Name:

Hon. Prof. Dr.-Ing. Hartmut Helmke
Ostfalia
Hochschule für angewandte Wissenschaften
Fakultät für Informatik

Matrikelnummer:	*Punktzahl:*
Ergebnis:	

Freiversuch ☐ *F1* ☐ *F2* ☐ *F3* ☐

Klausur im WS 2021/22:

Die verschiedenen Programmierparadigmen von C++ — Lösungen

Informatik Bachelorstudiengang ☐ *Informatik Masterstudiengang* ☐

Hilfsmittel wie Bücher und Skripte und eigene Notizen sind erlaubt.
Die Nutzung eines Computers z.B. mit einer Programmierumgebung oder einem Compiler ist gar nicht erlaubt!
Austausch von Hilfsmitteln mit Kommilitonen ist **nicht** erlaubt !
Die Kommunikation mit anderen Personen (zur Klausur) ist während der Klausur nicht erlaubt.

Bitte notieren Sie auf **allen** Blättern, die in die Bewertung eingehen sollen, Ihren Namen und Ihre Matrikelnummer.
Auf eine absolut korrekte Anzahl der Blanks und Zeilenumbrüche braucht bei der Ausgabe nicht geachtet zu werden. Dafür werden keine Punkte abgezogen.

Hinweis: In den folgenden Programmfragmenten wird die globale Variable `datei` verwendet. Hierfür kann der Einfachheit halber die Variable `cout` angenommen werden. Die Variable `datei` diente bei der Klausurerstellung lediglich dazu, automatisch eine Lösungsdatei zu erstellen.
Wir befinden uns jeweils im Namensraum `std`, d.h., ein `using namespace std;` dürfen Sie in jeder Codedatei annehmen. Außerdem dürfen Sie annehmen, dass für alle Code-Fragmente die erforderlichen *include*-Anweisungen für C++-Header-Dateien erfolgt sind. Syntaxfehler sind somit allenfalls unabsichtlich in den Programmfragmenten enthalten.

Für viele Aufgaben ist ein Extrablatt zu verwenden; bitte mit Namen und Matrikelnummer beschriften.

Geplante Punktevergabe

Punktziel	Im einzelnen	Pkte
Übungen:	XXX	
A1: 34 P.		
A2: 46 P.		
Summe 80+ SP.		

Der folgende Code zeigt die Deklaration der Klasse `Student`:

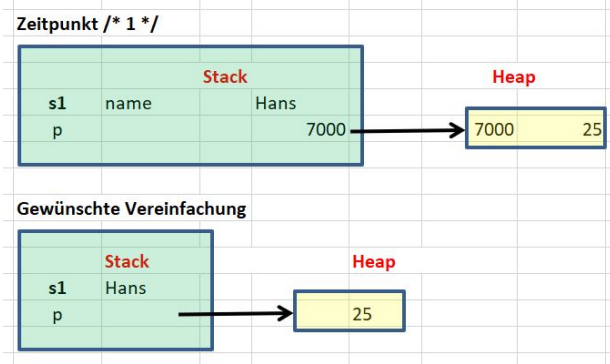
```
class Student {
public:
   Student(string n);
   Student(const Student& s2);
   Student(Student&& tmp);
   ~Student();
    string GetName() const { return name; }
   // functionaly for counting,
   //  how often <,==, etc. are called
   Student(string n, string n2);
    static string GetCounterValuesAsString();
    static void ResetCounters();
private:
    string name;
    friend bool operator<(
       const Student& s1, const Student& s2);
};
```

Die Implementierung der Deklarationen zeigt die Datei `Student.cxx`:

```
Student::Student(string n) :
   name(n) {
    datei << "+" << name << " ";
}
Student::~Student() {
    datei << "-" << name << " ";
}
Student::Student(const Student& s2) {
   name = s2.name;
    datei << "+Co " << name << " ";
}
Student::Student(Student&& s2) {
   name = s2.name;
    datei << "+Mo " << name << " ";
}
Student::Student(string n, string n2) {
   name = n + n2;
}
```

Die folgende Zeichnung skizziert eine vereinfachte Speicherbelegung zum Zeitpunkt /* 1 */ in der Funktion `simple`:

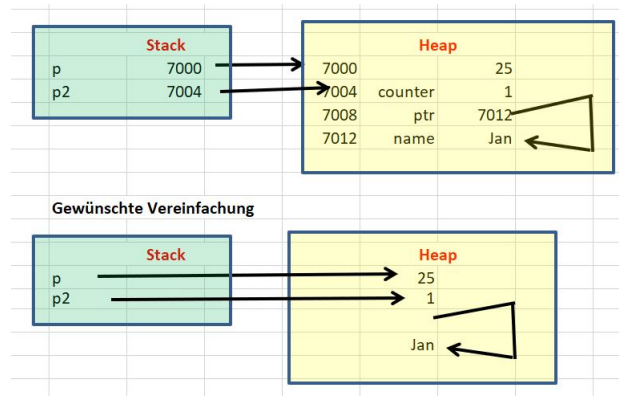
```
void simple() {
   Student s1("Hans");
    datei << s1.GetName();
   int* p = new int(25); /* 1 */
}
```



Sie dürfen und sollten diese Vereinfachung insbesondere für Instanzen von `string` verwenden. Die Variable `datei` und deren Argumente brauchen im Folgenden für die Speicherveranschaulichung auch nicht beachtet zu werden. Sie müssen auch nicht absolute Adressen angeben. Es reicht aus, wenn Sie *Pfeile* verwenden, um zu veranschaulichen, wohin ein Zeiger verweist. Links sollte immer der Stackspeicher sein und rechts der Heapspeicher.

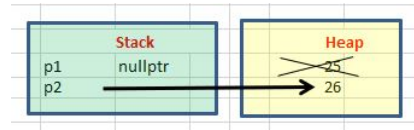
Die folgende Vereinfachung können Sie verwenden, um die Speicherbelegung von Instanzen der STL-Klasse `shared_ptr` zu veranschaulichen.

```
void simple2() {
   int* p = new int(25); /* 1 */
   shared_ptr<Student>  p2(new Student("Jan"));
}
```



Entsprechend verfahren Sie mit Instanzen der STL-Klasse `unique_ptr`.
Es folgt noch ein Beispiel zur Kennzeichnung von freigegebenem Speicher:

```
void simple3() {
   int* p1 = new int(25);
   int* p2 = new int(26);
   delete p1;
   p1 = nullptr;
}
```



Aufgabe 1 : Container und Komplexität

ca. 34 Punkte In dieser Aufgabe geht es überwiegend darum, herauszufinden, wie oft die Vergleichsoperatoren `operator==`, `operator<` bzw. `operator>` aufgerufen werden.
Dafür werden die folgenden Codefragmente verwendet.

```
bool operator>(
   const Student& s1, const Student& s2);
bool operator==(
   const Student& s1, const Student& s2);
```

```
// the following is needed as hash function
//   to insert Student into unordered_set etc.
template<>
struct hash<Student> {
    size_t operator()(const Student& s) const {
       return std::hash<std::string>().
          operator()(s.GetName());
    }
};

// unefficient hash function
struct hash1 {
    size_t operator()(const Student&) const {
       return 48774;
    }
};
```

Es folgen nun die Implementierungen der Funktionalität für die Vergleichsoperatoren etc.:

```
// global variables, only accessable in this file
  static int cntCmp = 0;
  static int cntEqu = 0;

  bool operator<(
     const Student& s1, const Student& s2) {
     ++cntCmp;
     return s1.name < s2.name;
  }
  bool operator>(
     const Student& s1, const Student& s2) {
     ++cntCmp;
     return s2 < s1; // calls also operator<
  }
  bool operator==(
     const Student& s1, const Student& s2) {
     ++cntEqu;
     return s1.GetName() == s2.GetName();
  }
```

```
/* Writing the values of the two counters
   into a stream and returning it as string */
string Student::GetCounterValuesAsString() {
    stringstream stre;
    stre << "cntCmp: " << cntCmp <<
       ", cntEqu: " << cntEqu;
    return stre.str();
}
void Student::ResetCounters() {
    cntCmp = 0;
    cntEqu = 0;
}
```

Das folgende Listing zeigt eine Schablonenfunktion zum Füllen eines Containers, die mit verschiedenen Containern aufgerufen werden kann. Die Klasse `Student` wurde bereits vorgestellt.

```
/* Filling cont in a loop from x to y
   with Tom0, Tom2, ... Tom100.
   Depending on cont type dupcliates
   are inserted or not. We return
   the value of the counters as a string. */
template <typename TCont>
string Fill (TCont& cont) {
    // Resetting the counters to 0
    Student::ResetCounters();
    /* Creating a lot of instances of Student with
       different and equal names; please ignore
       output to datei created in this loop.*/
    for (int i = 11; i < 1001; ++i) {
       cont.insert(Student("Tom",
          to_string(i % 101)));
    }
    datei << "\n" << cont.size() << " elems\n";
    return Student::GetCounterValuesAsString();
} // Fill
```

Es folgt eine Spezialisierung, sodass auch ein Aufruf mit einer Instanz einer Liste erfolgen kann.

```
/* Specialization, so that it is usable for lists
   same as before, but push_back called
   instead of insert */
template <>
string Fill ( list <Student>& cont) {
    Student::ResetCounters();
    // ignore output to datei of this loop
    for (int i = 11; i < 1001; ++i) {
       cont.push_back(Student("Tom",
          to_string(i % 101)));
    }
    return Student::GetCounterValuesAsString();
}// Fill
```

Nun füllen wir unterschiedliche Container jeweils mit Elementen Tom0 , Tom2 bis ... Tom100 Dabei werden in zwei verschiedenen Maps die Werte der Zähler und die Größe der Container festgehalten:

```
/* We fill different containers and
   store the value of the counters and print
   them afterwards. Then we try to find
   in the containers with global algorithm
   and with special methods and print at the
   end the counter values.
*/
void ContainerTemp() {
    map<string, string> cntMapFill;
    map<string, size_t> mSi;

    set<Student, less<Student>> cont1;
    cntMapFill["1set<Student, less<Student>>"] =
       Fill(cont1);
    mSi["1set<Student, less<Student>>"] =
       cont1.size();
```

```
set<Student, greater<Student>> cont2;
cntMapFill["2set<Student, greater<Student>>"] =
    Fill(cont2);
mSi["2set<Student, greater<Student>>"] =
   cont2.size();

multiset<Student, less<Student>> cont3;
cntMapFill["3multiset<Student,less<Student>>"]=
    Fill(cont3);
mSi["3multiset<Student, less<Student>>"] =
   cont3.size();

unordered_set<Student, hash<Student>> cont4;
cntMapFill["4unordered_set, efficient"] =
    Fill(cont4);
mSi["4unordered_set, efficient"] =
   cont4.size();

unordered_multiset<Student, hash<Student>> cont5;
cntMapFill["5unordered_multiset, efficient"] =
    Fill(cont5);
mSi["5unordered_multiset, efficient"] =
   cont5.size();

unordered_set<Student, hash1> cont6;
cntMapFill["6unordered_set, not efficient"] =
    Fill(cont6);
mSi["6unordered_set, not efficient"] =
   cont6.size();

list<Student> cont7;
cntMapFill["7list<Student>"] = Fill(cont7);
mSi["7list<Student>"] = cont7.size();
```

a.) (5 P.)
Welche Größe haben die Container, d.h. zu welcher Ausgabe führt der folgende Code?

```
datei << "\nNow the output of the sizes\n";
for (const auto& it : mSi) { /* 1 */
   datei << it.first << ": " << it.second << "\n";
}
```

Tragen Sie am besten Ihre Werte direkt in das folgende Listing ein:

```
1set<Student, less<Student>>:

2set<Student, greater<Student>>:

3 multiset<Student, less<Student>>:

4 unordered_set, efficient :

5 unordered_multiset, efficient :

6 unordered_set, not efficient :

7 list<Student>:
```

Lösung:

```
1set<Student, less<Student>>: 101
2set<Student, greater<Student>>: 101
3 multiset<Student, less<Student>>: 990
4 unordered_set, efficient : 101
5 unordered_multiset, efficient : 990
6 unordered_set, not efficient : 101
7 list<Student>: 990
```

b.) (3 P.)
Begründen Sie Ihre Entscheidungen, d.h. warum enthält z.B. **nicht** jeder Container 101 Elemente?
——— Bitte Extrablatt verwenden.———

Lösung:
Die Schleife wird insgesamt 990 mal durchlaufen. Allerdings werden nur 101 verschiedene Instanzen von `Student` erzeugt. `set` und `unordered_set` enthalten keine Dubletten. Daher sind hier jeweils nur 101 Elemente enthalten.

c.) (7 P.) Begründen Sie für das folgende Codefragment, wie oft für welchen Container bei der Erzeugung die einzelnen Vergleichsoperatoren aufgerufen werden.

```
datei << "\nNow the output of the counters\n";
for (const auto& it : cntMapFill) { /* 1 */
   datei << it.first << ":\n "
      << it.second << "\n";
}
```

Es führt zu der Ausgabe:

```
Now the output of the counters
1set<Student, less<Student>>:
  cntCmp: 11967, cntEqu: 0
2set<Student, greater<Student>>:
  cntCmp: 21576, cntEqu: 0
3 multiset<Student, less<Student>>:
  cntCmp: 11944, cntEqu: 0
4 unordered_set, efficient :
  cntCmp: 0, cntEqu: 901
5 unordered_multiset, efficient :
  cntCmp: 0, cntEqu: 1311
6 unordered_set, not efficient :
  cntCmp: 0, cntEqu: 51695
7 list<Student>:
  cntCmp: 0, cntEqu: 0
```

Die exakten Zahlen können Sie sicherlich nicht begründen, aber die relativen Größen. Argumentieren Sie mit der Containergröße sowie z.B. mit O(1), O(log N), O(N), O(N^2) und warum `cntCmp` und/oder `cntEqu` verwendet werden.
——— Bitte Extrablatt verwenden.———

Lösung:
Bei den Containern `set` und `multiset` erfolgt die Suche durch den Vergleichsoperator operator<. Daher wird jeweils nur `cntCmp` erhöht. Sie sind jeweils als binäre Bäume implementiert.
In `1set` wird insgesamt 990 mal versucht, ein Element

einzufügen. Insgesamt enthält der Container nur 101 Elemente, denn Dubletten sind nicht möglich. Der Baum hätte im Idealfall überall eine maximale Tiefe von höchstens 7 ($log_2 101 \approx 7$). In dem Fall dürften wir ca. $7*990$ Vergleiche erwarten Der Tiefe des Baums wird aber nicht überall exakt ausgeglichen. Daher sind es etwas mehr. Die Komplexität für das Einfügen von N Elemente in einen Baum der Größe M ist noch $O(N * logM)$.

`2set`: Hier wird jeweils 2mal der Zähler erhöht, da operator> jeweils operator< aufruft. Die Komplexität für das Einfügen von N Elemente in einen Baum der Größe M ist 2 * $O(N * logM)$.

In `3multiset` wird insgesamt 990 mal versucht, ein Element einzufügen. Insgesamt enthält der Container im Durchschnitt $990/2 = 495$ Elemente. Der Baum hätte im Idealfall überall eine maximale Tiefe von höchstens 10 ($log_2 990 \approx 10$) und eine durchschnittliche Tiefe von 9. In dem Fall dürften wir ca. $9*990$ Vergleiche erwarten Das kommt gut mit der Messung hin. Die Komplexität für das Einfügen von N Elemente in einen Baum der Größe M ist $O(N * logM)$.

Bei den assoziativen Containern (Hashes) `unordered_set` und `unordered_multiset` erfolgt die Suche durch den Gleichheitsoperator operator==. Daher wird nur jeweils `cntEqu` erhöht.

`4unordered_set`: Es wird versucht, 990 Elemente einzufügen. 101 werden eingefügt. Die Anzahl der Aufrufe ist mit 901 aber noch geringer als 990. Wie kann das sein? Der Vergleich erfolgt ja nur bei einer Kollision, d.h. sehr häufig ist überhaupt kein Element an der mit der hash-Funktion berechneten Stelle gefunden worden, sodass hier gar kein Vergleich mit operator== erfolgen muss. Manchmal gibt es aber auch mehr als eine Kollision. Die Komplexität für das Einfügen von N Elemente in einen assoziativen Container der Größe M ist $O(N) * O(1)$.

`5unordered_multiset`: Es wird versucht, 990 Elemente einzufügen. Alle werden eingefügt. Der Container ist voller als `4unordered_set`. Daher gibt es mehr Kollisionen und deshalb ist die Anzahl der Aufrufe von operator== hier etwas höher. Die Komplexität für das Einfügen von N Elemente in einen assoziativen Container der Größe M ist $O(N) * O(1)$.

`6unordered_set, not efficient`: Es wird versucht, 990 Elemente einzufügen. 101 werden eingefügt. Im Durchschnitt enthält der Container ca. 90 Elemente. Jeder Zugriff wird zunächst mit der dummy hash-Funktion `hash1` auf den gleichen hash-Wert abgebildet. Man könnte also sogar $90 * 990 \approx 90.000$ Zugriffe erwarten. Es sind aber deutlich weniger. Das Kollisionsmanagement der STL-Imlementierung des Hashes ist somit effizienter, d.h. nach ca. 50 Versuchen wird für jedes Element ein freier Platz gefunden.

`7list`: Es wird immer am Ende eingefügt. Die Einfüge-Position muss nicht erst aufwändig bestimmt werden. Daher sind beide Zähler gleich 0.

d.) (3 P.)

Schätzen Sie grob ab (mit Begründung), wie oft im folgenden Code-Fragment bei Nutzung des STL- Algorithmus `find` die einzelnen Vergleichsoperatoren aufgerufen werden.

```
cntMapAlgo["1set<Student, less<Student>>"] =
   FindWithAlgo(cont1);
cntMapAlgo["2set<Student, greater<Student>>"]=
   FindWithAlgo(cont2);
cntMapAlgo["3multiset<Student,less<Student>>"]=
   FindWithAlgo(cont3);
cntMapAlgo["4unordered_set, efficient"] =
   FindWithAlgo(cont4);
cntMapAlgo["5unordered_multiset, efficient"]=
   FindWithAlgo(cont5);
cntMapAlgo["6unordered_set, not efficient"] =
   FindWithAlgo(cont6);
cntMapAlgo["7list<Student>"] =
   FindWithAlgo(cont7);
```

```
datei << "\nCounters from algorithm find\n";
for (const auto& it : cntMapAlgo) { /* 2 */
   datei << it.first << ":\n "
      << it.second << "\n";
}
```

Die Implementierung von `FindWithAlgo` ist:

```
/* trying to find each element Tom1, Tom2, ... Tom100
     in cont serveral times.
   We return the value of the counters as a string. */
template <typename TCont>
string FindWithAlgo(TCont& cont) {
   // Reset counters for next call
   Student::ResetCounters();
   int misCnt = 0;
   for (int i = 11; i < 1001; ++i) {
      Student hlp("Tom", to_string(i % 101));
      if (find(cont.begin(),
            cont.end(),hlp) == cont.end()) {
         ++misCnt;
      }
   }
   return "missing " + to_string(misCnt) + " "
      + Student::GetCounterValuesAsString();
}// FindWithAlgo
```

Ihre Aufgabe ist somit den folgenden Text (direkt hier auf dem Aufgabenblatt) zu ergänzen und anschließend in der nächsten Teilaufgabe auf einem Extra-Blatt Ihre Schätzungen zu begründen. Als Hilfe sind die Zähler für die erste Ausgabe schon genannt. **Achtung:** Achten Sie darauf, ob `cntCmp` und / oder `cntEqu` erhöht werden.

```
Counters from algorithm find
1set<Student, less<Student>>:
  missing 0 cntCmp: 0, cntEqu: 50300

2set<Student, greater<Student>>:
  missing 0 cntCmp:    cntEqu:

3multiset<Student,less<Student>>:
  missing 0 cntCmp:       cntEqu:
```

```
4unordered_set, efficient:
  missing 0 cntCmp:     cntEqu:

5unordered_multiset, efficient:
  missing 0 cntCmp:   cntEqu:

6unordered_set, not efficient:
  missing 0 cntCmp:      cntEqu:

7list<Student>:
  missing 0 cntCmp:      cntEqu:
```

e.) (5 P.) Die exakten Zahlen können Sie sicherlich nicht abschätzen, aber die Größenordnungen. Argumentieren Sie mit der Containergröße sowie mit O(1), O(log N), O(N), O(N^2) etc.

——— Bitte Extrablatt verwenden.———————

Lösung:

```
Counters from algorithm find
1set<Student, less<Student>>:
  missing 0 cntCmp: 0, cntEqu: 50300
2set<Student, greater<Student>>:
  missing 0 cntCmp: 0, cntEqu: 50680
3 multiset <Student, less <Student>>:
  missing 0 cntCmp: 0, cntEqu: 486180
4 unordered_set , efficient :
  missing 0 cntCmp: 0, cntEqu: 49680
5 unordered_multiset , efficient :
  missing 0 cntCmp: 0, cntEqu: 486180
6 unordered_set , not efficient :
  missing 0 cntCmp: 0, cntEqu: 51300
7 list <Student>:
  missing 0 cntCmp: 0, cntEqu: 49680
```

Es wird jeweils mit dem gleichen Algorithmus `find` der Container von Anfang bis Ende durchsucht. Sobald das Element gefunden ist, wird die Suche abgebrochen. Dieses ist immer der Fall (missing hat immer den Wert 0). Zum Vergleich wird immer der operator== verwendet. Daher ist `cntCmp` auch immer 0.

Die Container enthalten jeweils 101 verschiedene Elemente, d.h. nach ca. 50 Aufrufen wird im Durchschnitt das Element gefunden. Die Anzahl der Aufrufe von operator== beträgt hier also etwa: Größe des Containers mal Anzahl der Vergleiche (ca. 50) bis zum Finden.

Bei `1set` und `2set` ergibt sich die gleiche Größenordnung, da zum Suchen der operator== und nicht der operator> verwendet wird. operator== ist für beide gleich.

f.) (5 P.)

Es folgt eine Schablonenfunktion zur Suche im Container mit den Container-spezifischen Methoden `find`:

```
/* trying to find each element Tom1, Tom2,
   .. Tom100 in cont serveral times. This
   time we use the special method of cont.
   We return the value of the counters as a string. */
template <typename TCont>
string FindWithMethod(TCont& cont) {
   // Reset counters for next call
   Student::ResetCounters();
   int misCnt = 0;
   for (int i = 11; i < 1001; ++i) {
      Student hlp("Tom", to_string(i % 101));
      if (cont.find(hlp) == cont.end()) {
         ++misCnt;
      }
   }
   return "missing " + to_string(misCnt) + " "
      + Student::GetCounterValuesAsString();
}// FindWithMethod
```

Außerdem die Spezilisierung für den Container `list`, der keine eigene Methode `find` besitzt.

```
/* Specialization for list , which does
      not implement method find*/
template <>
string FindWithMethod(list<Student>& cont) {
   return FindWithAlgo(cont);
}// FindWithMethod
```

Ermitteln Sie, wie oft für welchen Container beim Suchen mit der Container-spezifischen Methode `find` die einzelnen Vergleichsoperatoren aufgerufen werden. Als Hilfe dient Ihnen das folgende Codefragment:

```
   cntMapMeth["1set<Student, less<Student>>"] =
      FindWithMethod(cont1);
   cntMapMeth["2set<Student, greater<Student>>"]=
      FindWithMethod(cont2);
   cntMapMeth["3multiset<Student,less<Student>>"]=
      FindWithMethod(cont3);
   cntMapMeth["4unordered_set, efficient"] =
      FindWithMethod(cont4);
   cntMapMeth["5unordered_multiset, efficient"]=
      FindWithMethod(cont5);
   cntMapMeth["6unordered_set, not efficient"]=
      FindWithMethod(cont6);
   cntMapMeth["7list<Student>"] =
      FindWithMethod(cont7);
```

```
   datei << "\nCounters from method find\n";
   for (const auto& it : cntMapMeth) { /* 3 */
      datei << it.first << ":\n "
         << it.second << "\n";
   }
```

Ein Teil der Ausgabe ist als Hilfe hier schon angegeben:

```
Counters from method find
1set<Student, less<Student>>:
  missing 0 cntCmp: 11632, cntEqu: 0
```

Bei den folgenden Ausgaben überlegen und begründen Sie (im folgenden Aufgabenteil) bitte selber die Wer-

te.
Achtung: Achten Sie darauf, ob `cntCmp` und / oder `cntEqu` erhöht werden.

Die exakten Zahlen können Sie sicherlich nicht immer angeben, aber zumindest die Größenordnungen.

Schätzungen für die Werte können Sie hier notieren.

```
Counters from method find
1set<Student, less<Student>>:
  missing 0 cntCmp: see above, cntEqu: 0
2set<Student, greater<Student>>:
  missing 0 cntCmp:      cntEqu:

3multiset<Student, less<Student>>:
  missing 0 cntCmp:      cntEqu:

4unordered_set, efficient:
  missing 0 cntCmp:     cntEqu:

5unordered_multiset, efficient:
  missing 0 cntCmp:    cntEqu:

6unordered_set, not efficient:
  missing 0 cntCmp:      cntEqu:

7list<Student>:
  missing 0 cntCmp:     cntEqu:
```

g.) (6 P.) Begründen Sie Ihre Schätzungen. Argumentieren Sie mit der Containergröße sowie mit O(1), O(log N), O(N), O(N^2) etc.

——— Bitte Extrablatt verwenden.———

Lösung:

```
Counters from method find
1set<Student, less<Student>>:
  missing 0 cntCmp: 11632, cntEqu: 0
2set<Student, greater<Student>>:
  missing 0 cntCmp: 21772, cntEqu: 0
3 multiset <Student, less <Student>>:
  missing 0 cntCmp: 19319, cntEqu: 0
4 unordered_set , efficient :
  missing 0 cntCmp: 0, cntEqu: 990
5 unordered_multiset , efficient :
  missing 0 cntCmp: 0, cntEqu: 990
6 unordered_set , not efficient :
  missing 0 cntCmp: 0, cntEqu: 49680
7 list <Student>:
  missing 0 cntCmp: 0, cntEqu: 49680
```

Die Suche erfolgt nun mit für jeden Container speziellen Methoden. Bei den binären Bäumen ist es die Anzahl der Vergleichsaufrufe. Die Komplexität für das Finden von N Elemente in einen Baum der Größe M ist daher $O(N * logM)$. N beträgt immer 990. M beträgt für das `set` 101 und für das `multiset` 990, sodass wir von ca. 7*990 bzw. 10*990 Vergleichsoperationen ausgehen können. Es wurde bereits zuvor bei der Baumerzeugung begründet, warum die wahre Anzahl etwas größer ist, weil die Bäume nicht exakt ausgeglichen sind. Die doppelte Anzahlt für src2set wurde dort auch schon begründet.
Überraschend mag zunächst sein, dass es für die assoziativen Container mit der effizienten Hashfunktion `hash` exakt 990 Aufrufe sind und es offensichtlich zu keiner Kollisionen kommt. Jedes der 101 Elemente wird somit auf einen anderen hash-Wert abgebildet, in der Tat erscheint dieses ungewöhnlich. Es gibt natürlich viele gleiche Werte in diesen Containern. Es reicht aber aus, einen davon zu finden.
Für die nicht effiziente Hash-Funktion `hash1` findet man an der berechneten Stelle fast nie das gesuchte Element. Man muss dann linear weitersuchen und findet im Durchschnitt nach ca. 50 Aufrufen das gesuchte Element, da es 101 verschiedene Elemente gibt.
Bei der Liste gibt es exakt die gleiche Anzahl wie bei Verwendung des Algorithmus; es ist der gleiche Code, der hier durchlaufen wird.
Die Übereinstimmung der Anzahl bei `7list` und `6unordered_set` muss nicht so sein. Würden die Zahlen zufällig aus dem Intervall 0 bis 100 in die Container eingefügt, ergäben sich z.B. sehr wahrscheinlich unterschiedliche Werte.

Aufgabe 2 : Objekterzeugung und -zerstörung

ca. 46 Punkte

a.) (3 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f1` ?

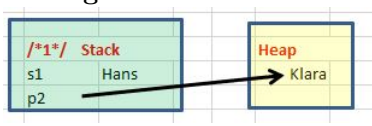
```
void f1() {
   Student s1("Hans");
   Student* ps2 = new Student("Klara");/*2*/
}
```

Lösung:

```
+Hans +Klara -Hans
```

b.) (3 P.)
Skizzieren Sie auf einem Extra-Blatt entsprechend der Beispiel-Zeichnungen auf Seite 2 die Speicherbelegungen zum Zeitpunkt /*2*/.

Lösung:



c.) (9 P.) Gegeben ist die Funktionen `f2`:

```
void f2() {
   unique_ptr<Student> p1(new Student("Tom"));
   datei << "\nBefore p2" << endl; /* 1 */
   unique_ptr<Student> p2 = nullptr;
   // Student(n) calls normal constructor
   // and creates temporary object
   p2 = make_unique<Student>(Student("Paula"));
   datei << "\nBefore p1=p2" << endl; /*2*/
   p1 = ::move(p2); // p1 responsible now
   datei << "\nEnde\n" << endl; /* 3 */
}
```

Welche Ausgaben erfolgen nach `datei`? Bitte gleich hier auf dem Aufgabenblatt notieren. Einige Ausgaben sind hier schon angegeben. Falls keine Ausgabe zwischen den vorhandenen Ausgaben erfolgt, kennzeichnen Sie dies mit mit `keine Ausgabe`. Es bietet sich vermutlich an, die folgenden Aufgabe zur Veranschaulichung der Speicherbelegung parallel zu bearbeiten.

```
Output of f2:


Before p2


Before p1=p2


Ende
```

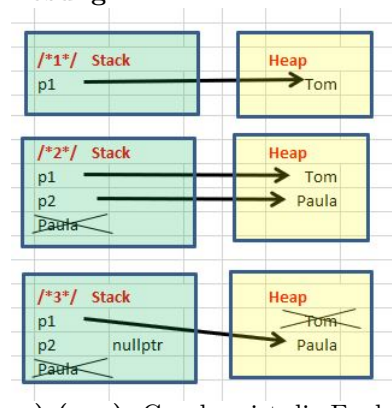
Lösung:

```
+Tom
Before p2
+Paula +Mo Paula -Paula
Before p1=p2
-Tom
Ende

-Paula
```

d.) (13 P.)
Skizzieren Sie auf einem Extra-Blatt die Speicherbelegungen zum Zeitpunkt /*1*/, /*2*/ und /*3*/ in drei separaten Zeichnungen (siehe Beispiel-Zeichnungen auf Seite 2 als Referenz).

——— Bitte Extrablatt verwenden.———

Lösung:



e.) (8 P.) Gegeben ist die Funktion `f3`:

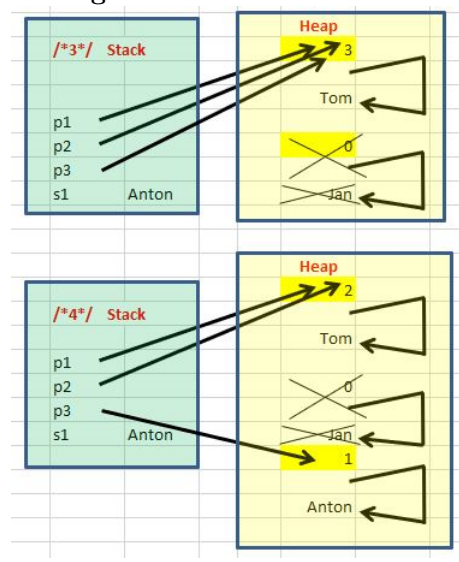
```
void f3() {
   shared_ptr<Student> p1(new Student("Tom"));
   datei << "\nbefore p2\n";
   shared_ptr<Student>p2(
      make_shared<Student>(Student("Jan"))); /*1*/
   datei << "\nbefore p3=p1\n";
   shared_ptr<Student> p3 = p1;
   datei << "\nBefore p2=p1\n"; /*2*/
   p2 = p1;
   Student s1("Anton");
   datei << "\nAfter s1(Anton) " << endl; /*3*/
   p3 = make_shared<Student>(s1);
   datei << "\nEnde" << endl; /*4*/
}
```

Welche Ausgaben erfolgen nach `datei`? Sie dürfen gleich hier auf dem Aufgabenblatt notieren. Einige Ausgaben sind hier schon angegeben. Falls keine Ausgabe zwischen den vorhandenen Ausgaben erfolgt, kennzeichnen Sie dies mit `keine Ausgabe`. Es bietet sich vermutlich an, die folgenden Aufgabe zur Veranschaulichung der Speicherbelegung parallel zu bearbeiten.

```
Output of f3:

before p2


before p3=p1
```

```
Before p2=p1


After s...


Ende
```

Lösung:



Lösung:

```
+Tom
 before  p2
+Jan  +Mo  Jan  -Jan
 before  p3=p1

Before  p2=p1
-Jan  +Anton
 After  s1(Anton)
+Co  Anton
Ende
-Anton  -Anton  -Tom
```

f.) (10 P.) Skizzieren Sie auf einem Extra-Blatt die Speicherbelegungen zum Zeitpunkt /*1*/, /*2*/, /*3*/ und /*4*/ in vier separaten Zeichnungen (siehe Beispiel-Zeichnungen auf Seite 2 als Referenz).
——— Bitte Extrablatt verwenden.———

Lösung: