

Name:

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

Matrikelnummer:		Punktzahl:	
Ergebnis:			
Freiversuch	<input type="checkbox"/>	F1	<input type="checkbox"/>
		F2	<input type="checkbox"/>
		F3	<input type="checkbox"/>

Klausur im WS 2017/18:

Die verschiedenen Programmierparadigmen von C++

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 wird manchmal die lokale Variable *file* sowie die globale Variable *datei* verwendet. Hierfür kann der Einfachheit halber die Variable *cout* angenommen werden. Die Variablen dienen bei der Klausurerstellung lediglich dem Zweck der Ausgabeumlenkung.

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 sowie `int` und Zeiger jeweils 4 Bytes im Speicher belegen. Die detaillierte Implementierung der Klasse `string` können Sie vernachlässigen. Nehmen Sie einfach 4 Byte (z.B. `char[4]`) an. 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	Sonderpunkte	erreicht
Laboraufgaben: max. 30 P.		
A1: 18 P.		
A2: 41 P.		
A3: 21 P.		
Summe 80+30 P.		

Die Klasse `GuestBase` wird in (fast) allen folgenden Aufgaben benötigt:

```

/* Basic information of guest,
which is stable during planing
*/
class GuestBase{
public:
    GuestBase(int u, string n, int w);
    ~GuestBase();
    GuestBase(const GuestBase & cp);
    int GetWakeUpTime() { return wakeUpTime; }
    string GetName() { return name; }
    int GetUsageTime() { return usageTime; }

private:
    int wakeUpTime;
    string name;
    int usageTime;
};

```

Die zugehörigen Methoden zur Objekterzeugung und -zerstörung sind wie folgt definiert (wichtig sind vor allem die Ausgaben nach `datei`):

```

GuestBase::GuestBase(int u, string n, int w) {
    datei << "+B " << n << " ";
    wakeUpTime = w;
    name = n;
    usageTime = u;
}
GuestBase::~GuestBase() {
    datei << "-B " << name << " ";
}
GuestBase::GuestBase(const GuestBase & cp) {
    datei << "+CB " << cp.name << " ";
    wakeUpTime = cp.wakeUpTime;
    name = cp.name;
    usageTime = cp.usageTime;
}

```

Die Klasse `Guest` hat folgende Deklaration:

```

/* dynamic info of a guest, which changes
depending on sequence number of guest */
class Guest{
public:
    Guest(int u, string n, int w);
    ~Guest();
    Guest(const Guest& cp);
    Guest(Guest&& cp);
    int GetWakeUpTime(){return pg->GetWakeUpTime();}
    string GetName() {return pg->GetName(); }
    int GetUsageTime() {return pg->GetUsageTime(); }
    int GetEntryTime() const { return entryTime; }

private:
    GuestBase* pg;
    int entryTime;
};

```

Die zugehörigen Methoden zur Objekterzeugung und -zerstörung sowie der Vergleichsoperator sind wie folgt definiert:

```

Guest::Guest(int u, string n, int w) {
    datei << "+G ";
    pg = new GuestBase(u, n, w);
    entryTime = 23*60 + 59; // 23:59 Uhr
}
Guest::~Guest(){
    datei << "-G ";
    delete pg;
}
Guest::Guest(const Guest& cp) {
    datei << "+CG ";
    pg = new GuestBase(* ( cp.pg) );
    entryTime = cp.entryTime;
}
Guest::Guest(Guest&& cp) {
    datei << "+MG ";
    pg = cp.pg;
    entryTime = cp.entryTime;
    cp.pg = nullptr;
}
bool operator<(Guest& g1, Guest& g2){
    return g1.GetUsageTime() < g2.GetUsageTime();
}

```

Außerdem gibt es noch den Ausgabeoperator:

```

ostream& operator<<(ostream& str, Guest& g){
    str << g.GetName() << " ";
    PrintAsTime(str, g.GetEntryTime());
    str << " (";
    PrintAsTime(str, g.GetWakeUpTime());
    str << ", " << g.GetUsageTime();
    str << ")";
    return str;
}

```

mit den zwei Hilfsfunktionen zur formatierten Zeitausgabe:

```

/* Ausgabe von t, wenn kleiner 10, dann
mit vorangestellter 0 */
void PrintWithZeros(ostream& str, int t){
    if (t < 10){
        str << "0" << t;
    }
    else {
        str << t;
    }
}
/* gibt die Minuten eines Tages in Stunden
und Minuten mit fuhrenden Nullen aus, also
z.B. 604 ist : 10:04, 595 ist 09:55 */
void PrintAsTime(ostream& str, int t){
    PrintWithZeros(str, t / 60);
    str << ":";
    PrintWithZeros(str, t % 60);
}

```

Aufgabe 1 : Schleifen und Lambda-Ausdrücke

ca. 18 Punkte

a.) (2,5 P.) Zu welcher Ausgabe (in die hier erzeugte Datei) führt die Ausführung der folgenden Codezeilen? In der gesamten Aufgabe 1 sollen Sie die Ausgaben nach `datei`, d.h., die Objekterzeugung und Objektzerstörung, nicht betrachten.

```
ofstream file ("prG2.txt", ios::out);
for (int i = 6; i < 9; ++i) {
    Guest g1(i, "Hans", 600);
    file << g1.GetName() << " ";
}
```

(*— Lösung hier notieren. —*)

b.) (2,5 P.) Zu welcher Ausgabe (in die hier erzeugte Datei) führt die Ausführung der folgenden Codezeilen?

```
ofstream file ("prG3.txt", ios::out);
for (int i = 16; i > 12; --i) {
    Guest g1(i, "Paul", 600);
    file << g1.GetName() << " ";
}
```

(*— Lösung hier notieren. —*)

c.) (3 P.) Erklären Sie, warum eine `const`-Deklaration des Parameters `g` vom Typ `Guest` im Ausgabeoperator zu einem Syntaxfehler führen würde (**Hinweis:** siehe *unglückliche* Deklaration der Klasse `Guest`).

(*— Lösung hier notieren. —*)

d.) (3 P.) Ein Codebeispiel soll nun zunächst den Code des Ausgabeoperators verdeutlichen:

```
ofstream file ("prG1.txt", ios::out);
Guest g1(3, "Hans", 599);
Guest g2(6, "Petz", 610);
file << g1 << "\n" << g2 << "\n";
```

Die Ausführung dieser Zeilen führt zur Ausgabe:

```
Hans:23:59 (09:59, 3)
Petz:23:59 (10:10, 6)
```

Zu welcher Ausgabe (in die hier erzeugte Datei) führt die Ausführung der folgenden Codezeilen?

```
vector<Guest> vg;
for (int i = 2; i < 5; ++i) {
    // Kopie von Guest wird in Vektor gepackt
    vg.push_back(Guest(i, "Paul", 600));
}
ofstream file ("prG4.txt", ios::out);
for (auto iter : vg) {
    // nur usageTime bei jedem Gast anders
    file << iter << "\n";
}
```

(*— Lösung hier notieren. —*)

e.) (7 P.) Zu welcher Ausgabe (in die hier erzeugte Datei) führt die Ausführung der folgenden Codezeilen?

```
ofstream file ("prG5.txt", ios::out);
vector<Guest> vg;
for (int i = 1; i < 5; ++i) {
    // Kopie von Guest wird in Vektor gepackt
    vg.push_back(Guest(i, "Paul", 600));
}
int sum = 0;
auto lambda =
    [&sum](Guest& g) {sum += g.GetUsageTime();};

sum = 0;
for_each(vg.begin(), vg.end(), lambda);
file << "sum is " << sum << "\n";

sum = 0;
for_each(++(vg.begin()), (vg.end())--, lambda);
file << "sum is " << sum << "\n";

for_each((vg.begin()++), (--vg.end()), lambda);
file << "sum is " << sum << "\n";
```

(*— Lösung hier notieren. —*)

Aufgabe 2 : Instanzerzeugung

ca. 41 Punkte

Für diese Aufgabe sind die bereits vorgestellten Methoden von `GuestBase` und `Guest` zur Objekterzeugung und -zerstörung und hier insbesondere die Ausgaben nach `datei` von Interesse.

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

```
void g1(){
    GuestBase gu1(1, "Hans", 600);
    GuestBase gu2(5, "Paul", 602);
}
```

(*— Lösung hier notieren. —*)

b.) (4 P.) Welche Ausgabe ergibt der Aufruf von `g2`?

```
void g2(){
    Guest gu1(1, "Elke", 600);
    Guest gu2(5, "Lola", 602);
}
```

(*— Lösung hier notieren. —*)

c.) (3 P.) Welche Ausgabe ergibt der Aufruf von `g5`?

```
void g5(){
    GuestBase* pg1=new GuestBase(3, "Lora", 600);
    GuestBase* pg2 = pg1;
}
```

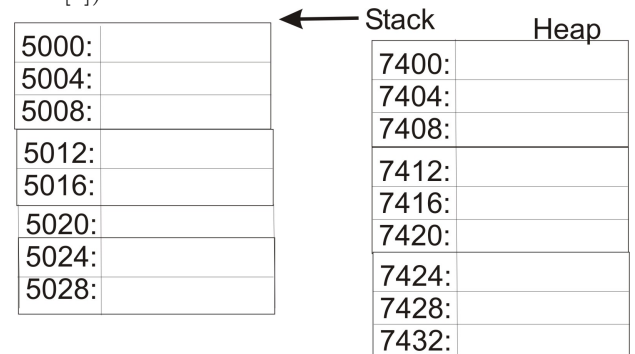
(*— Lösung hier notieren. —*)

d.) (3,5 P.) Welche Ausgabe ergibt der Aufruf von `g6`?

```
void g6(){
    GuestBase pg1=new GuestBase(3, "Hake", 600);
    GuestBase g2 = *pg1; /*1*/
}
```

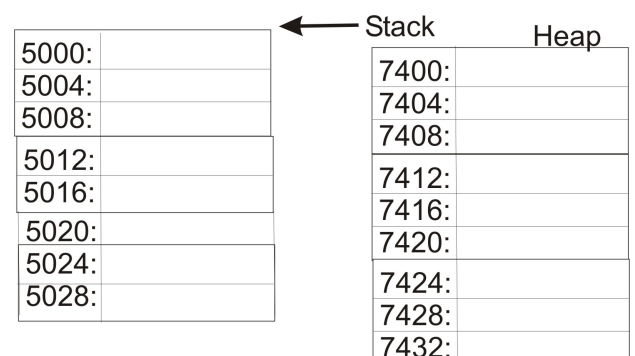
(*— Lösung hier notieren. —*)

e.) (6 P.) Veranschaulichen Sie grafisch die Stack- und Heap-Speicherbelegung nach Ausführung der mit `/*1*/` gekennzeichneten Programmzeile in `g6`. Nehmen Sie vereinfachend für `string` 4 Byte an (also wie `char[4]`).



f.) (8 P.) Veranschaulichen Sie grafisch die Stack- und Heap-Speicherbelegung nach Ausführung von `/*1*/` nach Aufruf von `g4` (string wieder 4 Byte)?

```
void funk34(Guest g){
    datei << "funk34\n "; /*1*/
}
void g3(){
    Guest gu1(1, "Nora", 600);
    funk34(gu1);
    datei << "dann ";
    GuestBase gu2(6, "Petz", 610);
    datei << "Ende\n ";
}
void g4(){
    funk34(Guest(1, "Nora", 600));
    datei << "dann ";
    GuestBase gu2(6, "Petz", 610);
    datei << "Ende\n ";
}
```



g.) (13,5 P.) Welche Ausgaben (nach `datei`) ergeben die Aufrufe von `g3` und `g4`?

(*— Ausgabe von g3 —*)

(*— Ausgabe von g4 —*)

Aufgabe 3 : Badplanung

ca. 21 Punkte

a.) (5 P.) Zur Optimierung der Gästereihenfolge mit vollständiger Permutation wurde folgende Funktion implementiert:

```
double getBestSequenceByFullPermut(
    const vector<Guest>& inpSeq,
    vector<Guest>& bestSeq )
{
    int nFak = 1; // n Fakultät
    for (int i = 0; i < inpSeq.size(); ++i) {
        nFak *= (i + 1);
    }
    double bestSeqValue=1e14; // ganz schlechter Wert
    vector<Guest> testSeq = inpSeq;
    for (int i = 0; i < nFak; ++i) {
        double seqValue = evaluateSequence(testSeq);
        if (seqValue < bestSeqValue) {
            bestSeq = testSeq;
            bestSeqValue = seqValue;
        }
        // Bilde naechste Permutation
        createNextPermutation(testSeq);
    } // for i
    return bestSeqValue;
}
```

Zur Bestimmung der Laufzeiten für unterschiedliche Anzahl von Gästen dient das folgende Testprogramm:

```
void mainTestFullPermut() {
    for (int i = 10; i < 17; ++i){
        vector<Guest> guests;
        // Erzeuge Sequenz mit i Guest
        for (int j = 0; j < i; ++j) {
            guests.push_back(
                Guest(i + j, "xyzu", 600 - 2 * j + i));
        }
        vector<Guest> bestSeq;
        // Laufzeitmessung
        K_TIMER timer; timer.start_timer();
        getBestSequenceByFullPermut(guests, bestSeq);
        cout << "Laufzeit fuer "
            << i << " Gaeste "
            << timer.stop_timer() << " sec\n";
    } // for i
}
```

Hierdurch wurden folgende Ausgaben ermittelt:

```
Laufzeit fuer 10 Gaeste 0.016 sec
Laufzeit fuer 11 Gaeste 0.171 sec
Laufzeit fuer 12 Gaeste 2.044 sec
Laufzeit fuer 13 Gaeste 8.159 sec
Laufzeit fuer 14 Gaeste 5.413 sec
Laufzeit fuer 15 Gaeste 8.487 sec
Laufzeit fuer 16 Gaeste 8.486 sec
```

Sie sehen, dass die Laufzeiten zunächst exponentiell mit der Anzahl der Gäste wachsen, aber bereits beim Übergang von 12 nach 13 Gästen scheint etwas nicht zu stimmen. Finden Sie in der Funktion `getBestSequenceByFullPermut` den Fehler. Hinweis: $13! \approx 6,2270 \cdot 10^9 > 2^{31}$. Wie kann der Fehler korrigiert werden?

(*— Lösung hier notieren. —*)

b.) (6 P.) Der folgende Code zur Berechnung von $n!$ (n Anzahl der Gäste)

```
int nFak = 1;
for (int i = 0; i < guests.size(); ++i) {
    nFak *= (i + 1);
}
```

ist (in der vorliegenden Implementierung von der Klasse `Guests` mit Ausgabe von Instanzerzeugung und -zerstörung) wesentlich schneller als

```
int j = 1;    nFak = 1;
for (auto iter: guests) {
    nFak *= (j++);
}
```

Warum ist das so? Begründen Sie Ihre Antwort. Beziehen Sie bei Ihrer Antwort auch die folgende for-Schleife ein:

```
for (vector<Guest>::iterator iter1 = guests.begin();
     iter1 != guests.end(); ++iter1) {
    nFak *= (j++);
}
```

Ist hier die Laufzeit eher vergleichbar mit der ersten oder der zweiten Implementierung (`auto iter : guests`)?

(*— Lösung hier notieren. —*)

c.) (10 P.) Implementieren Sie die Template-Funktion `MyMinElement`, die das Minimum entsprechend dem dritten Argument in einem Container bestimmt, der durch zwei Iteratoren beschrieben wird.

Die folgenden Aufrufe sollen also möglich sein:

```
vector<Guest> guests;
vector<int> iVec;
```

```
// minimales Element mit operator<
auto minlter = MyMinElement(
    guests.begin(), guests.end(), less<>() );
datei << "Min guests is " << *minlter << "\n";
int v = 0;
// Minimum ungleich v bestimmen
auto minlter2 = MyMinElement(
    iVec.begin(), iVec.end(),
    [v](int a, int b){
        if (a == v) { return false; }
        else if (b == v) { return true; }
        else return a < b; }
    );
```

(*— Lösung hier notieren. —*)