

Hashing

Dietmar Kühl
Bloomberg LP
dkuhl@bloomberg.net

Overview

- why hashing?
- hash functions
- hash values for user defined types
- variations in hashing containers

Value Look-Up

key -> value

Value Look-Up

$k \rightarrow v$

```
vector<pair<K, V>> c = ...;  
auto it = find_if(begin(c), end(c),  
                 [=](auto&& p){ return v == p.first; });  
if (it == end(c)) { not-found }  
else { use it->second }
```

Value Look-Up

$k \rightarrow v$

```
vector<pair<K, V>> c = ...;  
auto it = lower_bound(begin(c), end(c),  
                      [=](auto&& p){ return k < p.first; });  
if (it == end(c) || it->second != k) { not-found }  
else { use it->second }
```

implications: $k < x$ is defined and c is sorted accordingly

Value Look-Up

$k \rightarrow v$

```
map<K, V> c = ...;  
auto it = c.find(k);
```

```
if (it == end(c)) { not-found }  
else { use it->second }
```

implications: $k < x$ is defined

Value Look-Up

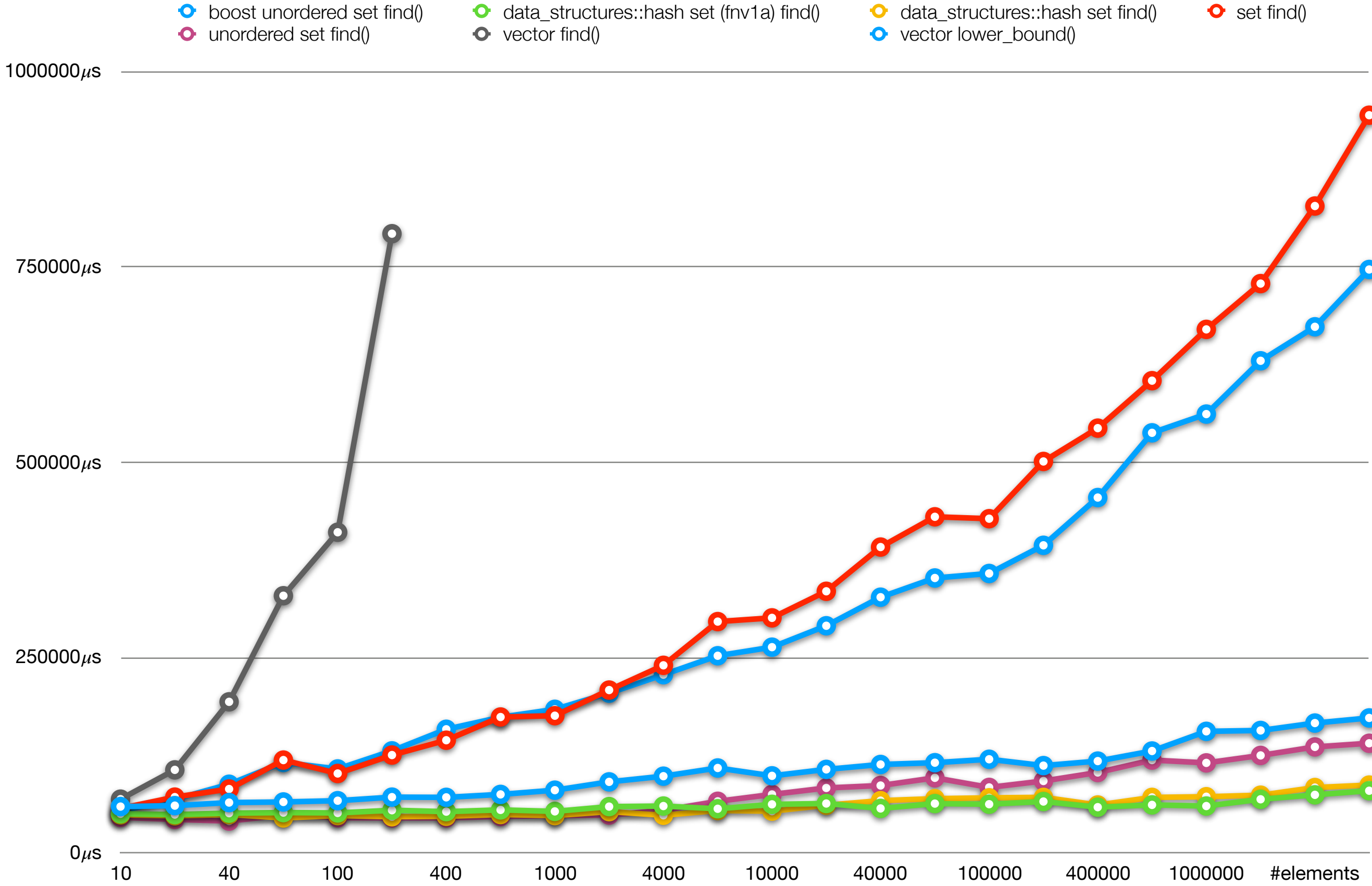
$k \rightarrow v$

```
unordered_map<K, V> c = ...;  
auto it = c.find(k);
```

```
if (it == end(c)) { not-found }  
else { use it->second }
```

implications: hash(k) is defined

Comparing Algorithms



Hash Caveats

- *expected* $O(1)$ access
 - $O(1)$ access assumes at most few hash conflicts
- worst case access: $O(n)$ (or $O(\log n)$ when using a tree to deal with conflicts)
- computing a better hash is often slower

Hashes as DoS Target

- hash algorithms are exploited by denial of service attacks:
 - adversary feeds keys known to create conflicts
- counter measures:
 - use variation of hash algorithms
 - use strong hash algorithms creating few conflicts
 - *seed* the hash value with random values

Scope of Hashes

- always the same hash value, e.g., for persistent storage
⇒ use a strong hash algorithm to counter conflicts
- the same hash value within a process
⇒ use a fairly strong hash algorithm seeded per run
- the same hash value for the short-lived containers
⇒ different seed per container for a faster hash algorithm

How To: Hash Algorithm

Do **not** design your own hash algorithm!

use a suitable existing one instead

Considerations

- the hash computation should be fast
- hash values should be uniformly distributed
 - to allow hash tables to use any bits for bucketing
 - flip 1 bit in the input \Rightarrow flip about 1/2 of result bits
- predictable hash \Rightarrow potential for denial of service attack

General Layout of Hashing Algorithms

- some **initial** setup
- **combining** all bytes into an intermediate state
 - should change for each bit changed in the input
- **reduce** the intermediate state into one integer

Jenkins

```
template <typename Range>
uint32_t jenkins(Range range) {
    Result result{};
    for (uint8_t c: range) {
        result += c;
        result += result << 10;
        result ^= result >> 6;
    }
    result += result << 3;
    result ^= result >> 11;
    return result += result << 15;
}
```

FNV1

```
template <typename Range>
std::uint32_t fnv1(Range range) {
    std::uint32_t result{0x811c9dc5};
    for (unsigned char octet: range) {
        result *= std::uint32_t{16777619};
        result ^= octet;
    }
    return result;
}
```


Pearson

```
template <typename FwdIt>
std::uint32_t pearson(FwdIt b, FwdIt e) {
    unsigned char const (&table)[256] = { /* ... */ };
    uint32_t result{};
    for (int s = 0; s != 32; s += 8) {
        unsigned byte = table[(*b + s / 8) % 256];
        for (uint8_t c: range{++FwdIt{b}, e})
            byte = table[byte ^ c];
        result |= Result{byte} << s;
    }
    return result;
}
```

Pearson (unrolled)

```
template <typename InIt, std::size_t... I>
std::uint32 pearsonr(InIt b, InIt e,
                    std::index_sequence<I...>) {
    uint8_t const (&table)[256] = { /*...*/ };
    uint8_t byte[] = { table[*b + I] % 256... };
    for (uint8_t c: range{++b, e}) {
        (void)((byte[I] = table[byte[I] ^ c]) |...);
    }
    return ((uint32_t{byte[I]} << (I * 8)) |...);
}
```

Murmur

```
template <typename Rndlt> std::uint32_t murmur(Rndlt begin, Rndlt end){
    std::uint32_t hash{seed}, len{end - begin};
    for (; 3 < end - begin; begin += 4) {
        std::uint32_t k{decode(begin)};
        k = rotate<15>(k * 0xcc9e2d51) * 0x1b873593;
        hash = rotate<13>(hash ^ k) * 5 + 0xe6546b64;
    }
    std::uint32_t remain{};
    for (; begin != end; ++begin)
        remain = (remain << 8) | std::uint8_t(*begin);
    remain = rotate<15>(remain * 0xcc9e2d51) * 0x1b873593;
    hash ^= remain; hash ^= len;
    hash ^= hash >> 16; hash *= 0x85ebca6b;
    hash ^= hash >> 13; hash *= 0xc2b2ae35;
    return hash ^= hash >> 16;
}
```

User Defined Types

- $a == b \Rightarrow \text{hash}(a) == \text{hash}(b)$
- wanted: $\text{hash}(a) == \text{hash}(b) \Rightarrow a == b$ with high probability
- just feed the bytes of the type to a hash algorithm?
 - **don't** hash pointers or non-salient attributes
 - there'll be dragons: $-0.0 == 0.0$
 - padding bits will haunt you

Example

```
struct foo {  
    std::unique_ptr<int> p;  
    float                f;  
    short               i;  
    char                c;  
};
```

```
foo f0{ std::make_unique<int>(0x12345678), -0.0, 0x1234, 'A' };  
foo f1{ std::make_unique<int>(0x12345678), 0.0, 0x1234, 'A' };  
std::cout << "f0=" << f0 << "\n" << "f1=" << f1 << "\n";
```

```
f0=[ 70 03 c0 4f ea 7f 00 00 00 00 00 80 34 12 41 00 ]
```

```
f1=[ 20 29 c0 4f ea 7f 00 00 00 00 00 00 34 12 41 00 ]
```

hash_expansion()

- basic idea: recursively decompose objects and add bits:

```
hash_code hash_expansion(hash_code h, X const& value) {  
    h = hash_code::combine(std::move(h), value.member1);  
    h = hash_code::combine(std::move(h), value.member2);  
    return h;  
}
```

- using hash_expansion()/hash_code uses a default algorithm

hash_expansion()

- basic idea: recursively decompose objects and add bits:

```
hash_code hash_expansion(hash_code h, X const& value) {  
    h = hash_code::combine(std::move(h), value.member1);  
    h = hash_code::combine(std::move(h), value.member2);  
    return h;  
}
```

- using hash_expansion()/hash_code uses a default algorithm

hash_expansion()

- basic idea: recursively decompose objects and add bits:

```
template <typename HashCode>
hash_code hash_expansion(HashCode h, X const& value) {
    h = HashCode::combine(std::move(h), value.member1);
    h = HashCode::combine(std::move(h), value.member2);
    return h;
}
```

- using a template argument allows customised algorithms

hash_expansion()

- structs just append their respective members
- unions/variants append the active member and descriptor
- optional appends flag and, if present, the data
- sequences append values and the size
- performance specialisation:
contiguous sequences without padding directly added

unordered_map<K, V, H>

- default hash: `std::hash<K>`
 - one to hash them all
- custom hash function: can still be used:

```
unordered_map<K, V, custom<K>> map;
```

```
unordered_map<K, V, stateful<K>> map(stateful<K>(seed));
```

std::hash<T>

- needs to use `hash_expansion(hash_code, T)` if available:
 - using the default is what most users do
- hash needs to change between each run:
 - otherwise users will depend on hash values
 - \Rightarrow impossible to use an improved default algorithm

Transparent Hash

- consider avoiding temporary objects:

```
unordered_map<string, T> map = ...;
```

```
char buffer[100];  
auto size = read(buffer);
```

```
auto it = map.find(string_view(buffer, buffer + size));
```

- string and string_view need to produce the same hash
- it needs to be detectable that hashes are compatible

unordered_map<K, V>

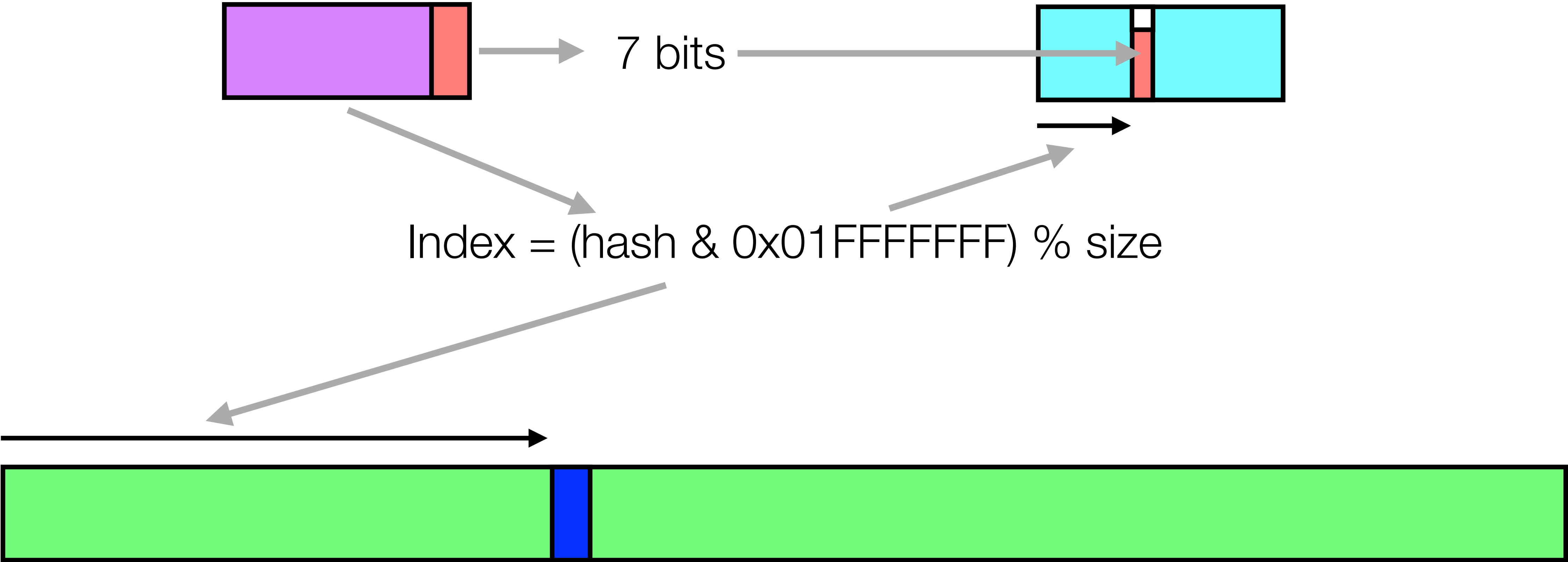
- a good starting point for an associative container
- not necessarily the best alternative, though:
 - element stability \Rightarrow more memory and memory access
 - iterator stability \Rightarrow can't even use "open addressing"

Google's Hash Idea

<https://youtu.be/ncHmEUmJZf4>

- use open addressing as collision handling
 - store the elements directly in the array
- use a descriptor array with one char per element
 - high bit to indicate empty and other bits deleted flag
 - other 7 bits used from the hash
 - ⇒ SIMD instructions to locate the likely cell

Google's Hash Idea



Lock-Free Hash

<http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/>

- reasonably common use case:
 - one updating thread, multiple reading threads
- lock-free hash map is quite viable
 - an implementation ships, e.g., Thread Building Blocks

Summary

- hash maps are an important data structure of efficiency
- a good hashing function is needed for effective use
- `hash_expansion()` for hashes for user-defined types
- hash maps come in variations, e.g., for collision resolution

Questions



Thank You!

Dietmar Kühl dkuhl@bloomberg.net