Übungsstunde

Woche 12

Adel Gavranović
adel.gavranovic@inf.ethz.ch

Übersicht

Heutige Themen

Intro

Iterators

`llvec::init`

Q&A

Dynamic vs Static

Pointers

Operator Overloading

EBNF

Rekursion

Outro

Links

- polybox zum Material für die Übungsstunden
- Mail an Assistenten

Follow-up aus vorheriger Übungsstunde

Letzte Übungsstunde kamen noch Fragen zum Beispielcode ”Rational Numbers and Classes” aus der Lektion auf. Hier ein paar hilfreiche Links zum Verständnis:

- ▸ `operator++(int)`
- ▸ `double()`

Ich weiss nicht wie direkt das prürel ist, aber sicher nicht schlecht, wenn man diese Arten vor overloading auch kennt. Nichts daran sollte neu oder sehr überraschend für euch sein.

Informationen

- Die Sache mit den ”Speicherslots”. . .
- Es gibt momentan ein Problem mit der Punktevergabe in der Bonus Exercise 1. Ist aber in Abklärung und wird gefixt
- Definitionen von `iostream` -operatoren (z.B. `operator>>`) können nur Out-of-Class gemacht werden[1]
- Kleinigkeit zu Default Constructors
- Timeouts in Bonus Exercise 2 (gibt eine Aufgabe die recht anfällig dafür ist)

[1] ▸ cppreference

Kommentare zu [code]expert

E10:T1 Finite Rings

- Was sind Invarianten? (`INV` vs. `PRE`)
  - Bonus: Was bringen sie uns? (Tipp: (`operator-`) und Range)
- Seid nicht so scheu und verwendet Member direkt statt extra Variablen zu initialisieren

E10:T2a Complex Numbers

- benutzt Funktionen, die ihr bereits implementiert habt (bspw. `operator==`)
- schaut euch die MS an für paar elegante Tricks

E10:T2b Calculator for Complex Numbers

- basically nur copy-and-paste, aber es kam zu Problemen bei einigen. Mehr Infos in der nächsten Übungsstunde

Fragen zu [code]expert eurerseits?

Lernziele

- ☐ Iteratoren verwenden können
- ☐ Einfache Container implementieren können

Iteratoren

- Iteratoren sind dazu da, um über einen Container zu iterieren, sei das ein `std::set`, `std::vector` oder euer eigener Container
- Sie funktionieren sehr ähnlich wie Pointers und haben etwa die gleiche Syntax: `++it` bewegt sie quasi eins weiter, `*it` lässt auf die zugrundeliegenden Daten/Objekte zugreifen
- Die Initialisierung ist etwas gewöhnungsbedürftig (siehe Summary)
- Ich stell sie mir als *fancy pointers* vor

Sets (Mengen) vs. Vectors (Vektoren)

Was sind die Unterschiede (mathematisch betrachtet) zwischen Mengen und Vektoren?

Iterators in Code (`std::vector`)

```
1  std::vector<int> cont = {8, 3, 1, 4, 6, 9};
2
3  for ( std::vector<int>::iterator it = cont.begin();
4        it != cont.end();
5        ++it) {
6
7      std::cout << *it << " ";
8  }
```

Iterators in Code (`std::set`)

```
1  std::set<int> cont = {8, 3, 1, 4, 6, 9};
2
3  for ( std::vector<int>::iterator it = cont.begin();
4        it != cont.end();
5        ++it) {
6
7      std::cout << *it << " ";
8  }
```

Exercise "llvec::init"

Description

The files `vector_linkedlist.h` and `vector_linkedlist.cpp` contain a simplified version of the llvec-vector from the lecture slides. Implement the constructor that initializes the vector with all elements from the iterator.

Hints:
How can you add the first element from the iterator?
How can you add any other element from the iterator?

Persönlicher Tipp

Verschwendet keine Zeit damit, die anderen Memberfunktionen zu verstehen. Schaut euch einfach die `PRE/POSTs` und Kommentare dazu an.

Exercise "`llvec::init`"

Simplere Beschreibung

Implementiere den Constructor `llvec::llvec(begin, end)`.

Dieser Constructor initialisiert einen neuen `llvec` und fügt die Werte ein, die ein einem anderen `llvec` zwischen `begin` und `end` liegen.

In der `llvec`-class: Basics

```
1  struct llnode {
2    int value;
3    llnode* next;
4  };
5
6  llnode* head;
```

In der `llvec`-class: `const_iterator` class

```
1  class const_iterator {
2        const llnode* node;
3     public:
4        const_iterator(const llnode* const n);
5
6        // PRE: Iterator does not point to the element
7        //      beyond the last one.
8        // POST: Iterator points to the next element.
9        const_iterator& operator++(); // Pre-increment
10
11       // POST: Return the reference to the number at
12       //       which the iterator is currently pointing.
13       const int& operator*() const;
14
15       // True if iterators are pointing to different elements.
16       bool operator!=(const const_iterator& other) const;
17
18       // True if iterators are pointing to the same element.
19       bool operator==(const const_iterator& other) const;
20 };
```

In der `llvec`-class: Member-Funktionen

```
1  // Default Constructor
2  llvec();
3
4  // PRE: begin and end are iterators pointing to
5  //      the same vector and begin is before end.
6  // POST: The constructed llvec contains all
7  //       elements between begin and end.
8  llvec(const_iterator begin, const_iterator end);
9
10 // POST: e is prepended to the vector.
11 void push_front(int e);
12
13 // POST: Returns an iterator that points
14 //       to the first element.
15 const_iterator begin() const;
16
17 // POST: Returns an iterator that points
18 //       after the last element.
19 const_iterator end() const;
```

Visualisierung von `llvec`

Lösungskonzept für ”`llvec::init`”

”llvec::init” Lösung

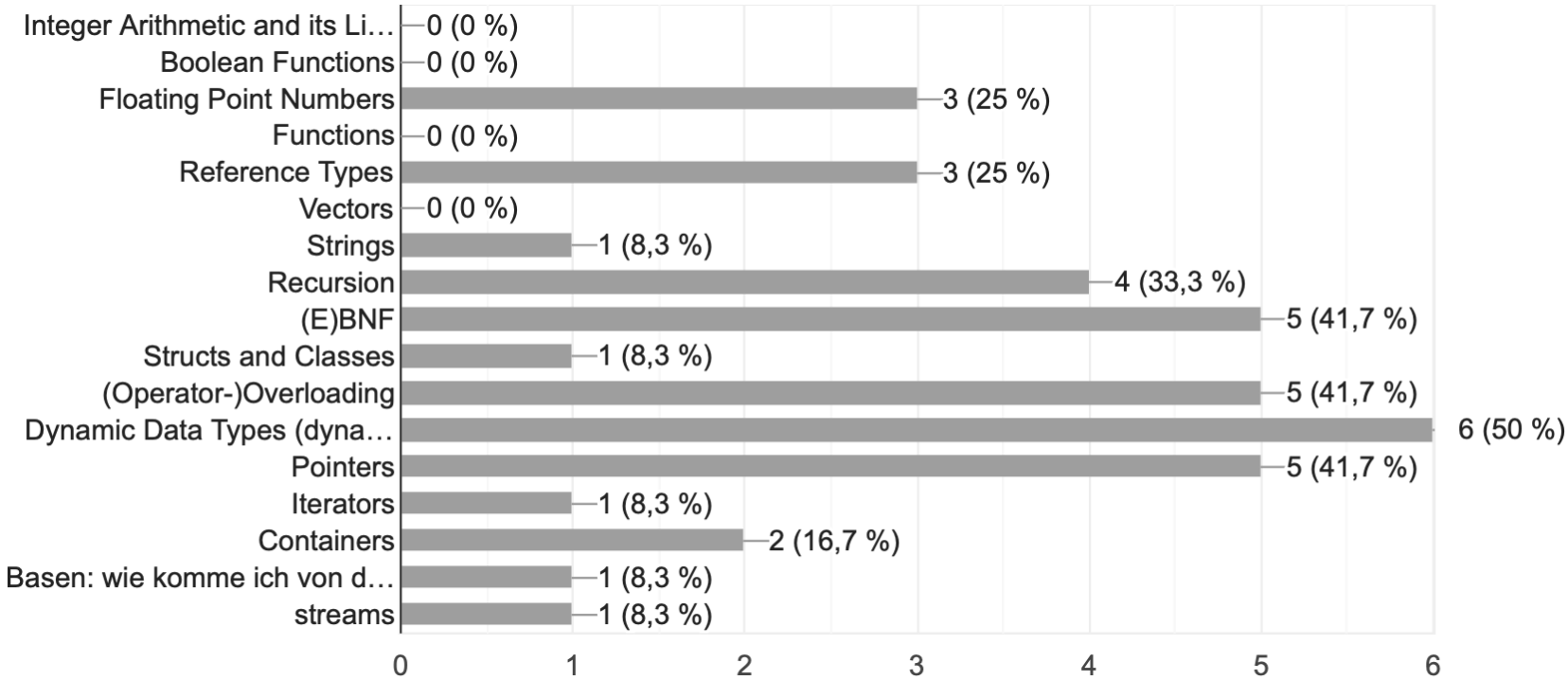
```
1  llvec::llvec(llvec::const_iterator begin,
2               llvec::const_iterator end) {
3    this->head = nullptr;
4    if (begin == end) {
5      return;
6    }
7
8    llvec::const_iterator it = begin;
9    // Let's add the first element from the iterator.
10   this->head = new llnode{*it, nullptr};
11   ++it;
12   llnode* current_node = this->head;
13   // Let's add all the remaining elements.
14   while(it != end){
15      current_node->next = new llnode{*it, nullptr};
16      current_node = current_node->next;
17      ++it;
18   }
19 }
```

Exercise "`llvec::init`"

Aufgabe

Versucht diese Aufgabe heute Abend noch einmal und macht euch dazu Skizzen von den Nodes und wie der ll-Vektor aussieht.

”Welche Konzepte möchtest du gerne kommende Lektion (noch einmal) anschauen?”



Stand: 07.12.21, 11:30

Eure Fragen

- ”Bei Pointern fände ich es interessant, wie man herausfinden kann, woher Compiler errors kommen, da diese nicht mehr so eindeutig sind”
  - In den allermeisten Fällen sind es ”Segmentation Faults” (*SegFaults*)[2], die sind aber eigentlich keine Compiler Errors
  - Compiler Errors bei Pointern kommen glaube ich nur vor, wenn man `const` Pointer verändert
- ”Müssen wir Float Umrechnungen für alle Basen können, oder nur von Dezi $\rightarrow$ Binär (und vice versa)?”
  - Leider ja, aber die wichtigsten sind sicher Basis 2, 8, und 16.
- ”Basen: wie komme ich von der einen in die andere? Rechentipps?”
  - Tricks kenne ich leider keine, aber das Vorgehen ist identisch mit dem für Basis 2. Vielleicht zur Sicherheit einen Zwischenschritt via Basis 10

[2] ▸ stackoverflow , ▸ Wikipedia

Eure Fragen

- ”Tipps für Coding unter druck/ während Prüfung (hab noch keine Coding Aufgabe bei den self assessments geschafft)?”
  - Flashcards für ”Theorieteil” (Anki!)
  - Speedrun Coding Exercises (Üben, üben, üben!)
  - Aufgaben, die man nicht sofort lösen kann, einfach skippen und am Schluss anschauen (gilt für alle Prüfungen)
  - Wenig/er Koffein an der Prüfung (ernsthaft)
- ”Ich glaube ich habe bis anhin noch nicht so wircklich verstanden, was r- und l-values sind. Das nochmal ganz kurz anzuschneiden wär schön :)”
  - L-values haben einen ”Slot” irgendwo im Speicher, d.h. man kann den Inhalt dieses ”Slots” ändern. ”L” rührt daher, dass diese Werte beim Zuweisungsoperator (=) immer auf der *linken* Seite stehen
  - R-values haben *keinen* ”Slot”
  - Gewisse Operatoren und Expressions geben entweder einen L- oder einen R-Value zurück. (Pre- vs. Post-incr. nachschauen!)

Eure Fragen

- ”functions mit reference type, z.b `int &foo(int x1, int x2)` und `const int & foo` ”
  - Was genau ist hier die Frage?
- ” `this->` nochmals anschauen, und der zusammenhang mit pointers”
  - `this` ist ein Pointer, der zu dem Objekt zeigt, in dem es verwendet wird
  - `->` ist ein Operator, der den vorangehenden Pointer (oft ein `this`) dereferenziert (also den direkten Zugriff auf das darunterliegende Objekt ermöglicht)

Eure Fragen

- ”bei operator overloading, nochmal die grundkonzepte anschauen, also nicht wie es geht sondern warum? wieso kann ich beim aufruf der funktion `operator+(int x, int y) a+b` schreiben und muss nicht `+(a,b)` schreiben, oder warum funktioniert das `+` mit einem funktionsargument, und das im zusammenhang mit `this` und `other` ”
  - Wieso das so funktioniert weiss ich auch nicht
  - zu `this` : siehe die vorherige Frage/Antwort dazu
  - zu `other` : das ist einfach der Name, dem wir dem zweiten Objekt in einer binären Operation (`+` , `-` , `/` , etc.) geben
  - Wann genau man nur ein Argument als Input braucht und wann zwei, schaut man am besten in der vorherigen Exercises nach und schreibt es sich auf die Zusammenfassung

Eure Fragen

- ”istream/ostream genau gleich wie `std::cin` und `std::cout` , warum nicht einfach `std::string` ? und was ist ifstream?”
  - `std::string` ist *kein* stream, sondern quasi ein `std::vector<char>` mit Spezialfähigkeiten
  - `std::string` sind quasi der Datentyp für Wörter und Sätze
  - ein `ifstream` ist ein input-file-stream, also eine Art ”Zufluss via einem File” (nzz.txt, hello.txt, etc.) statt via der Console (std::istream)

Dynamic vs Static

- Wofür?
  - Dynamisch allozierter Speicher wird verwendet, wenn wir Sachen initialisieren möchten, die ihren ursprünglichen Scope überleben sollen
  - Statische (”normale”) Objekte werden dekonstruiert (gelöscht), sobald der Scope in dem sie init’ed wurden endet (alsi wenn die ”}” erreicht ist). Dies ist insb. bei Funktionen der Fall, welche ja auch Scopes sind
  - Ihr sollt wissen, wie man mit Pointern umgeht, um eben mit diesen ”langlebigen” dynamischen Objekten arbeiten zu können
- Wann?
  - Wenn man ein Objekt ausserhalb seines ursprünglichen Scopes am leben erhalten möchte
  - Wenn es die Exercise Description verlangt
- Wie?
  - Mit dem Keyword `new`
  - Zu jedem `new` sein `delete` ! (mehr dazu später)

Dynamic Variables vs Static Variables

```
1  // statically (normally) allocated variable
2  int n = 42;
3  // accessing it
4  n = 1;
5
6  // dynamically allocated variable
7  int* d = new int(42);
8  // accessing it
9  *d = 1;
```

Dynamic Arrays

```
1  // dynamically allocated array of variables
2  int* d = new int[5];
3  // accessing and modifying
4  d[0] = 1337;    // first int in array
5  d[4] = 42;        // last int in array
```

Summary zu Dynamic Data Types

new	**Objekt mit dynamischer Lebensdauer** erstellen.

Mit new wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener Konstruktor aufgerufen wird.

Der Rückgabewert von new ist ein *Pointer* auf das neu erstellte Objekt.

```
Class My_Class {
public:
    My_Class (const int i) : y (i) { std::cout << "Hello"; }
    int get_y () { return y; }
private:
    int y;
};

...
My_Class* ptr = new My_Class (3); // outputs Hello
My_Class* ptr2 = ptr; // another pointer to the new object
std::cout << (*ptr).get_y(); // Output: 3
...
```

Summary zu Dynamic Data Types

`new ...[]`	**Ranges mit dynamischer Lebensdauer und Länge** erstellen.

```
int n; std::cin >> n;
int* range = new int[n];
// Read in values to the range
for (int* i = range; i < range + n; ++i) std::cin >> *i;
```

Pointers

- Wir benutzen Pointer hauptsächlich, um dynamisch allozierte Objekte zu verwalten (und schwierige Prüfungsfragen zu stellen)
- Sie sind nicht viel mehr als eine Adresse im Speicher
- Wenn wir sie `std::cout` -en, dann sehen sie meist so aus: `0xDB11E0` , also einfach eine Hexidezimalzahl

Pointers

Operator-overloading

Operator-overloading

EBNF

EBNF

Recursion

Recursion

Tipps für [code]expert

E12:T1 Lexicographic comparison

- "**Note**: Using `std::string` comparison functions from the standard library is not allowed. Note that the operators on strings such as `==` are implemented by `std::string` , and thus counts as library functions."
- Iteration (mit `while()`) statt Rekursion

E12:T2 Dynamic Queue

- Skizze anfertigen!
- `print_reverse()` benötigt Rekursion. Sehr cooler Trick, ist quasi "durchgehen bis zum Schluss und dann Printen"

E12:T3 Decomposing a Set into Intervals

- 3 Steps
  - Input verwalten
  - Intervalle finden
  - Intervalle printen
- Vergesst nicht: das Set ist bereits geordnet

Allgemeine Fragen?

Bis zum nächsten Mal

- ☐ macht eure Hausaufgaben
- ☐ bleibt gesund