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 2018/19:

Die verschiedenen Programmierparadigmen von C++ — Lösungen

Informatik Bachelorstudiengang ☐ *Informatik Masterstudiengang* ☐

Hilfsmittel sind bis auf Computer, Handy etc. erlaubt !	Bitte Aufgabenblätter mit abgeben !
Austausch von Hilfsmitteln mit Kommilitonen ist **nicht** erlaubt !	
Anwesenheit von Handys, Smartphones etc. bei Klausurteilnehmern im Hörsaal nicht erlaubt.	
Sie sind vor Beginn der Klausur am Dozentenpult abzugeben!	

Bitte notieren Sie auf **allen** Blättern Ihren Namen bzw. 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 werden manchmal die lokalen Variablen *file*, *file1*, *file2* sowie die globale Variable *datei* verwendet. Hierfür kann der Einfachheit halber die Variable *cout* angenommen werden. Die Variablen dienten bei der Klausurerstellung lediglich dazu, automatisch eine Lösungsdatei zu erstellen.

Meistens kann die Lösung direkt auf dem Aufgabenblatt notiert werden. Extrablätter bitte mit Namen und/oder Matrikelnummer versehen.
Gehen Sie davon aus, dass `double` 8 Bytes, `int` und Zeiger jeweils 4 Bytes sowie `char` ein Byte im Speicher belegen. Aufrufe von `new` sollen jeweils den nächsten zusammenhängenden ausreichend großen freien Speicherbereich beginnend bei den Adressen ab 7400 liefern. Die Speicherbelegung soll hier auf Stack und Heap jeweils von den tieferen zu den höheren Adressen verlaufen.
Wenn nicht anders angegeben, befinden wir uns jeweils im Namensraum `std`, d.h. `using namespace std;` dürfen Sie in jeder Codedatei annehmen.

Geplante Punktevergabe

Punktziel	Im einzelnen	Pkte
Tests/Team: max. 30 P.		
A1: 24 P.		
A2: 32 P.		
A3: 24 P.		
Summe 80+30 P.		

Sie finden bereits auf dieser Seite C++-Code, den Sie erst später benötigen, um Ihnen das Umblättern zu erleichtern. Hier zunächst die Schnittstelle der Klasse `Guest`, die Sie bei den Aufgaben 2 und 3 benötigen:

```
class Guest{
public:
   Guest();
   Guest(char* n, int w, int u);
   ~Guest();
   Guest(const Guest& cp);
   Guest(Guest&& cp);
   Guest& operator=(const Guest& cp);
   Guest& operator=(Guest&& mp);

   int GetWakeUpTime() const { return wakeUpTime; }
   const char* GetName() const { return name; }
   int GetUsageTime() const { return usageTime; }
   int GetEntryTime() const { return entryTime; }
   void SetEntryTime(int e) { entryTime = e; }
private:
   // a guest needs 16 Bytes of memory
   void CopyAllAttributes(const Guest& cp);
   enum {NAME_SIZE=4};

   char name[NAME_SIZE]; // needs 4 bytes
   int wakeUpTime; // 4 bytes
   int usageTime; // 4 bytes
   int entryTime; // 4 bytes
};
```

Die Implementierung der Klassenschnittstelle:

```
Guest::Guest() {
   datei << "+G-Standard ";
   wakeUpTime = 0;
   usageTime = 0;
   name[0] = '\0'; // empty string
   entryTime = 0;
}
Guest::Guest(char* n, int w, int u) {
   datei << "+G " << n << " ";
   wakeUpTime=w;
   usageTime=u;
   // copy all bytes of n to name
   // char* strcpy(char*, const char*);
   strcpy(name, n);
   entryTime = 719; // 11:59 Uhr
}
Guest::~Guest(){
   datei << "-G " << name << " ";;
}

// copies all attributes from cp to *this
void Guest::CopyAllAttributes(const Guest& cp) {
   usageTime = cp.usageTime;
   wakeUpTime = cp.wakeUpTime;
   entryTime = cp.entryTime;
   strcpy(name, cp.name);
}
Guest::Guest(const Guest& cp) {
   datei << "+GC ";
   CopyAllAttributes(cp);
}
Guest::Guest(Guest&& cp) {
   datei << "+GM ";
   CopyAllAttributes(cp);
}
Guest& Guest::operator=(const Guest& cp) {
   datei << "op= ";
   CopyAllAttributes(cp);
   return *this;
}
Guest& Guest::operator=(Guest&& mp) {
   datei << "move= ";
   CopyAllAttributes(mp);
   return *this;
}
```

Die Klasse `Node` wird bei Aufgabe 3 benötigt:

```
class Node {
public:
   Node(const Guest guest, Node* parent,
        int gVal, int hVal);
   ~Node();
   Node* GetParent(){return parent;}
   const Node* GetParent() const {return parent;}
   int GetGValue() const {return gVal;}
   const char* GetName() const {
      return guest.GetName();
   }
   static void recursiveDelete(Node* node);
private:
   Guest guest;
   Node* parent;
   int gVal; /* waiting of this node*/
   int hVal; /*estimated for children */
   vector<Node*> children; // to avoid mem leaks
};
```

Konstruktor und Destruktor seien wie folgt gegeben:

```
Node::Node(const Guest g, Node* par,
           int gV, int hV) {
   guest = g;
   parent = par;
   gVal = gV;
   hVal = hV;
}
Node::~Node() {}
```

Aufgabe 1 : Schleifen, STL etc.

ca. 24 Punkte

a.) (2,5 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welche Ausgabe liefert der Aufruf des folgenden Programmfragments in `datei`?

```
void loop1() {
   int zaehler = 0;
   for (int i = 23; i < 28; ++i) {
      ++zaehler;
   }
   datei << zaehler << " runs \n";
}
```

Lösung:

```
5 runs
```

b.) (2,5 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welche Ausgabe liefert der Aufruf des folgenden Programmfragments in `datei`?

```
void loop2() {
   int zaehler = 0; ;
   for (int i = 23; i < 28; i++) {
      zaehler++;
   }
   datei << zaehler << " runs \n";
}
```

Lösung:

```
5 runs
```

c.) (5 P.) Gegeben seien die folgenden 5 Funktionen `par1` bis `par5`.

```
void par1(int ar) {
   ar = 113;
}
void par2(int* ar) {
   *ar = 213;
}
void par3(int& ar) {
   ar = 313;
}
void par4(int&& ar) {
   ar = 414;
}
void par5(int** ar) {
   **ar = 515;
}
```

Geben Sie jeweils einen möglichen Aufruf für die Funktionen an, sodass die Variable `j` direkt oder indirekt übergeben wird, d.h. was ist jeweils für `???` einzusetzen?

```
void caller () {
   int j = 4;
   par1( ???            );
   datei << "par1 " << j << "\n";

   par2( ???          );
   datei << "par2 " << j << "\n";
```

```
   par3( ???             );
   datei << "par3 " << j << "\n";

   par4( ????             );
   datei << "par4 " << j << "\n";

   int* pntJ = &j;
   par5( ???       pntJ   );
   datei << "par5 " << j << "\n";
}
```

Lösung:

```
void caller () {
   int j = 4;
   par1(j);
   datei << "par1 " << j << "\n";
   par2(&j);
   datei << "par2 " << j << "\n";
   par3(j);
   datei << "par3 " << j << "\n";

   par4(move(j));
   datei << "par4 " << j << "\n";
   int* pntJ = &j;
   par5(&pntJ);
   datei << "par5 " << j << "\n";
}
```

d.) (5 P.) Zu welcher Ausgabe in `datei` führt der Aufruf von `caller`?

Lösung:

```
par1 4
par2 213
par3 313
par4 414
par5 515
```

e.) (9 P.) Zu welchen Ausgaben in `datei` führt der Aufruf von `erase`?

```
void erase() {
   vector<int> cont;
   cont.push_back(5);
   for (int i = 2; i < 7; ++i) {
      cont.push_back(i);
   }

   // Ausgabe des Container-Inhalts nach datei
   copy(cont.begin(), cont.end(),
        ostream_iterator <int>(datei, " "));
   datei << "\n";

   auto iter = remove(cont.begin(), cont.end(), 5);
   copy(cont.begin(), cont.end(),
        ostream_iterator <int>(datei, " "));
   datei << "\n";

   cont.erase( iter , cont.end());
   copy(cont.begin(), cont.end(),
        ostream_iterator <int>(datei, " "));
   datei << "\n";
}
```

Lösung:

```
5 2 3 4 5 6
2 3 4 6 5 6
2 3 4 6
```

Aufgabe 2 : Instanzerzeugung

ca. 32 Punkte

a.) (3 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf von `g1`?

```
void g1(){
   Guest gu1("Ida", 2, 600);
}
```

Lösung:

```
+G Ida -G Ida
```

b.) (13 P.) Welche Ausgabe ergibt der Aufruf von `g2a` (`create` wird erst im Teil 2 und 3 verwendet)?

```
Guest create(Guest& arg) {
   arg.SetEntryTime(400);
   datei << " create\n ";
   return arg;
}
void g2a() {
   Guest g1("Udo", 602, 5);
   Guest g2;
   datei << g2.GetEntryTime() << " ";
   datei << " Ende\n ";
}
```

Lösung:

```
+G Udo +G-Standard 0 Ende
 -G  -G Udo
```

Welche Ausgabe ergibt der Aufruf von `g2b`?

```
void g2b() {
   Guest g1("Udo", 602, 5);
   create(g1);
   datei << g1.GetEntryTime() << " ";
   datei << " Ende\n ";
}
```

Lösung:

```
+G Udo create
 +GC -G Udo 400 Ende
 -G Udo
```

Welche Ausgabe ergibt der Aufruf von `g2c`?

```
void g2c() {
   Guest g1("Udo", 602, 5);
   Guest g2;
   g2 = create(g1);
   datei << g1.GetEntryTime() << " ";
   datei << g2.GetEntryTime() << " ";
   datei << " Ende\n ";
}
```

Lösung:

```
+G Udo +G-Standard create
 +GC move= -G Udo 400 400 Ende
 -G Udo -G Udo
```

c.) (5 P.) Welche Ausgabe ergibt der Aufruf von `g3`?

Beginnen Sie am besten mit den Zeichnungen der folgenden Teilaufgabe.

```
void g3(){
   Guest* p= new Guest("Udo", 602, 5); /*1*/
   if (p != nullptr) {
      Guest gu1("Pet", 600, 1); /*2*/
   }
   Guest gu2("E", 500, 4); /*3*/
   delete p;
}
```

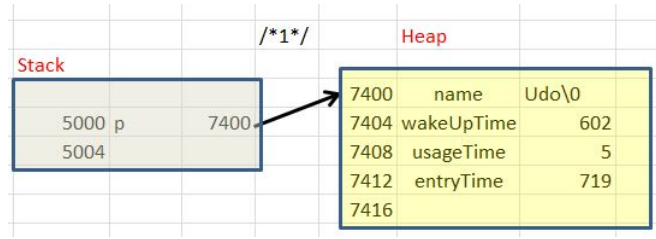
Lösung:

```
+G Udo +G Pet -G Pet +G E -G Udo -G E
```

d.) (11 P.) Veranschaulichen Sie grafisch die Stack- **und** Heap-Speicherbelegung zu den verschiedenen Zeitpunkten.

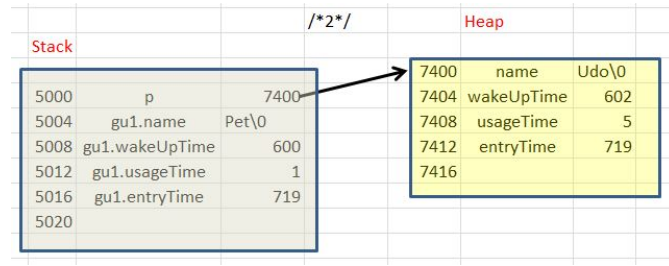
Speicherbelegung bei /*1*/ zeichnen:

Lösung:



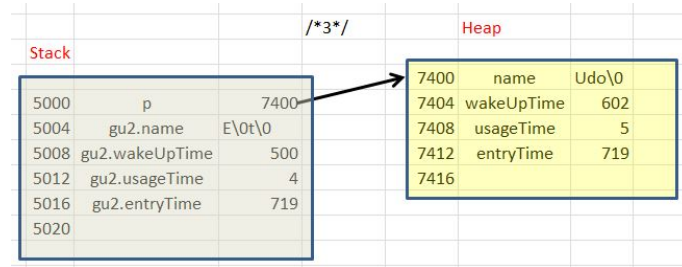
Speicherbelegung bei /*2*/ zeichnen:

Lösung:



Speicherbelegung bei /*3*/ zeichnen:

Lösung:



Aufgabe 3 : Tree Search

ca. 24 Punkte

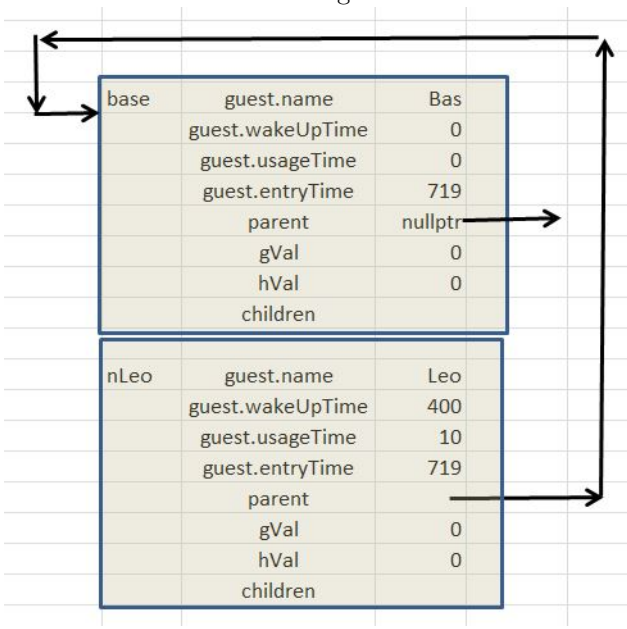
a.) (6 P.) Die folgende Funktion *versucht* einen Ausschnitt aus einem Suchbaum zu erzeugen. Das Attribut `Node::children` wird noch nicht verwendet.

```
Node createTree() {
   Guest bas("Bas", 0, 0);
   Guest leo("Leo", 400, 10);
   Node base(bas, nullptr, 0, 0);
   Node nLeo(leo, &base, 0, 0); /* 1 */
```

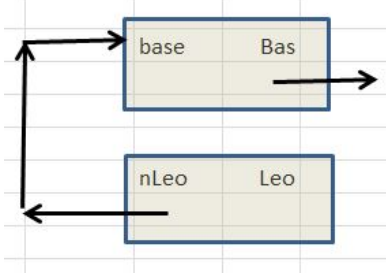
```
   Guest evi("Evi", 406, 3);
   Guest pap("Pap", 405, 6);

   Node nEvi(evi, &base, 0 ,0);
   Node nPap(pap, &base, 0, 0);

   Node nLeoEvi(evi, &nLeo, 4, 5);
   Node nLeoPap(pap, &nLeo, 5, 4);
   Node n6(pap, &nLeoEvi, 11, 0);
   Node n7(evi, &nLeoPap, 7, 0); /*2*/
```

```
   return n7;
}
```

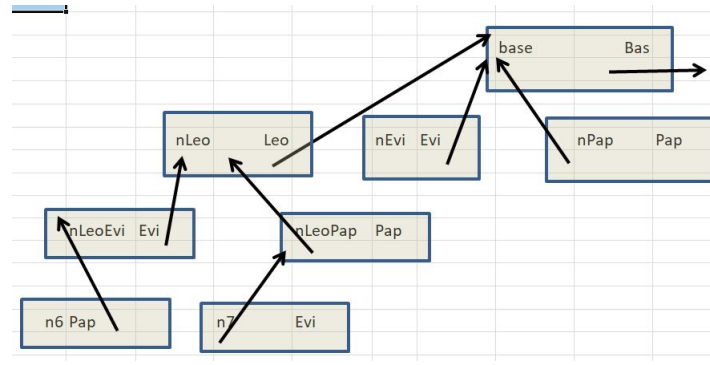
Skizzieren Sie den Suchbaum zum Zeitpunkt /*2*/, d.h. welche Instanzen von `Node` und `Guest` gibt es und wie verweisen diese aufeinander? Zum Zeitpunkt /*1*/ würde die Skizze z.B. wie folgt aussehen:



Sie müssen Ihre Zeichnung nicht so detailliert angeben. Die folgende Detailtiefe genügt:



Lösung:



b.) (3 P.) Zu welcher Ausgabe auf den Bildschirm würde der Aufruf von `printUntilRoot(&n7, cout);` am Ende der Funktion `createTree` (vor der `return`-Anweisung) führen:

```
void printUntilRoot (const Node* arg, ofstream& file)
{

   const Node* base = arg;

   while (base->GetParent() != nullptr) {
       file << base->GetName() << endl;
      base = base->GetParent();
   }

}
```

Lösung:

```
Evi
Pap
Leo
```

c.) (8 P.) Der Rückgabewert von `createTree` ist allerdings ziemlich nutzlos, z.B. führt der folgende Code bei Ausführung von `printUntilRoot` zu einem undefinierten Verhalten:

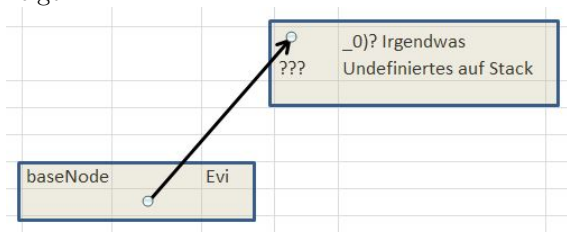
```
void callCreateTree () {
   Node baseNode = createTree(); /* 3 */
   // Ausgabedatei erzeugen
   ofstream file ("createTree2.txt", ios::out);
   // Baum ausgeben
    printUntilRoot (&baseNode, file);
}
```

Erklären Sie das Problem z.B. durch Skizzierung der Speicherbelegung bei /*3*/ auf einem Extrablatt. Geben Sie dabei auch an, bis wann die Ausgabe in `file` noch definiert abläuft und in welcher Codezeile warum mit undefiniertem Verhalten zu rechnen ist.

Lösung:

In `createTree` werden eine Menge Instanzen von `Node` miteinander durch Zeiger verbunden. Allerdings liegen sämtliche dieser Instanzen inklusive der Instanzen von `Guest` auf dem Stack. Nach Verlassen der Funktion gibt es diese Instanzen also nicht mehr. Einzig eine Kopie `n7` wird beim Verlassen von `createTree` erzeugt und steht damit in der aufrufenden Funktion zur Verfügung. Das Attribut `parent` zeigt allerdings bereits auf ein Element auf dem Stack, das damit schon

einen undefinierten Wert hat, wie die folgende Grafik zeigt:



In der aufgerufenen Funktion printUntilRoot ist die erste Auswertung von `base->GetParent() != nullptr` noch definiert. Der Zeiger verweist zwar auf etwas Undefiniertes. Trotzdem ist er verschieden von `nullptr`. Damit wird die `while`-Schleife betreten und die Ausgaben sind auch definiert, denn das erste Objekt (Kopie von `n7`) liegt im gültigen Stackbereich. Nun wird aber der Zeiger `parent` zugewiesen, der zwar verschieden von `nullptr` ist, aber schon auf etwas Undefiniertes verweist. Die Auswertung der Bedingung in der `while`-Schleife ist noch definiert. Es ergibt sich `true`. Spätestens die Ausgabe über die get-Methoden führt aber zu undefiniertem Verhalten. Ohne Release-Modus kann sich nun auch eine Todschleife ergeben. Es erfolgen irgendwelche Ausgaben, bis irgendwann die Zuweisung `base->GetParent();` tatsächlich mal einen Wert gleich dem `nullptr` zuweist und damit die `while`-Schleife terminiert. Über die undefinierten Zeiger könnte aber auch ein Zugriff auf verbotene Speicherbereiche versucht werden, sodass dann hoffentlich das Laufzeitsystem der Codeausführung ein Ende bereitet. Eine Ausgabe könnte also z.B. sein.

```
Evi
ZXV8890undefiniertxYZUui
```

d.) (7 P.) Wie müsste `createTree` verändert werden, damit der obige Aufruf von `callCreateTree` zum gewünschten Verhalten führt? Die Klasse `Node` soll hierbei nicht verändert werden. Die einzige Änderung soll die folgende Benutzung des Attributes `children` im Konstruktor betreffen, damit man Memory-Leaks auf dem Heap-Speicher vermeiden kann.

```
Node::Node(const Guest g, Node* par,
           int gV, int hV) {
   datei << "+N ";
   guest = g;   parent = par;
   if (par != nullptr) {
      par->children.push_back(this);
   }
   gVal=gV;      hVal=hV;
   datei << "\n";
}
```

Skizzieren Sie die Verbesserung von `createTree` auf einem Extrablatt (Sie dürfen sinnvoll abkürzen).

Lösung:

Die Instanzen von `Node` müssen auf dem Heap angelegt werden. Die Instanzen von Guest dürfen auch auf dem Stack angelegt werden, da sie beim Konstruktoraufruf von Node ohnehin als Kopie übergeben werden und damit auf dem Heap landen. Ansonsten müsste noch Extra-Code zum Aufräumen implementiert werden.

```
Node createTree() {
   Guest bas("Bas", 0, 0);
   Guest leo("Leo", 400, 10);
   Node* base = new Node(bas, nullptr, 0, 0);
   Node* nLeo = new Node(leo, base, 0, 0);

   Guest evi("Evi", 406, 3);
   Guest pap("Pap", 405, 6);

   Node* nEvi = new Node(evi, base, 0 ,0);
   Node* nPap = new Node(pap, base, 0, 0);

   Node* nLeoEvi = new Node(evi, nLeo, 4, 5);
   Node* nLeoPap = new Node(pap, nLeo, 5, 4);
   Node* n6 = new Node(pap, nLeoEvi, 11, 0);
   Node* n7= new Node(evi, nLeoPap, 7, 0);
   return *n7;
}
```

Wir haben nicht einmal ein Speicherleck, wenn wir nach Benutzung des durch `createTree` erzeugten Suchbaumes `deleteSearchTree` mit irgendeiner Knoteninstanz aus dem Suchbaum aufrufen.

```
/** if node has no parent we return node
Otherwise we recursively recall this
function with parent of node.
*/
Node* findRoot(Node* node) {
   if (node->GetParent() == nullptr) {
      return node;
   }
   else {
      return findRoot(node->GetParent());
   }
}

/** First we delete all children of node
    and then we delete node itself.
*/
void Node::recursiveDelete(Node* node) {
   for (auto iter : node->children) {
      recursiveDelete (iter);
   }
   delete node;
}

void deleteSearchTree(Node& node) {
   Node* root = findRoot(&node);
   Node::recursiveDelete(root);

}
```