# Übungsblatt 0

Abgabe via Moodle. Deadline Fr., den 28. Juni

Um eine Klausurzulassung zu erhalten, müssen mindestens 25% der Punkte auf diesem Blatt erreicht werden! Es wird keine Fristverlängerung gewährt.

## Allgemeine Regeln

- Alle Aufgaben sind in C++17 zu bearbeiten. Im Zweifelsfall muss die Abgabe mittels `x64-64 gcc 14.1` auf `https://compiler-explorer.com/` funktionieren (wenn der entsprechende Code in eine einzelne Datei zusammen kopiert wurde).
- Ändern Sie **niemals** die Signatur von existierenden Funktionen. Wir verwenden zusätzliche Tests, die die vorgegebenen Signaturen benötigen. Sie dürfen aber neue Funktionen hinzufügen.
- Wir erwarten, dass alle Abgaben ohne Warnungen und Fehler kompilieren; i.A. bewerten wir nicht kompilierende Abgaben mit 0 Punkten (beachten Sie, dass wir `-Werror` nutzen und somit Warnungen als Fehler behandelt werden)
- Geben Sie alle Quelldateien (keine Kompilate oÄ) in einem ZIP-Archiv ab. Wenn Sie die Bonusaufgabe bearbeiten, fügen Sie zusätzlich die aufgezeichneten Rohdaten und deren Auswertung der ZIP-Datei bei.

## Tipps

- Nutzen Sie das beigefügte Skript `build.sh` zum Kompilieren; evtl. muss die Zeile `CXX=g++` angepasst werden, falls Sie einen anderen C++ Compiler verwenden.
- Wenn Sie *nicht* das Skript nutzen, schalten Sie Warnung in Ihrem Compiler an, z.B.

```
g++     --std=c++17 -Wall -Wextra -Wpedantic -Werror -g -O0 tests.cpp -o tests
clang++ --std=c++17 -Wall -Wextra -Wpedantic -Werror -g -O0 tests.cpp -o tests
```

- Der Vorständigkeit-halber liegt dem Projekt auch eine `cmake` Konfigurationsdatei bei. Wir empfehlen Ihnen `Debug`-Builds zu nutzen, um möglichst aussagekräftige Fehlermeldungen zu bekommen. (z.B. mittels `cmake -DCMAKE_BUILD_TYPE=Debug ..`).

## Testfälle

Sie werden wiederholt gebeten, Testfälle zu schreiben. Hierfür haben wir ein kleines Rahmenwerk erstellt. Um einen neuen Testfall zu schreiben, gehen Sie wie folgt vor:

1. Öffnen Sie die Datei `tests.cpp` und erstellen eine neue Funktion `bool test_XXX()` wobei XXX durch einen geeigneten Namen ersetzt werden soll.
2. Implementieren Sie den Test. Jede Testfunktion gibt `true` zurück, falls der Test klappte, und `false` sonst. Um den Vorgang zu vereinfachen, haben wir Ihnen folgende Hilfsfunktionen erstellt:

- `fail_if(cond)`: Wenn die Bedingung `cond` als `true` ausgewertet wird, wird eine Fehlermeldung ausgegeben und die Funktion direkt mit `false` verlassen.
- `fail_if_eq(a, b)`: Verursacht einen Fehler, wenn $a$ und $b$ gleich sind.
- `fail_unless(cond)`, `fail_unless_eq(a, b)`: die Negationen der beiden Varianten oben. So stellt `fail_unless_eq(a, b)` sicher, dass $a$ und $b$ identisch sind.

  Tipp: Beachten Sie, dass bei eingeschalteten Warnung z.B. auch geprüft wird, dass keine vorzeichenbehafteten Datentypen (etwa `int`) mit vorzeichenlosen Datentypen (etwa `size_t`) verglichen werden. Das tritt z.B. auf, wenn die Länge eines Containers mit einem `int` verglichen wird; ggf. müssen Sie also explizite Casts nutzen:

  ```
  // Vollständiger Test siehe ‘test_push_front()‘ in ‘tests.cpp‘
  for(int i=0; i<10; ++i) {
      fail_unless_eq(lst.size(), static_cast<size_t>(i));
      lst.push_front(i);
  }
  ```

3. Rufen Sie die Funktion auf. Erstellen Sie hierzu in `main()` eine neue Zeile `run_test(test_XXX);`.
4. Bauen Sie das Projekt (`bash build.sh`) und führen Sie den Test aus (`./tests`). Tipp: Wenn Sie Bash nutzen, können Sie beide Befehle verbinden: `bash build.sh && ./tests` . Dabei sorgt `&&` dafür, dass das Programm nur ausgeführt wird, wenn des Kompilieren klappte.

Testfälle sollten immer einen sinnvollen Umfang haben; so ist es z.B. wichtig zu prüfen, ob eine leere Folge korrekt sortiert wird — wäre dies jedoch der einzige Test eines Sortieralgorithmus, wäre dies nicht ausreichend.

### std::unique_ptr

In der Vorlesung konzentrieren wir uns i.d.R. auf die Kernaufgaben von Datenstrukturen und vernachlässigen oft, dass einmal allokierter Speicher (`new`) auch wieder freigeben werden muss (`delete`). Vergessen wir dies, kommt es zu Speicherlecks (leaks).

Um die Speicherverwaltung zu automatisieren, verwendet modernes C++ sog. `smart pointer`, wobei der einfachste der sog. `std::unique_ptr` (kurz UP) ist. Ein UP besitzt einen Pointer (auf diesen können wir mit `get` zugreifen) und gibt den Speicherbereich wieder frei, sobald der UP selbst zerstört wird. Diese Gültigkeitsanalyse wird vom Compiler durchgeführt und hat keinerlei Laufzeitkosten!

```
{
    // Die bevorzuge Methode einen unique_ptr zu erstellen ist mittels
    // std::make_unique. Die Funktion ruft automatisch ‘new Item(123)‘ auf.
    auto item = std::make_unique<Item>(123);

    // Wir können dann den Pointer mittels get anfragen:
    std::cout << "Item verwaltet den Pointer " << item.get() << "\n";

    // Aber auch ganz konfortabel -> nutzen (ist äquivalent zu ‘get()->‘):
    std::cout << "Item hat den Wert " << item->get_value() << "\n".

    // item geht ‘out-of-scope‘ und ruft automatisch ‘delete item.get()‘ auf
}
```

```
// Äquivalent ohne Smart Pointer wäre
{
    Item* item = new Item(123);
    std::cout << "item zeigt auf die Adresse " << item << "\n";
    std::cout << "item hat den Wert " << item->get_value() << "\n";
    delete item;
}
```

Ein UP muss aber nicht immer ein Objekt verwalten und kann auch *leer* sein (dann liefert `get()` den Nullpointer `nullptr` zurück). Wir können prüfen, ob ein UP gerade ein Objekt verwaltet, indem wir den UP `up` in ein `bool` casten:

```
if (up)  {std::cout << "Enthält ein Objekt\n";}
if (!up) {std::cout << "Ist leer\n";}
```

Die namensgebende Eingeschaft von UPs ist, dass es nur einen eindeutigen UP gibt, das ein Objekt verwaltet. Wir können daher keine Kopie eines UPs erstellen (da beide UPs bei ihrer Zerstörung das Objekt löschen würden und es zu eine *double-free* Fehler käme). Daher dürfen wir UPs nur *moven*. Ein Move verschiebt den Besitz einer verwalteten Adresse in einen anderen UP und lässt den alten UP leer zurück. Wichtig ist dabei, dass die eigentlich verwalteten Daten nicht verändert werden:

```
// Klassische Pointer dürfen kopiert werden, es sind ja nur "Adressen":
int* ptr1 = new int;
int* ptr2 = ptr1;

// Bei unique_ptr geht das nicht:
auto up1 = std::make_unique<int>(123);
// auto up2 = up1; // DAS IST ILLEGAL!

// Wir können aber den Besitz weitergeben:
auto up2 = std::move(up1);
assert(up2); // up2 hat jetzt den Besitz
assert(!up1); // up1 ist leer!

// Natürlich dürfen wir aber die unterliegende Adresse beliebig kopieren:
int* ptr3 = up2.get();
int* ptr4 = ptr3;
```

### Aufgabe 1 (*Listen aufräumen, 10 Punkte*)

Unique Pointer passen sehr gut auf die Semantik einer einfach-verkettenen Liste. Wenn wir den `next` Pointer als Unique-Pointer ausführen, schlagen wir mehrere Fliegen mit einer Klappe:

- Das letzte Element *besitzt* keinen Nachfolger (anders als in der Vorlesung, nutzen wir keine zyklischen Listen!). Wir können also direkt mittels `if (!current->next)` prüfen, ob wir das Ende der Liste erreicht haben. Alternativ können wir aber auch immer noch mit `nullptr` vergleichen:

  ```
  for(Item* current = dummy.next.get();
      current != nullptr;
      current = current->next().get())
  {
      std::cout << current->get_value() << "\n";
  }
  ```

- Wenn wir das Dummy/Head-Element löschen, sorgen die Unique-Pointer automatisch dafür, dass alle Nachfolger gelöscht werden:

```
{
    auto head        = std::make_unique<Item>(1);
    head->next       = std::make_unique<Item>(2);
    head->next->next = std::make_unique<Item>(3);

    // head geht hier out-of-scope und wird zerstört. bevor ‘head‘ zerstört wird,
    // wird rekursiv aber head->next->next, dann head->next und schließlich head gelöscht.
}
```

- Wir können mit z.B. bei `std::unique_ptr<Item> pop_front()` einen UP zurückgeben. Damit ist ganz klar geregelt: Der aufrufende Code bekommt das entnommen Objekt zurückgeben, besitzt es und kann es frei verwenden. Sobald es out-of-scope geht, wird es automatisch zerstört. Ohne Smart Pointer hätte die Dokumentation regeln müssen, wer den Speicher wieder freigibt.

```
List lst; // hier fehlt noch Code, der die Liste füllt
{
    auto item = list.pop_front();
    std::cout << "item hat den Wert " << item->get_value() << "\n";
    // item geht out-of-scope und der Speicher wird automatisch freigegeben.
    // Beachten Sie, dass pop_front() sicherstellt, dass item keinen Nachfolger hat;
    // somit wird nur das eine Item gelöscht.
}
```

UPs sind somit sehr komfortabel und sicher. Daher verwendet unsere Implementierung der einfach-verketteten Liste in der Datei `list.hpp` Smart Pointer. Gerade im Kontext von verketteten Listen gibt es aber eine subtile Fehlerquelle: Die Nachfolger werden rekursiv gelöscht. Wenn der Compiler keine `tail-recursion` Optimierung durchführt (z.B. weil wir nicht `-O2` übergeben haben), hat der Callstack eine Tiefe linear in der Listenlänge und läuft irgendwann über.

Sie können diesen Fehler forcieren, indem Sie in der Funktion `test_destruct` (in der Datei `tests.cpp`) eine längere Liste erzeugen; wenn Sie den Wert von 1000 auf 100000 (oder größer — der genaue Wert hängt vom System ab) erhöhen und ohne Optimierungen kompilieren, sollte das Programm abstürzen. (Falls nicht, haben Sie Glück — tun Sie im Folgenden so, als hätten Sie den Fehler gesehen ;))

Lösen Sie das Problem wie folgt: **Implementieren** Sie unterhalb der Zeile (`List(List&) = delete;`) einen Destruktor für `List`. Dieser Destruktor soll solange in einer Schleife `pop_front()` aufrufen, bis die Liste leer ist. Nutzen Sie dabei aus, dass –wie oben beschrieben– `pop_front()` einen UP zurück liefert, der automatisch gelöscht wird.

*Tipp*: Diese Teilaufgabe dient nur dazu, Sie mit dem Code vertraut zu machen. Die Lösung kann extrem kurz sein.

### Aufgabe 2 (*push_back_item, 10 Punkte*)

*Denken Sie bei dieser und allen folgenden Aufgaben daran, dass die Reihenfolge von Elementen in einer Liste allein durch die next-Pointer definiert werden. Die `value`-Einträge von Items sollen zu keinem Zeitpunkt verändert oder getauscht werden.*

Die Liste verfügt bereits über eine `push_front` und `pop_front` Implementierung. Zudem, ist auch `push_back` vorbereitet (und soll nicht verändert werden). Diese Methode verweist aber nur auf die unvollständige Methode `push_back_item`. **Implementieren** Sie `push_back_item` vollständig.

1. Erweitern Sie die Klasse `List` um ein privates Datenelement `last` vom Typ `Item*`. Es handelt sich also um einen klassichen Pointer und keinen smart pointer! Dieser soll folgende Datenstrukturinvariante erfüllen: `last` zeigt immer auf den letzten Eintrag der Liste oder auf `&dummy`, falls die Liste leer ist.
2. Um die Invariante aufrecht zu erhalten, müssen mindestens folgende Stellen angepasst werden:
   a) Der Standard-Konstruktor `List() {}` muss erweitert werden
   b) Die Methoden `push_front` und `extract_after` können das letzte Element verändern und müssen dann `last` anpassen
3. Vervollständingen Sie nun `Item* push_back_item(std::unique_ptr<Item>&& item)`. Die Signatur erzwingt mittels `&&`, dass ein UP als Argument hinein ge*moved* wird. Die Liste übernimmt also den Besitzt von `item` und darf es als ihr Eigentum betrachten:
   a) `item` muss nun ans Ende der Liste angefügt werden
   b) `last` und `size` müssen entsprechend angepasst werden
4. Fügen Sie mindestens einen Testfall hinzu, der `push_back` (nicht `push_back_item`) prüft. Sie können sich hierbei am vorhandenen Test `test_push_front` orientieren. Fügen Sie weitere Testfälle hinzu, die `push_back` aufrufen, nachdem `last` durch (i) `push_front` und (ii) `pop_front` verändert wurde.

### Aufgabe 3 (*concat, 5 Punkte*)

Die Funktion `void concat(List& other)` soll die Liste `other` an die aktuelle Liste (`this`) in Zeit $O(1)$ anhängen und dabei `other` leer zurück lassen. Die Implementierung ist auskommentiert, da sie erst funktioniert, sobald das Datenfeld `last` zur Verfügung steht; Sie können diese Aufgabe daher erst bearbeiten, wenn Aufgabe 2 abgeschlossen ist.

Die auskommentierte Implementierung wurde mit Hilfe von Copilot erstellt und enthält einen (typischen) Fehler. Erstellen Sie Testfälle, die den Fehler nachweisen, beschreiben Sie den Fehler kurz in einem Kommentar im Testfälle und korrigieren Sie ihn dann.

### Aufgabe 4 (*move_into_if, 10 Punkte*)

Wir betrachten die folgende Methode:

```
template <typename Predicate>
void move_into_if(List& append_to_if_true, Predicate &&predicate);
```

Diese Methode iteriert von Anfang bis zum Ende über die aktuelle Liste (`this`). Für jedes $x$ ruft die Funktion `predicate(x->get_value())` auf, die einen Wahrheitswert zurückliefert. Falls das Prädikat `true` ist, wird $x$ aus der aktuellen Liste entfernt und ans Ende von `append_to_if_true` angehängt. Viele Funktionen der C++ Standardbibliothek verwenden dieses Prädikats-Pattern, um generische Algorithmen für konkrete Entscheidungen zu implementieren (siehe z.B. `find_if`, `partition`, uvm.). In der Regel werden die Prädikate dabei mittels lambda-Ausdruck übergeben, z.B.:

```
List lst, lst_even;
for(int i=0; i<10; ++i) lst.push_back(i); // Enthält [0, 1, ..., 9]

auto is_even = [] (const List::Value& val) { return val % 2 == 0; };
lst.move_into_if(lst_even, is_even);

std::cout << lst << std::endl;      // gibt "[1, 3, 5, 7, 9]" aus.
std::cout << lst_even << std::endl; // gibt "[0, 2, 4, 6, 8]" aus.
```

Vervollständigen Sie die Implementierung von `move_into_if` in der Datei `list.hpp`. Nutzen Sie hierfür die Methoden `push_back_item` und `extract_after(before)`. Implementieren Sie Testfälle.

### Aufgabe 5 (*QuickSort, 15 Punkte*)

Implementieren Sie die Methode `sort` in dem Sie eine geeignete QuickSort Variante aus der Vorlesung umsetzen. Sie können sich dabei an der 3-Partitionen Variante orientieren, es fällt aber einfacher, wenn Sie nur zwei Partition nutzen. Nutzen Sie das erste Element der Eingabe als Pivot (siehe `pop_front`). Implementieren Sie den Paritionierungsschritt mittels `move_into_if`. Zum Kombinieren der Teillisten sind `concat` und `push_back_item` hilfreich. Die Methode soll zudem die Anzahl der Vergleiche zwischen Elementen der Liste berechnen und als Rückgabewert liefern.

Erstellen Sie Testfälle um die Korrektheit Ihres Algorithmus zu prüfen. Hierbei können Sie die Methode `is_sorted` nutzen.

### Aufgabe 6 (*Visualisierung, 10 Bonuspunkte*)

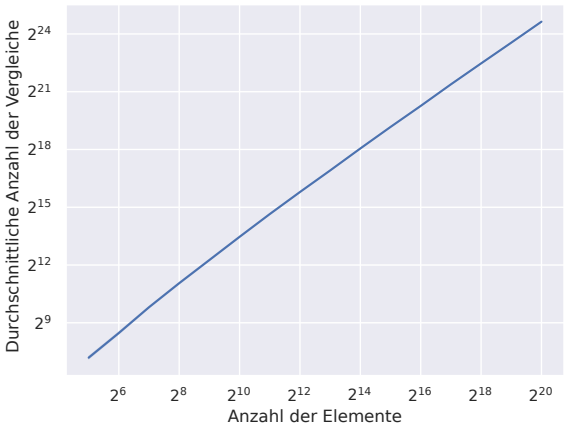
*Diese Aufgabe ist eine Bonusaufgabe, die nicht bearbeitet werden muss; hier erhaltene Punkte können genutzt werden, um einen etwaigen Punktverlust bei früheren Aufgaben dieses Blattes abzufedern.*

Das Programm in der Datei `sort.cpp` ruft die Funktion `sort` für verschiedene Problemgrößen $n$ auf und zeichnet die Anzahl an Vergleichen auf. Das Ergebnis wird in die CSV-Daten `compares.csv` geschrieben — pro Experiment eine Komma-getrennte Zeile, wobei in der ersten Spalte die Anzahl der Elemente und in der zweiten Spalte die Anzahl der Vergleiche gespeichert wird. Die ersten Zeile könnten so aussehen:

```
num_items,num_compares
32,170
32,127
32,165
32,134
```

Jedes Experiment wird mehrfach wiederholt um statistisch aussagekräftige Ergebnisse zu bekommen. Jeder Run nutzt eine zufällige Permutation paarweise verschiedener Zahlen in der Eingabe. **Wichtig:** Die Simulation ist relativ langsam (ca. 100s Gesamtlaufzeit). Sie sollten daher umbedingt ein optimiertes Kompilat nutzen (z.B. in dem Sie `-O3` Ihrem Compiler übergeben, oder `cmake -DCMAKE_BUILD_TYPE=Release ..` aufrufen. Das `build.sh` Skript kompiliert `sort.cpp` bereits optimiert).

Das Python-Skript `plot_compares.py` kann genutzt werden (im Header der Datei stehen Infos zum Installieren der Abhängigkeiten), um die CSV-Datei zu lesen und zu visualisieren — jedes andere Werkzeug (inkl. einer Tabellenkalkulation) wird aber akzeptiert. Das Skript erzeugt folgenden Plot:




Dieser Plot lässt relativ wenig Rückschlüsse zu. In der Vorlesung haben wir jedoch gezeigt, dass bei zufälliger Pivotwahl in Erwartung höchstens $2n \ln n$ 3-Wege Vergleiche benötigt werden. Eine übliche Analysetechnik besteht nun darin, die Y-Achse genau durch diese Vorhersage zu teilen. Würden sich dann für alle $n$ der Wert 1 ergeben, wäre unsere Analyse optimal gewesen.

- **Erzeugen** Sie einen entsprechend skalierten Plot, z.B. indem Sie das Python-Skript modifizieren. Es empfiehlt sich auf der X-Achse ein Log-Skala und auf der Y-Achse eine Lineare-Skala zu wählen (in `matplotlib` heißt die entsprechende Funktion `plt.semilogx(base=2)`). Interpretieren Sie diesen kurz und nehmen Sie insb. Bezug auf die Annahmen, die in der Vorlesung getroffen wurden.
- Die Zeile `std::iota(values.begin(), values.end(), 0)` in `sort.cpp` füllt den Vektor `values` mit aufsteigenden Zahlen von 0 bis $n-1$. Kommentieren Sie diese Zeile aus — dann ist der Vektor mit $n$ aufeinander folgenden Nullen gefüllt. Wiederholen Sie das Experiment (wählen Sie zunächst kleineren Wert für `max_n`), erstellen Sie einen neuen Plot und prüfen Sie, ob die Vorhersage noch stimmt. Falls nicht, schätzen Sie die asymptotische Laufzeit ab, teilen die Y-Achse durch die neue Abschätzung und erklären Sie das Phänomen.