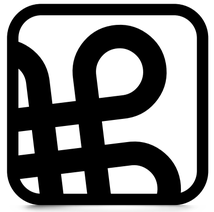


Macoun

ProjectWizards

Crashkurs Objective-C++

Frank Illenberger

Was ist Objective-C++?

Was ist Objective-C++?

- Sprachvariante von Objective-C
- Syntax von C++ und Objective-C kann gemischt werden
- Erweitert C++ auf gleiche Weise wie Objective-C C erweitert
- Compiler: clang & gcc

Zwei Welten nebeneinander

- Semantiken beider Sprachen werden nicht vereint
- C++-Klassen können nicht von Objective-C-Klassen abgeleitet werden
- C++-Referenzen können nicht auf Objective-C-Objekte verweisen
- Objective-C nutzt nicht die C++-Namespaces

Zwei Welten nebeneinander

- Ausnahmen
  - Exceptions sind vereint
  - nur in „Modern Runtime“
  - Memory-Management ist vereint (ARC)

Wozu brauche ich das?

„Within C++, there is a much smaller and cleaner language struggling to get out.“

Bjarne Stroustrup

„There are only two things wrong with C++: The initial concept and the implementation.“

Bertrand Meyer

Wozu brauche ich das?

- Objective-C mit C++ aufpeppen – nicht umgekehrt
- Ein mächtiges Werkzeug im Werkzeugkasten des Objective-C-Entwicklers
- An Stellen verwenden, an denen die Abstraktionen von Objective-C versagen
- Effizienz bei gleichzeitig hoher Abstraktion

Effizienz

Effizienz

- Objective-C kennt (fast) nur Heap-Objekte
- Allokation ist kostspielig
- Alle Cocoa-Container enthalten keine Objekte, sondern nur Pointer auf Objekte
- alle Objekte eines Containers müssen einzeln auf dem Heap allokiert werden

Übliche Alternativen

- C-structs
  - leichtgewichtig auf dem Stack
  - NSRange, NSPoint, NSRect etc.
- Nackte malloc-Blöcke als Container
  - effizient am Stück allokierbar

Übliche Alternativen

- Nachteile von C-structs
  - geringe Abstraktion
  - können keine Logik enthalten
  - Gefahr von Seiteneffekten
  - kein automatisches Retain-Counting (ARC)

Übliche Alternativen

- Nachteile von malloc
  - Umständliches memory management
  - geringe Abstraktion = viel Arbeit
  - sehr fehleranfällig
  - ja, wirklich!
  - Buffer overruns

Typische malloc-Fallen

Uninitialisierter Speicher

```
MyStruct* array = malloc(20 * sizeof(MyStruct));
int amount = array[0].amount;
```

- `array` enthält Zufallsdaten
- Abhilfe: `memset` oder `calloc`

Speicherleck

```
char* str = malloc(sizeof(char) * 20);
…
if(errorCode != 0)
  return 0;    // early exit
…
free(str)
return count;
```

- Abhilfe: Static Analyzer

Buffer Overrun

```
MyStruct* array = malloc(sizeof(MyStruct) * n + 1);
for(int i=0; i<=n; i++)
   array[i].amount = 42;
```

- Führt zu schwer reproduzierbaren Fehlern
- Schwer zu finden, auch mit Static Analyzer

Praxisbeispiele mit Objective-C++

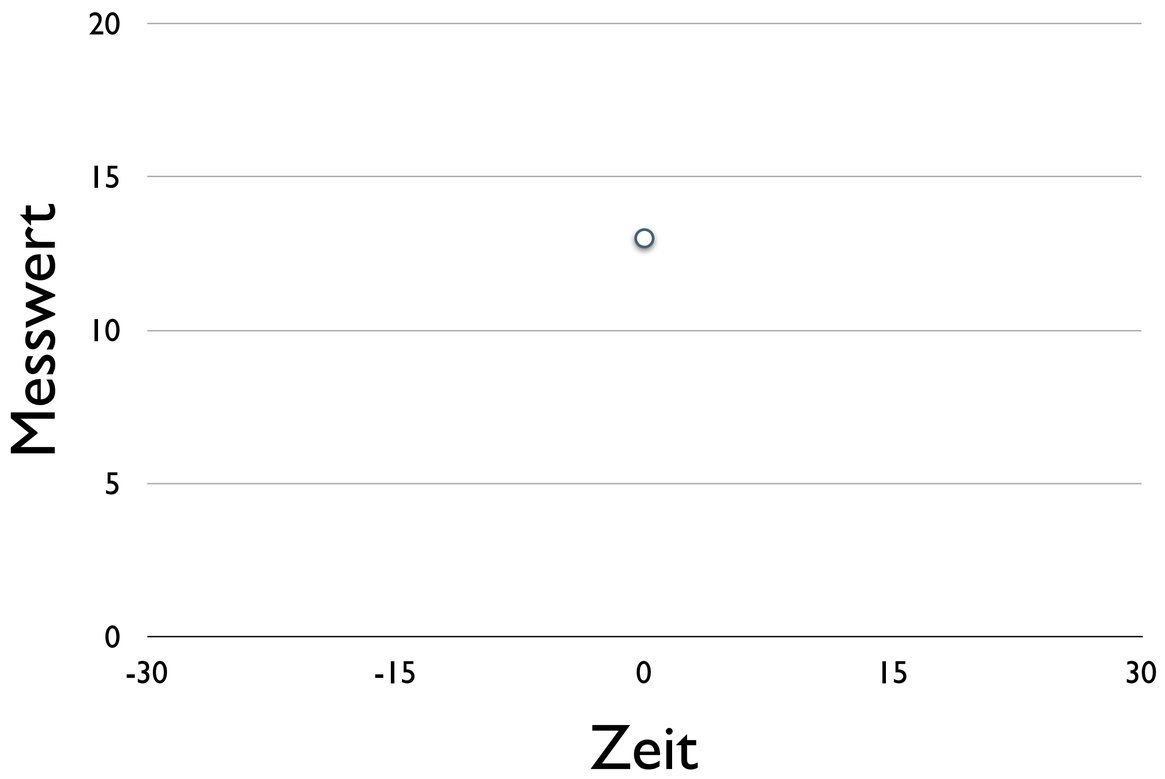
Beispielcode

github.com/depth42/CrashkursObjectiveCPP

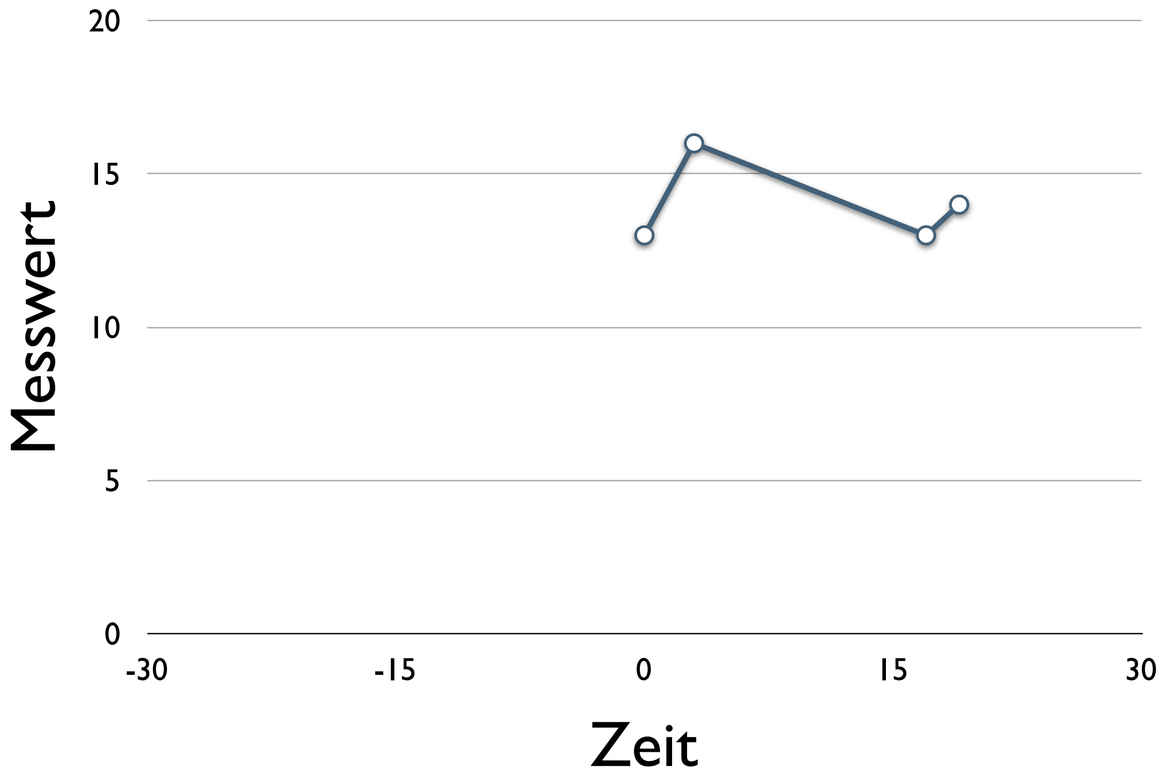
Beispiel: Sensordaten

- Messstation liefert gespeicherte Datentripel:
  - Zeitstempel
  - skalarer Messwert (z.B. Temperatur)
  - Sensor-ID
- Tripel können sequentiell zeitlich auf- oder absteigend abgeholt werden.
- Cocoa-Client soll Messdaten um einen zentralen Zeitpunkt herum in Bündeln abholen
- Client soll unbekannte Messwerte in der Zeit zwischen zwei Zeitstempeln interpolieren

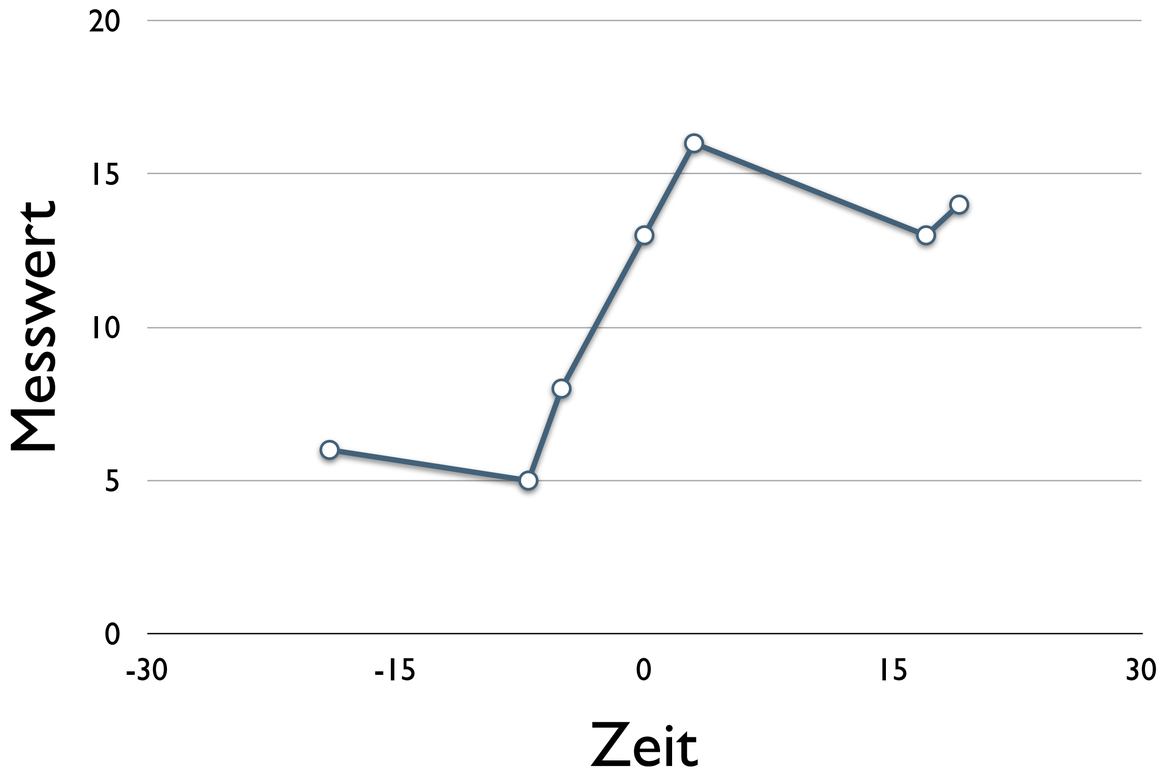
Beispiel: Sensordaten



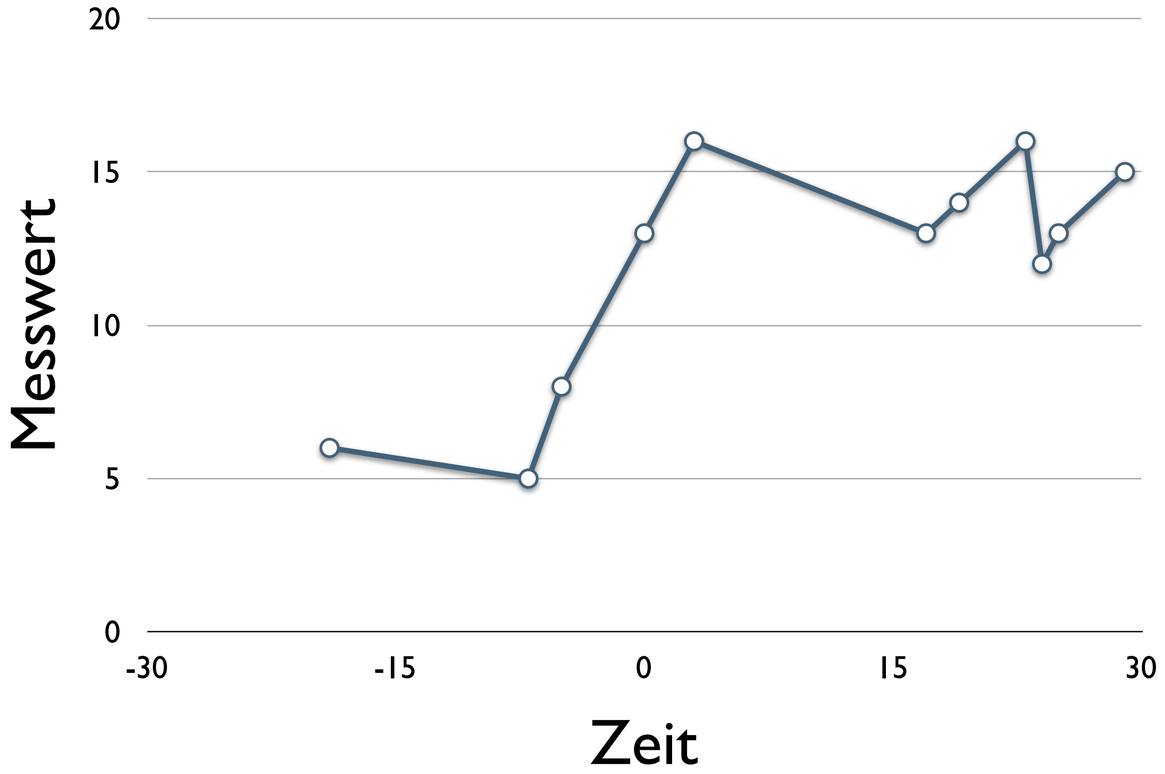
Beispiel: Sensordaten



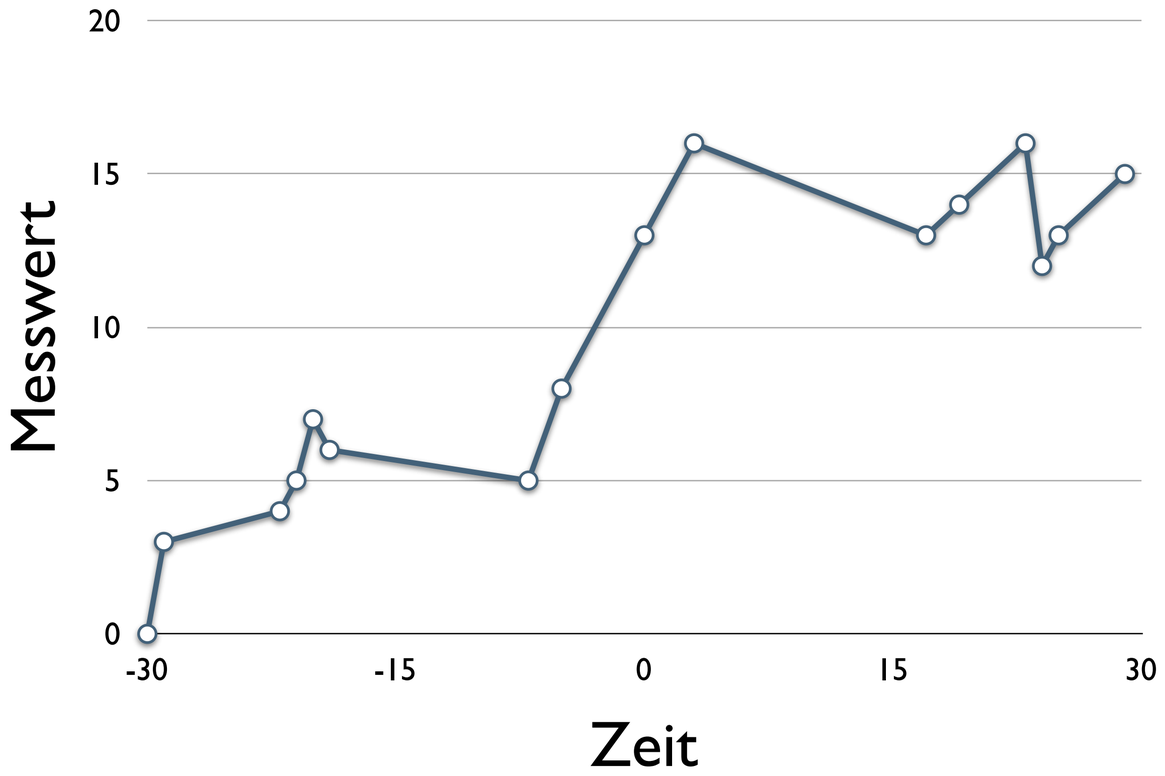
Beispiel: Sensordaten



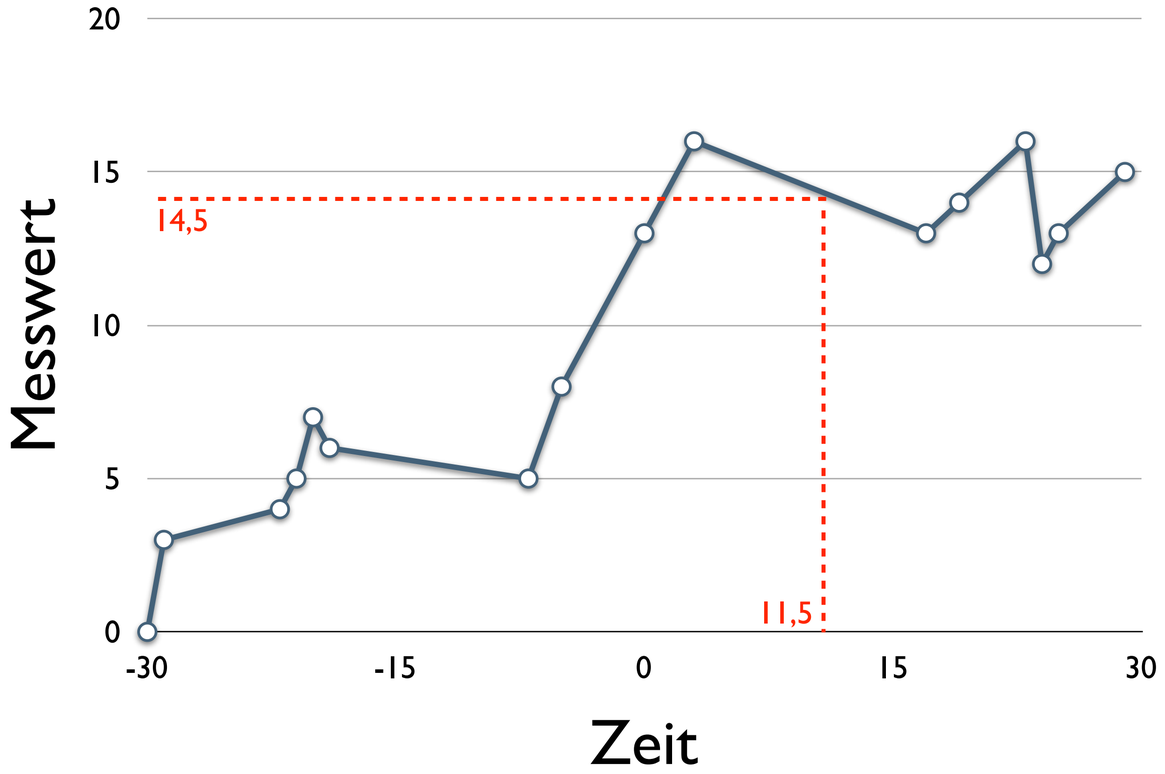
Beispiel: Sensordaten



Beispiel: Sensordaten



Beispiel: Sensordaten



Beispiel-Implementierungen

- Objective-C
- Nacktes C
- Objective-C++
- Benchmarking

Implementierung in Objective-C

Objective-C: Messpunkt

```
@interface COCSensorReading : NSObject

@property (nonatomic, readonly)     NSTimeInterval  time;
@property (nonatomic, readonly)     double          value;
@property (nonatomic, readonly)     NSUUID*         sensorID;

- (id)initWithTime:(NSTimeInterval)time
             value:(double)value
          sensorID:(NSUUID*)sensorID;

- (double)interpolatedValueAtTime:(NSTimeInterval)time
       betweenReceiverAndReading:(COCSensorReading*)targetReading;

@end
```

Objective-C: Messpunkt

```
- (double)interpolatedValueAtTime:(NSTimeInterval)time
        betweenReceiverAndReading:(COCSensorReading*)nextReading
{
    NSParameterAssert(self.time <= time);
    NSParameterAssert(nextReading);
    NSParameterAssert(nextReading.time >= time);

    NSTimeInterval timeA = self.time;
    NSTimeInterval timeB = nextReading.time;
    double valueA = self.value;
    double valueB = nextReading.value;

    return valueA + (valueB - valueA) * (time - timeA) / (timeB - timeA);
}
```

NSMutableArray als Container für Messpunkte

```
#import "COCSensorReading.h"

@implementation COCSensorDemoObjC
{
    NSMutableArray* _readings; // COCSensorReading
}

- (id)init
{
    if(self = [super init]) {
        _readings = [NSMutableArray array];
    }
    return self;
}
```

Bestimmung der Randzeiten

```
- (NSTimeInterval)earliestTime
{
    if(_readings.count > 0)
        return ((COCSensorReading*)_readings[0]).time;
    else
        return self.defaultTime;
}

- (NSTimeInterval)latestTime
{
    if(_readings.count > 0)
        return ((COCSensorReading*)_readings.lastObject).time;
    else
        return self.defaultTime;
}
```

Hintanhängen eines Bündels von Messpunkten

```
- (void)appendBatchOfSensorReadings
{
    [self.class provideRandomValues:self.batchSize
                           startTime:self.latestTime
                          usingBlock:^(NSTimeInterval iTime,
                                       double iValue,
                                       NSUUID* iSensorID)
     {
         [_readings addObject:
          [[COCSensorReading alloc] initWithTime:iTime
                                           value:iValue
                                        sensorID:iSensorID]];
     }];
}
```

Vornanstellen eines Bündels von Messpunkten

```
- (void)prependBatchOfSensorReadings
{
    NSMutableArray* batch = [NSMutableArray arrayWithCapacity:self.batchSize];
    [self.class provideRandomValues:-self.batchSize
                           startTime:self.earliestTime
                          usingBlock:^(NSTimeInterval iTime,
                                       double iValue,
                                       NSUUID* iSensorID)
     {
         [batch addObject:
          [[COCSensorReading alloc] initWithTime:iTime
                                           value:iValue
                                        sensorID:iSensorID]];
     }]; …
```

Vornanstellen eines Bündels von Messpunkten

```
…
NSMutableArray* newReadings = [NSMutableArray
        arrayWithCapacity:self.batchSize + _readings.count];
    [batch enumerateObjectsWithOptions:NSEnumerationReverse
                            usingBlock:^(COCSensorReading* iReading,
                                         NSUInteger idx,
                                         BOOL* stop)
     {
         [newReadings addObject:iReading];
     }];

    [newReadings addObjectsFromArray:_readings];
    _readings = newReadings;
}
```

Vornanstellen eines Bündels von Messpunkten

```
…
NSMutableArray* newReadings = [NSMutableArray
        arrayWithCapacity:self.batchSize + _readings.count];
    [batch enumerateObjectsWithOptions:NSEnumerationReverse
                            usingBlock:^(COCSensorReading* iReading,
                                         NSUInteger idx,
                                         BOOL* stop)
     {
         [newReadings addObject:iReading];
     }];

    [newReadings addObjectsFromArray:_readings];
    _readings = newReadings;
}
```

$O(n^2)$

Nachschlagen und Interpolieren

```
- (double)interpolatedValueAtTime:(NSTimeInterval)time
{
    NSUInteger index = [self insertionIndexOfReadingForTime:time];
    NSAssert(index != NSNotFound, nil);
…
    return [_readings[index-1]
            interpolatedValueAtTime:time
          betweenReceiverAndReading:_readings[index]];
}
```

Nachschlagen und Interpolieren

```
- (NSUInteger)insertionIndexOfReadingForTime:(NSTimeInterval)time
{
    Class readingClass = COCSensorReading.class;
    return [_readings indexOfObject:@(time)
                      inSortedRange:NSMakeRange(0, _readings.count)
                            options:NSBinarySearchingInsertionIndex
                    usingComparator:^NSComparisonResult(id o1, id o2)
            {
                NSTimeInterval time1, time2;
                if([o1 isKindOfClass:readingClass])
                {
                    time1 = ((COCSensorReading*)o1).time;
                    time2 = time;
                }
                else
                {
                    time1 = time;
                    time2 = ((COCSensorReading*)o2).time;
                } …
```

Nachschlagen und Interpolieren

```
…
                if(time2 > time1)
                    return NSOrderedAscending;
                else if(time2 < time1)
                    return NSOrderedDescending;
                else
                    return NSOrderedSame;
            }];
}
```

Demo

Implementierung in nacktem C

Messpunkt struct

```
typedef struct
{
    NSTimeInterval      time;
    double              value;
    __unsafe_unretained NSUUID* sensorID;
} SensorReading;

@implementation COCSensorDemoPlainC
{
    SensorReading*  _readings;
}
```

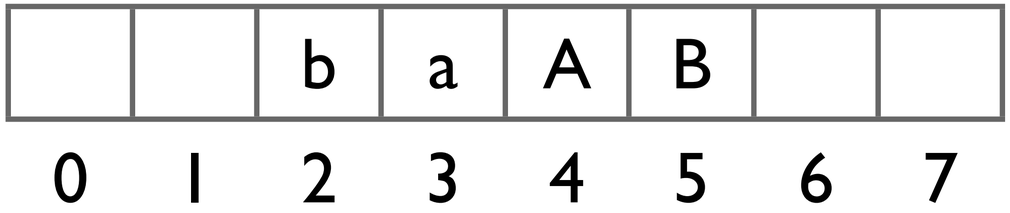
Messpunkt struct

```
typedef struct
{
    NSTimeInterval      time;
    double              value;
    __unsafe_unretained NSUUID* sensorID;
} SensorReading;

@implementation COCSensorDemoPlainC
{
    SensorReading*  _readings;
}
```

kein ARC!

Double-ended Queue (deque) für Arme

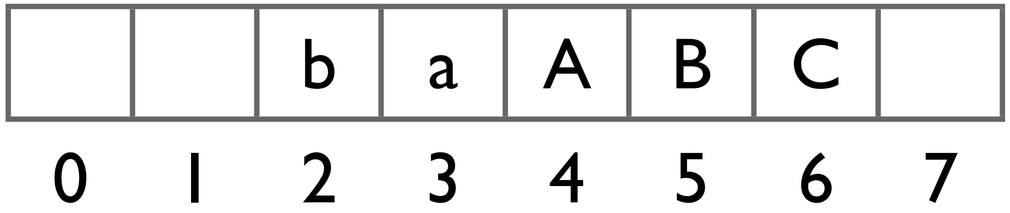
```
backCount = 2
backSize  = 4
```




```
frontCount = 2
frontSize  = 4
```

Double-ended Queue (deque) für Arme

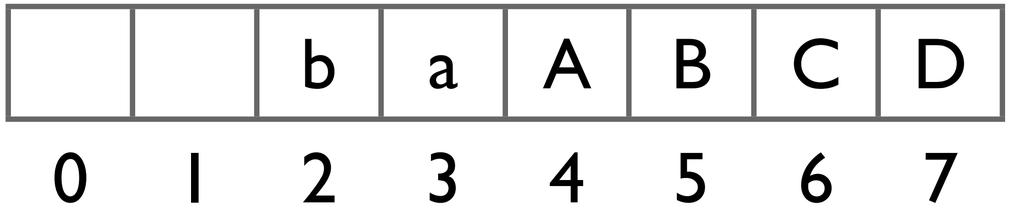
```
backCount = 3
backSize  = 4
```




```
frontCount = 2
frontSize  = 4
```

Double-ended Queue (deque) für Arme

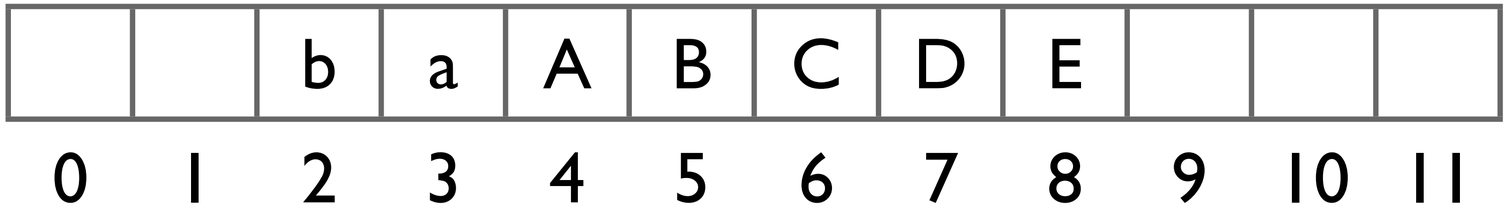
```
backCount = 4
backSize  = 4
```




```
frontCount = 2
frontSize  = 4
```

Double-ended Queue (deque) für Arme

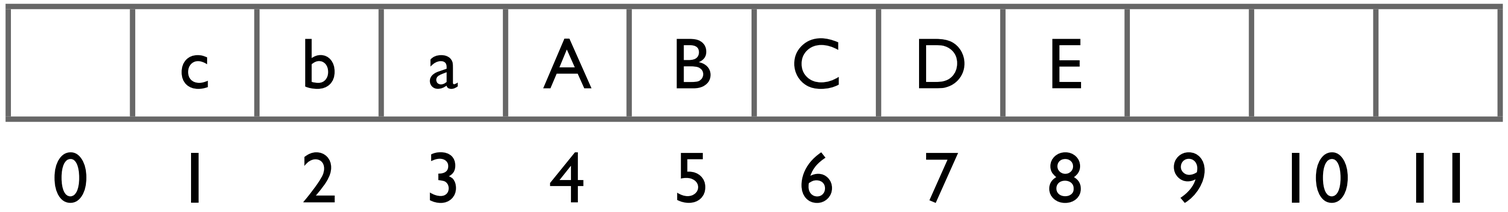
```
backCount = 5
backSize  = 8
```




```
frontCount = 2
frontSize  = 4
```

Double-ended Queue (deque) für Arme

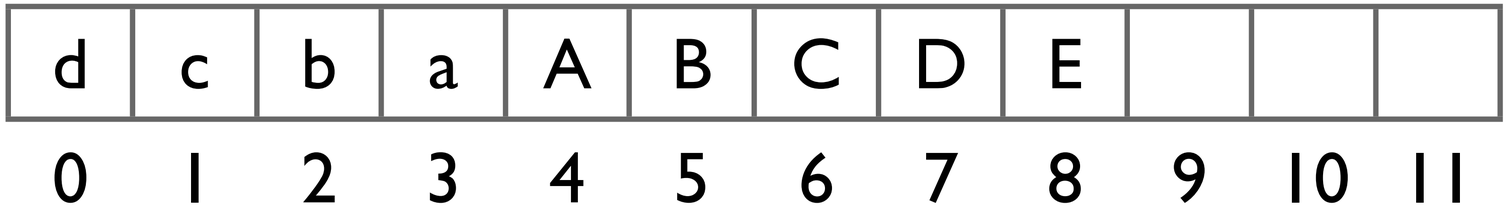
```
backCount = 5
backSize  = 8
```




```
frontCount = 3
frontSize  = 4
```

Double-ended Queue (deque) für Arme

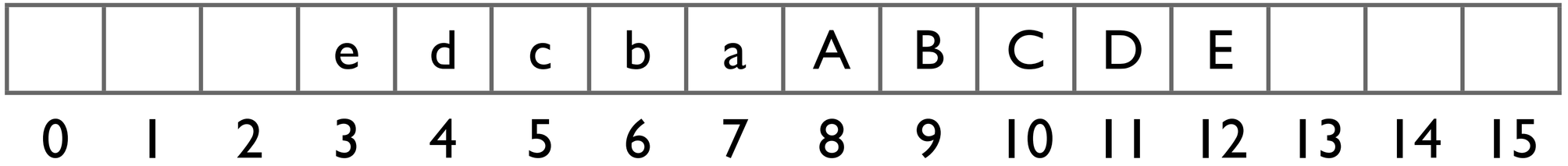
```
backCount = 5
backSize  = 8
```




```
frontCount = 4
frontSize  = 4
```

Double-ended Queue (deque) für Arme

```
backCount = 5
backSize  = 8
```




```
frontCount = 5
frontSize  = 8
```

Double-ended Queue für Arme

```
typedef struct
{
    NSTimeInterval      time;
    double              value;
    __unsafe_unretained NSUUID* sensorID;
} SensorReading;

@implementation COCSensorDemoPlainC
{
    SensorReading*  _readings;
    NSUInteger      _backSize;
    NSUInteger      _backCount;
    NSUInteger      _frontSize;
    NSUInteger      _frontCount;
}
```

Hintanhängen eines Bündels von Messpunkten

```
- (void)addSensorReadingToBack:(SensorReading)reading
{
    if(!_readings) {
        _backSize = InitialReadingsSize;
        _readings = malloc(_backSize * sizeof(SensorReading));
    }
    else if(_backCount == _backSize) {
        _backSize = (_backSize > 0) ?
                        _backSize * 2 : InitialReadingsSize;
        _readings = realloc(_readings,
                (_frontSize + _backSize) * sizeof(SensorReading));
    }
    NSAssert(_backCount < _backSize, nil);
    _readings[_frontSize + _backCount] = reading;
    _backCount++;
}
```

Vornanstellen eines Bündels von Messpunkten

```
- (void)addSensorReadingToFront:(SensorReading)reading
{
    if(!_readings) {
        _frontSize = InitialReadingsSize;
        _readings = malloc(_frontSize * sizeof(SensorReading));
    }
    else if(_frontCount == _frontSize) {
        NSUInteger prevFrontSize = _frontSize;
        _frontSize = (_frontSize > 0) ?
                        _frontSize * 2 : InitialReadingsSize;

    …
```

Vornanstellen eines Bündels von Messpunkten

```
…
SensorReading* newReadings =
            malloc((_frontSize + _backSize) * sizeof(SensorReading));

        memcpy(newReadings + _frontSize - _frontCount,
               _readings + prevFrontSize - _frontCount,
               (_frontCount + _backCount) * sizeof(SensorReading));
        free(_readings);
        _readings = newReadings;
    }
    NSAssert(_frontCount < _frontSize, nil);
    _readings[_frontSize - _frontCount - 1] = reading;
    _frontCount++;
}
```

Nachschlagen und Interpolieren

```
- (NSInteger)insertionIndexOfReadingForTime:(NSTimeInterval)time
{
  return [self insertionIndexOfReadingForTime:time
                                     minIndex:0
                                     maxIndex:self.numberOfReadings];
}
```

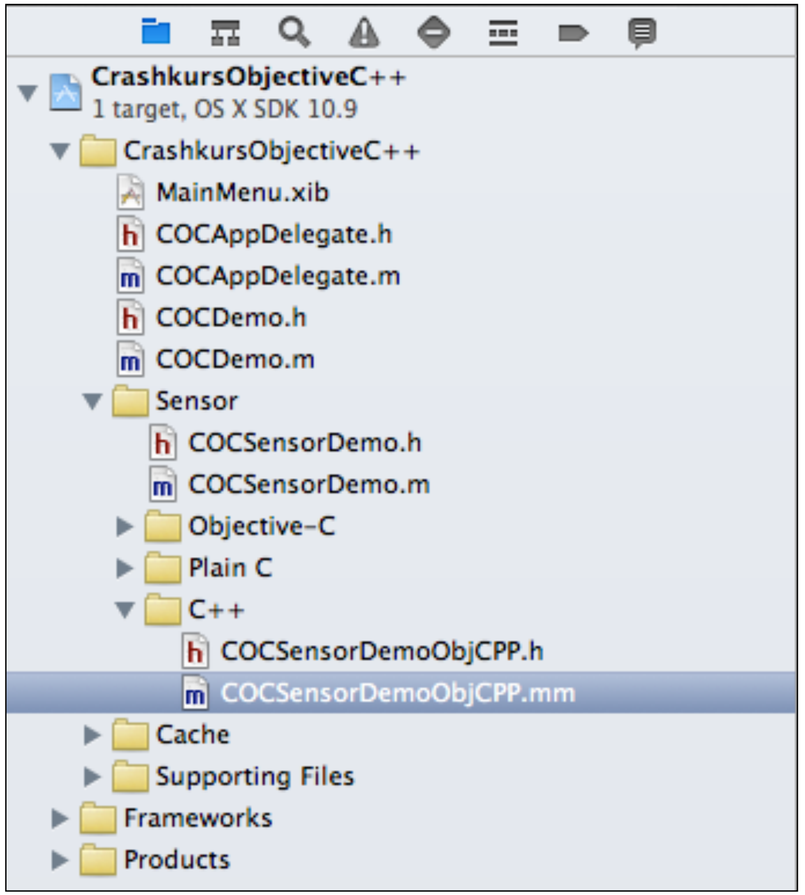
Nachschlagen und Interpolieren

```
- (NSInteger)insertionIndexOfReadingForTime:(NSTimeInterval)searchTime
                                   minIndex:(NSInteger)minIndex
                                   maxIndex:(NSInteger)maxIndex
{
    if (maxIndex < minIndex) return minIndex;
    NSInteger midIndex = (minIndex + maxIndex) / 2;
    NSTimeInterval timeAtMidIndex = [self readingAtIndex:midIndex]->time;
    if(searchTime == timeAtMidIndex)
        return midIndex;
    else if(timeAtMidIndex > searchTime)
        return [self insertionIndexOfReadingForTime:searchTime
                                           minIndex:minIndex
                                           maxIndex:midIndex - 1];
    else
        return [self insertionIndexOfReadingForTime:searchTime
                                           minIndex:midIndex + 1
                                           maxIndex:maxIndex];
}
```

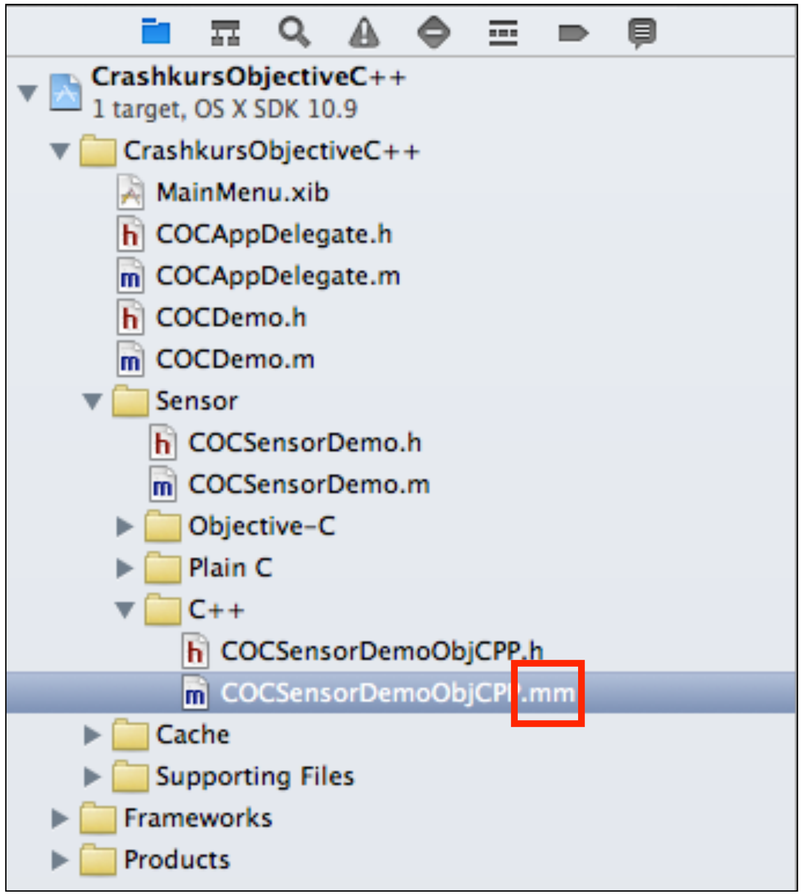
Demo

Implementierung in Objective-C++

Mit dem Dateisuffix .mm wechselt man zum Objective-C++-Compiler



Mit dem Dateisuffix .mm wechselt man zum Objective-C++-Compiler



Messpunkt struct

```
struct SensorReading {
    NSTimeInterval  time;
    double          value;
    NSUUID*         sensorID;
};
```

- Kein typedef nötig
- Vollweriges Objekt mit Methoden
- class und struct sind synonym in C++

Messpunkt struct

```
struct SensorReading {
    NSTimeInterval  time;
    double          value;
    NSUUID*         sensorID;
};
```

- automatische ARC-Unterstützung
- retain / release beim Setzen
- release wenn struct zerstört wird
- Bei Stackobjekten, auch wenn ihr Scope endet!

Interpolationsmethode

```
struct SensorReading {
    NSTimeInterval  time;
    double          value;
    NSUUID*         sensorID;

    double interpolatedValue(NSTimeInterval interpolationTime,
                             SensorReading& nextReading) {
        NSCParameterAssert(time <= interpolationTime);
        NSCParameterAssert(nextReading.time >= interpolationTime);

        return value + (nextReading.value - value)
        * (interpolationTime - time) / (nextReading.time - time);
    }
};
```

Collection

```
#import <deque>

@implementation COCSensorDemoObjCPP
{
    std::deque<SensorReading> _readings;
}
```

- Standard Library (libc++)
- Namespace „std“
- Double-ended queue container
- Hocheffizient: Freier Zugriff,
  Einfügen vorne & hinten skalieren mit O(1)

Collection

```
#import <deque>

@implementation COCSensorDemoObjCPP
{
    std::deque<SensorReading> _readings;
}
```

- ivar
- kein Pointer!
- deque liegt direkt in ivar
- ein malloc weniger als in NSMutableArray

Collection

```
#import <deque>

@implementation COCSensorDemoObjCPP
{
    std::deque<SensorReading> _readings;
}
```

- Template Parameter
- Bestimmt Typ der im Container enthaltenen Objekte
- Container enthält Objekte, keine Pointer auf Objekte!
- kein Heap malloc pro Objekt nötig!

Collection

```
#import <deque>

@implementation COCSensorDemoObjCPP
{
    std::deque<SensorReading> _readings;
}
```

- Volle ARC-Unterstützung!
- Destruktor von _readings wird aufgerufen, in dealloc von Obj-C-Instanz
- enthaltene ARC-Objekte in SensorReadings werden dann ebenfalls released

Bestimmung der Randzeiten

```
- (NSTimeInterval)earliestTime
{
    if(_readings.size() > 0)
        return _readings.front().time;
    else
        return self.defaultTime;
}

- (NSTimeInterval)latestTime
{
    if(_readings.size() > 0)
        return _readings.back().time;
    else
        return self.defaultTime;
}
```

Hintanhängen von Messpunkten

```
- (void)appendBatchOfSensorReadings
{
    [self.class provideRandomValues:self.batchSize
                           startTime:self.earliestTime
                          usingBlock:^(NSTimeInterval iTime,
                                       double iValue,
                                       NSUUID* iSensorID)
     {
         _readings.push_back({ iTime, iValue, iSensorID});
     }];
}
```

- Literal erzeugt temporäres Stackobjekt
- C++-11 ermittelt den Typ automatisch
- Alternativ: emplace_back kommt ohne Stackobjekt aus

Vornanhängen von Messpunkten

```
- (void)prependBatchOfSensorReadings
{
    [self.class provideRandomValues:-self.batchSize
                           startTime:self.latestTime
                          usingBlock:^(NSTimeInterval iTime,
                                       double iValue,
                                       NSUUID* iSensorID)
     {
         _readings.push_front({ iTime, iValue, iSensorID});
     }];
}
```

Nachschlagen und Interpolieren

```
#import <algorithm>

- (double)interpolatedValueAtTime:(NSTimeInterval)time
{
    auto insert = std::lower_bound(_readings.begin(),
                                   _readings.end(),
                                   time,
                                   [] (const SensorReading& reading,
                                       double iTime) {
                                       return reading.time < iTime;
                                   }); …
```

- Sammlung von Algorithmen, die auf Container arbeiten
- Unabhängig von Containern definiert, funktionieren somit auf vielen Containern
- Exzellent dokumentiert (z.B. www.cplusplus.com)

Nachschlagen und Interpolieren

```
#import <algorithm>

- (double)interpolatedValueAtTime:(NSTimeInterval)time
{
    auto insert = std::lower_bound(_readings.begin(),
                                   _readings.end(),
                                   time,
                                   [] (const SensorReading& reading,
                                       double iTime) {
                                       return reading.time < iTime;
                                   }); …
```

- Binärsuche. Sucht nach erstem Objekt in vorgegebenem Collection-Bereich, das nicht kleiner ist als angegebener Wert.
- Flexible Varianten: upper_bound, equal_range, binary_search

Nachschlagen und Interpolieren

```
#import <algorithm>

- (double)interpolatedValueAtTime:(NSTimeInterval)time
{
    auto insert = std::lower_bound(_readings.begin(),
                                   _readings.end(),
                                   time,
                                   [] (const SensorReading& reading,
                                       double iTime) {
                                       return reading.time < iTime;
                                   }); …
```

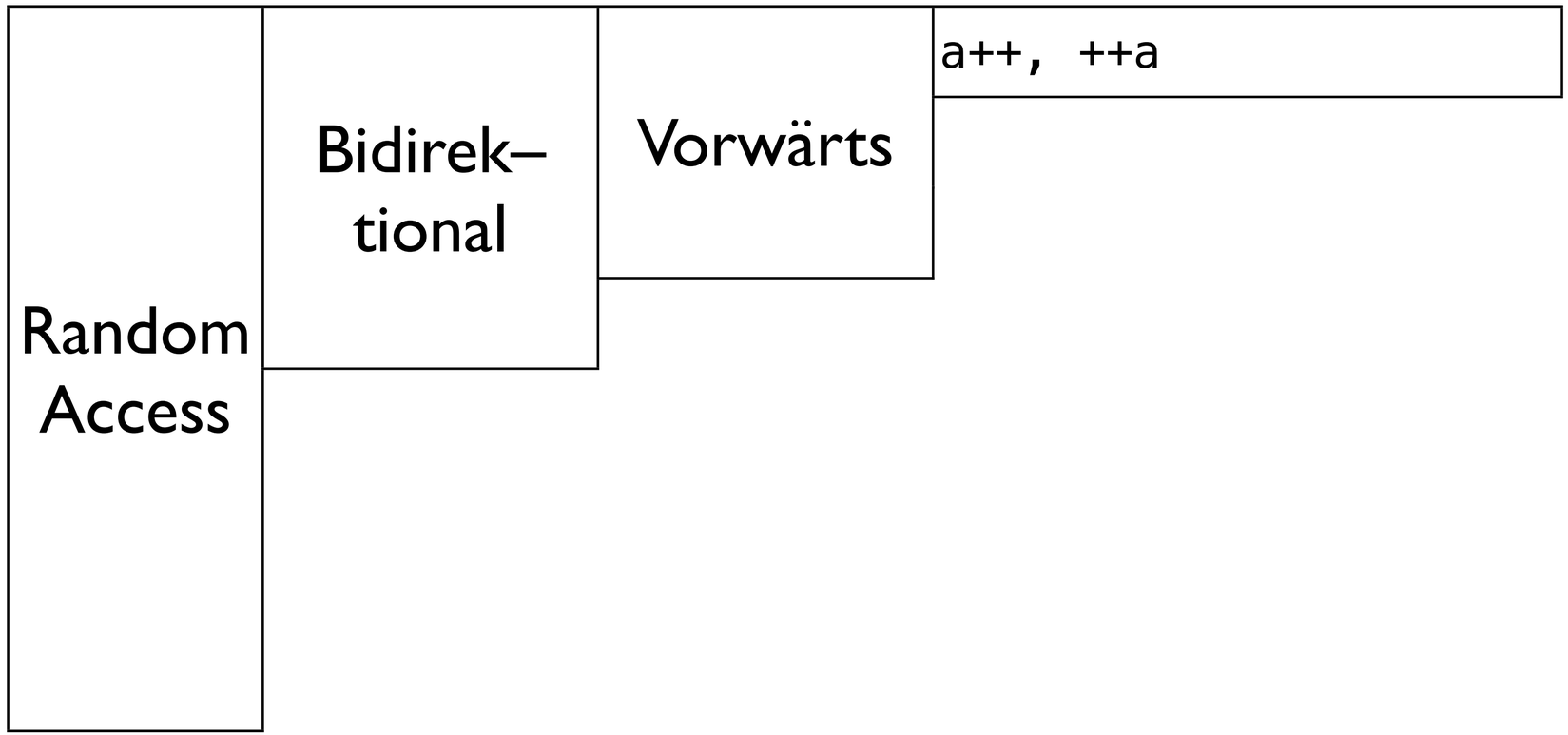
- Der Suchbereich wird durch zwei Iteratoren abgegrenzt

Iterator

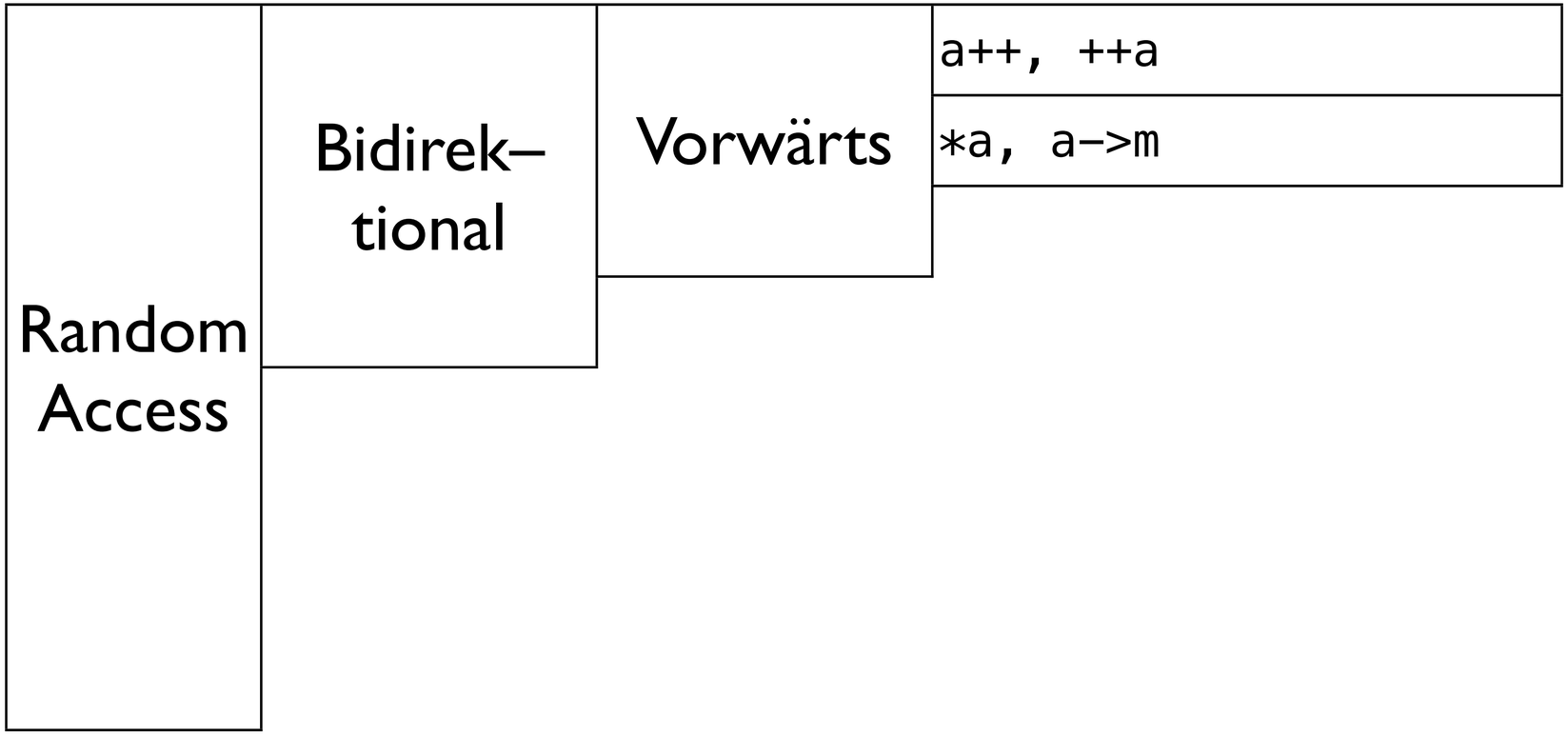
- Ein Iterator ist ein Objekt
- Zeigt auf ein Element in einem Container
- Verallgemeinerter Pointer
- Iteriert über die Elemente mit Hilfe von Operatoren

Iterator-Operatoren

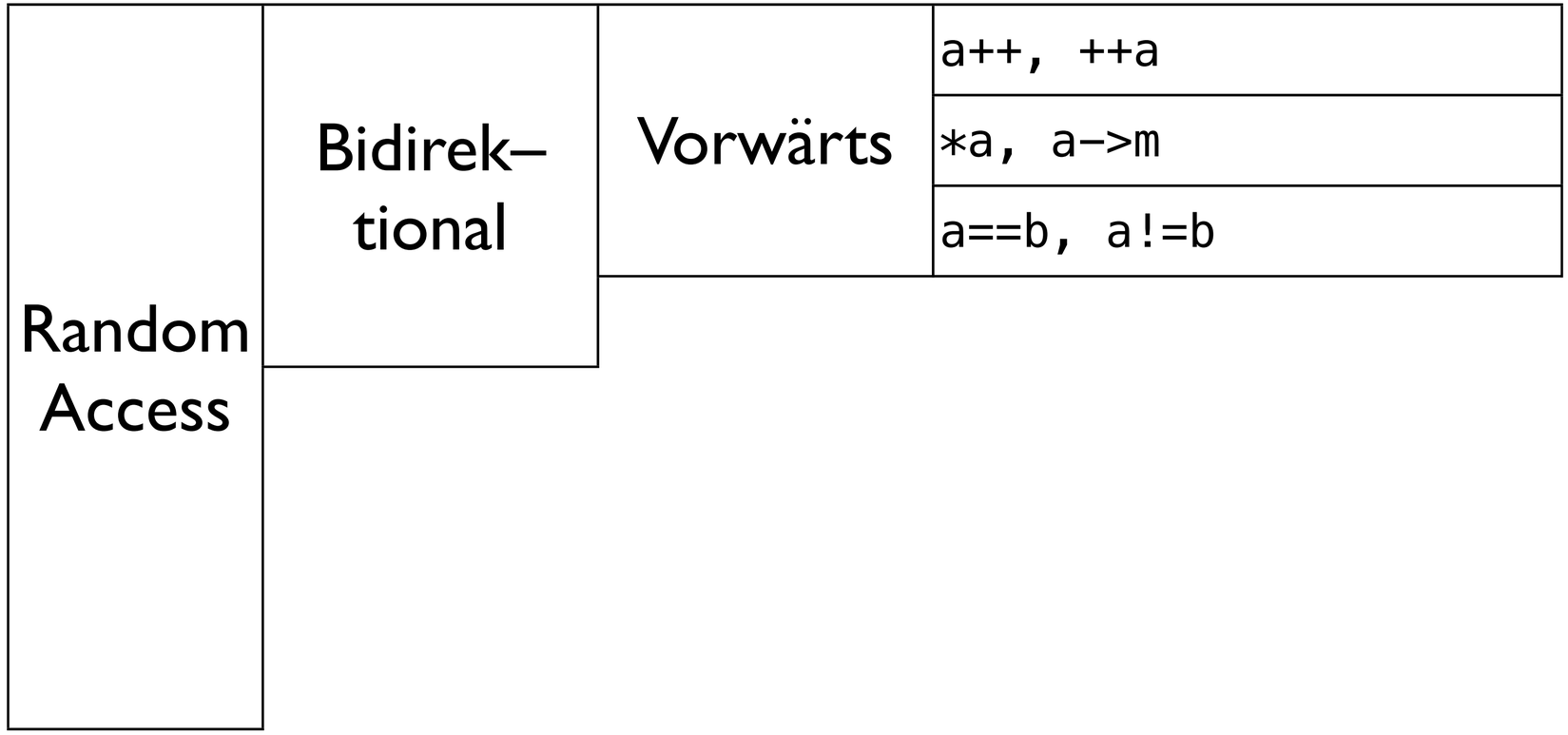
Iterator-Operatoren



Iterator-Operatoren



Iterator-Operatoren



Iterator-Operatoren

<table>
<tr><td rowspan="4">Random Access</td><td rowspan="3">Bidirek–tional</td><td rowspan="3">Vorwärts</td><td><code>a++, ++a</code></td></tr>
<tr><td><code>*a, a->m</code></td></tr>
<tr><td><code>a==b, a!=b</code></td></tr>
<tr><td></td><td><code>--a, a--, *a--</code></td></tr>
</table>

Iterator-Operatoren

<table>
<tr><td rowspan="5">Random Access</td><td rowspan="3">Bidirek–tional</td><td rowspan="2">Vorwärts</td><td><code>a++, ++a</code></td></tr>
<tr><td><code>*a, a->m</code></td></tr>
<tr><td><code>a==b, a!=b</code></td></tr>
<tr><td></td><td><code>--a, a--, *a--</code></td></tr>
<tr><td colspan="2"></td><td><code>a+n, n+a, a-n, a-b</code></td></tr>
</table>

Iterator-Operatoren

<table>
<tr><td rowspan="6">Random Access</td><td rowspan="4">Bidirek–tional</td><td rowspan="3">Vorwärts</td><td>a++, ++a</td></tr>
<tr><td>*a, a->m</td></tr>
<tr><td>a==b, a!=b</td></tr>
<tr><td></td><td>--a, a--, *a--</td></tr>
<tr><td colspan="2" rowspan="3"></td><td>a+n, n+a, a-n, a-b</td></tr>
<tr><td>a<b, a>b, a<=b, a>=b</td></tr>
</table>

Iterator-Operatoren

<table>
<tr><td rowspan="7">Random Access</td><td rowspan="3">Bidirek–tional</td><td rowspan="2">Vorwärts</td><td><code>a++, ++a</code></td></tr>
<tr><td><code>*a, a->m</code></td></tr>
<tr><td><code>a==b, a!=b</code></td></tr>
<tr><td></td><td><code>--a, a--, *a--</code></td></tr>
<tr><td colspan="2" rowspan="4"></td><td><code>a+n, n+a, a-n, a-b</code></td></tr>
<tr><td><code>a<b, a>b, a<=b, a>=b</code></td></tr>
<tr><td><code>a+=n, a-=n</code></td></tr>
</table>

Iterator-Operatoren

<table>
<tr><td rowspan="8">Random Access</td><td rowspan="3">Bidirek–tional</td><td rowspan="3">Vorwärts</td><td>a++, ++a</td></tr>
<tr><td>*a, a->m</td></tr>
<tr><td>a==b, a!=b</td></tr>
<tr><td></td><td>--a, a--, *a--</td></tr>
<tr><td colspan="2" rowspan="5"></td><td>a+n, n+a, a-n, a-b</td></tr>
<tr><td>a<b, a>b, a<=b, a>=b</td></tr>
<tr><td>a+=n, a-=n</td></tr>
<tr><td>a[n]</td></tr>
</table>

Container-Methoden

Container-Methoden

begin	Zeigt auf das erste Element.

Container-Methoden

begin	Zeigt auf das erste Element.
end	Zeigt auf das theoretische Element nach dem letzten. Darf nicht dereferenziert werden.

Container-Methoden

begin	Zeigt auf das erste Element.
end	Zeigt auf das theoretische Element nach dem letzten. Darf nicht dereferenziert werden.
rbegin	Zeigt auf das letzte Element.

Container-Methoden

begin	Zeigt auf das erste Element.
end	Zeigt auf das theoretische Element nach dem letzten. Darf nicht dereferenziert werden.
rbegin	Zeigt auf das letzte Element.
rend	Zeigt auf das theoretische Element vor dem ersten. Darf nicht dereferenziert werden.

Container-Methoden

begin	Zeigt auf das erste Element.
end	Zeigt auf das theoretische Element nach dem letzten. Darf nicht dereferenziert werden.
rbegin	Zeigt auf das letzte Element.
rend	Zeigt auf das theoretische Element vor dem ersten. Darf nicht dereferenziert werden.
cbegin, cend crbegin, crend	Liefern Const-Iteratoren, mit denen der Inhalt der Elemente nicht verändert werden kann.

Iterator-Schleifen

```
std::deque<int> myDeque;
for(int i=1; i<=5; i++)
    myDeque.push_back(i);

for(std::deque<int>::iterator it  = myDeque.begin();
                              it != myDeque.end(); ++it)
    NSLog(@"%d", *it);
```

Iterator-Schleifen

```
std::deque<int> myDeque;
for(int i=1; i<=5; i++)
    myDeque.push_back(i);

for(auto it  = myDeque.begin();
         it != myDeque.end(); ++it)
    NSLog(@"%d", *it);
```

Iterator-Schleifen

```
std::deque<int> myDeque;
for(int i=1; i<=5; i++)
    myDeque.push_back(i);

for(auto iElement : myDeque)
    NSLog(@"%d", iElement);
```

Nachschlagen und Interpolieren

```
#import <algorithm>

- (double)interpolatedValueAtTime:(NSTimeInterval)time
{
    auto insert = std::lower_bound(_readings.begin(),
                                   _readings.end(),
                                   time,
                                   [] (const SensorReading& reading,
                                       double iTime) {
                                       return reading.time < iTime;
                                   }); …
```

- Comparator als Lambda-Ausdruck
- Ähnlich wie Objective-C-Blocks
- Gibt zurück, ob erster Wert vor zweiten gehört

Nachschlagen und Interpolieren

```
#import <algorithm>

- (double)interpolatedValueAtTime:(NSTimeInterval)time
{
    auto insert = std::lower_bound(_readings.begin(),
                                   _readings.end(),
                                   time,
                                   [] (const SensorReading& reading,
                                       double iTime) {
                                       return reading.time < iTime;
                                   }); …
```

- Refererenz auf Objekt in Collection
- Sicherer als Pointer, kann nicht NULL sein
- Const erhöht die Sicherheit

Nachschlagen und Interpolieren

```
#import <algorithm>

- (double)interpolatedValueAtTime:(NSTimeInterval)time
{
    auto insert = std::lower_bound(_readings.begin(),
                                   _readings.end(),
                                   time,
                                   [] (const SensorReading& reading,
                                       double iTime) {
                                       return reading.time < iTime;
                                   });
    if(insert == _readings.begin())
        return (*insert).value;             // we do not extrapolate
    else if(insert == _readings.end())
        return (*(insert-1)).value;         // we do not extrapolate
    else
        return (*(insert-1)).interpolatedValue(time, *insert);
}
```

Demo

Container-Arten

vector	Dynamisches Array wie NSMutableArray
deque	Dynamisches Array mit zwei flexiblen Enden
list	Doppelt verkettete Liste
forward_list	Einfach verkettete Liste
array	Array mit fester Größe

Container-Arten

set	wie NSMutableSet
multiset	wie NSCountedSet, ein Element kann mehrfach enthalten sein.
unordered_map	Hashtable, wie NSMutableDictionary
unordered_multimap	Hashtable, mehrere Elemente können gleichen Key besitzen
map, multimap	Assoziativ wie unordered_map, aber als Binärbaum implementiert

Beispiel: Caching

- Speichern von teuer zu berechnenden Ergebnissen
- Ablegen in assoziativem Container
- mehr als ein Schlüsselwert (compound key):

```
- (id)cachedObjectForNumberValue:(double)number
                     stringValue:(NSString*)string
                  creationBlock:(id (^)(double number,
                                        NSString* string))block;
```

Implementierung in Objective-C

Cache: Container

```
@implementation COCCacheDemoObjC
{
    NSMutableDictionary*  _cache;
}

- (id)init
{
    if(self = [super init]) {
        _cache = [[NSMutableDictionary alloc] init];
    }
    return self;
}
```

Cache: Compound Key

```
- (id)cachedObjectForNumberValue:(double)number
                     stringValue:(NSString*)string
                  creationBlock:(id (^)(double number,
                                        NSString*
string))block
{
    NSParameterAssert(string);
    NSAssert(_cache, nil);

    NSString* key = [NSString stringWithFormat:@"%f-%@",
                     number, string];

    id object = _cache[key];

…
```

Cache: Compound Key

```
- (id)cachedObjectForNumberValue:(double)number
                     stringValue:(NSString*)string
                  creationBlock:(id (^)(double number,
                                        NSString*
string))block
{
    NSParameterAssert(string);
    NSAssert(_cache, nil);

    NSString* key = [NSString stringWithFormat:@"%f-%@",
                     number, string];

    id object = _cache[key];

…
```

WzT?

Cache: Compound Key

```
@interface COCCacheKey : NSObject <NSCopying>

- (id)initWithNumberValue:(double)numberValue
              stringValue:(NSString*)stringValue;

@property (nonatomic, readonly)          double    numberValue;
@property (nonatomic, readonly, copy) NSString* stringValue;

@end
```

Cache: Compound Key

```
#import "COCCacheKey.h"

@implementation COCCacheKey

- (id)initWithNumberValue:(double)numberValue
              stringValue:(NSString*)stringValue
{
    NSParameterAssert(stringValue);
    if(self = [super init])
    {
        _numberValue = numberValue;
        _stringValue = [stringValue copy];
    }
    return self;
}
```

Cache: Compound Key

```
- (BOOL)isEqual:(id)object
{
    if(![object isKindOfClass:COCCacheKey.class]) return NO;
    COCCacheKey* key = object;
    return (key.numberValue == _numberValue)
        && [key.stringValue isEqualToString:_stringValue];
}

- (NSUInteger)hash
{
    NSUInteger hash =  17;
    hash = 37 * hash + _numberValue;
    hash = 37 * hash + _stringValue.hash;
    return hash;
}

- (id)copyWithZone:(NSZone*)zone
{
    return self;
}
```

Cache: Container

```
@implementation COCCacheDemoObjC
{
    NSMutableDictionary*  _cache; // COCCacheKey -> id
}

- (id)init
{
    if(self = [super init]) {
        _cache = [[NSMutableDictionary alloc] init];
    }
    return self;
}
```

Cache: Compound-Key

```
- (id)cachedObjectForNumberValue:(double)number
                     stringValue:(NSString*)string
                  creationBlock:(id (^)(double number,
                                        NSString* string))block
{
    NSParameterAssert(stringValue);
    NSAssert(_cache, nil);
    COCCacheKey* key = [[COCCacheKey alloc] initWithNumberValue:number
                                                    stringValue:string];
    id object = _cache[key];
    if(!object && block)
    {
        object = block(number, string);
        NSAssert(object, nil);
        _cache[key] = object;
    }
    return object;
}
```

Demo

Implementierung in Objective-C++

Cache: Compound Key

```
struct CacheKey
{
    double      numberValue;
    NSString*   stringValue;

    bool operator==(const CacheKey& key) const {
        return numberValue == key.numberValue
            && [stringValue isEqualToString:key.stringValue];
    }
};
```

Cache: Compound Key

```
namespace std {
    template<>
    struct hash<CacheKey>
    {
        size_t operator()(const CacheKey& key) const
        {
            size_t hash = 17;
            hash        = 37 * hash + key.numberValue;
            hash        = 37 * hash + key.stringValue.hash;
            return hash;
        }
    };
}
```

```
- (id)cachedObjectForNumberValue:(double)number
                     stringValue:(NSString*)string
                   creationBlock:(id (^)(double number, NSString* string))block
{
    NSParameterAssert(string);
    CacheKey key = { number, string };
    if(block) {
        __strong id& object = _cache[key];
        if(!object && block) {
            object = block(number, string);
            NSAssert(object, nil);
        }
        return object;
    }
    else {
        auto objectIterator = _cache.find(key);
        if(objectIterator == _cache.end())
            return nil;
        else
            return (*objectIterator).second;
    }
}
```

```
- (id)cachedObjectForNumberValue:(double)number
                     stringValue:(NSString*)string
                  creationBlock:(id (^)(double number, NSString* string))block
{
    NSParameterAssert(string);
    CacheKey key = { number, string };
    if(block) {
        __strong id& object = _cache[key];
        if(!object && block) {
            object = block(number, string);
            NSAssert(object, nil);
        }
        return object;
    }
    else {
        auto objectIterator = _cache.find(key);
        if(objectIterator == _cache.end())
            return nil;
        else
            return (*objectIterator).second;
    }
}
```

Stackobjekt

```
- (id)cachedObjectForNumberValue:(double)number
                     stringValue:(NSString*)string
                  creationBlock:(id (^)(double number, NSString* string))block
{
    NSParameterAssert(string);
    CacheKey key = { number, string };
    if(block) {
        __strong id& object = _cache[key];
        if(!object && block) {
            object = block(number, string);
            NSAssert(object, nil);
        }
        return object;
    }
    else {
        auto objectIterator = _cache.find(key);
        if(objectIterator == _cache.end())
            return nil;
        else
            return (*objectIterator).second;
    }
}
```

Immer Referenz auf Bucket

```
- (id)cachedObjectForNumberValue:(double)number
                     stringValue:(NSString*)string
                   creationBlock:(id (^)(double number, NSString* string))block
{
    NSParameterAssert(string);
    CacheKey key = { number, string };
    if(block) {
        __strong id& object = _cache[key];
        if(!object && block) {
            object = block(number, string);
            NSAssert(object, nil);
        }
        return object;
    }
    else {
        auto objectIterator = _cache.find(key);
        if(objectIterator == _cache.end())
            return nil;
        else
            return (*objectIterator).second;
    }
}
```

map-Iteratoren

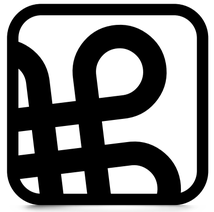
```
typedef pair<const Key, T> value_type;

unordered_map<Key, T>::iterator it;
(*it).first;        // Schlüssel (von Typ Key)
(*it).second;       // Gemappter Wert (vom Typ T)
(*it);              // Der "Elementwert"(vom Typ pair<const Key, T>)

it->first;         // identisch zu (*it).first
it->second;        // identisch zu (*it).second
```

Zusammenfassung

- Objective-C++
  - bietet mächtige Collections
  - mit hoher Abstraktion
  - und optimaler Performance
  - die sich wunderbar mit herkömmlichen Objective-C verbinden lassen
  - Jedes malloc ist ein Fehler!

Fragen?

Vielen Dank



Macoun