Moving Faster

Everyday efficiency in modern C++

Alan Talbot

LTK Engineering Services

May 2018

A 30 Year Tale

90 WPM = 450 CPM = 7.5 CPS

1988

Mac II – 1 thread – 16 MHz

2 million instructions / character

2018

ThinkPad T580 – 8 threads – 2 GHz

2 billion instructions / character

What are we doing wrong?

- Thinking it doesn’t matter
  - Computers are so fast they can do anything
  - And anyway the compiler will take care of it
- Designing like it doesn’t matter
  - Suboptimal container choice
  - Overuse of the heap
- Coding like it doesn’t matter
  - Tuning vs. optimizing
  - Values vs. references

When does efficiency matter?

- Typing? Really???
  - The world record for typing speed is 216 WPM set by Stella Pajunas
  - In 1946 on an IBM electric typewriter
- Where does it say it has to go faster?
  - Original code ran in 3 weeks
  - New code ran in around 5 hours (100 X)
- Test automation for TrainOps rail simulator
  - 2700 tests take about 1:10 (70 minutes) today
  - New approach will reduce that to about 20 seconds (200 X)

Where does efficiency matter?

- Tune, don't optimize: the 5% - 95% rule
- This is necessary but not sufficient
- Many programs do *not* spend all their time in a small bit of code
- If you write a big, complicated program without attention to performance, you'll likely have a big slow program
- TrainOps is an example of both kinds of program
- Write (almost) everything optimally

Writing optimal code

- But optimized code is ugly!
- In C++ *optimal* code can be elegant
- In fact, optimal code is the most correct code, and so the most elegant code
- If an optimal solution really is ugly, you can hide it behind an elegant zero-overhead abstraction
- The most elegant, correct and optimal code should be the most idiomatic code

Compiler optimizations

- Today's compilers are very smart
- But they can't fix design problems
- And they can't get around all coding inefficiencies

```
for (int i = 0; i < zillion; i++)

vector<string> v;
v.push_back("hello");
```

Caches

- Many of us grew up with simple architectures
  - Floating point operations used to be slow
  - The golden rule of efficiency used to be: reduce FP divides
  - Floating point is now faster than integer
- Caches are big, and much faster than main memory
  - But they are not big compared to main memory
  - And they get smaller as they get faster
  - So locality of reference is critical
  - Which means size matters
  - Cache misses are the new FP divides

$\boxed{\text{Avoid cache misses}}$

Dynamic allocation

- Heap allocations are the most expensive things we commonly do
- Deallocations are the second most expensive
- So reduce them as much as possible, ideally to zero
  - Count your allocations
- Most of the Standard containers and almost any use of new (or smart pointers) uses the heap
- Heap allocations are in arbitrary locations, so locality of reference is terrible between separate allocations

Avoid allocations

Static allocation

- Local variables and function parameters are on the stack (when not in registers)
- The stack is contiguous, so it has great locality of reference
- It's used constantly so it's always in the cache
- Unless something is pretty big, if it has local scope and fixed size it should be on the stack

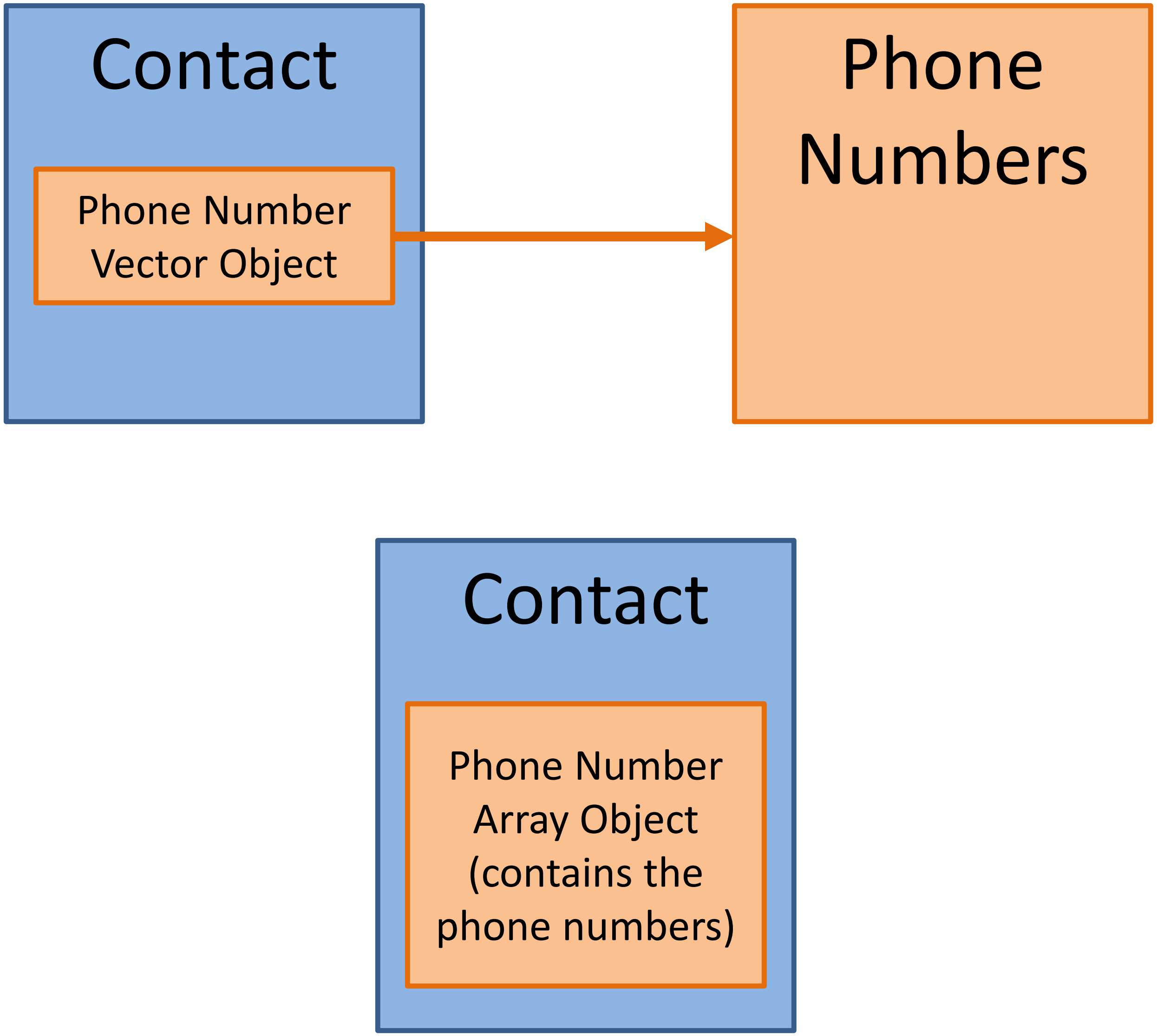
Registers

- Caches are fast, but nothing is faster than a register
- There are lots of registers and compilers use them
- Many stack variables are never actually on the stack, they are in registers for their entire lifetime
- Taking the address of a variable means it can't (only) be in a register
- This of course includes all dynamic allocation

Embedded objects

- A contact object needs a list of phone numbers, do you use **vector** or **array**?
- Using **vector** will mean that you will have two unrelated allocations for each contact:
  - The contact object (which contains the vector itself)
  - The phone number vector contents
- Using **array** means that you have to:
  - Pick a maximum number of phone numbers
  - Deal with the container not knowing how many there actually are (arrays have a fixed size)

Embedded objects



Embedded objects

- If you can live with the fixed maximum size, embedding the list will be much more efficient
  - Much better locality of reference yields faster access
  - Much faster allocation/deallocation
  - Much less space (assuming maximum size is small)
  - Sadly we don't have a fixed capacity vector
- This is a perfect case for a "short string optimization" which does both
  - Sadly we don't have an SSO vector

Sharing space

- Unions
  - Unions used to be pretty useless, but they got fixed in C++11
  - A union lets you share space when two things are either mutually exclusive or have non-overlapping lifetimes
  - Saving space saves time (because of caching)
  - Unions are only reasonable if the active type is clear from the context
  - But unions are never really safe

Sharing space

- Variants
  - Variants are new in C++17
  - They wrap a union with a type management API, and include a type tag (int64) that permits runtime type checking
  - They trade a little bit of space for a lot of convenience and safety
  - But if you are trying to save 4 bytes by superimposing two ints, variant won't help (because of the type tag overhead)

Pass by value

- Pass simple things by value
  - Built-in types (int, long long, double)
  - Maybe your simple types (16 bytes)
  - Value semantics are a Good Thing
  - But remember, you are making a copy
- Pass things by value when you need to modify a copy
  - There is no point in taking a const & parameter if you are immediately going to make a copy anyway
- Pass shared_ptr by value to share ownership
  - But move it to transfer ownership

Pass by const reference

- Pass most other things by const reference
- There is no penalty for passing by reference over value for built-in types, so in generic code just do it
- Pass shared_ptr by const reference if you aren’t sharing or transferring ownership
- But be careful, you have to think about the lifetime of the referenced object
- And there are efficiency considerations
- So watch Nicolai Josuttis's presentation:
  - https://www.youtube.com/watch?v=PNRju6_yn3o&t=2710s

Pass by non-const reference

- Generally avoid passing by non-const reference
- It makes code much harder to understand and reason about
- Depending on what you are modifying, value semantics may be fast enough anyway (if move is possible)
- But it is sometimes necessary or much more efficient
- Do it to avoid loss of container capacity,
  which can lead to reallocation

Passing vector by value

```
vector<int> load_numbers(vector<int> v)
{
   for (int i = 1; i <= 1000; ++i)
      v.push_back(i);
   return v;
}

vector<int> v;
for (int n = 0; n < 9; ++n)
{
   v.clear();
   v = load_numbers(v);
}
```

Passing vector by value

```
vector<int> load_numbers(vector<int> v)
{  // copy constructor     size 0, capacity 0
   for (int i = 1; i <= 1000; ++i)
      v.push_back(i);   // 10 allocations
   return v;            // move constructor
}

vector<int> v;
for (int n = 0; n < 9; ++n)
{                       // size 1000, capacity 1066
   v.clear();           // size    0, capacity 1066
   v = load_numbers(v); // move assignment
}
```

Passing vector by r-value reference

```
vector<int> load_numbers(vector<int>&& v)
{
   for (int i = 1; i <= 1000; ++i)
      v.push_back(i);
   return v;
}

vector<int> v;
for (int n = 0; n < 9; ++n)
{
   v.clear();
   v = load_numbers(move(v));
}
```

Passing vector by r-value reference

```
vector<int> load_numbers(vector<int>&& v)
{  // no constructor!      size 0, capacity 1000
   for (int i = 1; i <= 1000; ++i)
      v.push_back(i);   // 0 allocations
   return v;
}

vector<int> v;
for (int n = 0; n < 9; ++n)
{                        // size 1000, capacity 1000
   v.clear();            // size    0, capacity 1000
   v = load_numbers(move(v)); // move assignment
}
```

Passing vector by r-value reference

```
vector<int> load_numbers(vector<int>&& v)
{  // no constructor!      size 0, capacity 1000
   for (int i = 1; i <= 1000; ++i)
      v.push_back(i);   // 0 allocations
   return v;            // copy constructor
}

vector<int> v;
for (int n = 0; n < 9; ++n)
{                       // size 1000, capacity 1000
   v.clear();           // size    0, capacity 1000
   v = load_numbers(move(v)); // move assignment
}
```

Passing vector by r-value reference

```
vector<int> load_numbers(vector<int>&& v)
{  // no constructor!      size 0, capacity 1066
   for (int i = 1; i <= 1000; ++i)
      v.push_back(i);    // 0 allocations
   return move(v);       // move constructor
}

vector<int> v;
for (int n = 0; n < 9; ++n)
{                        // size 1000, capacity 1066
   v.clear();            // size    0, capacity 1066
   v = load_numbers(move(v)); // move assignment
}
```

Passing vector by non-const reference

```
void load_numbers(vector<int>& v)
{
   for (int i = 1; i <= 1000; ++i)
      v.push_back(i);
}

vector<int> v;
for (int n = 0; n < 9; ++n)
{
   v.clear();
   load_numbers(v);
}
```

Passing vector by non-const reference

```
void load_numbers(vector<int>& v)
{  // nothing                    size 0, capacity 1066
   for (int i = 1; i <= 1000; ++i)
      v.push_back(i);   // 0 allocations
}

vector<int> v;
for (int n = 0; n < 9; ++n)
{                       // size 1000, capacity 1066
   v.clear();           // size    0, capacity 1066
   load_numbers(v);     // nothing
}
```

Return by value

- Return by value unless you are writing an accessor
- Copy elision (Return Value Optimization) will help
  - It is mandatory under some circumstances in C++17
  - It means the local object is actually the call site object
- If RVO does not occur, an automatic move may occur
  - The compiler tries treating the return expression as an r-value, if that fails it treats it as an l-value
- RVO is better than a move, since nothing happens

Return rules

- Avoid std::move in your return—it will inhibit RVO
- Function return type must be the same as the type you are returning
- You can return a local variable or by-value parameter
- Multiple return statements will often prevent RVO
- Multiple return statements will not affect auto-move
- Conditional expressions in the return statement will often prevent auto-move
  - `return test ? move(x) : move(y);`
  - `if (test) return x; else return y;`

Return Examples - Good

```
foo make_foo()
{ foo x; return x; }

foo change_foo(foo x)
{ return x; }

foo change_foo(foo x, foo y)
{ return test ? move(x) : move(y); }

foo change_foo(foo x, foo y)
{ if (test) return x; else return y; }
```

Return Examples - Bad

```
foo make_foo()
{ foo f; return move(f); }

foo make_foo()
{ foo_like f; return f; }

foo change_foo(const foo& f)
{ return f; }

foo change_foo(foo x, foo y)
{ return test ? x : y; }
```

Moving

- Some things should not or cannot be copied
  - Container contents should not be copied if two copies are not required
  - RAII objects typically control resources which shouldn't be duplicated arbitrarily
  - Unique pointers are a good example
- These things can't be copied, so how do you:
    - Pass them to a function
    - Put them into a vector

Moving

- Move semantics solves these and other problems
- Move semantics can provide a big boost in efficiency
- However, many objects get **no** benefit from moving (vs. copying)
  - An object containing 1000 ints requires copying 4000 bytes whether you call it copy or move
  - This is a pretty obvious case, but what about an object containing one std:string?

The "rule of six"

- There are six special member functions that the compiler may generate for you:
  - Default constructor
  - Destructor
  - Copy constructor
  - Copy assignment operator
  - Move constructor
  - Move assignment operator
- If you write any of them you probably should write all of them
- **It is often the case that you don't need to write any of them**
- Read and watch Howard Hinnant's presentation:
  - http://howardhinnant.github.io/bloomberg_2016.pdf
  - https://youtu.be/vLinb2fgkHk

Moving a string

```
vector<string> v;
v.reserve(...);

string s = "Hello";
s += ", ";
s += "world!";
s += get_lots_of_other_stuff();

v.push_back(s);
```

Moving a string

```
vector<string> v;
v.reserve(...);

string s = "Hello";
s += ", ";
s += "world!";
s += get_lots_of_other_stuff();

v.emplace_back(s);
```

Moving a string

```
vector<string> v;
v.reserve(...);

string s = "Hello";
s += ", ";
s += "world!";
s += get_lots_of_other_stuff();

v.push_back(move(s));
```

Moving a string

```
vector<string> v;
v.reserve(...);

string s = "Hello";
s += ", ";
s += "world!";


v.push_back(move(s));
```

Moving a string

```
vector<string> v;
v.reserve(...);

string s = "Hello";
s += ", ";
s += "world!";


v.emplace_back(move(s));
```

Moving a string

```
vector<string> v;
v.reserve(...);

auto& s = v.emplace_back("Hello");
s += ", ";
s += "world!";
```

Moving a string

```
vector<string> v;
v.reserve(...);

v.push_back("hello");
v.push_back("world");
```

Moving a string

```
vector<string> v;
v.reserve(...);

v.push_back(move("hello"));
v.push_back(move("world"));
```

Moving a string

```
vector<string> v;
v.reserve(...);

v.emplace_back("hello");
v.emplace_back("world");
```

Perfect Forwarding

- Perfect forwarding is used in generic contexts to preserve the r or l value property of a parameter
- Use std::forward when you are passing a parameter (or parameter pack) on, and want the callee to take advantage of move semantics
- You need to use a forwarding reference (universal reference)

Add – A Case Study

```
struct foo {
   foo(int i, double d, char c, const string& s)
   : i(i), d(d), c(c), s(s){}

   int i;
   double d;
   char c;
   string s;
};


foo f(42, 3.1415, 'Q', "Bond");
```

Add – A Case Study

```
struct foo {
   template<typename STR>
   foo(int i, double d, char c, STR&& s)
   : i(i), d(d), c(c), s(forward<STR>(s)) {}

   int i;
   double d;
   char c;
   string s;
};

foo f(42, 3.1415, 'Q', "Bond");
```

Add – A Case Study

```
class bar {
   vector<foo> foos;
public:
   void add(const foo& f)
   {
      foos.push_back(f);
   }
};


bar b;
b.add(foo(42, 3.1415, 'Q', "Bond"));
```

Add – A Case Study

```
class bar {
   vector<foo> foos;
public:
   void add(int i, double d, char c,
            const char* s)
   {
      foos.push_back(foo(i, d, c, s));
   }
};

bar b;
b.add(42, 3.1415, 'Q', "Bond");
```

Add – A Case Study

```
class bar {
   vector<foo> foos;
public:
   void add(int i, double d, char c,
            const char* s)
   {
      foos.push_back({i, d, c, s});
   }
};

bar b;
b.add(42, 3.1415, 'Q', "Bond");
```

Add – A Case Study

```
class bar {
   vector<foo> foos;
public:
   void add(int i, double d, char c,
            const char* s)
   {
      foos.emplace_back(i, d, c, s);
   }
};

bar b;
b.add(42, 3.1415, 'Q', "Bond");
```

Add – A Case Study

```
class bar {
   vector<foo> foos;
public:
   void add(int i, double d, char c,
            const string& s)
   {
      foos.emplace_back(i, d, c, s);
   }
};

bar b;
b.add(42, 3.1415, 'Q', "Bond");
```

Add – A Case Study

```
class bar {
   vector<foo> foos;
public:
   template<typename... T>
   void add(T&&... t)
   {
      foos.emplace_back(forward<T>(t)...);
   }
};

bar b;
b.add(42, 3.1415, 'Q', "Bond");
```

Container Choice

- Why it matters
  - Each container has a purpose
  - Each container has different tradeoffs between speed and space and convenience
  - New containers are under consideration to offer more speed/space/convenience options

Container Choice

- When should I use vector?

Container Choice

- When ~~should~~ shouldn't I use vector?
- Vector is the go-to for dynamic storage
- Each other container offers special properties
  - Performance
  - Convenience

Vector

- Contiguous memory
- Fastest possible traversal
- Random access
- Growth invalidates everything *if it grows*
  - When you know the final size (even roughly) pre-reserve
  - Pre-reserve to avoid thrashing (if you can afford it)
  - You can shrink to fit later (but it may not help)
- Geometric growth behavior makes large vectors impossible unless the maximum size is known ahead of time

Array

- Contiguous memory
- Fastest possible traversal
- Random access
- Fixed size
- Local object (stack or embedded)

Vector vs. Array vs. C-array

- Use a vector if the size is not fixed
  - We may get a new container someday that addresses this limitation
- Use a C-array for simple C-like situations
  - Principle of using the simplest tool for the job
- Use an array when you need elaborate container-like behavior

Deque

- Clumps of contiguous memory (and an index)
- Fast traversal
- Random access
- Growth invalidates only iterators
- Linear growth behavior makes large deques work very well
- Clumps and index overhead make small deques very wasteful

Deque vs. Vector

- Use a vector if the container is small
- Pushing onto the front of a vector is faster until container size is surprisingly big
- Vector is much smaller for small containers
- Use a deque when the container may get large

List

- Node-based
  - Overhead is very high
  - Locality of reference is terrible
- Slow traversal
- No random access
- Growth invalidates nothing , nodes never move
- Linear growth behavior makes large lists possible, but overhead can be a problem
- Small lists are very wasteful and slow

List vs. Vector

- Use a vector if the container is small
- Inserting anywhere in a vector is faster until container size is surprisingly big
- Vector is much, much smaller for any size (but has the growth problem)
- Use a list when the container is large and you are doing lots of inserts and/or deletes in the middle
- Use a list when you can benefit from splicing

List vs. Deque vs. Vector

Counts	Container	Access Time (s)	Memory (GB)
10M/10	List	8.1	4.34
	Deque	3.7	3.05
	Vector	2.5	1.29
1M/100	List	6.7	3.33
	Deque	1.7	1.21
	Vector	2.3	0.50
100K/1000	List	6.4	3.23
	Deque	1.4	1.03
	Vector	8.6	0.43

A large number of small elements in a **map**. Each element has a container of ints. Ints were inserted one at a time at the front of the container, then removed one at a time from the front. Measurements were taken for **list**, **deque** and **vector**. In the **vector** case the correct size was reserved in advance. Access Time is the time spent adding and removing the ints; the time spent building and destroying the outer map was not included. Times were measured with the operating system's microsecond-precision interval timer. Memory usage is the working set after loading the ints.

Set/Map

- Node-based
  - Overhead is very high
  - Locality of reference is terrible
- Slow traversal
- No random access
- Growth invalidates nothing, nodes never move
- Linear growth behavior makes large containers possible, but overhead can be a problem
- Small containers are very wasteful and slow

Set/Map vs. Vector

- Use a vector if the container is small
- Arbitrary insertion is faster until the size is quite large
- Maintaining order is faster until the size is quite large
- Binary search on vector works just as well
- For smallish containers, linear search is faster
- Pre-reserve and the vector will not move
- Use a set or map when the container is large

Set vs. Vector

```
struct element {
    set<int>   container;
    int        id;
};
```

Set vs. Vector

```
struct element {
    vector<int>    container;
    int            id;

    void insert(int i)
    {
        for (auto it(container.begin()), end(container.end());
              it != end; ++it)
            if (*it >= i)
            {
                if (*it > i)
                    container.insert(it, i);
                return;
            }
        container.push_back(i);
    }
};
```

Set vs. Vector

Counts	Container	Access Time (s)	Memory (GB)
10M/10	Set	6.9	6.11
	Vector	4.8	1.45
1M/100	Set	8.3	4.95
	Vector	4.3	0.68
100K/1000	Set	12.6	4.83
	Vector	19.6	0.45

A large number of small elements in a **map**. Each element has an ordered, unique container of ints. Ints were inserted one at a time in decreasing order, then inserted one at a time in increasing order. Measurements were taken for **set** and **vector**. In the **vector** case the correct size was assumed to be unknown and was *not* reserved in advance. Access Time is the time spent inserting the ints; the time spent building and destroying the outer map was not included. Times were measured with the operating system's microsecond-precision interval timer. Memory usage is the working set after loading the ints.

One container to rule them all!

vector

Not moving

- Some things cannot (or should not) even be moved
  - Object may not get any benefit from moving (no owned resources) – moving is just as expensive as copying
  - Construction and destruction may have side effects (RAII)
    - Rocket: constructor launches, destructor blows up
    - DB: constructor establishes connection, destructor releases
  - You may want the object to be stable so you can keep references to it
    - Node-based containers are inherently stable
    - Vectors can be used in a stable way
- Large numbers of objects
  - May be expensive to construct more than once

Constructing in place

- When to use emplace
  - To avoid copying or moving
  - With unmovable types
  - To ensure lifetime stability
  - To get a default-constructed element

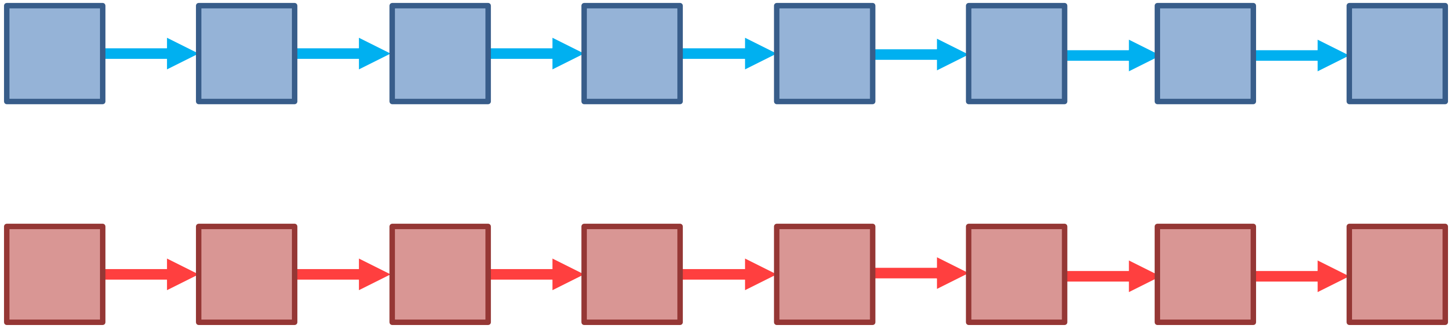
```
v.emplace_back().read(file);
```

- When not to use it
  - When it's a copy anyway (e.g. built-in types)
  - When you want to ensure that explicit constructors are not called (other things being equal)

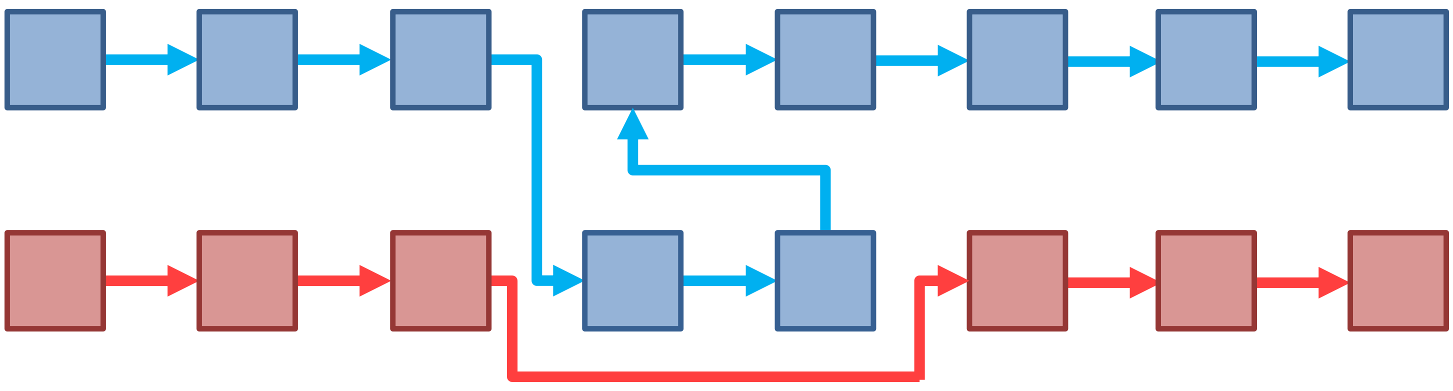
Splicing

- Lists have a splice method
- Splicing lets you transfer elements to another list for the price of a couple of pointer swaps
- This is a great example of not moving
- No references or iterators are invalidated
- It is one of the few reasons to use list

Splicing



Splicing



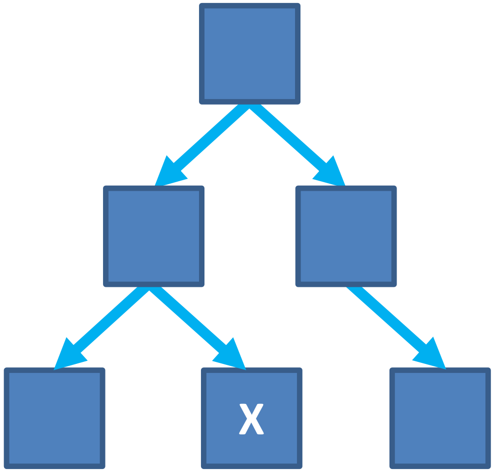
Node Extraction

- Until C++17, sets and maps did not have anything like a splice method
  - You could not change the key of an element
  - You could not transfer an element to another container
  - You had to create new elements and delete old ones

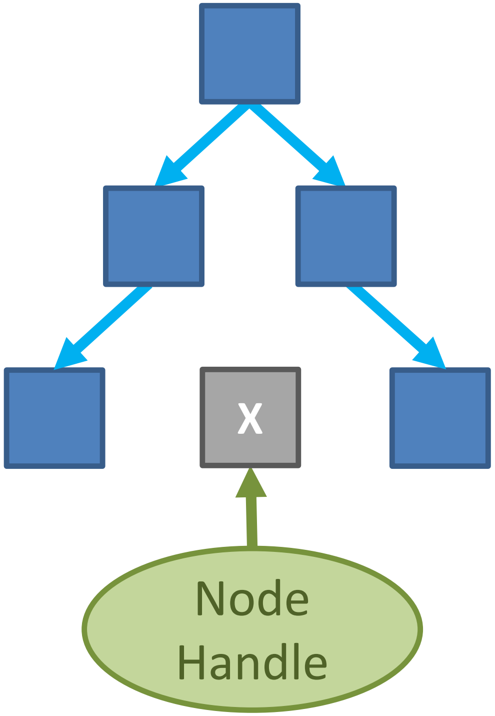
Node Extraction

- C++17 has node extraction
  - You can change the key of an element
  - You can transfer an element to another container with compatible nodes
    - e.g. map -> multimap
  - You can merge one container into another
  - You can remove an element and hold it for later use (even surviving the destruction of the container)
  - You can move an element out of a set

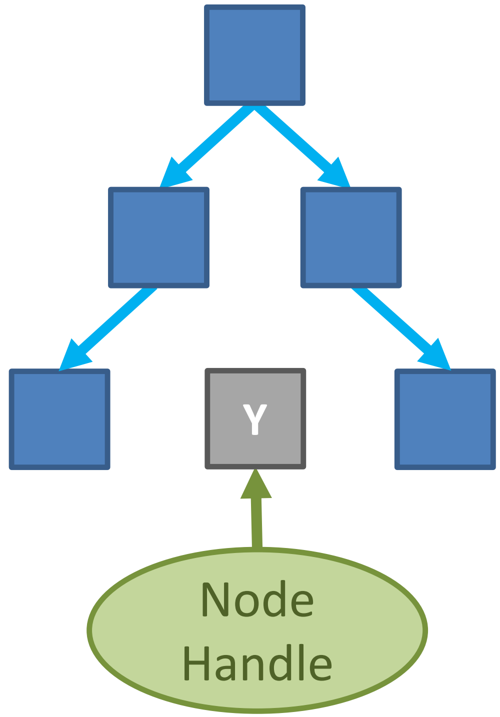
Changing an Element Key



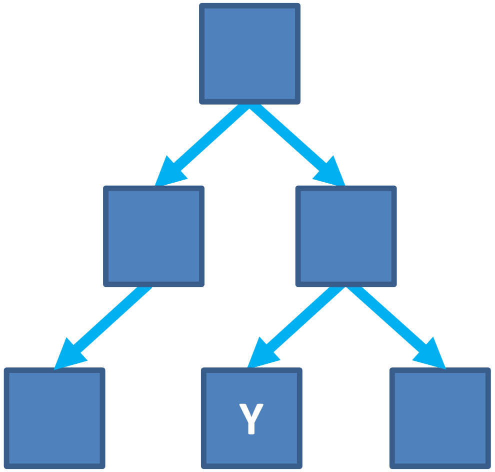
Changing an Element Key



Changing an Element Key



Changing an Element Key



Changing an Element Key

```
map<int, string> m{
   {1,"mango"}, {2,"papaya"}, {3,"guava"}
};

auto nh = m.extract(2);
nh.key() = 4;
m.insert(move(nh));

// m == {{1,"mango"}, {3,"guava"}, {4,"papaya"}}
```

Merging Sets

```
set<int> src{1, 3, 5};
set<int> dst{2, 4, 5};

dst.merge(src);

// src == {5}
// dst == {1, 2, 3, 4, 5}
```

Efficient Factories

```
auto new_record(const char* str)
{
   static int id = 0;
   map<int, string> temp;
   temp.emplace(++id, str);
   return temp.extract(temp.begin());
}

map<int, string> table;
table.insert(new_record("Hello"));
table.insert(new_record("World"));

// table == {{1,"Hello"}, {2,"World"}}
```

Questions and Discussion