TR1: C++ on the *Move*

Pete Isensee

Director
XNA Developer Connection
Microsoft

Welcome to GDC

- Do a little of everything
- Expand your horizons
- Talk with other developers
- Enjoy the city

C++ TR Defined

- Technical Report
- An “informative document”
- Not part of the C++ Standard
- May become part of a future standard
- Or not

TR1: A History

- Formal work began in 2001
- Most proposals came from Boost members
- Approved in 2006 by ISO
- In Spring 2006 all TR1 except math functions added to draft of next Standard
- Next Standard (C++0x) is due before the decade is out

What’s in TR1

- Fixed-sized arrays
- Hash tables
- Smart pointers
- Function objects
- Type traits
- Random number generators
- Tuples
- Call wrappers
- Regular expressions
- Advanced math functions

Why You Should Care

- Efficiency
- TR1 is
  - Lean and mean: more on that in a bit
  - Standard: learn once, use everywhere
  - Proven: avoid the not-invented-here syndrome
  - Available: today, on platforms you care about

Where To Find TR1

- Boost.org
- Dinkumware.com
- Gcc.gnu.org
- Metrowerks

Getting Started

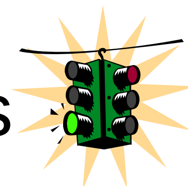
```
// TR1 header
#include <array>

// fully qualified namespace
std::tr1::array< int, 4 > a;

// or more likely ...
using namespace std::tr1;
array< int, 4 > a;
```

Arrays: True Containers at Last

- array< class T, size_t N >
- Size fixed at compile time; contiguous space
- Same performance as C-style arrays
- Member functions you'd expect
  - begin, end, size, op [], at, front, back
- Works with all std algorithms
- Can be initialized using standard array initializers

```
std::tr1::array< int,4 > a = { 1,2,3,4 };
```

Arrays: Take Control

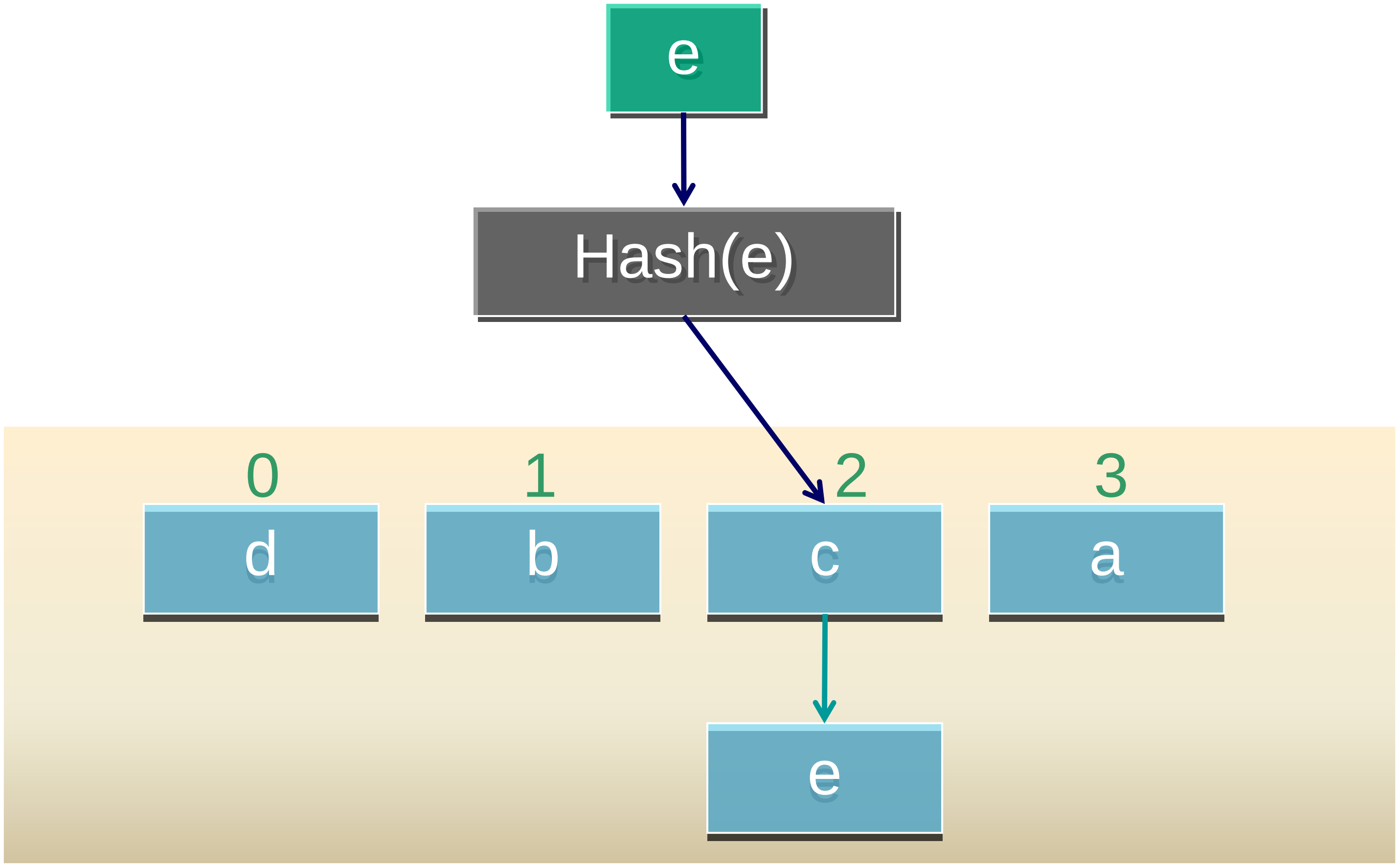
- Advantages 
  - Performance
  - Size is part of the type
  - Easy to convert array code to/from std::vector
  - More secure: must call data() to get ptr
- Disadvantages 

  - Not growable
  - No push_back, reserve, resize
  - Must call data() to get ptr: inconvenient
  - array.swap() is O(n)
    - vector is O(1)

Hash Tables

- Considered for original standard
- Basic concept
    - Super-fast search
    - You control the hash algorithm
    - Similar interface to other STL containers
    - Low overhead: on par with set/map

Hash Table Conceptual Diagram



Unordered Associative Containers

- Unordered
  - A traversal is not ordered like set/map
- Associative
  - Dictionary pairs (K,T)
- Containers
  - Work with std iterators and algorithms
- TR1 naming
  - std::tr1::unordered_ [multi] set< K >
  - std::tr1::unordered_ [multi] map< K,T >

Hash Table Code Example

```
#include <unordered_set>
using namespace std::tr1;
unordered_set<int> ht;

ht.insert( 41 ); // add elements
ht.insert( 11 );
ht.insert( 18 );

unordered_set<int>::iterator it;
it = ht.find( 41 ); // find element
```

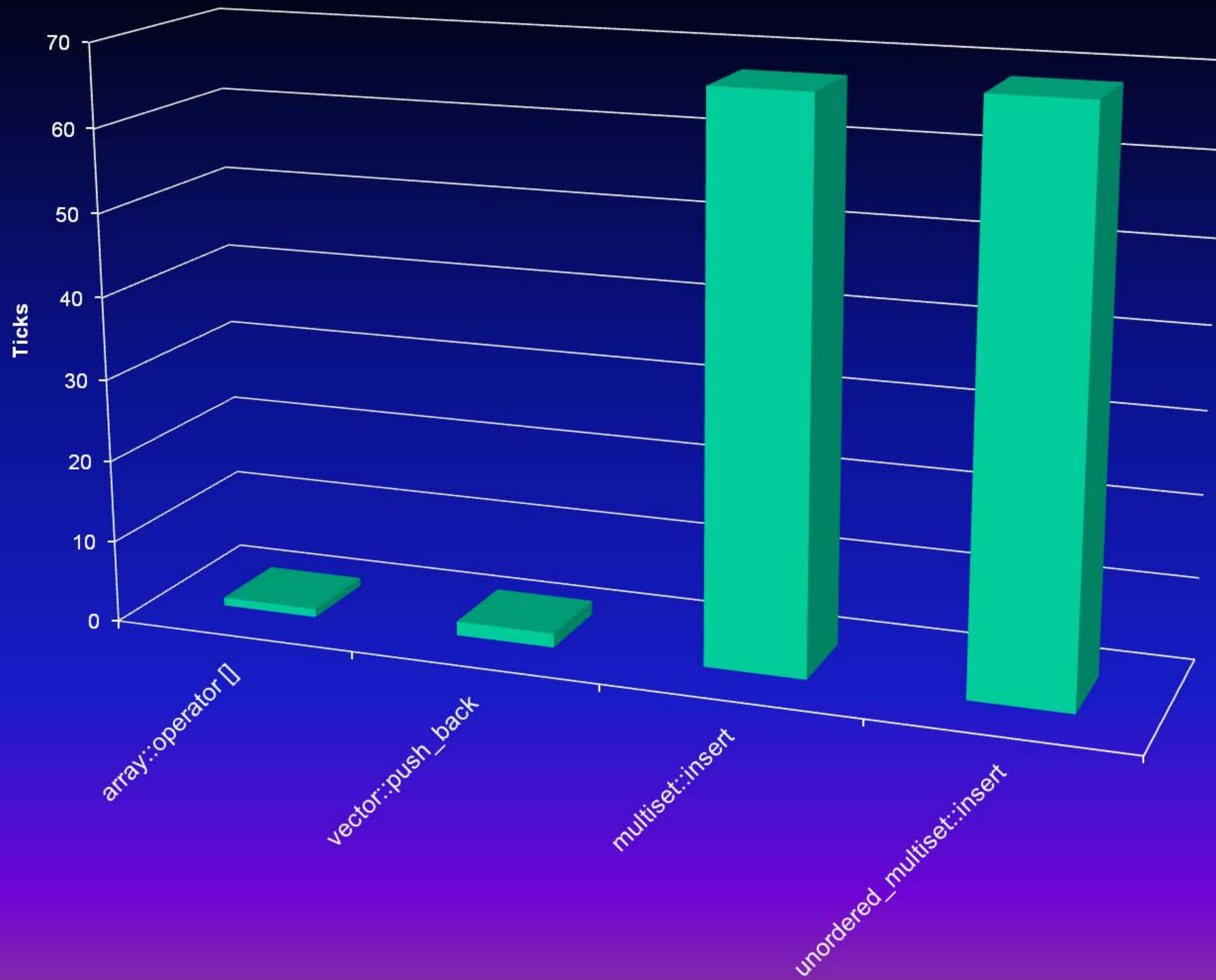
Real-World Performance

- Evaluated three types of objects
  - Light-weight: int
    - Hash = std::tr1::hash<int>
  - Medium-weight: std::complex<double>
    - Hash = (size_t)(real + imag)
  - Heavy-weight: bitmap
    - Allocates, copy ctor calls memcpy
    - Hash = (size_t)(sum of first few pixels)
- 50,000 items in hash table
- Tested on Xbox 360
  - Few perturbations = consistent results

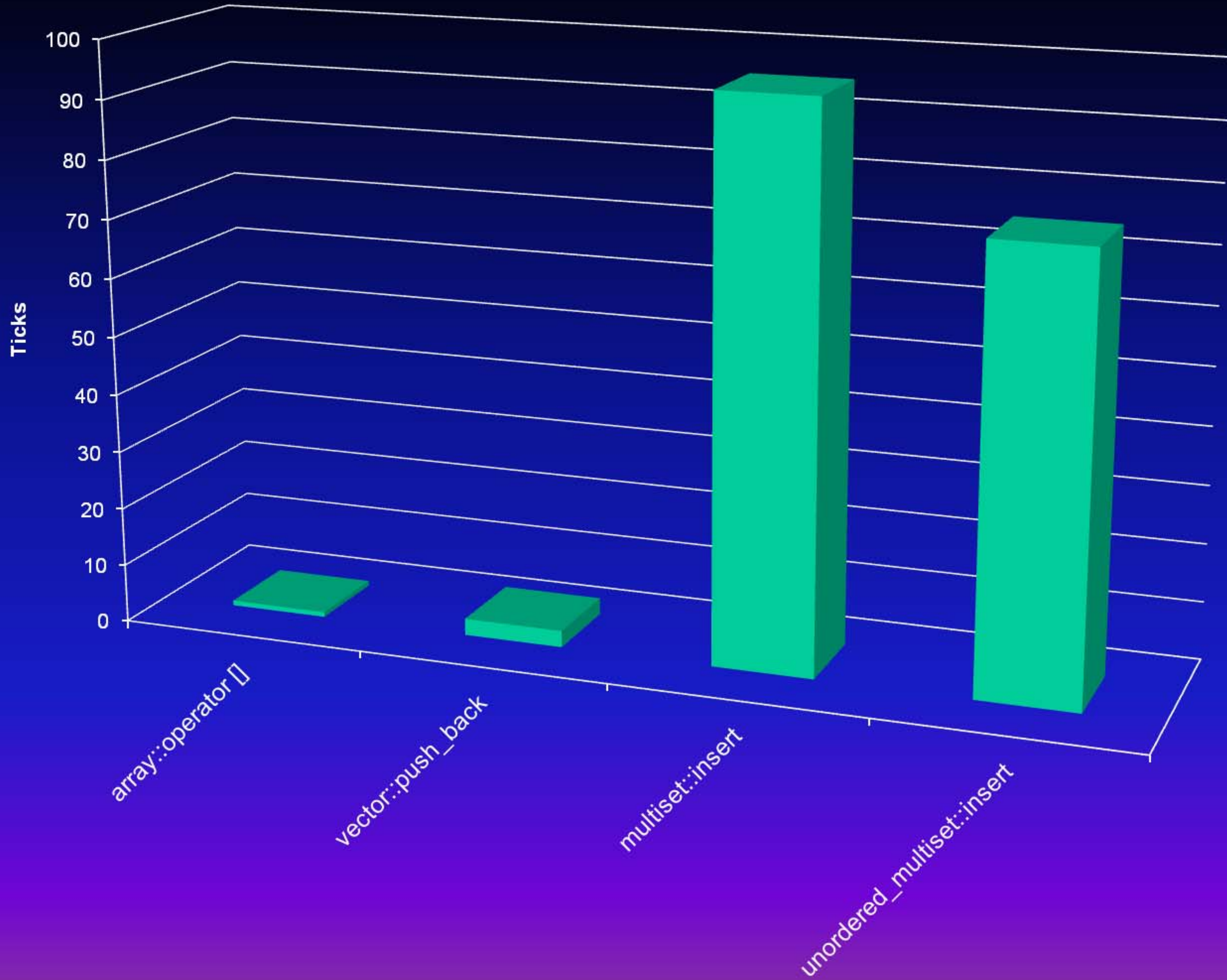
Element Insertion Performance

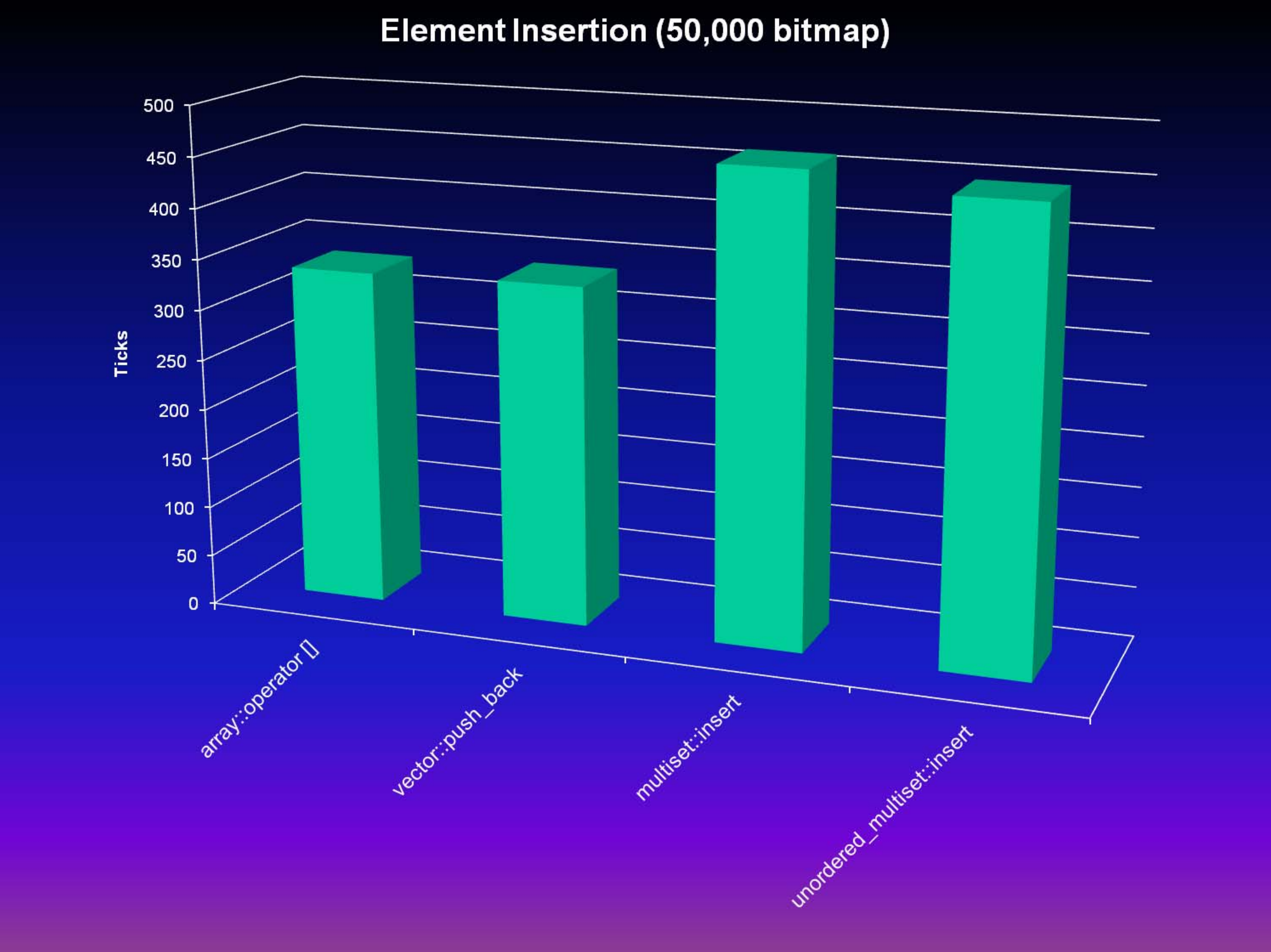
- Sequence containers (vector, deque)
  - Complexity: O(1)
- Associative containers (set, map)
  - Complexity: O(log n)
- Unordered associative containers
  - Average time complexity: O(1)
  - Worst-cast time complexity: O(n)
- What was real-world performance?

Element Insertion (50,000 int)

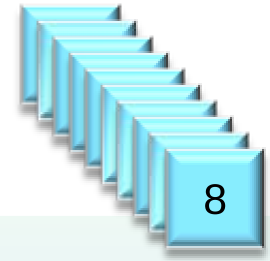


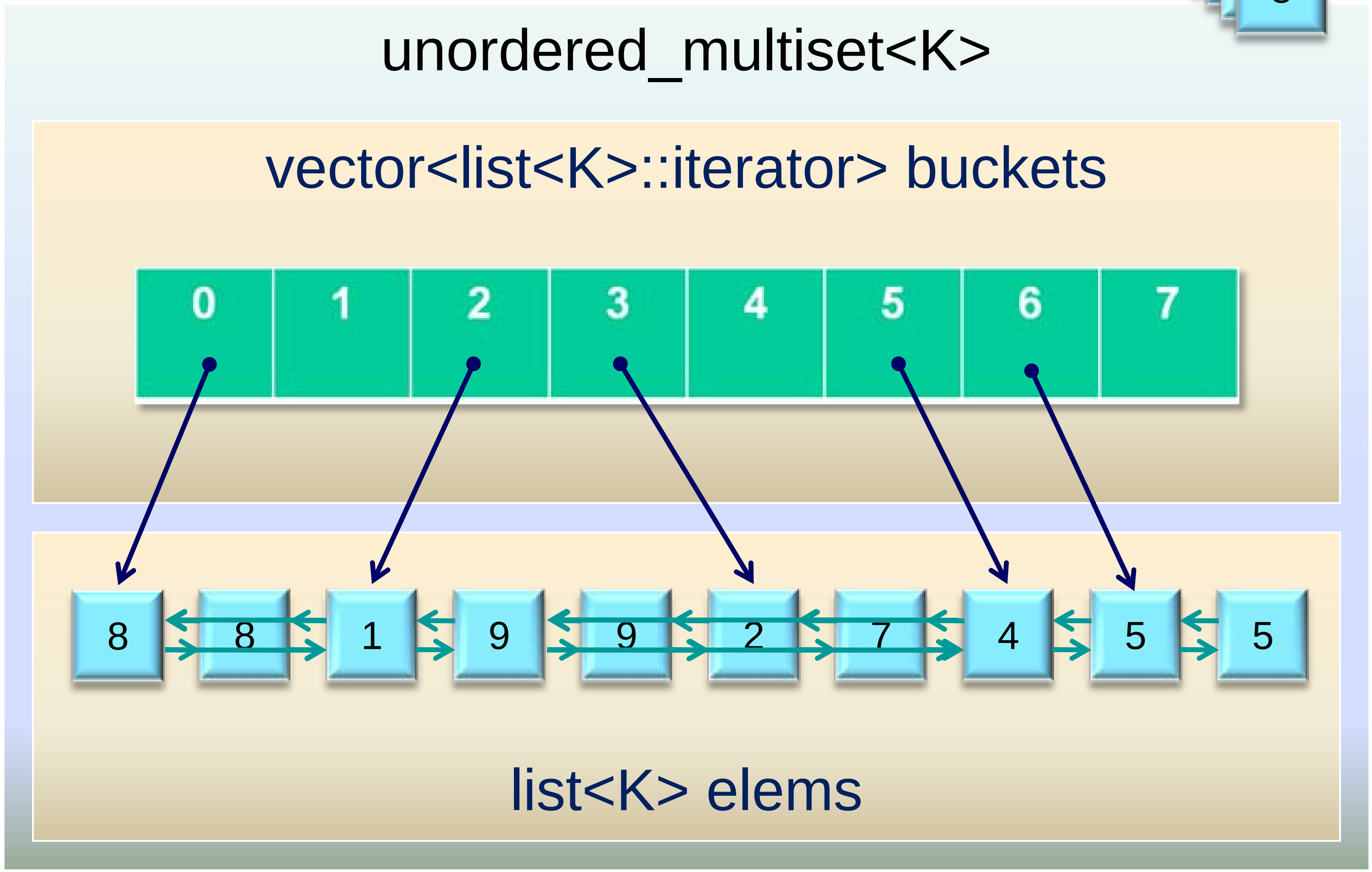
Element Insertion (50,000 complex<double>)




Element Insertion (50,000 bitmap)
Ticks
500
450
400
350
300
250
200
150
100
50
0
array::operator []
vector::push_back
multiset::insert
unordered_multiset::insert

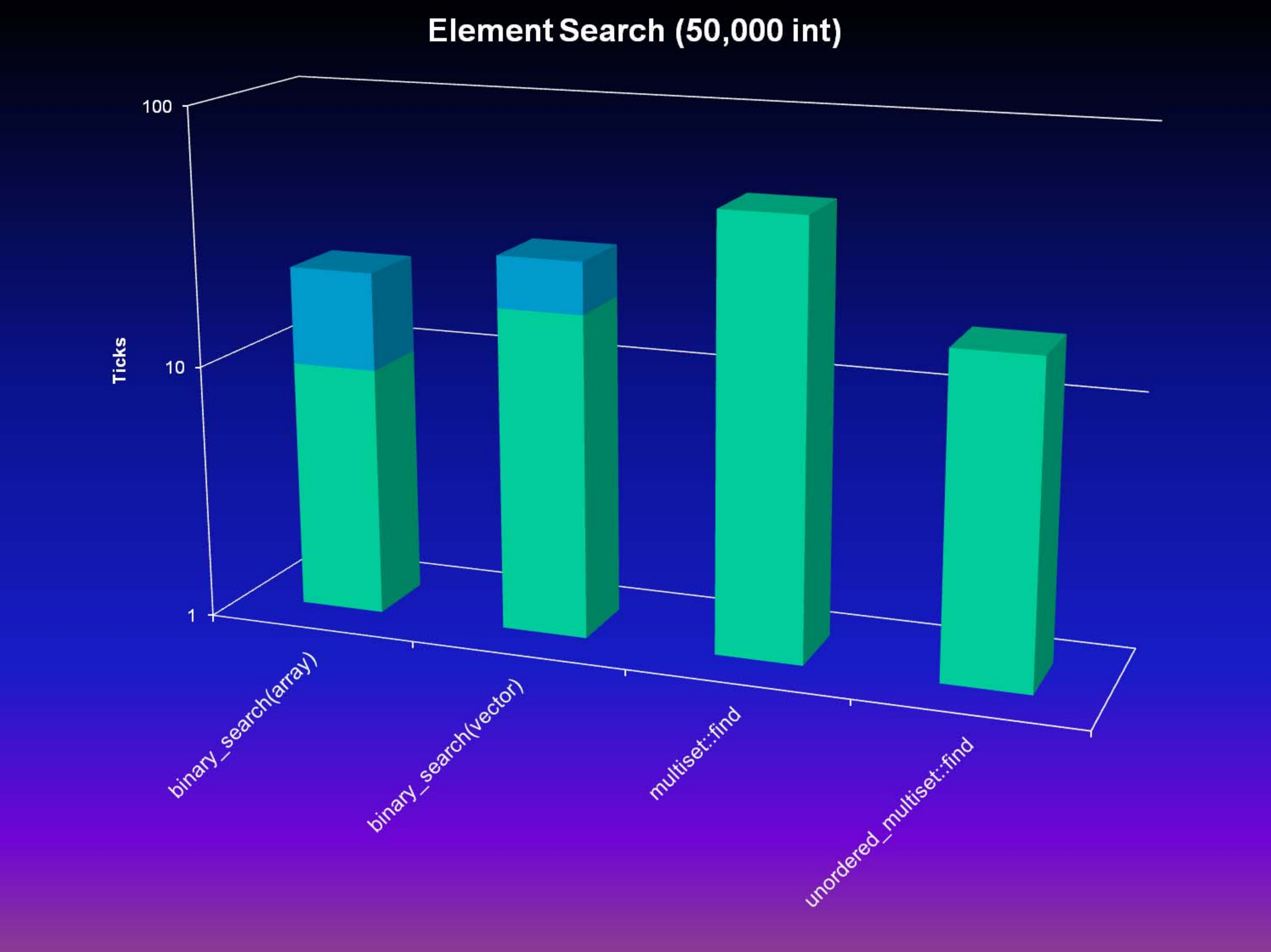
Implementation Example





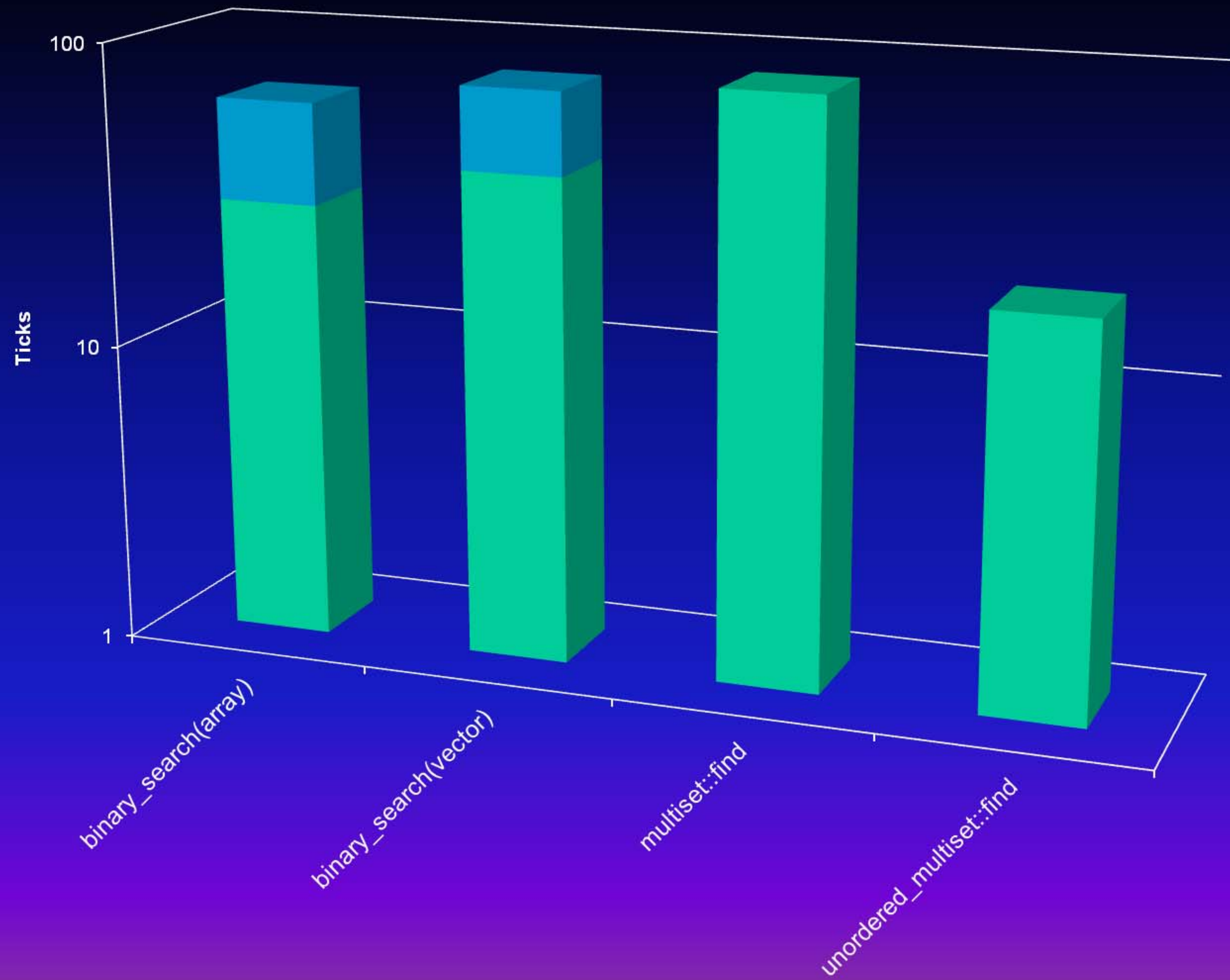
Search Performance

- Sequence containers (vector, array)
  - Complexity: O(n), O(log n) if sorted
- Associative containers (set, map)
  - Complexity: O(log n)
- Unordered associative containers
  - Average time complexity: O(1)
  - Worst-cast time complexity: O(n)
- What was real-world performance?

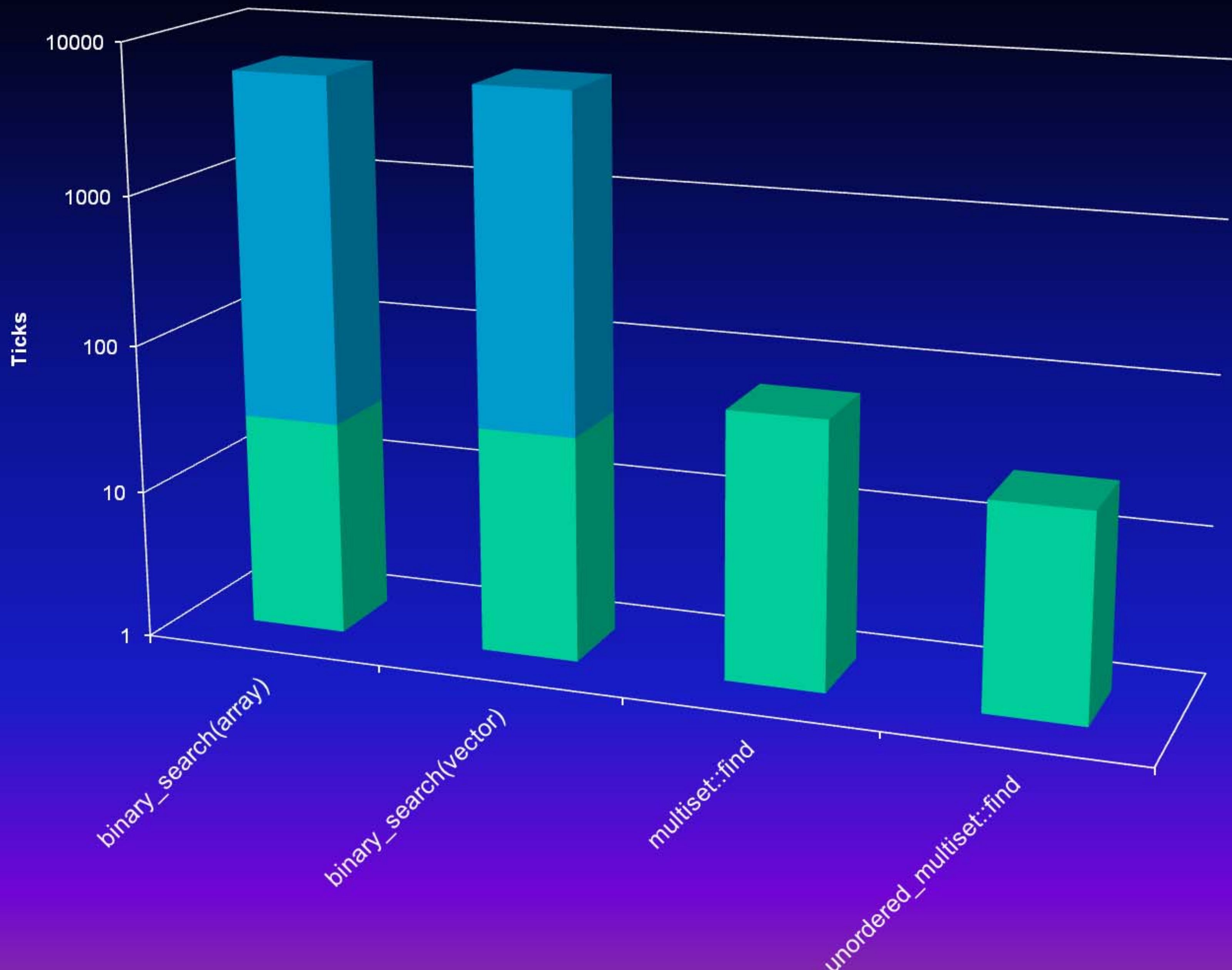


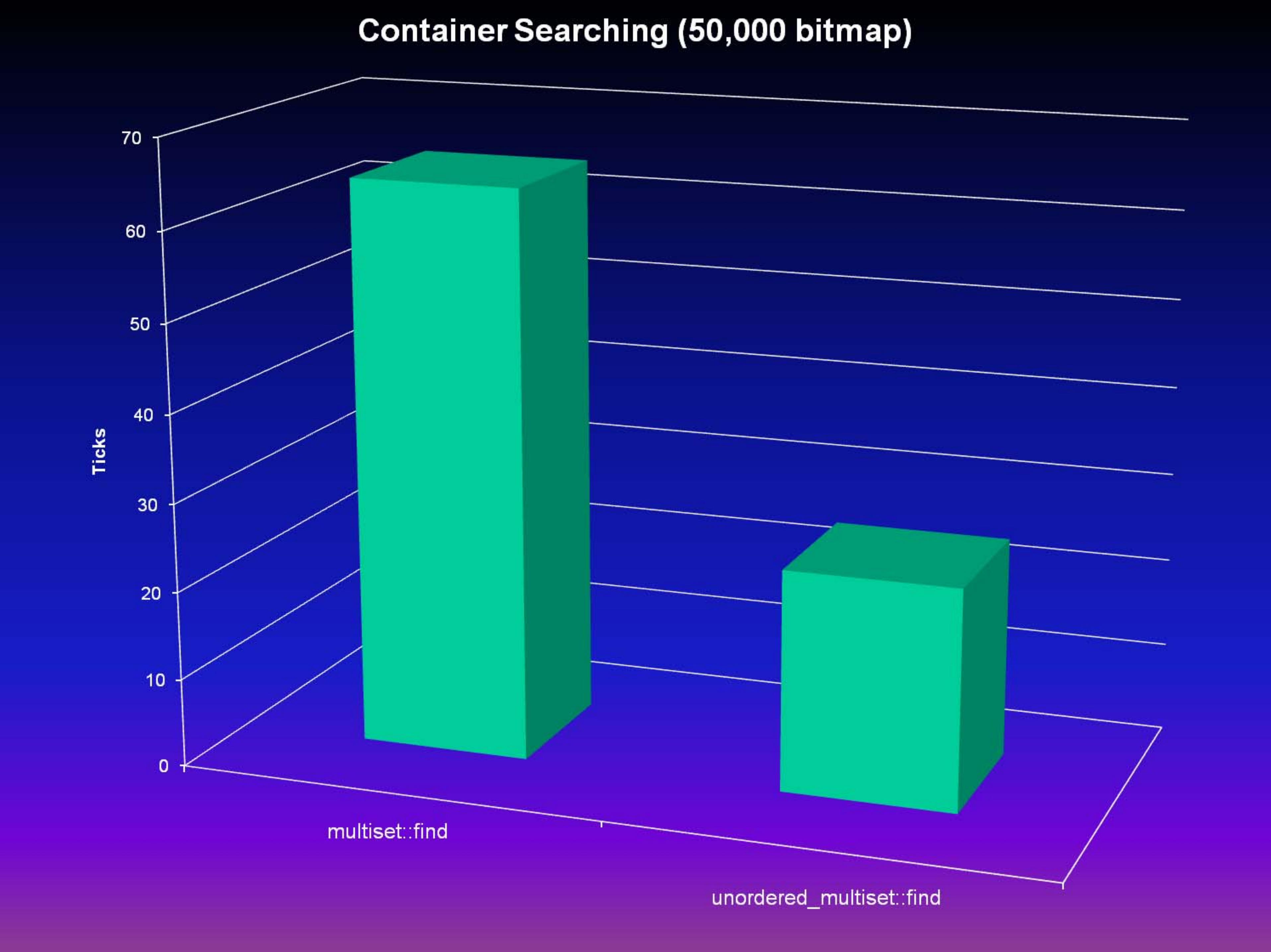
Element Search (50,000 int)
Ticks
100
10
1
binary_search(array)
binary_search(vector)
multiset::find
unordered_multiset::find

Element Search (50,000 complex<double>)



Element Search (50,000 bitmap)





Container Searching (50,000 bitmap)
Ticks
70
60
50
40
30
20
10
0
multiset::find
unordered_multiset::find

Hash Functions

- Choosing the right hash function is critical!
- Defaults provided for built-ins and std::string
- For examples, see <unordered_set>
- Hash references
    - *The Art of Computer Programming*, Vol 3, Knuth
    - *Algorithms in C++*, Sedgewick

Custom Hash Functions

```cpp
typedef std::complex<double> cpx;
struct CpxHash
{
    size_t operator()(const cpx& c) const
    {
        return (size_t)(c.real()+c.imag());
    }
};
// Specify hash fn as template param
unordered_set< cpx, CpxHash > ht;
```

Load Factors and Rehashing

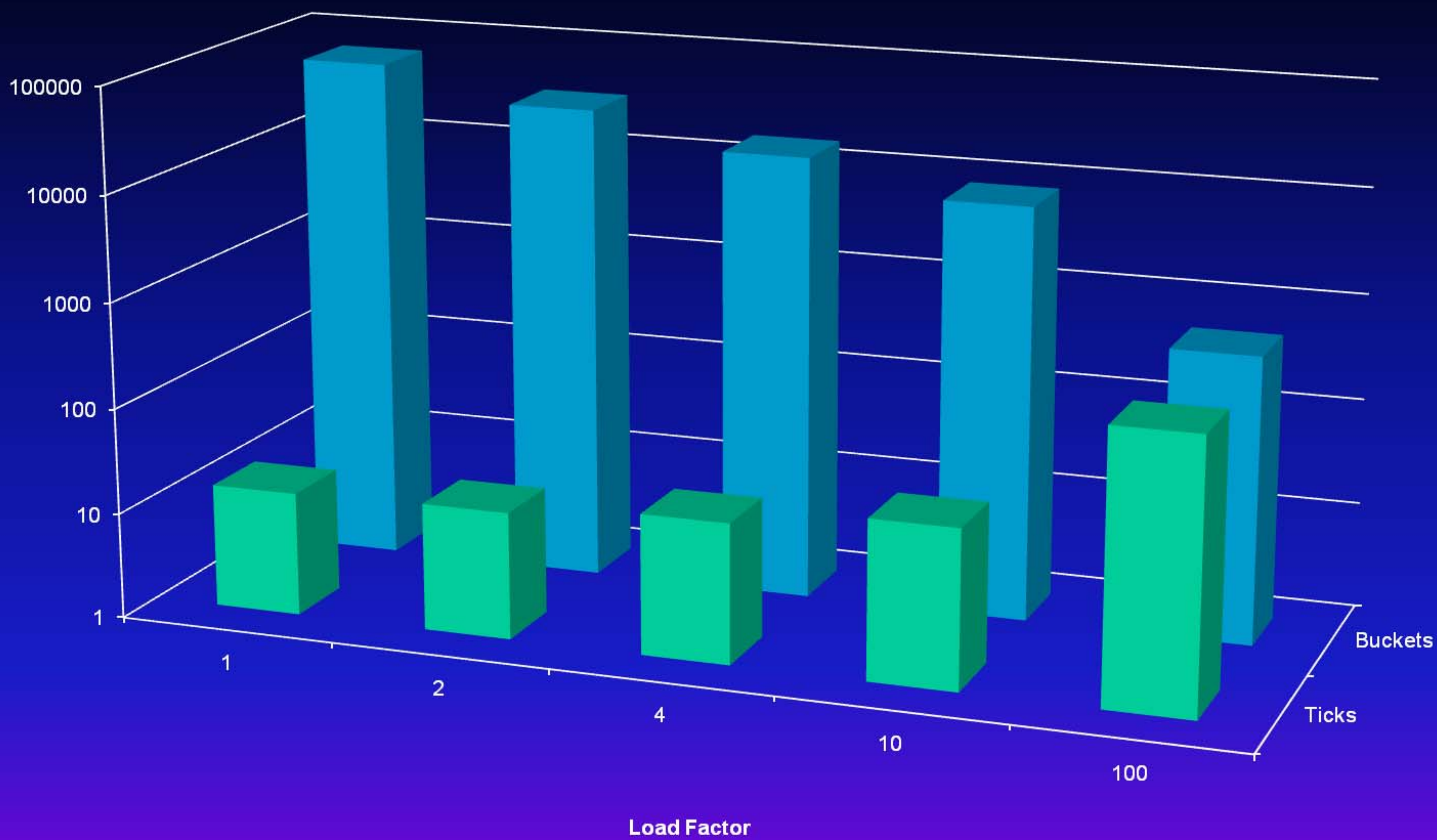
- Load factor = size() / bucket_count()
- Smaller load factor = better performance
- You control the maximum load factor

```
max_load_factor( float );
```

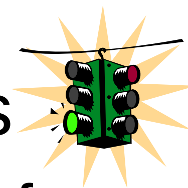
- When maximum exceeded, auto rehashes
- You can also rehash directly

```
rehash( size_type buckets );
```

Load Factors and Search Performance



Hash Tables: Take Control

- Advantages 
  - Search performance: O(1)
  - Tuning options (hash function, load factor)
  - Insertions/deletions amortized O(1)
  - Equivalent or smaller overhead/elem than set
- Disadvantages 

  - Not ordered
  - No set_union, set_intersection
  - No reverse traversal
  - Depends on great hash function
  - Requires key support op==()

Smart Pointers

- C++ resource lifetimes
  - Global
  - Stack-based
  - Static
- No direct support for resources that have an intermediate lifetime
- Enter shared_ptr<T>
  - Ensures resources are available as long as needed and disposed when no longer needed

High-Octane Power

- Smart ptr copied: ref count incremented
- Dtor decrements ref count
- Ref count goes to zero: ptr delete'd
- Initialize with raw ptr

```
shared_ptr<T> sp( new T(…) );
```

- Masquerades as a pointer for common usage

```
T t(*sp);   // T& operator*() const
sp->func(); // T* operator->() const
```

- Direct access available, too

```
p = sp.get(); // T* get() const
```

Extended Example, Part A

```
typedef std::tr1::shared_ptr<Bitmap> SPB;
struct SPBHash
{
    size_t operator()(const SPB& bp) const
    {  // hash = raw ptr address
        return (size_t)( bp.get() );
    }
};
// Store smart bitmaps for fast lookup
unordered_set< SPB, SPBHash > ht;
```

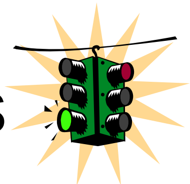
Extended Example, Part B

```
// Store bitmaps in hash table
SPB spb( new Bitmap( … ) );
ht.insert( spb );
ht.insert( … ); // Insert some more
// Fast lookup
unordered_set<SPB>::iterator it;
it = ht.find( spb );
int w = (*it)->GetWidth();
// Best part: no more code required
// No more resource leaks
```

Smart Pointer Performance

- Object management
  - Ctor: allocates block for ref count info
  - Copy: ref count update
  - Dtor: ref count update; deallocation calls
- Mgmt costs are insignificant for most objects
  - Smart ptrs generally wrap much more costly objs
- Smart ptr access is *equivalent* to raw ptrs
  - *sp produces same code as *p
  - sp-> produces same code as p->

Smart Ptrs: Take Control

- Advantages 
  - Automatic resource management
  - Avoid memory leaks
  - Can be stored in containers
  - Tested
- Disadvantages 

  - Ref count management overhead
  - Ref count memory overhead
  - Cycles require use of weak_ptr
  - May not be thread-safe; chk your implementation

TR1: Take it for a Spin!

Resources

- Contact me
  - Email: pkisensee@msn.com
  - Homepage: www.tantalon.com/pete.htm
  - Blog: pkisensee.spaces.live.com
- Useful websites
  - www.boost.org
  - www.dinkumware.com
  - gcc.gnu.org
- Books and magazines
  - C++ Standard Library Extensions, Pete Becker
  - Dr. Dobbs Journal; search on “TR1”