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 2022/23:

Die verschiedenen Programmierparadigmen von C++ — Lösungen

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 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 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	
Name und Matrikelnummer eingetragen 1	XXX	
A1: 30 P.		
A2: 49 P.		
Summe 80+ SP.		

Aufgabe 1 : Container und Komplexität

ca. 30 Punkte

Kurzversion dieser Aufgaben: Erklären Sie die Ergebnisse der Laufzeitmessungen aus der Tabelle auf der folgenden Seite 3.

Langversion:
Der folgende Code führt Laufzeitmessungen für sechs verschiedene Container durch. Zunächst ist die Erzeugung der Container gezeigt. Für jeden der Container wird eine Schablonen-Funktion aufgerufen.

```
// Creating a vector with vecSize random integers
// from the intervall [0.. vecSize/2[.
// Filling 6 different containers from the vector
// and making different runtime measurement
// for each of the 6 containers.
void ContainerExperiment1() {
   const int vecSize = 10000;
   vector<int> vec(vecSize);
   srand((unsigned)0); // init random generation
   FillVectorWithRandom(vec, vecSize, vecSize / 2);

   set<int> setCont;
   TestWithCont1(setCont, vec);
   multiset<int> msetCont;
   TestWithCont1(msetCont, vec);
   unordered_set<int> hashCont;
   TestWithCont1(hashCont, vec);
   unordered_multiset<int> mulhashCont;
   TestWithCont1(mulhashCont, vec);

   vector<int> vCont;
   TestWithContLin1(vCont, vec); // call func 2
   list<int> lCont;
   TestWithContLin1(lCont, vec); // call func 2
}
```

Implementierung des allgemeinen Templates:

```
// make container setCont empty and fill it
// with all elements from v by insert method
// and measure the time in milliseconds this
// needs by macro MT.
// then measure time to find all elements
// from v in new filled container cont
template <typename Cont>
void TestWithCont1(Cont cont,
   const vector<int>& v) {

   MT({ cont.clear();
   for (auto iter : v) {
      cont.insert(iter);
   }) }
   cout <<"container size "<< cont.size()<< "\n";

   // find all with algorithm
   MT({for (auto iter : v) {
      if (find(cont.begin(),
         cont.end(), iter) == cont.end()) {
         cerr << "found, error\n";
      } })}

   // find all not existing with algorithm
   MT({ for (auto iter : v) {
      if (find(cont.begin(),
         cont.end(), -iter - 1) != cont.end()) {
         cerr << "found, error\n";
      } }) }

   // find all with method
   MT({for (auto iter : v) {
      if (cont.find(iter) == cont.end()) {
         cerr << "not found, error\n";
      } })}

   // find all not existing with method
   MT({for (auto iter : v) {
      if (cont.find(-iter - 1) != cont.end()) {
         cerr << "not found, error\n";
      } }) }
}
```

Es wird zunächst der übergebene Container mit der Methode `insert` gefüllt. Die benötige Zeit hierfür wird ausgegeben. Der Algorithmus `find` sucht anschließend jedes Element. Danach wird mit find das negative jedes Elements gesucht. Dieses ist **nicht** vorhanden. Es dient somit zur Laufzeitmessung für nicht vorhandenen Elemente. Anschließend erfolgen noch zwei Laufzeitmessungen mit der Methode find des Containers. Die Bedingungen der `if`-Anweisungen sind nie erfüllt. Sie wurden eingefügt, um mögliche Optimierungen des Compilers zu unterbinden, da andernfalls der Code in der Schleife ggf. gar nicht ausgeführt worden wäre.
Für Instanzen von `list` und `vector` ist der Code nicht ausführbar. Deshalb ist hierfür eine andere Schablonen-Funktion implementiert:

```
template <typename Cont>
void TestWithContLin1(Cont cont,
   const vector<int>& v) {

   MT({ cont.clear();
   for (auto iter : v) {
      cont.push_back(iter);
   } } )
   cout <<"container size "<< cont.size()<< "\n";

   // find all with algo
   MT({for (auto iter : v) {
      if (find(cont.begin(),
           cont.end(), iter) == cont.end()) {
         cerr << "found, error\n";
      }})}

   // find all not there with algo
   MT({for (auto iter : v) {
      if (find(cont.begin(),
           cont.end(), -iter - 1) != cont.end()) {
         cerr << "found, error\n";
      }})}
}
```

Der Vollständigkeit halber sind am Ende dieser Aufgabe noch der Code des Makros MT sowie zur Füllung der Container mit Zufallszahlen angegeben. Dieser Code wird zum Lösen der Aufgaben nicht benötigt.

Die wiederholt durchgeführten Laufzeitmessungen ergaben folgende durchschnittliche Messwerte:

in ms	Insert	Find Algo		Find Method		Size
set	1,27	122	175	0,72	0,15	4343
multiset	1,82	454	957	0,86	0,11	10000
hash	0,58	60	129	0,10	0,08	4343
multihash	0,98	203	400	0,066	0,070	10000
vector	0,03	7	24			10000
list	0,69	61	232			10000

Alle Angaben sind in Millisekunden. Die Spalte `Insert` enthält die Zeit zum Befüllen des gesamten Containers mit den Pseudo-Zufallszahlen. Die beiden Spalten unter `Find-Algo` zeigen zuerst die Laufzeiten für die Suche für Elemente, die im Container vorhanden sind und anschließend für die gleiche Anzahl von Elementen, die im Container **nicht** vorhanden sind. Entsprechendes gilt für die beiden Spalten unter `Find-Method` für die Nutzung der Container-spezifischen Methoden `find`. Die Spalte `Size` gibt die Anzahl der Elemente im jeweiligen Container an.

a.) (5 P.)
Erklären Sie die unterschiedlichen Größen der Container, d.h. die verschiedenen Werte in der Spalte `Size`.

Lösung:
`set` und `unordered_set` sind sortierte Container und speichern im Unterschied zu `multiset` und `unordered_multiset` keine Dubletten. Wird `insert` für ein bereits vorhandenes Element aufgerufen, wird das vorhandenen einfach überschrieben, während es bei den `multi*` nochmals sortiert eingetragen wird. Da 10.000 Elemente erzeugt werden, es aber nur 5.000 verschiedene Zufallszahlen gibt, können maximal 5.000 verschiedene Zufallszahlen erzeugt werden. Es wird aber nicht jede Zufallszahl gleich häufig erzeugt. Daher werden einige Zufallszahlen sogar dreimal, viermal etc. erzeugt, sodass sich nur 4.343 verschiedene Zahlen ergeben. Instanzen von `vector` und `list` sind nicht sortiert. Es wird beim Erzeugen lediglich immer am Ende neu eingefügt, sodass auch hier 10.000 Elemente vorliegen. Alles hier genutzten Container, außer `vector`, nutzen zur Verwaltung der Elemente Heap-Speicher. Heap anzufordern und zu verwalten, erfordert zusätzliche Laufzeit.

b.) (3 P.)
Erklären Sie, warum für `vector` und `list` der Aufruf der Funktion `TestWithCont1` zu einem Compilerfehler führen würde und daher die *Spezialisierung* durch `TestWithContLin1` erforderlich ist.

Lösung:
Die Container `vector` und `list` verfügen weder über die Methode `insert` noch über die Methode `find`.

c.) (5 P.)
Erklären Sie, warum es in der Spalte `Find Algo` signifikante Laufzeit-Unterschiede zwischen der Suche im Container X und dem entsprechenden multi-X Container gibt, d.h., die Unterschiede zwischen 122 und 454, 60 und 203 bzw. 175 und 957 sowie 129 und 400 Millisekunden sollen erklärt werden. Auf die *Unterschiede in den Unterschieden* muss nicht eingegangen werden.

Lösung:
Es handelt sich um eine lineare Suche. Die multi-X Container sind um den Faktor 2,3 größer. Dieses erklärt noch nicht vollständig die Erhöhung, die sich zwischen den Faktoren 3,1 und 5,5 bewegt.

d.) (7 P.)
Erklären Sie, die unterschiedliche Einträge in der Spalte `Insert` vector ist am schnellsten, dann hasht/list, dann multihash (`unordered_multiset`), dann set und zum Schluss multiset. Verwenden Sie, wenn immer möglich, die O(x)-Schreibweise zur Argumentation.

Lösung:
Der Aufwand zum Einfügen in einen vector ist am geringsten. Der Aufwand ist konstant O(1). Die Einfügeposition, d.h. die Speicherstelle, ist bekannt. Auch beim hash ist der Aufwand im Mittel konstant O(1). Allerdings kommt es hier aufgrund der Dubletten zu Konflikten. Die Gleichheit muss hier zunächst ermittelt werden und dann erfolgt die Konfliktlösung. Der multi-hash enthält ca. 2.3 mal so viele Elemente und ist damit langsamer. Der Aufwand am Ende oder auch am Anfang einer Liste ist O(1). Die Endposition muss bei der doppelt verketteten Liste nicht erst gesucht werden.
Der Aufwand zum Einfügen an der richtigen Position in einem Baum (bei set und multiset) ist logarithmisch von der Größe N abhängig, d.h. $O(log_2 N)$. Im Mittel ist N bei unserem set also $log_2\ 4343/2 = log_2\ 2170 = 11$ und beim multiset $log_2\ 10000/2 = log_2\ 5000 = 12$, was den um 50% höheren Aufwand von multiset zu set nicht ganz erklärt. Der Verwaltungsaufwand beim multiset ist also größer.

e.) (5 P.)
Erklären Sie, warum die Nutzung der find-Methoden wesentlich schneller abläuft als mit der Verwendung des globalen find-Algorithmus (O(N)-Argumentation).

Lösung:
Beim find-Algorithmus haben wir linearen Aufwand, also ca. $\frac{10.000}{2}$ bzw. $\frac{4,343}{2}$. Bei der find-Methode für set und multiset ist der Aufwand logarithmisch also ca. $log_2\ \frac{10.000}{2} = 12$ bzw. $log_2 \frac{4,343}{2} = 11$, was einen Faktor von 20-50 ergibt. Eine weitere Zeitersparnis bringt die wesentlich einfachere Weiterschaltung der Iteratoren.
Beim hash, also bei `unordered_set` und bei `unordered_multiset` haben wir sogar nur konstantem Aufwand.

f.) (5 P.)
Erklären Sie, warum es mit dem **Algorithmus** `find` länger dauert, ein Element zu *finden*, was nicht im Container vorhanden ist, gegenüber dem Finden eines im Container vorhandenen Elements, d.h. erklären Sie die Unterschiede vom ersten zum zweiten Messwert in der Spalte `Find Algo`.
Lösung:
Es findet immer eine lineare Suche mit dem Algorithmus `find` statt. Ist das gesuchte Element vorhanden, wird es im Durchschnitt nach ca. $\frac{10.000}{2}$ bzw. $\frac{4,343}{2}$ Vergleichen gefunden. Ist es nicht vorhanden, müssen immer alle Elemente geprüft werden, was immer 10,000 bzw. 4,343 Vergleichen bedarf. Dieses erklärt in etwa die doppelte Laufzeit, aber nicht warum die Laufzeit manchmal fast viermal so groß ist.

Anhang STL

Der Makro MT startet eine Laufzeitmessung. Er führt alle Anweisungen, die im Argument anw übergeben wurden, aus, stoppt die Laufzeitmessung und gibt die Differenz in Millisekunden auf dem Bildschirm aus.

```
#define MT(anw)\
   {\
   C_CpuStopWatch clock_0; clock_0.start();\
   {anw}\
   clock_0.stop();\
   cout << clock_0.getMilliSec()  \
        << " ms" << std::endl;\
   }
```

Das Füllen der Instanz von `vector` mit Pseudo-Zufallszahlen erfolgt mit `FillVectorWithRandom` und der Hilfsfunktion `GetRandomInt`:

```
// Get a pseudo-random integer from
//  intervall  [0.. a_max[.
int GetRandomInt(int a_max) {
   int hlp = rand();
   double x = hlp / static_cast<double>(RAND_MAX)
      * static_cast<double>(a_max);
   return static_cast<int>(x);
}

// Empty vector a_vec and fill with a_cnt
// pseudo-random-numbers from the intervall
//  [0.. a_max[
void FillVectorWithRandom(
   vector<int>& a_vec, int a_cnt, int a_max) {
   a_vec.clear();
   for (int i = 0; i < a_cnt; ++i) {
      a_vec.push_back(GetRandomInt(a_max));
   }
}
```

Aufgabe 2 : Objekterzeugung und -zerstörung

ca. 49 Punkte
Die Klasse `Student` hat folgende Schnittstelle:

```
class Student {
public:
   Student(const char* n);
   // erzeugt cnt Noten von 1 bis 6.
   Student(const char* n, int cnt);
   ~Student();

   Student(const Student& s2);
   Student(Student&& s2); // Move-Constructor
   Student& operator=(const Student& s2);
   // Move-Zuweisungoperator
   void operator=(Student&& s2);
private:
   // Name am Ende '\0' zur Endekennzeichnung
   char* p_name;
   double* p_noten;
   int anz;   // Anzahl der Noten
    // anz / p_noten ohne Einfluss auf datei-Ausgabe
   void StupidShallowCopy(const Student& s2);
   void StupidShallowCopy(Student& s2);
   void MoveCopy(Student&& s2); // Hilfsfunktion
};
```

Die Implementierung der beiden Konstruktoren und des Destruktors ist:

```
Student::Student(const char* n) {
   datei << " +S " << PR(n);
   SaveCopyOnHeap(p_name, n);
   p_noten = nullptr;
   anz = 0;
}
Student::~Student() {
   datei << " -S " << PR(p_name);
#ifdef DEEP_COPY_ALREADY_IMPLEMENTED
   delete[] p_name;
   delete[] p_noten;
#endif
}

Student::Student(const char* n, int cnt) {
   datei << " +S " << PR(n);
   SaveCopyOnHeap(p_name, n);
   p_noten = nullptr;
   anz = 0;
   if (cnt > 0) {
      p_noten = new double[cnt];
      anz = cnt;
      for (int i = 0; i < anz; ++i) {
         p_noten[i] = (i % 5) + 1.0;
      }
   }
}
```

Auf die Implementierung der `delete`[]-Aufrufe im Destruktor wird noch verzichtet. Sonst würde sich ein Absturz ergeben, da der Kopierkonstruktor etc. aktuell noch nicht korrekt implementiert sind. Hier werden Sie in einem späteren Aufgabenteil für Abhilfe sorgen.

Der Aufruf der Funktion `f0`

```
void f0() {
   Student s1("ABBAcus");
// Kaul hat Noten,  1,2,3,4,5,1,2,3
   Student s2("Kaul", 8);
   datei << " Ende" << endl;/*1*/
}
```

führt also zur folgenden Ausgabe:

```
+S ABBAcus +S Kaul Ende
-S Kaul -S ABBAcus
```

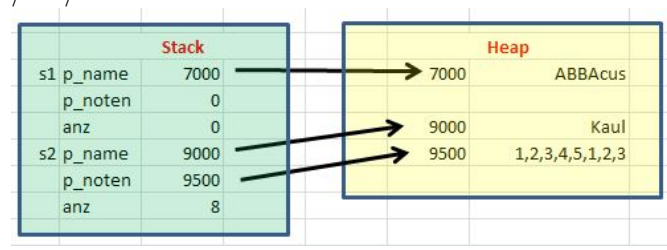
Die Hilfsfunktion `PR` dient zur Ausgabe der Bereiche, auf die der Zeiger `n` verweist, und falls `n` ein `nullptr` ist, wird `xxx` ausgegeben.

```
string PR(const char* p) {
   if (p != nullptr) {
      return string(p);
   }
   else {
      return "xxx";
   }
}
```

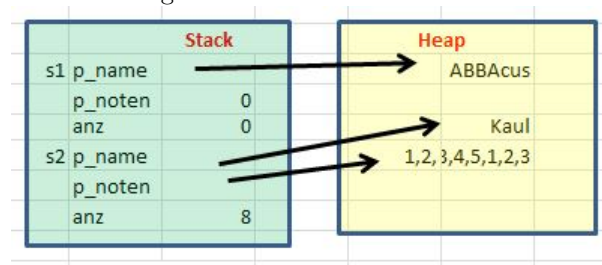
Zum sicheren Kopieren von Zeichenketten, die über `char*` realisiert sind, dient die folgende Funktion:

```
/* Erzeugt eine Kopie des Bereiches, auf den
   src verweist, auf dem Heap und anschliessend
   zeigt dest auf diesen neuen Speicherbereich.
   src wird nicht verändert. */
void SaveCopyOnHeap(
   char*& dest, const char* src) {
   if (src != nullptr) {
      auto len = strlen(src) + 1;
      dest = new char[len];
 // kopiert maximal len Bytes
 // vom Speicherbereich src nach dest
      strcpy_s(dest, len, src);
   }
   else {
      dest = nullptr;
   }
}
```

Die folgende Speicherdarstellung dient zur Veranschaulichung der Speicherbelegung zum Zeitpunkt /*1*/.



Zur Zeitersparnis verwenden Sie bitte jeweils folgende Vereinfachung:



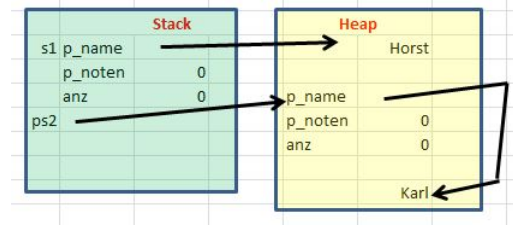
a.) (4 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f1` ?

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

Lösung:

```
+S Horst +S Karl -S Horst
```

b.) (6 P.)
Skizzieren Sie auf einem Extra-Blatt entsprechend der vereinfachten Beispiel-Zeichnungen auf Seite 5 die Speicherbelegungen zum Zeitpunkt /*2*/. **Lösung:**



c.) (3 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f2` ?

```
void f2() {
   Student* p1 = new Student("Karl");
   unique_ptr<Student> p2(new Student("Ute"));
   datei << " Ende ";
}
```

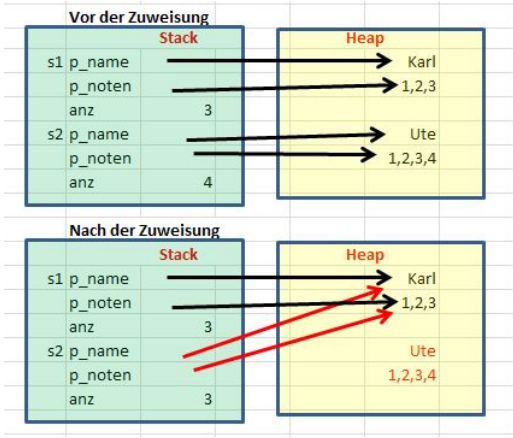
Lösung:

```
+S Karl +S Ute Ende -S Ute
```

d.) (6 P.)
Skizzieren Sie auf einem Extra-Blatt entsprechend der vereinfachten Beispiel-Zeichnungen auf Seite 5 die Speicherbelegungen zu den Zeitpunkten /*3*/ und /*4*/ beim Aufruf der Funktion `copy`.

```
void copy() {
   Student s1("Karl", 3);
   Student s2("Ute", 4); /* 3 */
   s2 = s1;
   datei << "\n Ende "; /* 4 */
}
```

Lösung:



Achtung: Bedenken Sie, dass die Klasse aktuell noch keinen brauchbaren Zuweisungs-Operator etc. hat, sondern lediglich flach kopiert, siehe Anhang Objekterzeugung.

e.) (9 P.)
Implementieren Sie nun auf einen Extra-Blatt einen korrekten Zuweisungsoperator für die Klasse, der eine tiefe Kopie durchführt. Nutzen Sie dazu den folgenden Code. Es genügt , wenn Sie die Funktion `CreateHeapAndCopyNoten` implementieren. Achten Sie auf die korrekte Implementierung der Typen der vier Parameter inklusive des Schlüsselwortes `const`. Bedenken Sie, dass Zeiger `nullptr` sein können oder `anz 0` enthält

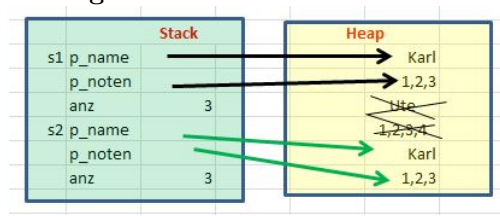
```
Student& Student::operator=(const Student& s2) {
   datei << " op= " << PR(s2.p_name);
   char* hlpName = nullptr;
   SaveCopyOnHeap(hlpName, s2.p_name);
   double* hlpNoten = nullptr;
   CreateHeapAndCopyNoten(hlpNoten,
      s2.p_noten, anz, s2.anz);
   delete[] p_noten;
   delete[] p_name;
   // Zeiger verbiegen
   p_noten = hlpNoten;
   p_name = hlpName;
   return *this;
}
```

Lösung:

```
void CreateHeapAndCopyNoten(
   double*& p_destNoten, const double* p_notenSrc,
   int& r_anzDest, int anzSrc) {
   p_destNoten = nullptr;
   if (p_notenSrc != nullptr && anzSrc > 0) {
      p_destNoten = new double[anzSrc];
      r_anzDest = anzSrc;
      for (int i = 0; i < r_anzDest; ++i) {
         p_destNoten[i] = p_notenSrc[i];
      }
   }
   else {
      r_anzDest = 0;
   }
}
```

f.) (4 P.)
Skizzieren Sie nochmals auf einem Extra-Blatt die Speicherbelegungen zum Zeitpunkt /*4*/ in der Funktion `copy`, wenn Sie von einer jetzt korrekten Implementierung der tiefen Kopie des Zuweisungsoperators ausgehen. Kennzeichen Sie freigegeben Speicher z.B. durch Durchstreichen.

Lösung:



g.) (4 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f3` ?
Gehen Sie bitte im Folgenden von einer korrekten Implementierung von Kopier-Konstruktor, Verschiebe-Konstruktor,, Zuweisungs-Operator und Verschiebe-Zuweisungsoperator aus, d.h. es wird mit Definition von `DEEP_COPY_ALREAY_IMPLEMENTED` übersetzt. Diese vier Methoden führen bei Ihrem Aufruf Ausgaben nach `datei` entsprechend dem Anhang durch, d.h. , `+S Co`, `+S Mv`, `op=` und `mv=`.

```
void f3() {
   Student s1("Karl", 3);
   Student s2(s1);
   datei << "\nEnde ";
}
```

Lösung:

```
 +S Karl +SCo Karl
Ende  -S Karl -S Karl
```

h.) (4 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f4` ?

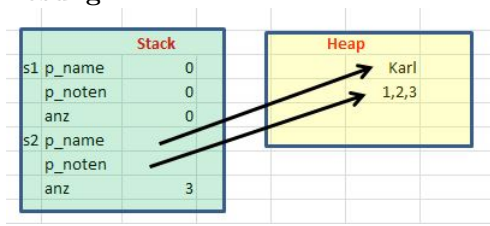
```
void f4() {
   Student s1("Karl", 3);
   Student s2(move(s1));
   datei << "\nEnde "; /*5*/
}
```

Lösung:

```
 +S Karl +SMv Karl
Ende  -S Karl -S xxx
```

i.) (4 P.)
Skizzieren Sie auf einem Extra-Blatt die Speicherbelegungen zum Zeitpunkt /*5*/ in der obigen Funktion `f4`.

Lösung:



j.) (5 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f5` ?

```
Student Create(const char* name) {
   datei << " Create ";
   Student hlp(name);
   return hlp;
}
void f5() {
   Student s2(Create("Karl"));
   datei << "\nEnde ";
}
```

Lösung:

```
 Create  +S Karl +SMv Karl -S xxx
Ende  -S Karl
```

Anhang Objekterzeugung

Der folgende Code zeigt die Implementierung von Kopier-Konstruktor, Verschiebe-Konstruktor,, Zuweisungs-Operator und Verschiebe-Zuweisungsoperator, wenn nur flach kopiert würde.

```
Student::Student(const Student& s2) {
   datei << " +SCo " << PR(s2.p_name);
   StupidShallowCopy(s2);
}
Student::Student(Student&& s2) {
   datei << " +SMv " << PR(s2.p_name);
   StupidShallowCopy(s2);
}
Student& Student::operator=(const Student& s2) {
   datei << " op= " << PR(s2.p_name);
   StupidShallowCopy(s2);
   return *this;
}
void Student::operator=(Student&& s2) {
   datei << " mv= " << PR(s2.p_name);
   StupidShallowCopy(s2);
}
```

```
void Student::StupidShallowCopy(const Student& s2) {
   p_name = s2.p_name;
   p_noten = s2.p_noten;
   anz = s2.anz;
}
void Student::StupidShallowCopy(Student& s2) {
   p_name = s2.p_name;
   p_noten = s2.p_noten;
   anz = s2.anz;
}
```