Informatik I – Übung 11

Pascal Schärli

pascscha@student.ethz.ch

17.05.2019

ETH zürich

Was gibts heute?

- Best-Of Vorlesung
  - Nullpointer
  - Container
  - Iteratoren
  - Linked List
- Vorbesprechung

Vorlesung

Nullpointer

- Wie kann man es schaffen, dass ein Pointer auf nichts zeigt?
- Es gibt in C++ den sogenannten *null* Pointer, welcher verwendet werden kann um eine Variable effektiv auf nichts zeigen zu lassen.

```cpp
#include <iostream>

int main() {

    int* p = new int(10);
    std::cout << p << std::endl;

    delete p;
    std::cout << p << std::endl;

    p = nullptr;
    std::cout << p << std::endl;

    return 0;
}
```

```
0x55beafb36e70
0x55beafb36e70
0
```

Syntaktischer Zucker

Beide Schreibweisen sind äquivalent, aber
das Rechte *(->)* wirkt eleganter

```cpp
#include <iostream>
#include <vector>

int main() {

    std::vector<int>* v = new std::vector<int>;

    (*v).push_back(3);

    std::cout << (*v).size() << std::endl;

    return 0;
}
```

```cpp
#include <iostream>
#include <vector>

int main() {

    std::vector<int>* v = new std::vector<int>;

    v->push_back(3);

    std::cout << v->size() << std::endl;

    return 0;
}
```

Container

- In C++ gibt es verschiedene Arten eine Sammlung von Daten zu speichern.
- Alle diese Arten nennt man Container
- Ihr kennt bereits ein paar Container: Arrays, Strings, Vektoren
- Es gibt aber noch andere Container:
  - Set
  - Queue
  - usw... *(Liste aller Container)*

Set

Ein Set ist ähnlich wie ein Vektor, aber es Speichert seine Elemente in Sortierter Reihenfolge ab.

Name	Vektor-Pendent	Funktionalität
`size`	`size`	Grösse (Länge) vom Set
`empty`	`empty`	true falls Set leer ist
`insert`	`push_back`	Element *sortiert* hinzufügen
`remove`	`pop_back`	Löscht Element *mit bestimmten Wert*
`begin`	`begin`	Iterator auf erstes Element im Set
`end`	`end`	Iterator auf Ende (eins nach dem letzten)

Eine komplette Liste findet ihr hier unter "Member Functions"

Iteratoren

- Pointers erlauben es uns zusammenhängende Speicherbereiche zu traversieren
- Andere Container können jedoch nicht mit solchen Pointers traversiert werden
- Aus diesem Grund stellen uns Containers *Iteratoren* zur Verfügung, welche uns diese Funktionalität ermöglichen.

Iteratoren

```cpp
#include <vector>
#include <iostream>

int main(){
    std::vector<int> cont = {8,3,1,4,6,9};

    for (std::vector<int>::iterator it = cont.begin(); it != cont.end(); ++it) {
        std::cout << *it << " ";
    }
    return 0;
}
```

```
8 3 1 4 6 9
```

Iteratoren

Der selbe Code funktioniert genau gleich bei anderen Containertypen, wie z.B. bei einem set:

```cpp
#include <set>
#include <iostream>

int main(){
    std::set<int> cont = {8,3,1,4,6,9};

    for (std::set<int>::iterator it = cont.begin(); it != cont.end(); ++it) {
        std::cout << *it << " ";
    }
    return 0;
}
```

```
1 3 4 6 8 9
```

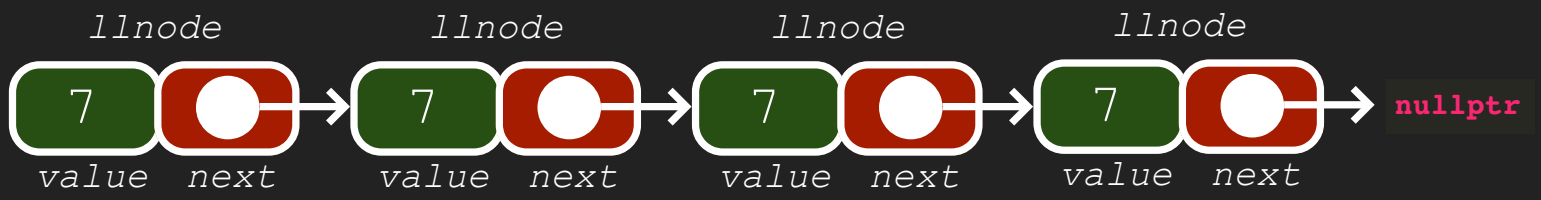
Linked List

- Letzte Woche haben wir gesehen, wie wir mit *new[]* Arrays mit einer bestimmten Grösse erstellt werden können.
- Dies ist eher unflexibel, da die Grösse des allozierten Speicherbereichs nicht verändert werden kann.
- Linked Lists erlauben einem dynamisch die grösse einer Liste beliebig zu ändern.

Linked List

- Um dieses Ziel zu erreichen speichern wir unsere Werte in sogenannten "Nodes"
- Nodes sind Structs, welche einerseits einen Wert (value) beinhalten aber auch einen Pointer (next) welcher auf die nächste Node in der Liste zeigt.

```
struct llnode {
  int value;
  llnode* next;
};
```



Linked List

- Alles diese Nodes organisieren wir in einer Klasse, welche alle Utilities Funktionen (zB *push_front* usw) implementiert.
- Sie haben auch noch einen Iterator (*const_iterator*) implementiert, welchen ihr nicht zwingend verstehen müsst.

```
class llvec {

  llnode* head;

public:
  // POST: e is appended at the beginning
  //        of the vector.
  void push_front(int e);
  // PRE: begin and end are iterators
  //       pointing to the same vector
  //       and begin is before end.
  // POST: All elements between begin and
  //        end are added to the vector.
  void extend(const_iterator begin,
              const_iterator end);
  // POST: Returns an iterator that points
  //        to the first element.
  const_iterator begin() const;
  // POST: Returns an iterator that points
  //        after the last element.
  const_iterator end() const;
};
```

llvec::extend

Implementiert die *extend* Funktion von dieser Linked List:

1. Findet die letzte Node vom Vektor:
   Benutzt den Iterator begin und geht solange weiter bis ihr das Ende erreicht habt:
   node->next == nullptr

2. Erstellt eine Node für jedes Element zwischen den Iteratoren *begin* und *end*.

```
class llvec {

  llnode* head;

public:
  // POST: e is appended at the beginning
  //        of the vector.
  void push_front(int e);
  // PRE: begin and end are iterators
  //       pointing to the same vector
  //       and begin is before end.
  // POST: All elements between begin and
  //        end are added to the vector.
  void extend(const_iterator begin,
              const_iterator end);
  // POST: Returns an iterator that points
  //        to the first element.
  const_iterator begin() const;
  // POST: Returns an iterator that points
  //        after the last element.
  const_iterator end() const;
};
```

llvec::extend

```
void llvec::extend(llvec::const_iterator begin,
                   llvec::const_iterator end) {
  if (begin == end) {
    return;
  }
  llvec::const_iterator it = begin;
  if (this->head == nullptr) {
    this->head = new llnode { *it, nullptr };
    ++it;
  }
  llnode *n = this->head;
  while (n->next != nullptr) {
    n = n->next;
  }
  for (; it != end; ++it) {
    n->next = new llnode { *it, nullptr };
    n = n->next;
  }
}
```

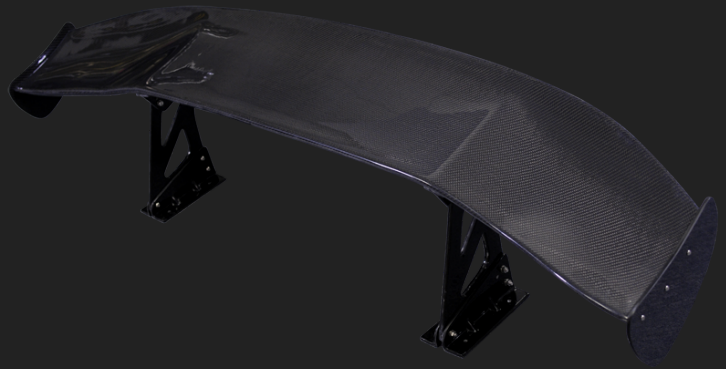
llvec::swap

- Implementiert eine swap methode, welche eine Node mit gegebenem Index mit ihrem nachfolger swapt.
- Ihr dürft davon ausgehen, dass alle Inputs so sind, dass es funktioniert.

llvec::swap

```
1  void llvec::swap(unsigned int index) {
2    if (index == 0) {
3      llnode* tmp = this->head->next;
4      this->head->next = this->head->next->next;
5      tmp->next = this->head;
6      this->head = tmp;
7    } else {
8      llnode* prev = nullptr;
9      llnode* curr = this->head;
10
11     // Find the element.
12     while (index > 0) {
13       prev = curr;
14       curr = curr->next;
15       index--;
16     }
17
18     // Swap with the next one.
19     llnode* tmp = curr->next;
20     curr->next = curr->next->next;
21     tmp->next = curr;
22     prev->next = tmp;
23   }
24 }
```

Vorbesprechung



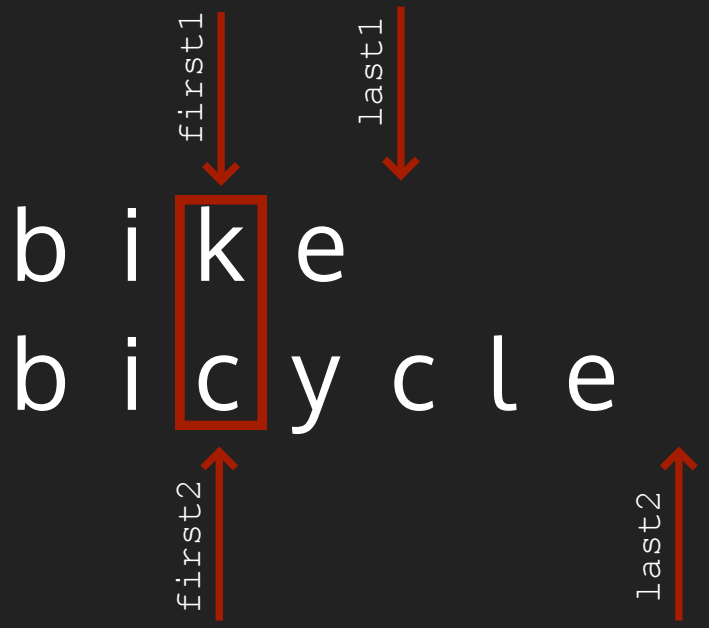
1: Lexicographic Comparison

Schreibt ein Programm, welches Strings lexikografisch vergleichen kann.

apple	<	banana	Alphabetisch sortiert nach ASCII ordering
bike	>	bicycle	Man beachtet nicht nur der erste Buchstabe
web	<	website	Kleinere Wörter sind auch lexikografisch kleiner

1: Lexicographic Comparison

```
bool lexicographic_compare(Iterator first1, Iterator last1,
                           Iterator first2, Iterator last2) {
```



c < k

⇩

bicycle < bike

1: Lexicographic Comparison

```
bool lexicographic_compare(Iterator first1, Iterator last1,
                           Iterator first2, Iterator last2) {
```

Tipp:

- Die Funktion lexicographic Compare gibt nur aus, ob string1 < string2 gilt.
- Die Aufgabe verlangt aber auch, dass ihr prüft wenn zwei Strings gleich sind.
- Wenn weder a < b noch b < a gilt, muss zwangsläufig a == b gelten.

2: Decomposing Sets

input:

```
3   5   8   6   5   1   2
- 1
```

std::set<int>:

```
1   2   3   5   5   6   8
```

output:

```
([1,3]U[5,6]U[8,8])
```

3: Dynamic Queue

- In dieser Aufgabe implementieren wir eine eigene Queue für Integer.
- Eine Queue ist eine Warteschlange, wer zuerst reinkommt kommt auch zuerst wieder raus.
- Wir implementieren die Funktion mit einer Linked-List

3: Dynamic Queue

```
struct Node {
    Node(const int _value, Node* _next) : value(_value), next(_next) { }
    const int value;
    Node* next;
};
```

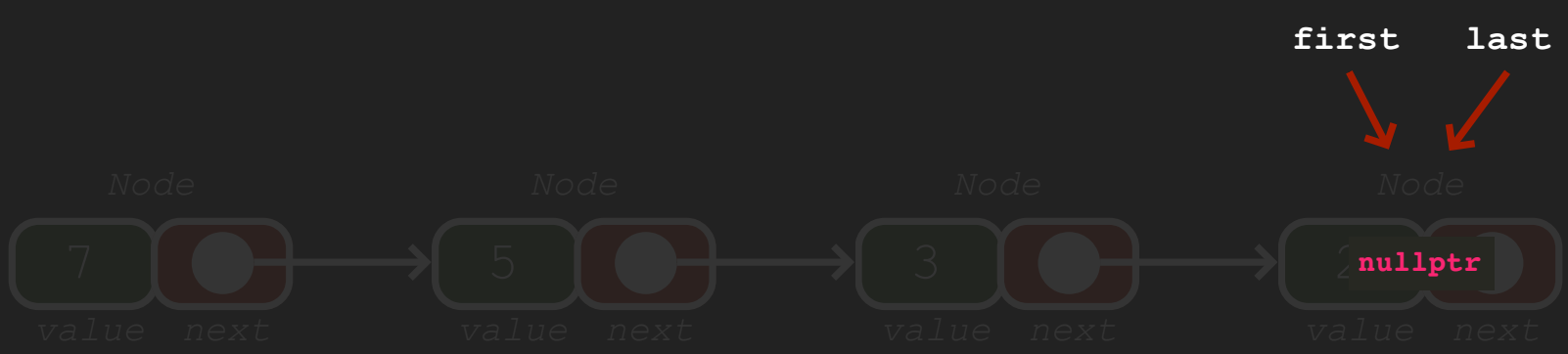
```
class Queue {
private:

    // Class invariant: first == nullptr iff last == nullptr
    Node* first; // pointer to first element of queue or nullptr is queue is empty.
    Node* last;  // pointer to last element of queue or nullptr if queue is empty.

    // PRE: -
    // POST: Ensures class invariant.
    void is_valid() const;

public:
    // PRE:  -
    // POST: Valid and empty queue.
    Queue();

    // PRE:  Valid queue.
    // POST: Valid queue with new node having value i added at end of queue.
```

3: Dynamic Queue Beispiel

```
1  #include <iostream>
2  #include "queue.h"
3
4  int main() {
5    Queue q;
6
7    q.enqueue(7);
8    q.enqueue(5);
9    std::cout << q.dequeue() << std::endl;
10   q.enqueue(3);
11   q.print(std::cout);
12   std::cout << std::endl;
13   q.print_reverse(std::cout);
14   std::cout << std::endl;
15   std::cout << q.dequeue() << std::endl;
16   std::cout << q.dequeue() << std::endl;
17   q.enqueue(2);
```

Output:

```
7
[5 3]
[3 5]
5
3
2
```



Viel Spass!


NEVER HAVE I FELT SO CLOSE TO ANOTHER SOUL
AND YET SO HELPLESSLY ALONE
AS WHEN I GOOGLE AN ERROR
AND THERE'S ONE RESULT
A THREAD BY SOMEONE WITH THE SAME PROBLEM
AND NO ANSWER
LAST POSTED TO IN 2003
WHO WERE YOU, DENVERCODER9?
WHAT DID YOU SEE?!