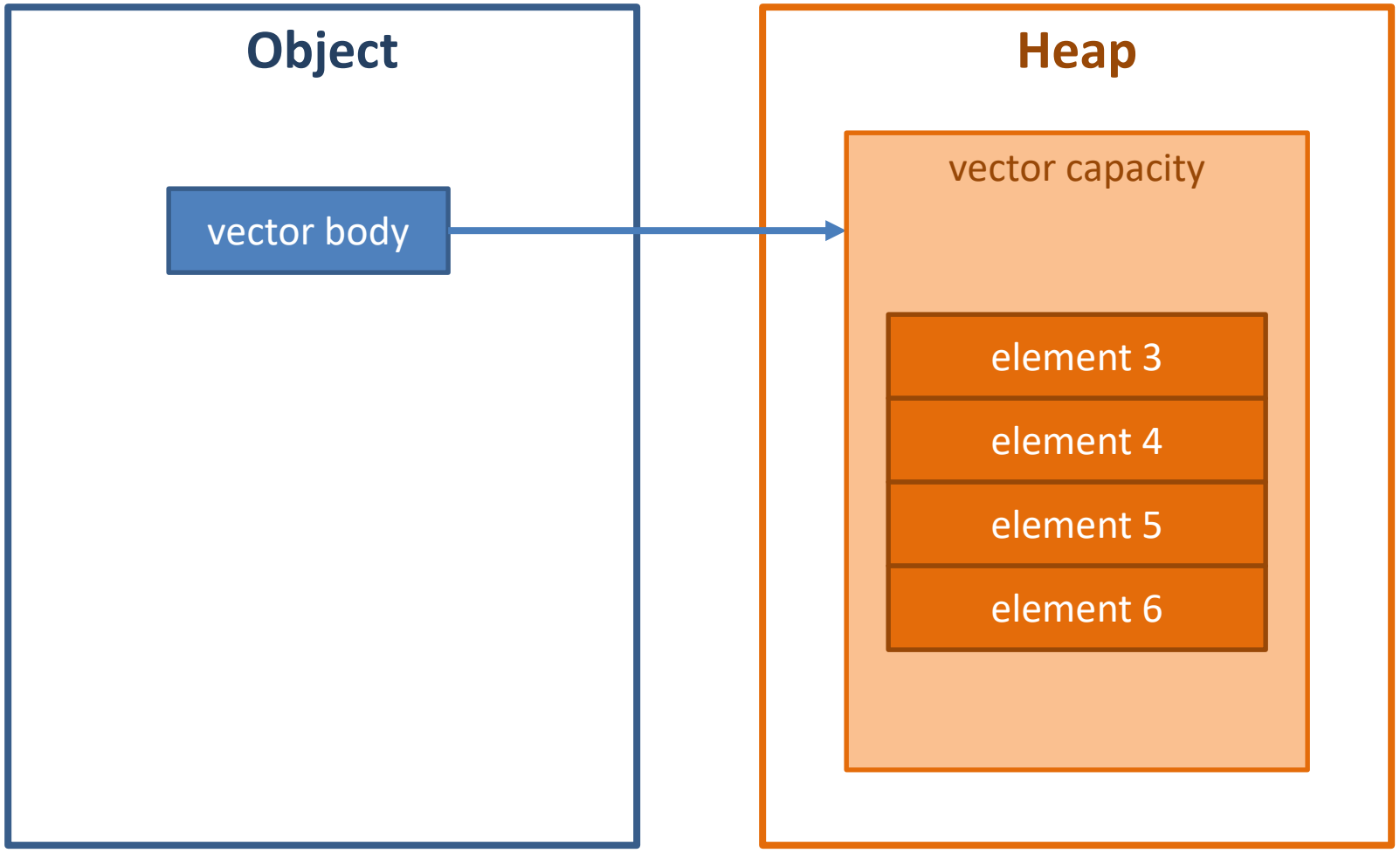
The Queue – Shifty Vector



The Queue – Shifty Vector



The Queue – Shifty Vector



The Queue – Shifty Vector

```
struct widget {
    widget() {
        fifo.reserve(size);
        front = fifo.begin();
    }

    // More below ...

    vector<int> fifo;
    vector<int>::iterator front;
};
```

The Queue – Shifty Vector

```
int pop()
{
    return *front++;
}

bool empty() const
{
    return front == fifo.end();
}
```

The Queue – Shifty Vector

```
void push(int i) {
      vector<int>::difference_type x;
      if (fifo.size() == fifo.capacity()) {
            if (front != fifo.begin())
                  fifo.erase(fifo.begin(), front);
            x = 0;
      }
      else
            x = front - fifo.begin();
      fifo.push_back(i);
      front = fifo.begin() + x;
}
```

The Queue – Shifty Vector

Container	Count	Reps	Size	Time (s)	Memory (MB)
List	10000	10000	10	35.4	95.7
Vector	10000	10000	10	10.3	19.6
Deque	10000	10000	10	5.7	55.9
Shift Vector	10000	10000	10	2.2	19.5

Don't Assume – Measure

- You can't depend on rules of thumb
  - **set** can be a very slow way to maintain a sorted list
  - It might be faster to look something up linearly than with a binary search
- Or even STL design
  - Vector does not have **push_front** or **pop_front**
  - But vector might be faster in your situation
- You have to try different implementations
  - And measure performance
  - Your assumption, which may be correct most of the time, could be a HUGE pessimization for your particular case
  - Sometimes you have to think about it rather hard and be a bit clever

Other Vectors

- Fixed capacity
  - Stores all elements locally; never uses the heap
  - Useful if you have a variable (but limited) number of elements
  - This is quite a common case (e.g. up to 4 phone numbers in a personnel record)
  - **std::array** is a fixed *size* vector (like a C array), not the same thing
- Local cache ("short string optimization")
  - Stores up to a specified number of elements locally, then uses the heap
  - This avoids the heap for common cases at the expense of local size
  - **std::string** typically does this, but you can't choose the local size

The Game

- Suppose we want a data structure with the following traits:
  - A fairly large number of objects (1000's)
  - Frequent turnover (insert/erase)
  - Pointers and iterators must remain valid with turnover
  - Order is not important
  - Rapid and reliable iteration is essential
- **list** is the only Standard Library solution that comes close
  - But the limitations mentioned above all apply
  - Lack of locality of reference will cause iteration to be slow and to vary in performance, even during a single pass

The Game

- Create a container of **count** elements
  - Elements are roughly 100 bytes
- Do the following **reps** times
  - Do **reads** times
    - Access each of **count** elements in turn
  - Delete every **stride**-th element in the container
  - Insert **count** / **stride** new elements

The Game – List

Container	Count	Reps	Reads	Stride	Time (s)	Memory (MB)
List	10000	1000	100	5	14.5	8.4

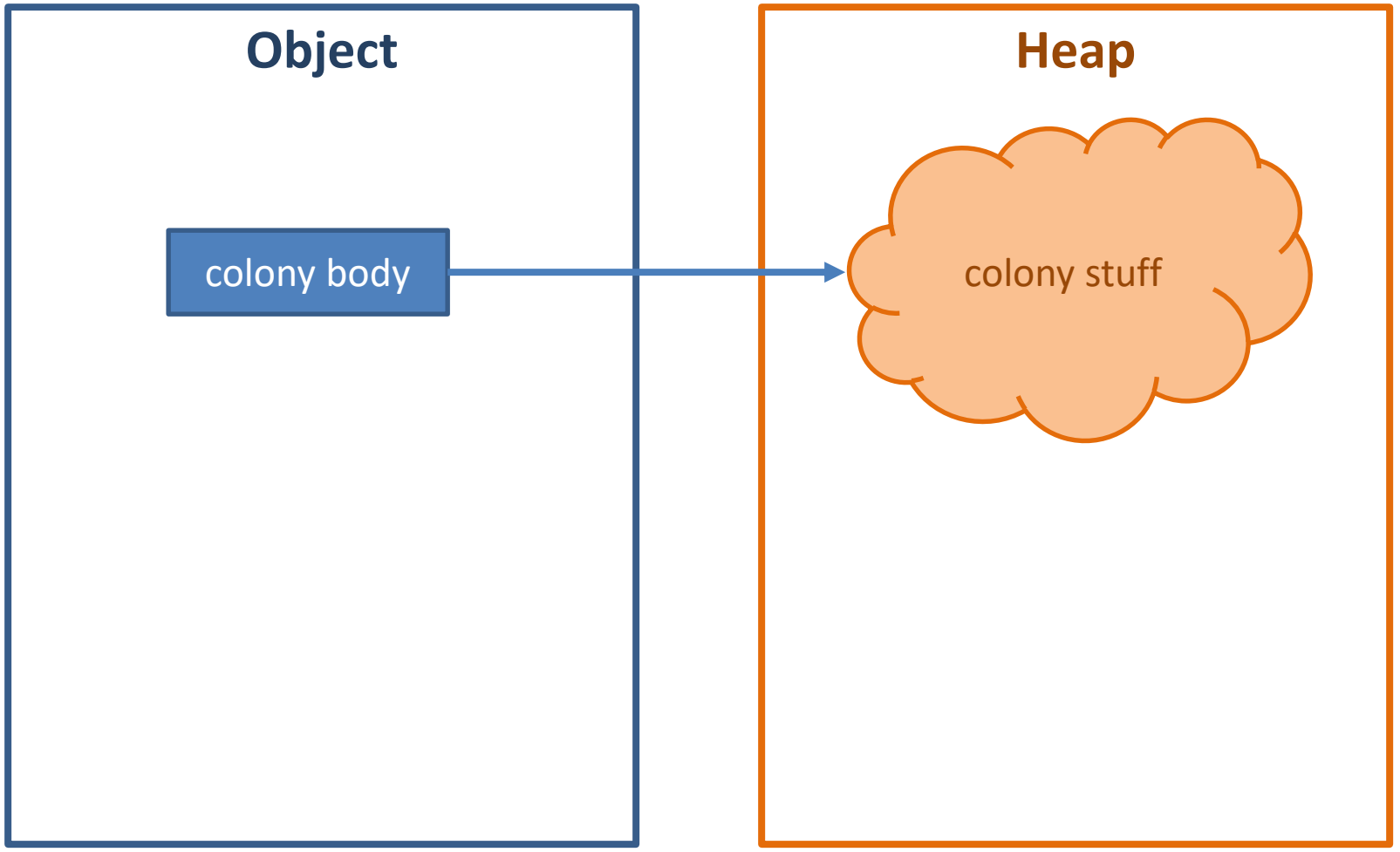
The Game – Colony

- **colony** container
  - An extension of the bucket array container design
  - Stores the elements in a series of blocks of increasing size
  - Pointers and iterators are not invalidated by **insert** or **erase**
  - Memory for deleted elements is reused by new elements or freed
  - A separate data structure called a skipfield marks deleted elements in a way which optimizes iteration
  - Other structures are maintained to identify deleted elements

https://www.youtube.com/watch?v=wBER1R8YyGY

https://plflib.org/colony.htm

The Game – Colony



? Allocations

The Game – Colony

Container	Count	Reps	Reads	Stride	Time (s)	Memory (MB)
List	10000	1000	100	5	14.5	8.4
Colony	10000	1000	100	5	2.8	7.7

Program Runs Slow

- Simulation took 7.25 hours on my fast 2019 workstation
  - It was a lot slower on the customer's laptop

Program Runs Slow

- Simulation took 7.25 hours on my fast 2019 workstation
  - It was a lot slower on the customer's laptop
- Ran 100 times slower at 0.1 sec time step than at 1.0 sec
  - That's exactly 10 times the work and 10 times the data generated

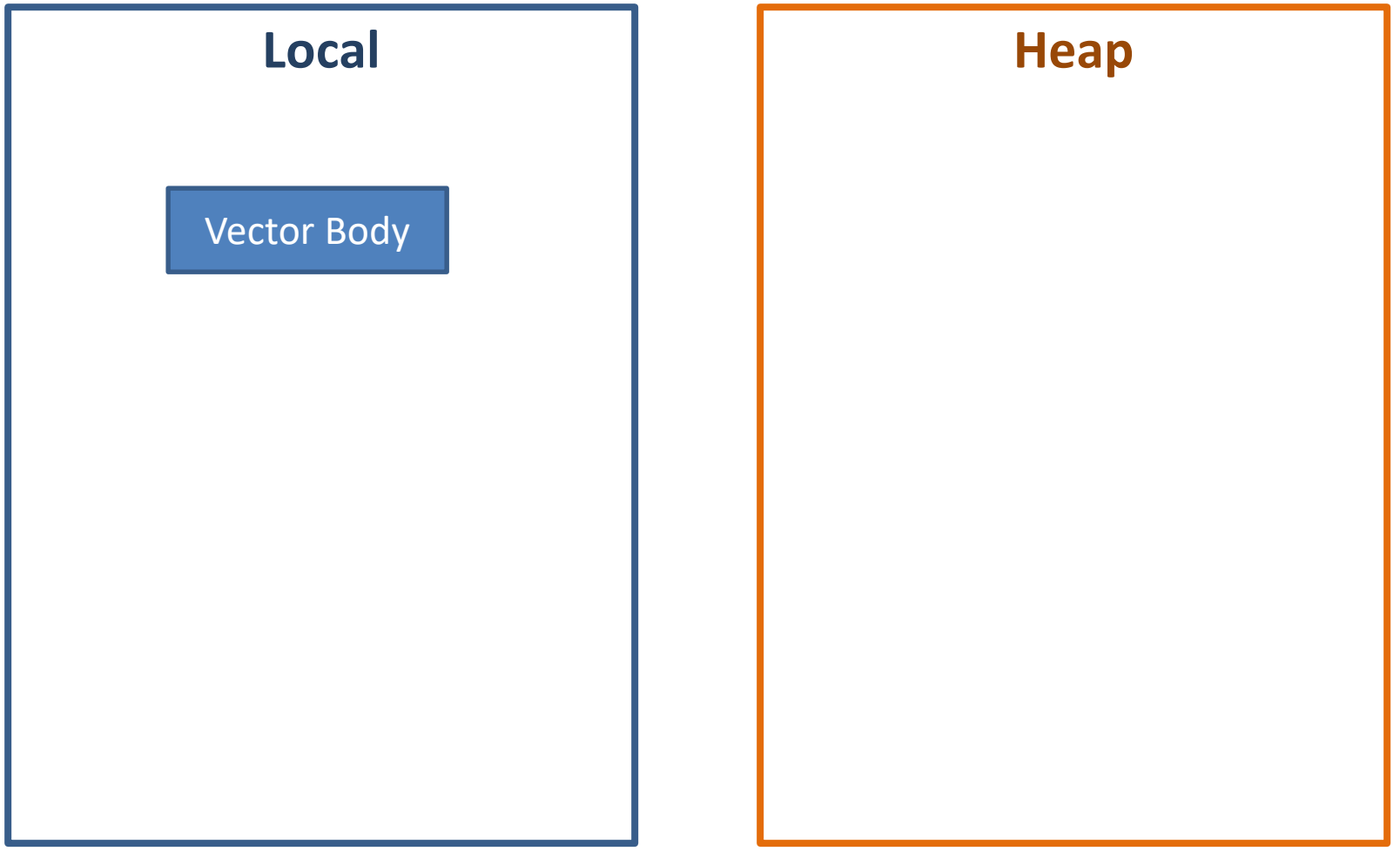
Program Runs Slow

- Simulation took 7.25 hours on my fast 2019 workstation
  - It was a lot slower on the customer's laptop
- Ran 100 times slower at 0.1 sec time step than at 1.0 sec
  - That's exactly 10 times the work and 10 times the data generated
- Profiling revealed lots of time spent in **vector** allocation
  - But I was doing all the right things: reserving, using move semantics

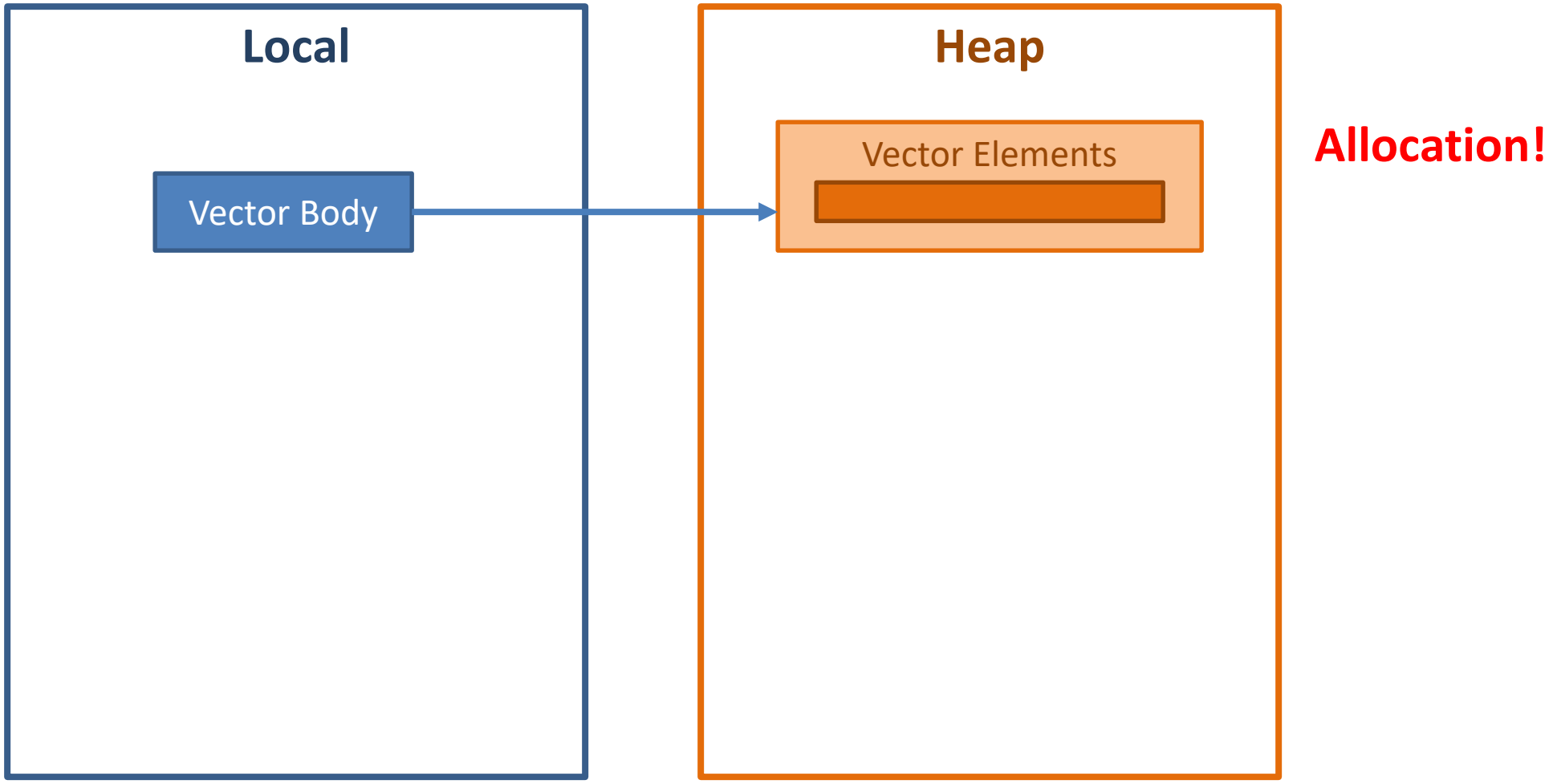
Program Runs Slow

- Simulation took 7.25 hours on my fast 2019 workstation
  - It was a lot slower on the customer's laptop
- Ran 100 times slower at 0.1 sec time step than at 1.0 sec
  - That's exactly 10 times the work and 10 times the data generated
- Profiling revealed lots of time spent in **vector** allocation
  - But I was doing all the right things: reserving, using move semantics
- Ran fine at 1.0 sec with electrical system, or at 0.1 without
  - Electrical system generates lots more data per time step

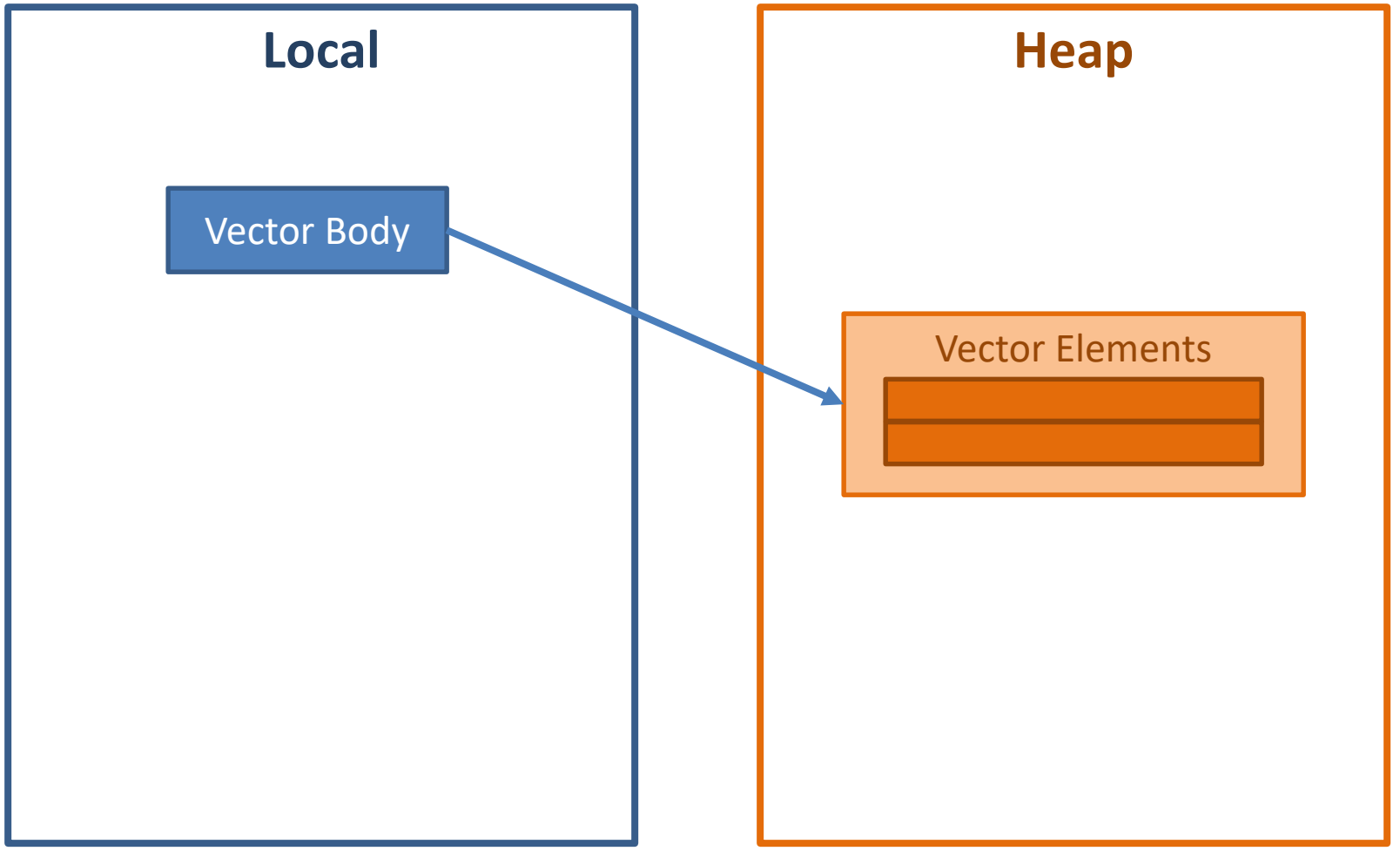
Vector – Growth



Vector – Growth

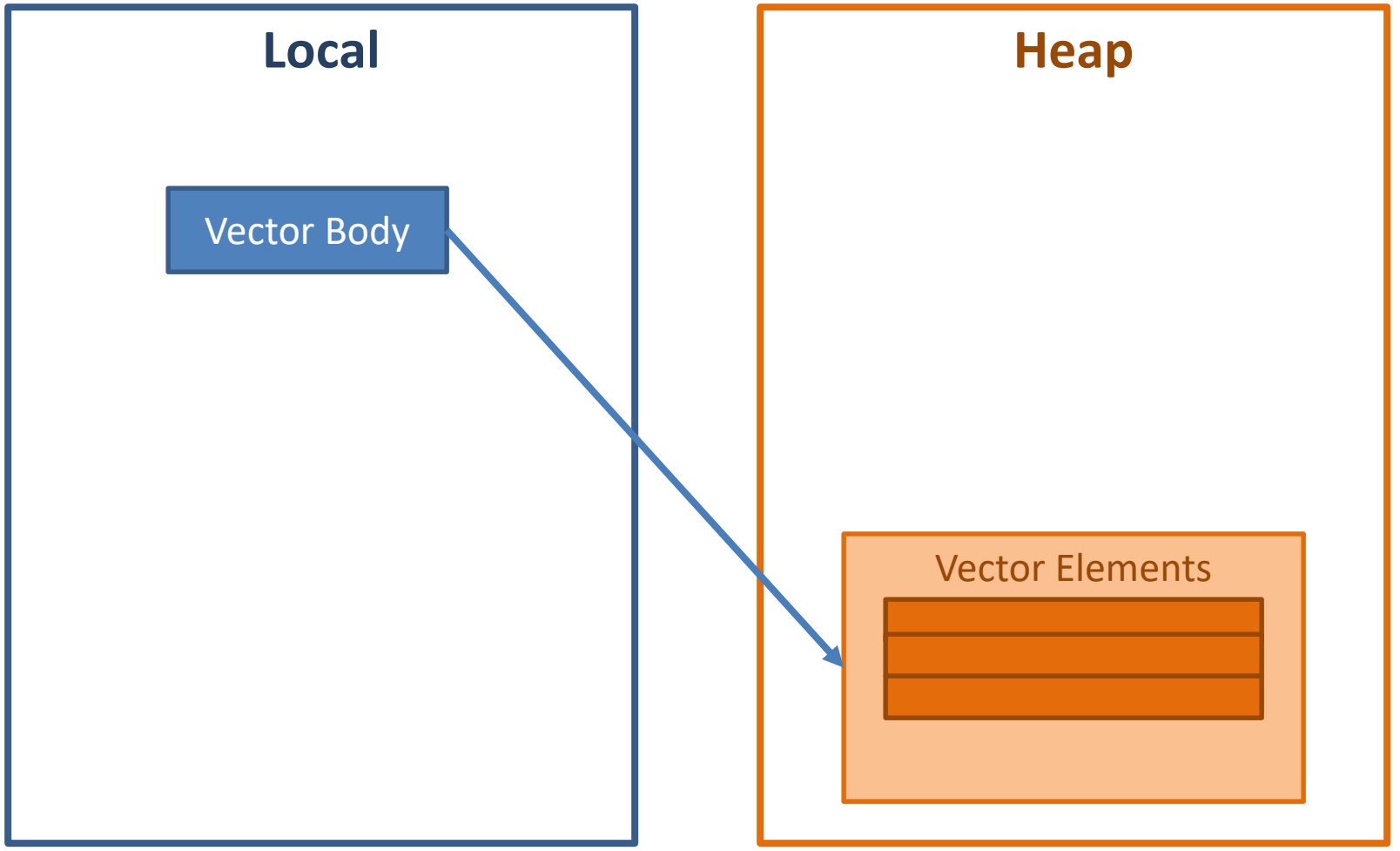


Vector – Growth



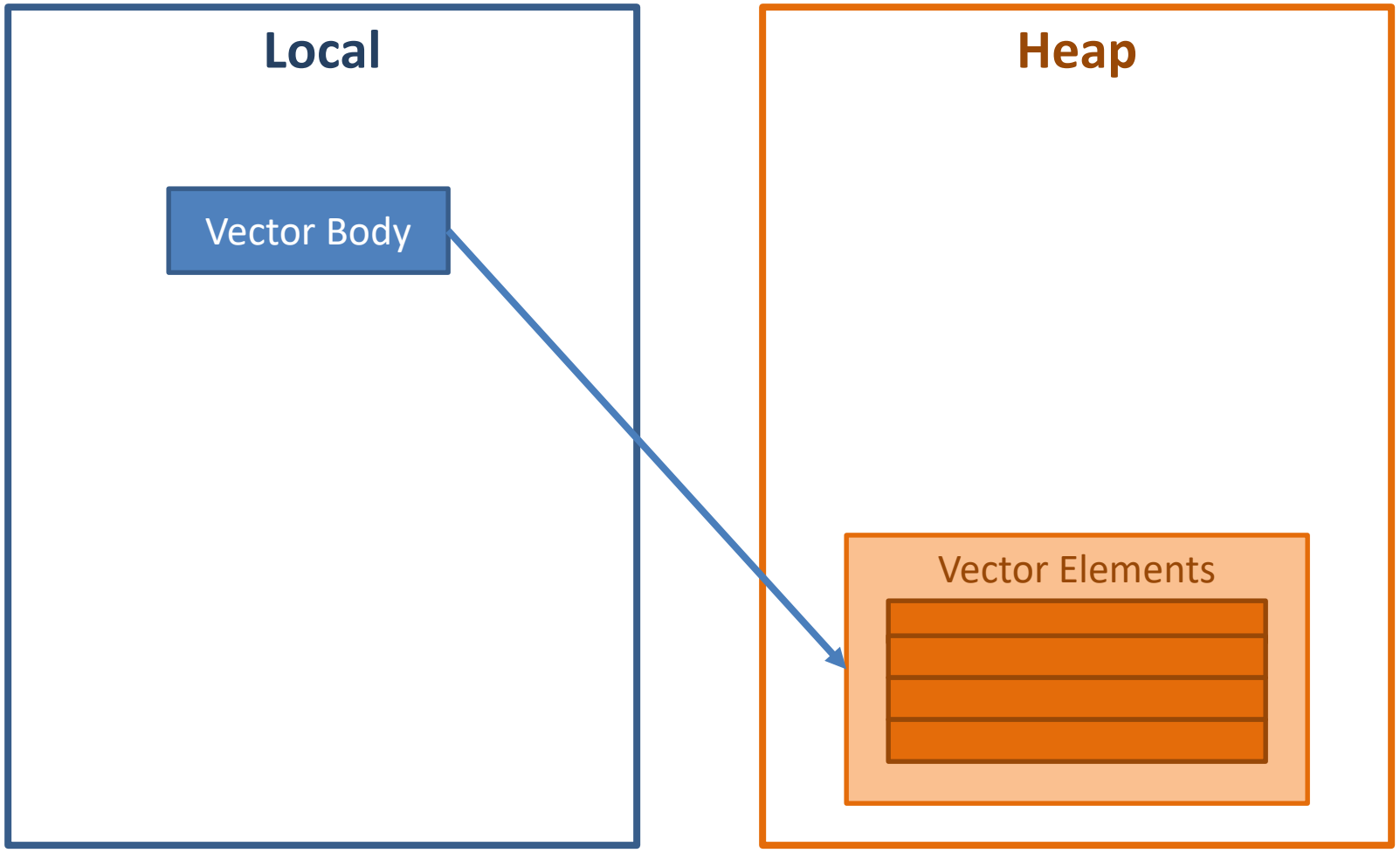
Allocation!

Vector – Growth

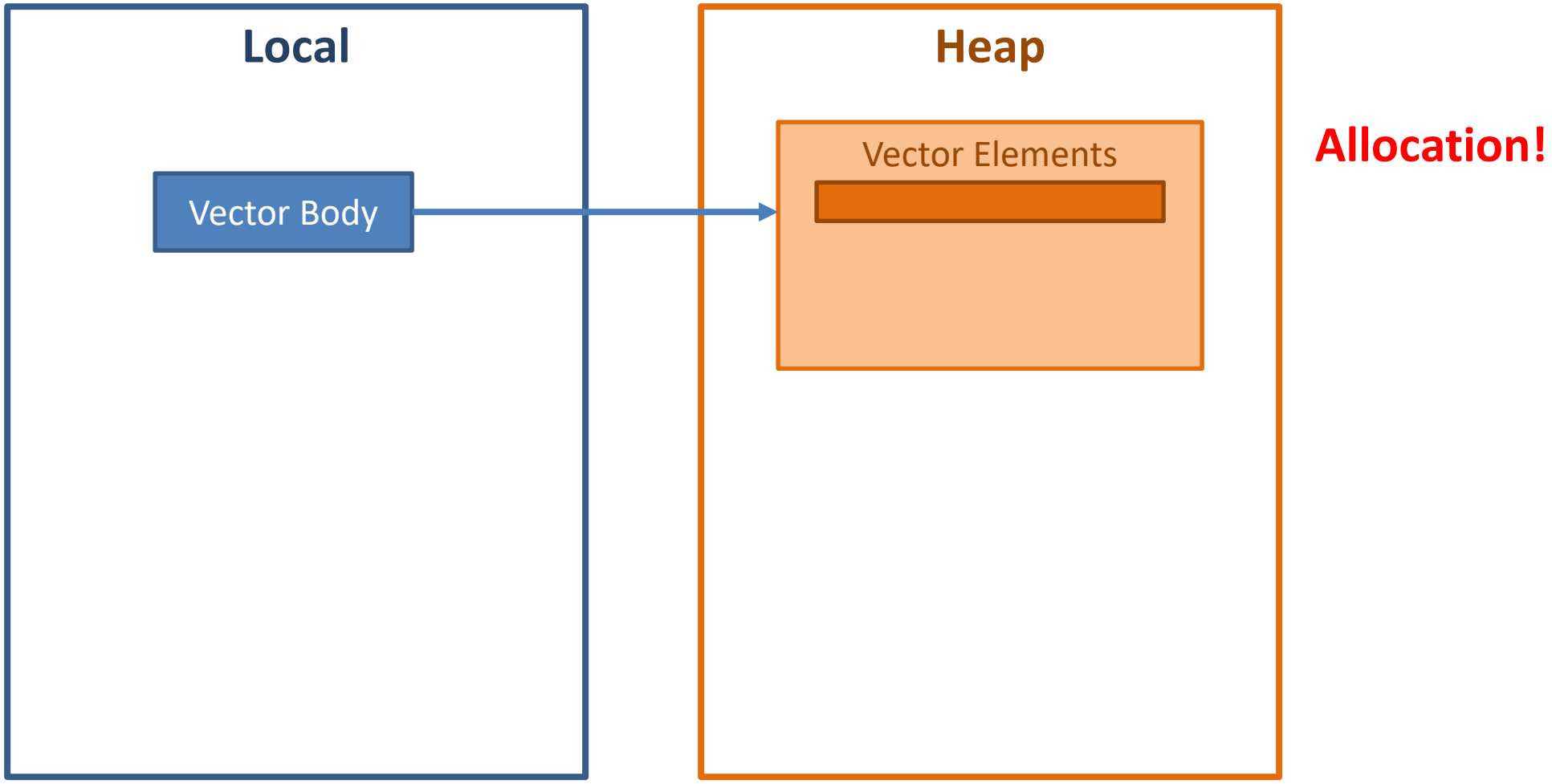


Allocation!

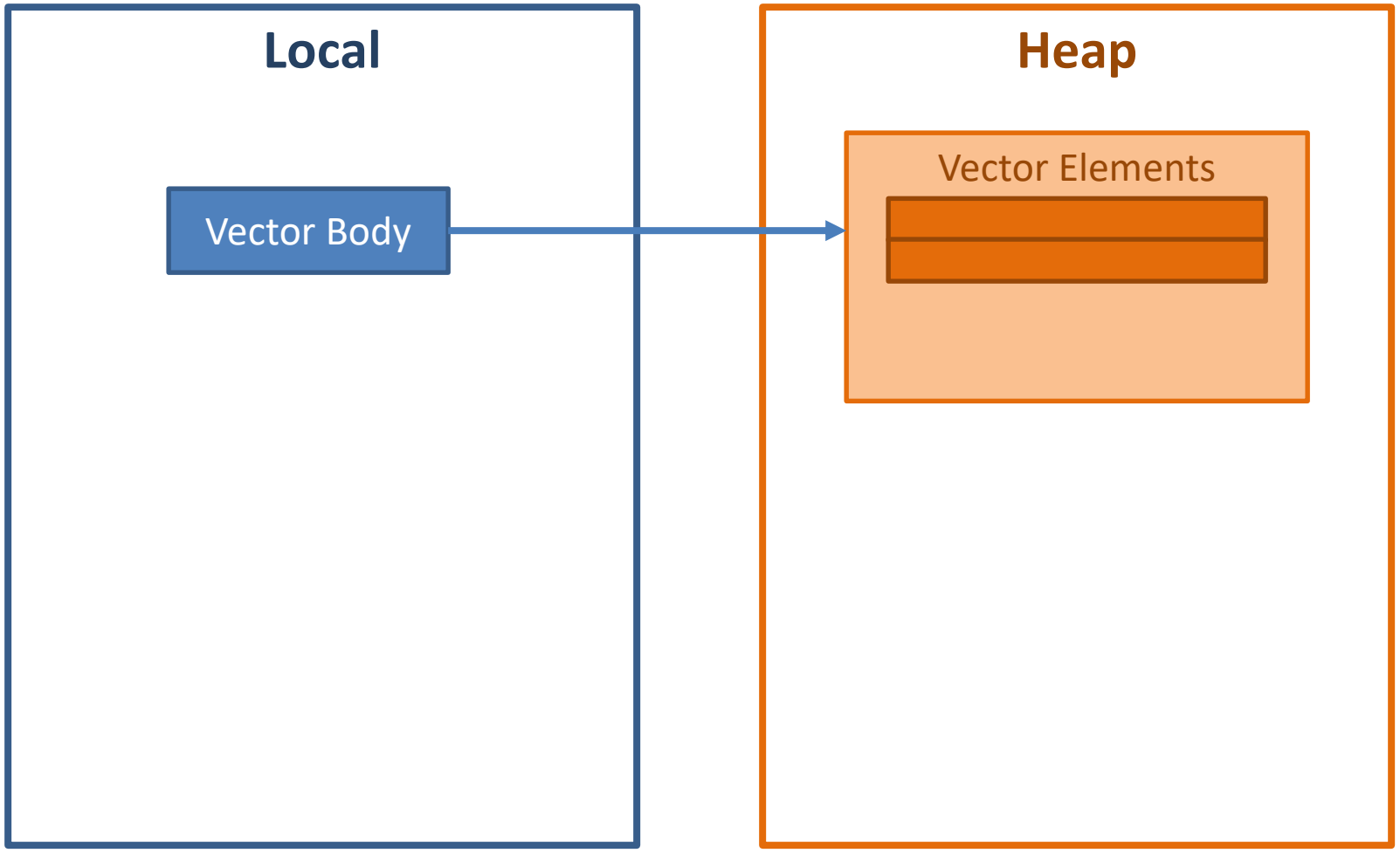
Vector – Growth



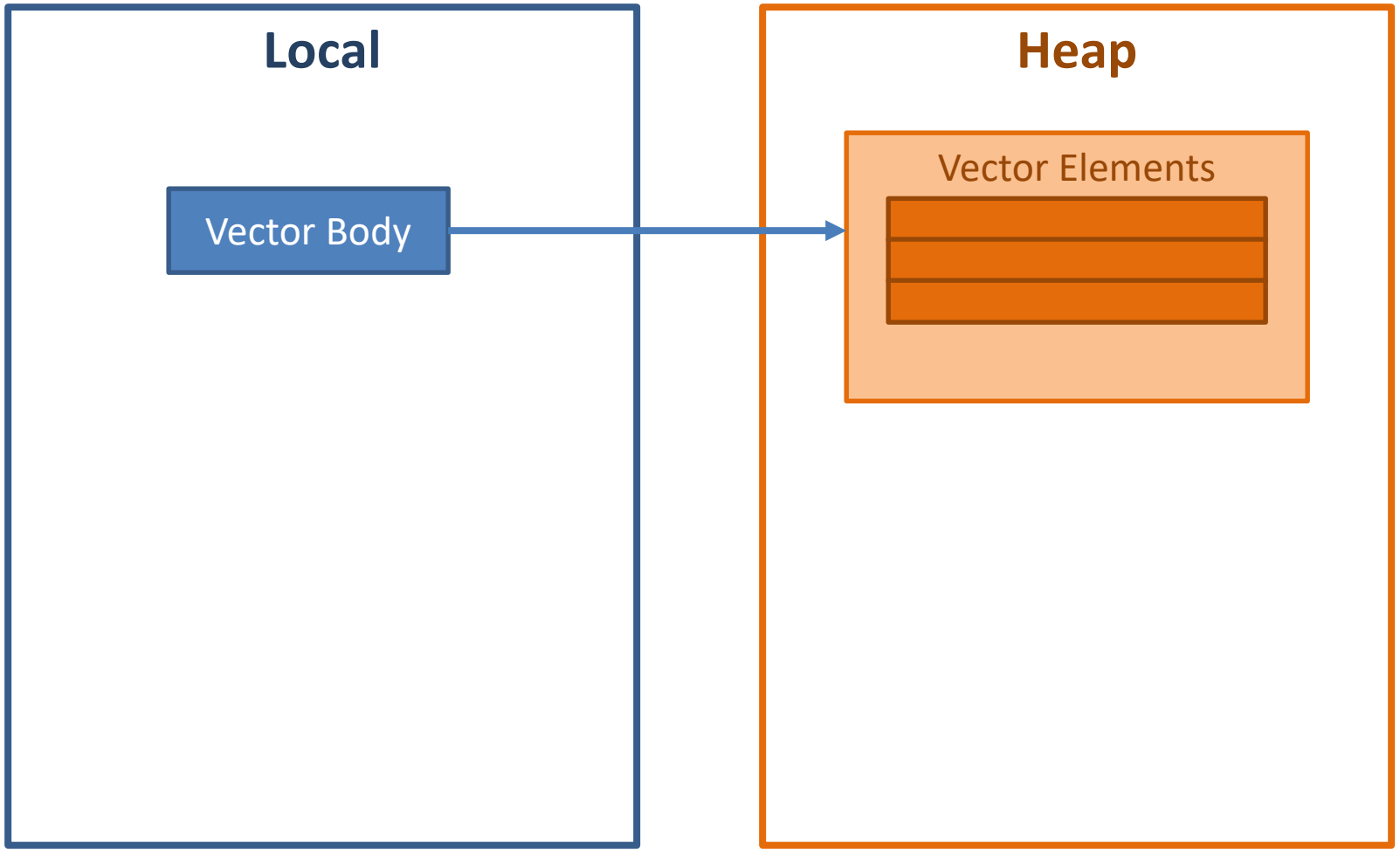
Vector – Growth with Reserve



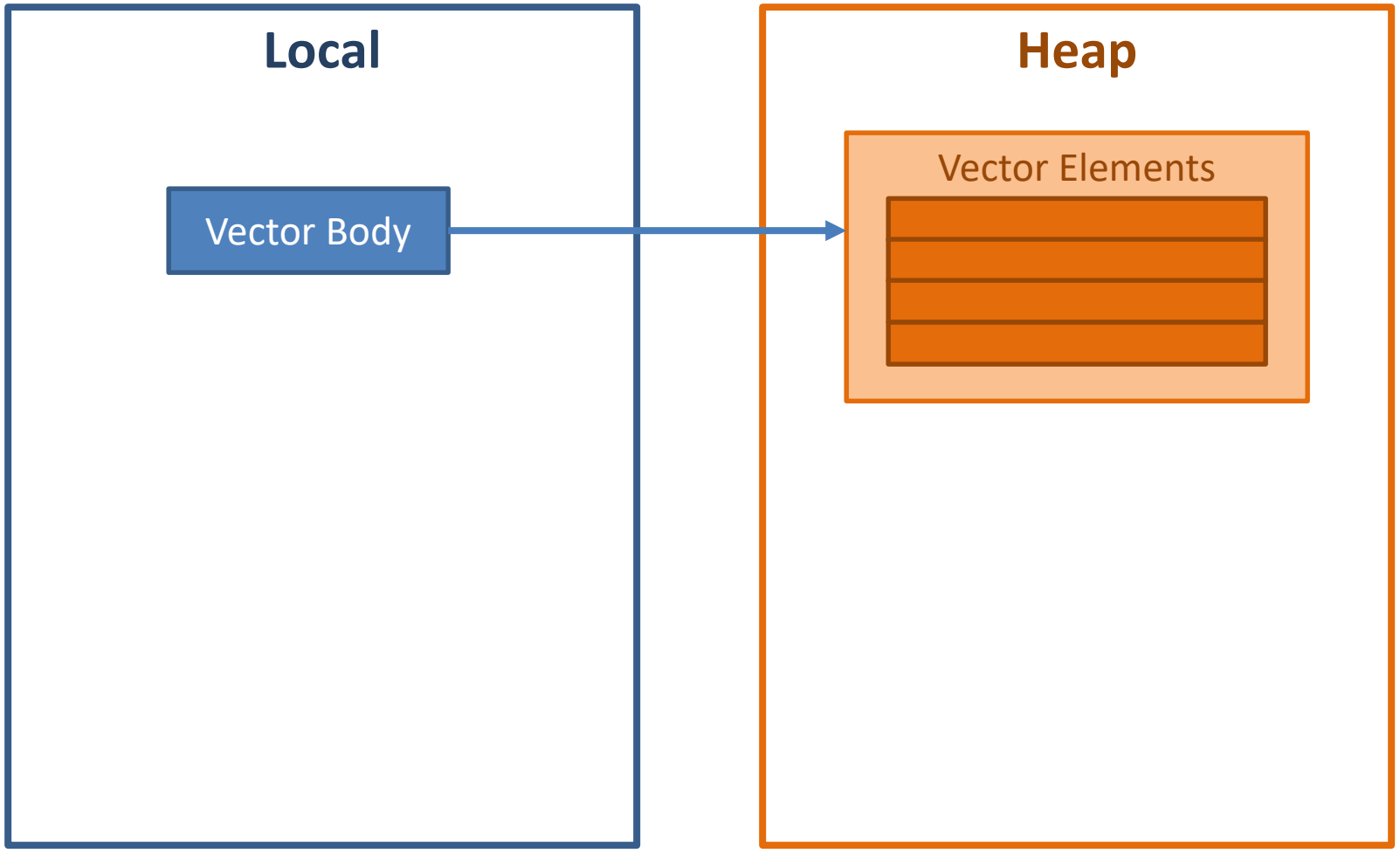
Vector – Growth with Reserve



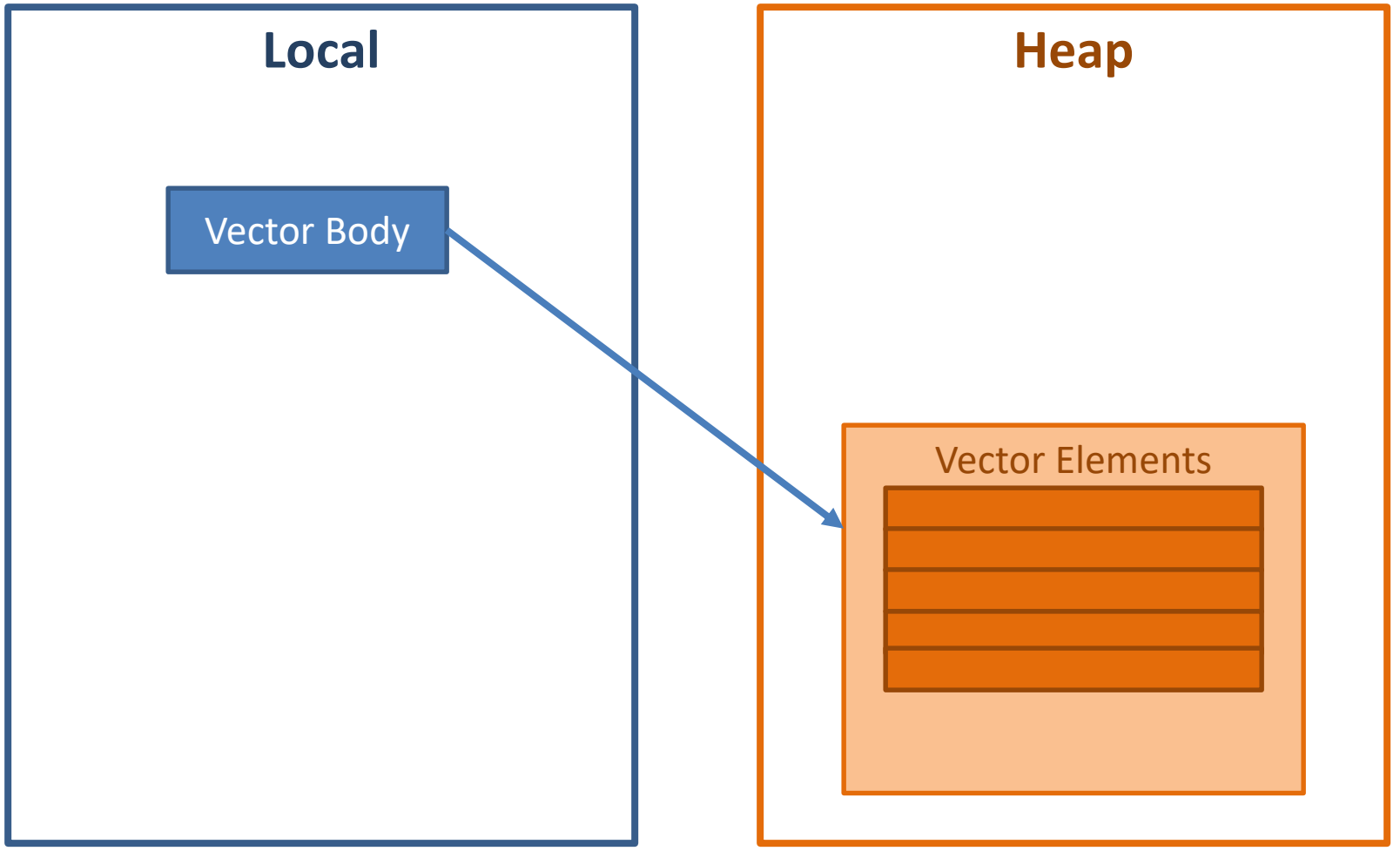
Vector – Growth with Reserve



Vector – Growth with Reserve

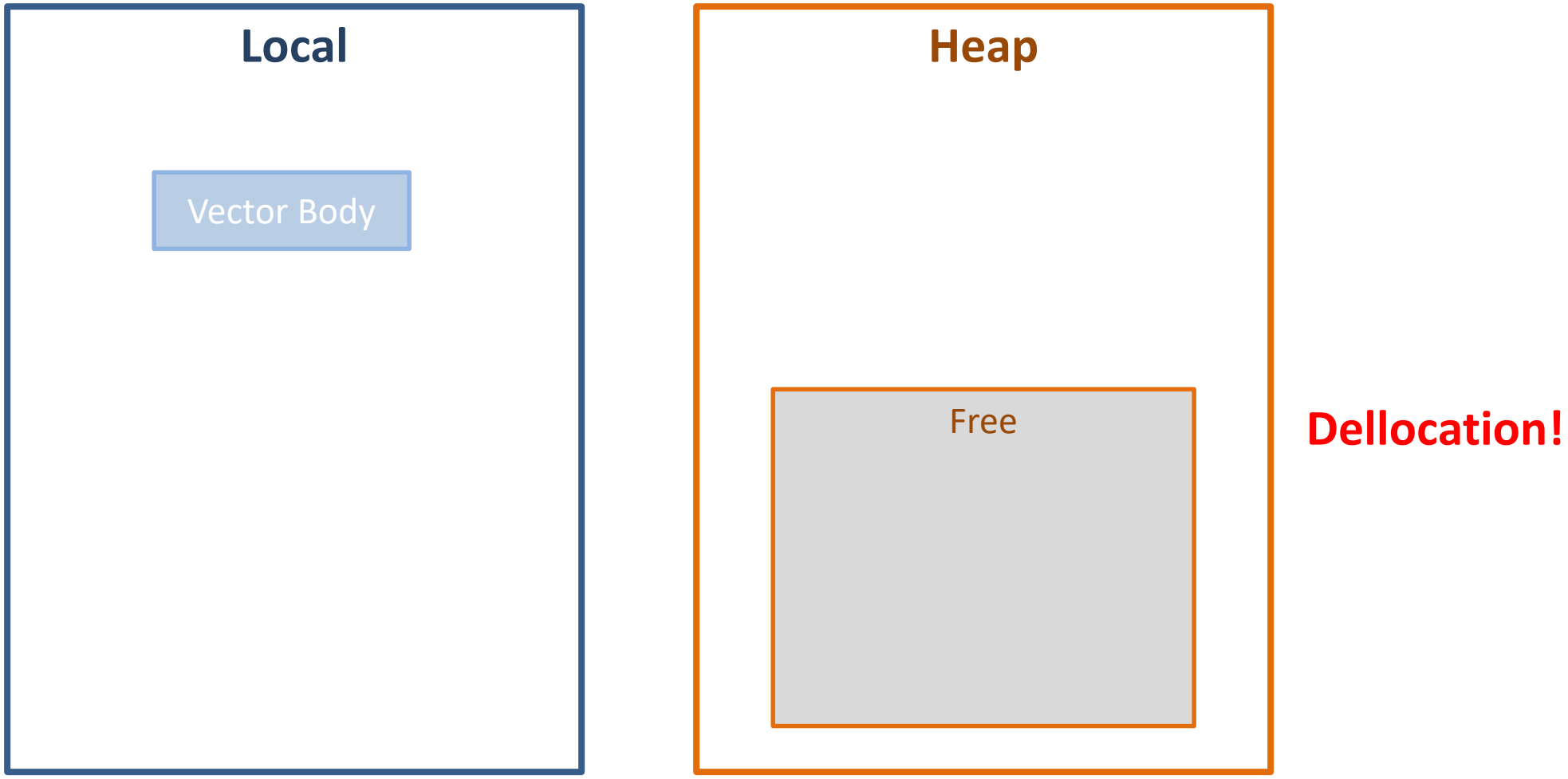


Vector – Growth with Reserve

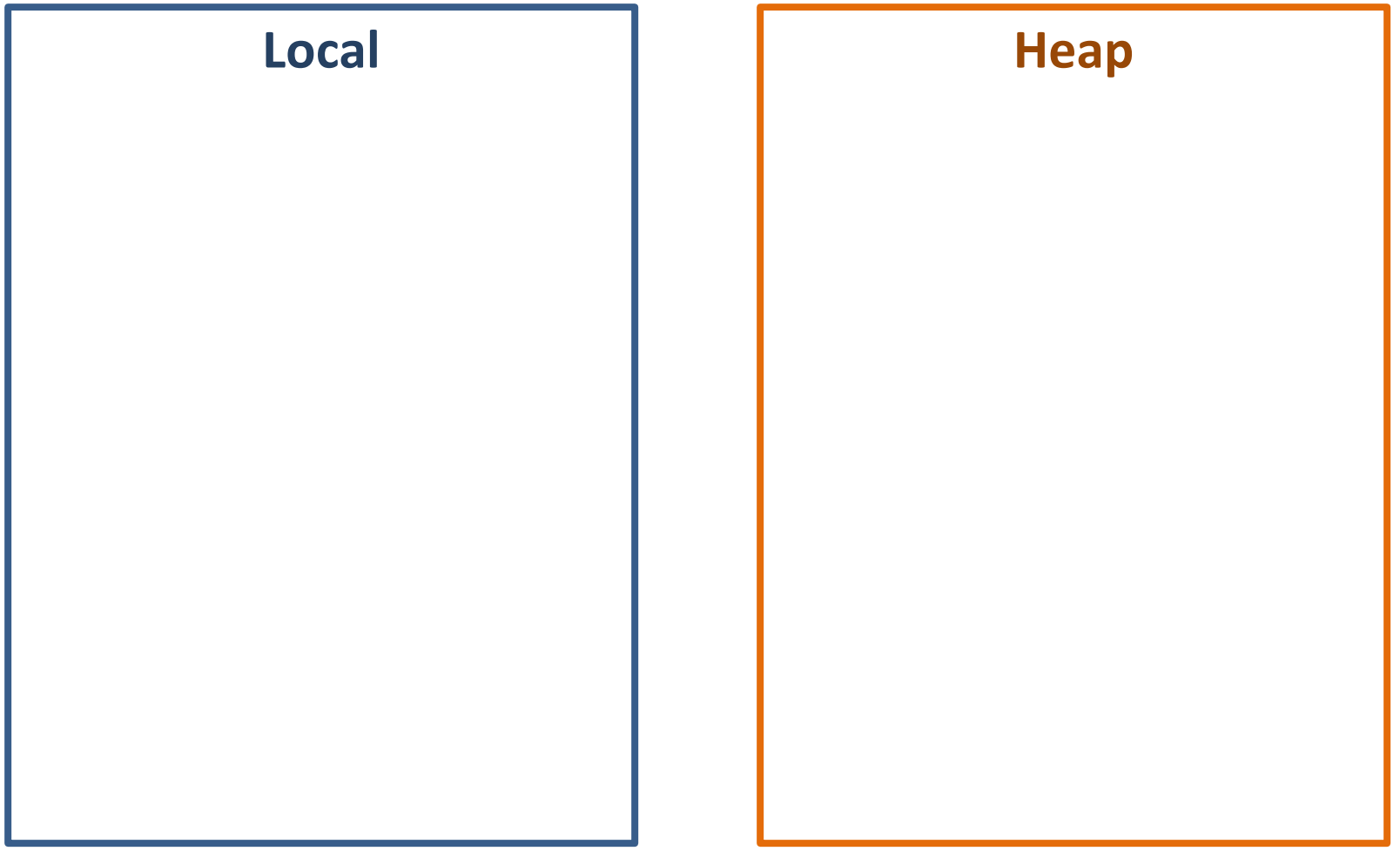


Allocation!

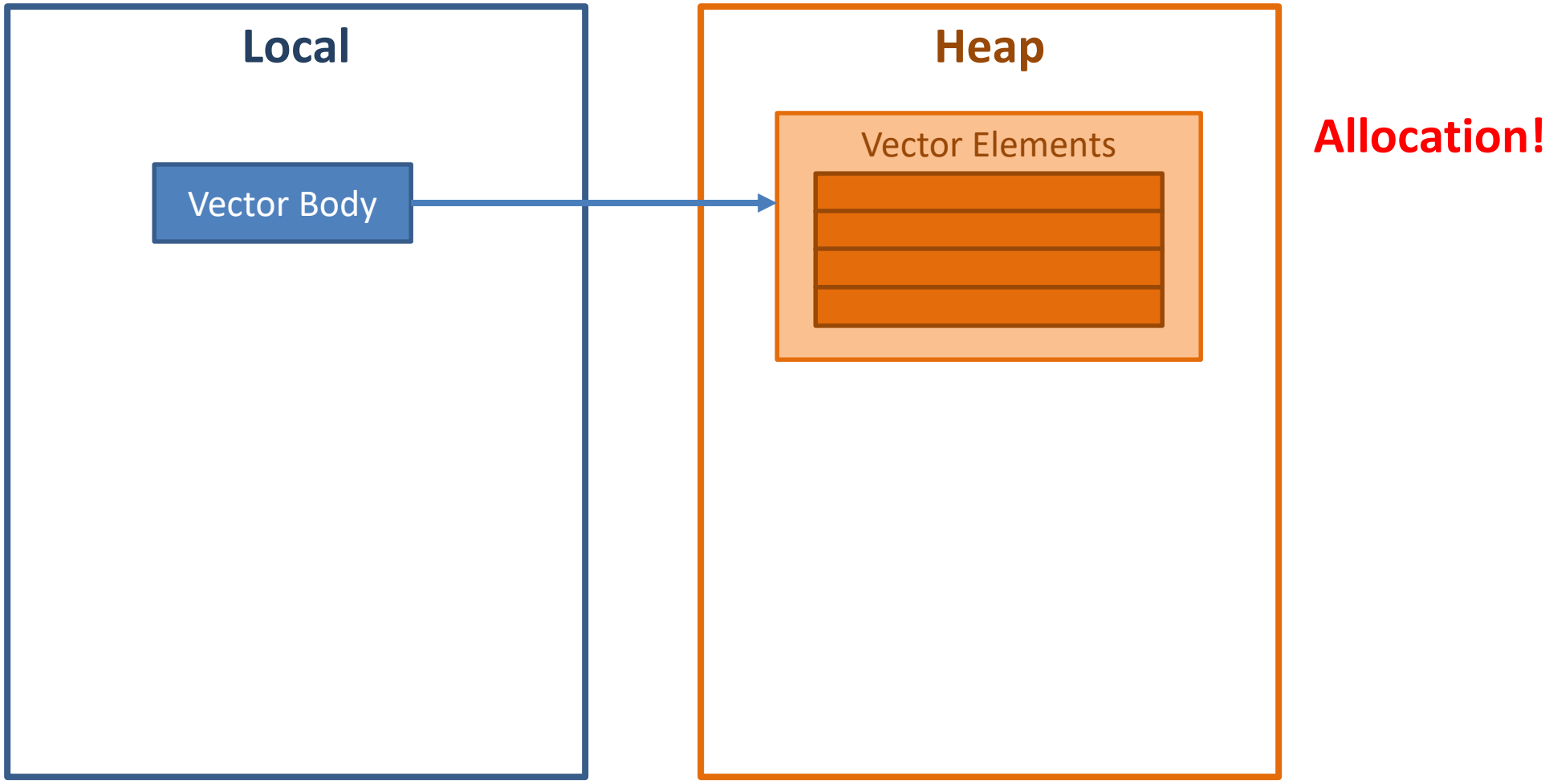
Vector – Creating Another



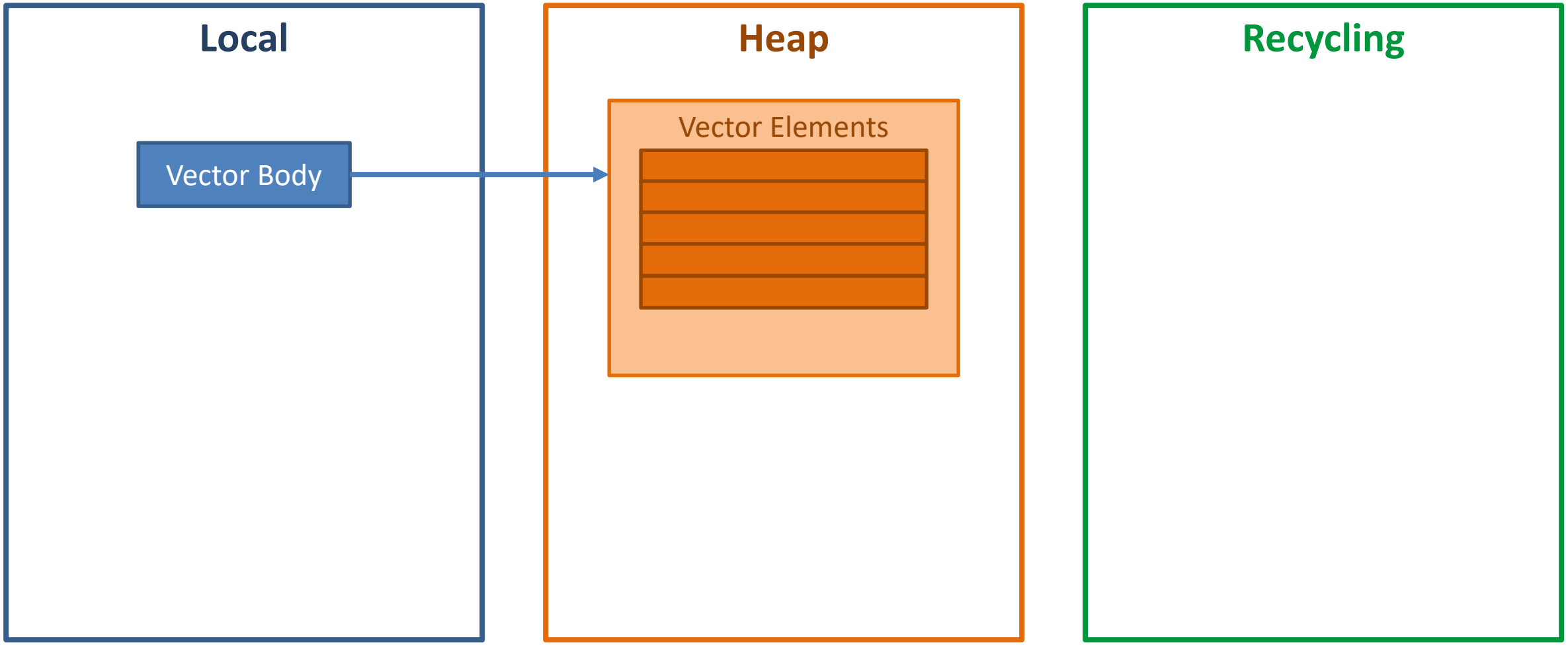
Vector – Creating Another



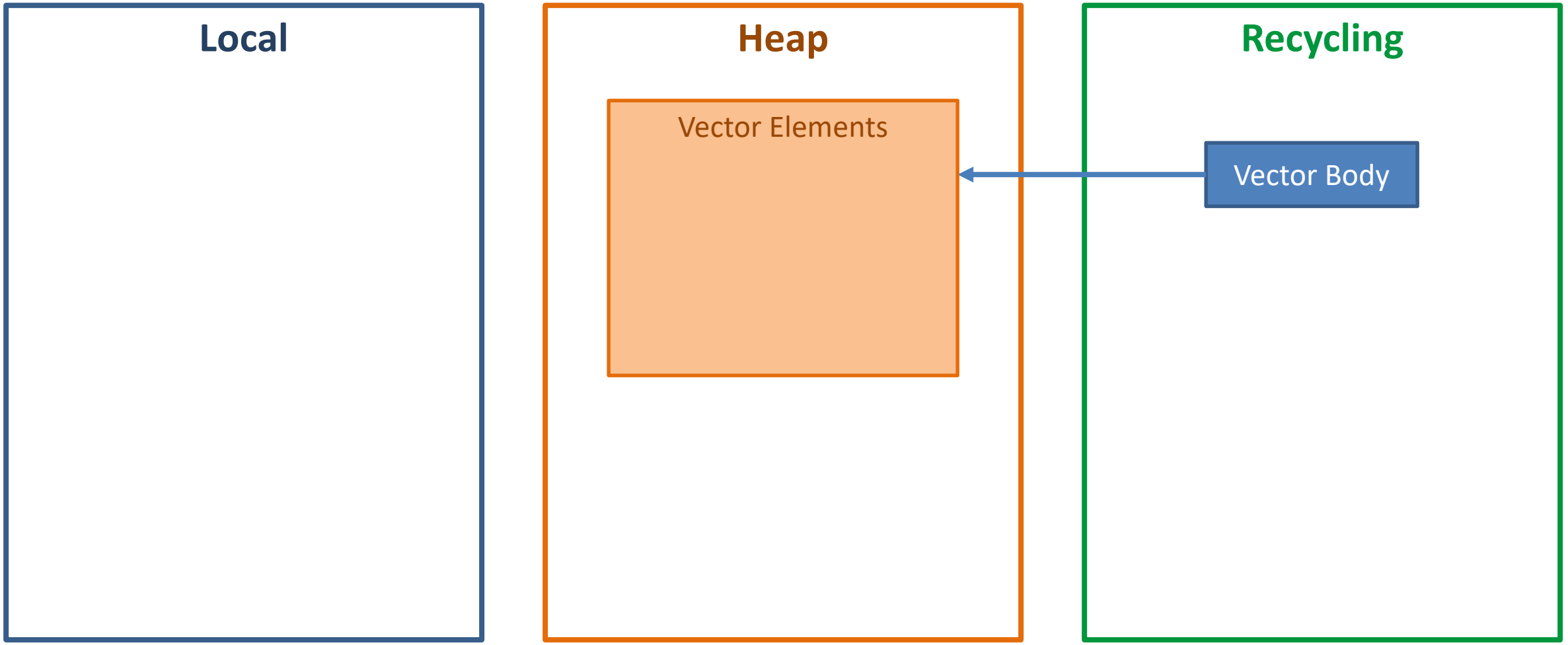
Vector – Creating Another



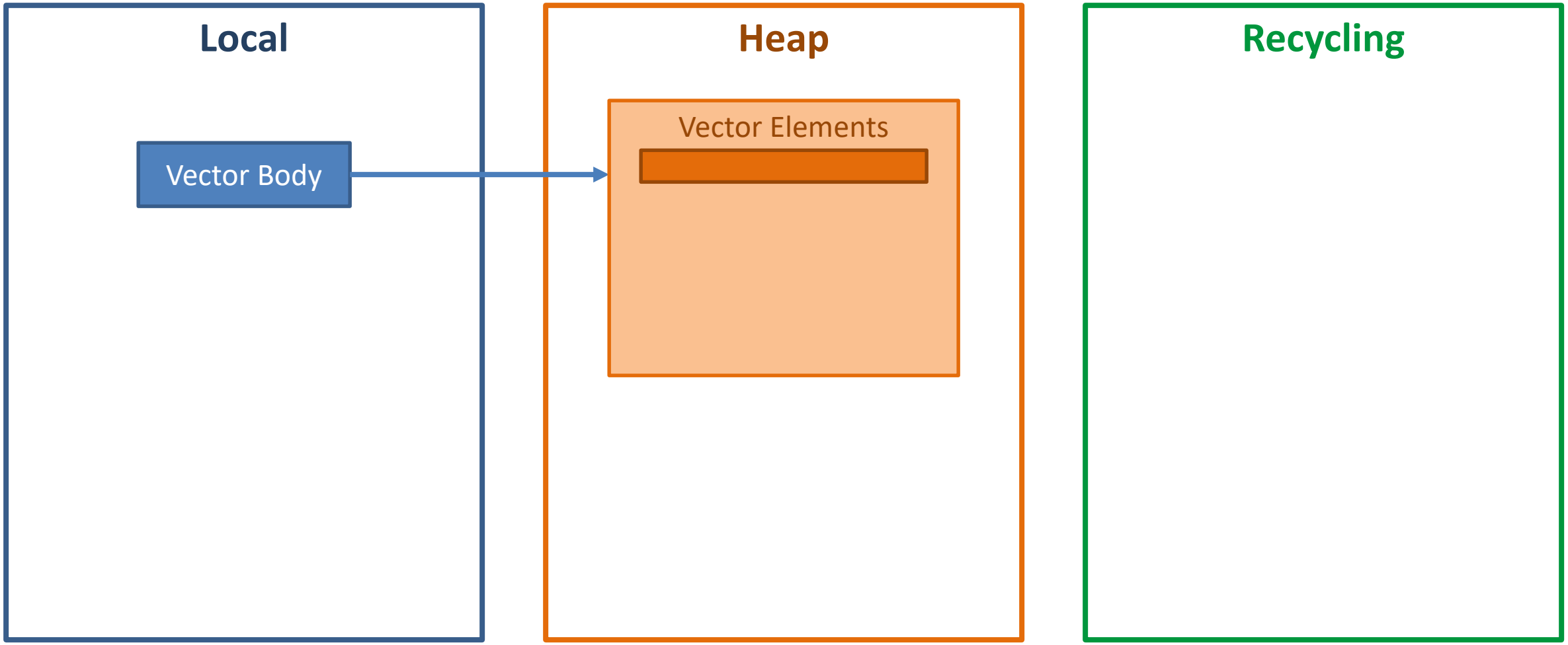
Vector – Recycling



Vector – Recycling



Vector – Recycling



Vector – Recycling

```
struct number_list {
    number_list()
    {
        if (recycling.empty())
            numbers.reserve(X);
        else
        {
            numbers = move(recycling.back());
            recycling.pop_back();
        }
    }
    ~number_list()
    {
        numbers.clear();
        recycling.push_back(move(numbers));
    }
    vector<int> numbers;
    static vector<vector<int>> recycling;
};
```

```
number_list(const number_list& rhs)
: number_list()
{
    numbers = rhs.numbers;
}
```

Program Runs Slow

- Simulation took 7.25 hours on my fast 2019 workstation
  - It was a lot slower on the customer's laptop
- Ran 100 times slower at 0.1 sec time step than at 1.0 sec
  - That's exactly 10 times the work and 10 times the data generated
- Profiling revealed lots of time spent in **vector** allocation
  - But I was doing all the right things: reserving, using move semantics
- Ran fine at 1.0 sec with electrical system, or at 0.1 without
  - Electrical system generates lots more data per time step
- With **vector** recycling, simulation took 35 min (instead of 435)
  - **A speed up of over 12 times**

Lessons

- Don’t make assumptions - measure
  - Rules of thumb may not apply to pinkies
  - Results on a server may be very different than on a laptop
  - You may want to try several different approaches
- Know your containers
  - Basic complexity is not the whole story
  - Learn how all the standard containers work
  - Other containers may offer significant improvements
- Know what’s going on in memory
  - Heap allocations are expensive
  - Locality of reference is critical
  - Random access of main memory can be much slower than sliding things in cache

How to Choose the Right Standard Library Container

And Why You Should Want Some More

CppCon 2019

Alan Talbot
cpp@alantalbot.com

September 17, 2019