

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 2015/16:

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 wird manchmal die globale Variable *datei* verwendet. Hierfür kann der Einfachheit halber die Variable *cout* angenommen werden. Die Variable *datei* diene lediglich bei der Klausurerstellung 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. Aufrufe von **new** sollen jeweils den nächstenzuhängenden ausreichend großen freien Speicherbereich ab den Adressen beginnend bei 7400 liefern. Die Speicherbelegung soll hier auf Stack und Heap jeweils von den tieferen zu den höheren Adressen verlaufen.

Geplante Punktevergabe

Planen Sie pro Punkt etwas mehr als eine Minute Aufwand ein.

Punktziel	Im einzelnen	Pkte
Laboraaufgaben: 10 +10 P.		
A1: 15 P.		
A2: 20 P.		
A3: 39 P.		
A4: 26 P.		
Summe 100 P.		

Aufgabe 1 : Schleifen

ca. 15 Punkte

a.) (1,5 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert liefert die Ausgabe der Variablen `zaehler`?

```
int zaehler = 0;
for (int i=10; i < 23; ++i) {
    ++zaehler;
}
datei << zaehler << " runs\n";
```

Lösung:

13 runs

b.) (1,5 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert liefert die Ausgabe der Variablen `zaehler`?

```
int zaehler = 0;
for (int i = 10; i < 23; i++) {
    zaehler++;
}
datei << zaehler << " runs\n";
```

Lösung:

13 runs

c.) (1,5 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert liefert die Ausgabe der Variablen `zaehler`?

```
int zaehler = 0;
for (int i = 11; i > 8; --i) {
    zaehler++;
}
datei << zaehler << " runs\n";
```

Lösung:

3 runs

d.) (1,5 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert liefert die Ausgabe der Variablen `zaehler`?

```
int zaehler = 0;
for (int i = 100; i > 20; i = i - 10) {
    zaehler++;
}
datei << zaehler << " runs\n";
```

Lösung:

8 runs

e.) (1,5 P.) Welche Ausgabe erzeugt das folgende Code-Fragment in `datei`?

```
vector<int> cont;
for (int i = 14; i < 18; ++i) {
    cont.push_back(i + 3);
}
for (unsigned int i = 0; i < cont.size(); ++i) {
    datei << cont[i] << " ";
}
```

Lösung:

17 18 19 20

f.) (1,5 P.) Welche Ausgabe erzeugt das folgende Code-Fragment in `datei`?

```
list<int> cont;
for (int i = 14; i < 18; ++i) {
    cont.push_back(i + 3);
}
list<int>::iterator iter = cont.begin();
while (iter != cont.end()) {
    datei << *iter << " ";
    ++iter;
}
```

Lösung:

17 18 19 20

g.) (2 P.) Welche Ausgabe erzeugt das folgende Code-Fragment in `datei`?

```
list<int> cont;
for (int i = 2; i < 14; ++i) {
    cont.push_back(i);
}
datei << *max_element(cont.begin(), cont.end());
datei << ", ";
datei << *find(cont.begin(), cont.end(), 12);
```

Lösung:

13, 12

h.) (4 P.)

Welche Ausgabe erzeugt das folgende Code-Fragment in `datei`?

```
void Print(int i) {
    datei << i << " ";
}
void funk8() {
    vector<int> cont;
    for (int i = 3; i < 6; ++i) {
        cont.push_back(i * i);
        cont.push_back(i);
    }
    for_each(cont.begin(), cont.end(), Print);
    datei << "\n";
    sort(++cont.begin(), cont.end()); // increasing
    for_each(cont.begin(), cont.end(), Print);
}
```

Lösung:

```
9 3 16 4 25 5
9 3 4 5 16 25
```

Aufgabe 2 : Instanzerzeugung

ca. 20 Punkte

Für diese Aufgabe und die folgende wird die folgende Klassendefinition verwendet.

```
class Task {
public:
    Task(int arg) {
        id = arg;
        datei << "+T" << id << " ";
        zeiten = nullptr;
        punkte = 2;
    }
    Task(int arg, int st, int copy){
        id = arg;
        datei << "+T" << id << " ";
        zeiten = new int[3];
        zeiten[0] = st;
        zeiten[1] = copy;
        zeiten[2] = -1;
        punkte = 0;
    }
    ~Task() {
        datei << "-T" << id << " ";
        delete[] zeiten;
        zeiten = nullptr;
    }
    int GetId() const {
        return id;
    }
private:
    int id;
    //! Array, mit Startzeit, Kopierzeit, geplante Zeit
    int* zeiten;
    int punkte;
    friend Task Make1From3(Task t1, Task t2, Task t3);
};
```

Veranschaulichen Sie jeweils grafisch die Stack- und/oder Heap-Speicherbelegung in **allen** folgenden Programmfragmenten nach Ausführung der mit /* 1*/ usw. gekennzeichneten Programmzeilen.

Beachten Sie bitte auch, dass bei fast allen Aufgaben, die Ausgabe des Funktionsaufrufs in den Stream datei zu notieren ist.

a.) (3 P.) Welche Ausgabe ergibt der Aufruf von f10?

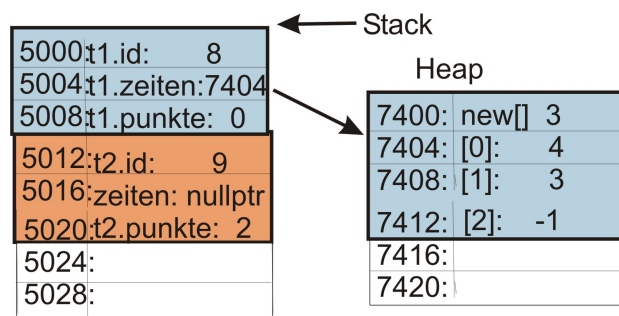
```
void f10() {
    Task t1(8, 4, 3);
    Task t2(9); /*1*/
}
```

Lösung:

```
f10
+T8 +T9 -T9 -T8
```

b.) (5 P.) Speicherbelegung zeichnen:

Lösung:



new[] 3 gibt in diesem Zusammenhang an, dass in dieser Speicherstelle für den anschließenden delete[]-Aufruf gespeichert ist, wie viel Array-Elemente mit new angefordert wurden. Für die Bewertung wird dieses aber nicht (negativ) berücksichtigt, falls es fehlen sollte.

Es muss natürlich **t2.zeiten nullptr** in Speicherstelle 5016

c.) (3 P.) Welche Ausgabe ergibt der Aufruf von f11?

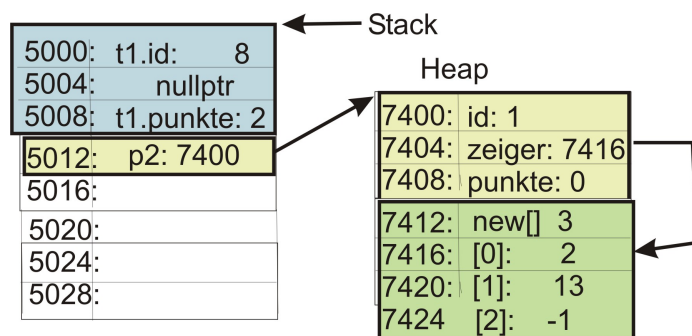
```
void f11() {
    Task t1(8);
    Task* p2 = new Task(1, 2, 13); /*1*/
}
```

Lösung:

```
f11
+T8 +T1 -T8
```

d.) (9 P.) Speicherbelegung zeichnen: Zeitpunkt /*1*/

Lösung:



In Zelle 5004 muss es natürlich **t1.zeiten nullptr** heißen. In Zelle 7404 steht die Variable zeiten und nicht zeiger.

Aufgabe 3 : Fläche und tiefe Kopie

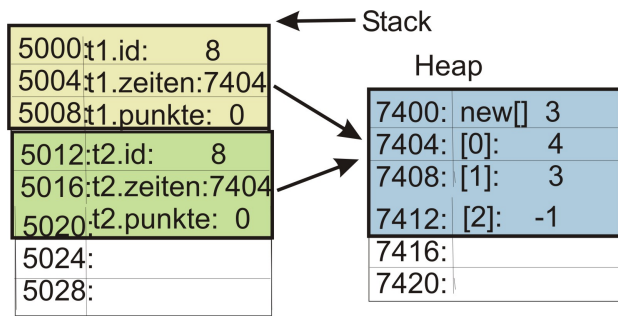
ca. 39 Punkte

a.) (5 P.) Veranschaulichen Sie die Speicherbelegung zum Zeitpunkt /*1*/ nach Aufruf von f21.

```
void f21() {
    Task t1(8, 4, 3);
    Task t2(t1); /*1*/
    datei << t2.GetId() << " ";
}
```

Speicherbelegung zeichnen: für f21:

Lösung:



b.) (5 P.) Erklären Sie mit Hilfe Ihrer Zeichnung, warum der Aufruf der Funktion `f21` zu einem Programmabsturz oder zumindest zu undefinierten Verhalten des Programms führt.

Lösung:

Aufgrund des fehlenden Kopierkonstruktors wird automatisch vom System ein Default-Kopierkonstruktor erzeugt, der die Instanzen der Klasse `Task` nur flach kopiert. Deshalb verweisen sowohl `t1.zeiten` als auch `t2.zeiten` auf den gleichen Speicherbereich im Heap. Wird eines der Elemente im Array, auf das `zeiten` verweist, bei `t1` geändert, erfolgt die Änderung auch bei `t2` und umgekehrt.

Das Hauptproblem ist in jedem Fall in diesem Mini-programm der Aufruf des Destruktors. Der Destruktor von `t2` gibt diesen Speicherbereich wieder frei. `t2.zeiten` wird auf `nullptr` gesetzt, allerdings verweist `t1.zeiten` nach wie vor auf diesen bereits wieder frei gegebenen Speicherbereich auf dem Heap. Beim Aufruf des Destruktors von `t1` wird versucht für diese Speicherbereich abermals `delete[]` aufzurufen. Dieses führt zu einem Laufzeitfehler oder zumindest zu undefiniertem Verhalten.

Zur Lösung des Problems muss somit ein Kopierkonstruktor (und auch ein Zuweisungsoperator) explizit implementiert werden, der eine tiefe Kopie durchführt.

c.) (5 P.) Erweitern Sie `Task` (mit oder ohne Ausgabe nach `datei`) direkt `inline` in der Klassenschnittstelle von `Task`, sodass die Funktion `f21` weiterhin syntaktisch korrekt ist und so, dass ihr Aufruf nun zu korrektem Verhalten führt.

Lösung:

```
Task(const Task& t2) :
    id(t2.id),
    punkte(t2.punkte) {
    datei << "+TC" << t2.id << " ";
    GetHeapSpaceFor3Ints(t2.zeiten);
}
```

Die private Methode `GetHeapSpaceFor3Int` wurde wie folgt implementiert. Sie dient zur einfachen Dokumentation der im Folgenden benötigten Dokumentation der Heapspeicheranforderungen. Für die Lösung des Problems ist diese Funktionalität aber nicht erforderlich.

```
static void IncHeapIncCntBy(int inc) {
    datei << "\n 3 on heap\n";
    heapIntCnt += inc;
}
int* GetHeapSpaceFor3Ints(){
    IncHeapIncCntBy(3);
    return new int[3];
}
void GetHeapSpaceFor3Ints(int* arg){
    zeiten = arg;
    if (arg != nullptr){
        IncHeapIncCntBy(3);
        zeiten = new int[3];
        for (int i = 0; i < 3; ++i){
            zeiten[i] = arg[i];
        } // for
    } // if
} // GetHeapSpaceFor3Ints
```

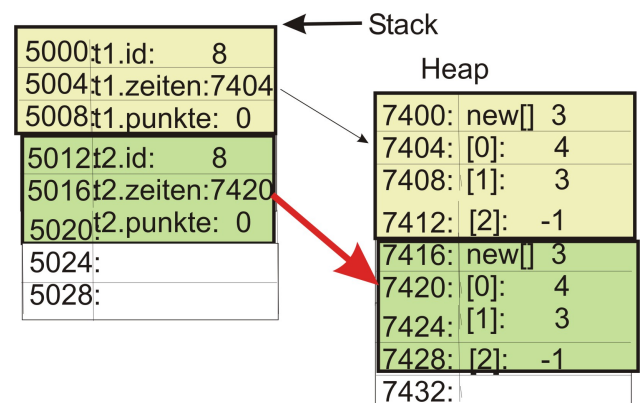
Der Zuweisungsoperator der Klasse könnte wie folgt implementiert werden:

```
Task& operator=(const Task& t2) {
    datei << "op=C" << t2.id << " ";
    int* hlp = new int[3];
    for (int i = 0; i < 3; ++i){
        hlp[i] = t2.zeiten[i];
    }
    punkte = t2.punkte;
    id = t2.id;
    delete[] zeiten;
    zeiten = hlp;
    return *this;
}
```

d.) (4 P.) Veranschaulichen Sie nun nochmals die Speicherbelegung beim Aufruf von `f21` nach Ihrer Verbesserung.

Speicherbelegung zeichnen: zum Zeitpunkt `/*1*/`:

Lösung:



Wir würden die folgenden Ausgaben erhalten (Leistung nicht in Aufgabenstellung gefordert):

```
f31
+T8
  3 on heap
+TC8
  3 on heap
8 -T8 -T8
```

e.) (6 P.) Gehen Sie im Folgenden davon aus, dass Ihre Verbesserung aus der vorherigen Teilaufgabe für die Klasse `Task` umgesetzt ist und dass Ihre Verbesserung auch eine geeignete Ausgab in den Stream `datei` erzeugt [z.B. `+TCo id`].

Das folgende Listing zeigt die Funktion `Make1From3`

```
/* Kombination von 3 Tasks zu einer indem die
Attribute addiert werden.*/
Task Make1From3(Task t1, Task t2, Task t3){
datei << "Make1From3 called\n";
Task hlp(0, -1, -1);
hlp.punkte = t1.punkte + t2.punkte + t3.punkte;
hlp.id = t1.id + t2.id + t3.id;
hlp.zeiten [0] = t1.zeiten [0] +
t2.zeiten [0] + t3.zeiten [0];
hlp.zeiten [1] = t1.zeiten [1] +
t2.zeiten [1] + t3.zeiten [1];
return hlp;
}
```

`Make1From3` wird in der Funktion `f32` aufgerufen.

```
Task t1(1, 1, 1);
Task t2(2, 2, 2);
Task t3(3, 3, 3);

Task t4 = Make1From3(t1, t2, t3);
datei << t4.GetId() << " ";
}
```

Wie viele `int`-Objekte werden durch Aufruf von `f32` bis zum Ende der Funktion auf dem Heap erzeugt, d.h. wie oft (multipliziert mit 3) wird der Code `new int [3];` ausgeführt.

Geben Sie hierzu die Ausgabe des Aufrufs von `f32` in den Stream `datei` an und beginnen Sie jeweils eine neue Zeile in Ihrer Ausgabe, sobald `new int [3];` ausgeführt wird. Ergänzen Sie dann jeweils den Text „3 `int` mit `new`“.

Hier dürfen Sie (müssen aber nicht) zur Erklärung Ihrer Überlegungen weiteren Text ergänzen. Ihre Ausgabe könnte z.B. wie folgt aussehen:

```
+T1 +T7
  3 int mit new // t1 mit 1,4,8 aufgerufen
-T7 +T11
  3 int mit new // Kopierkonstruktor
                //wird aufgerufen
1 operator= // abcde jfj
  3 int mit new
  +TCopy7
  3 int mit new
...
```

Geben Sie Ihre Ausgaben am besten auf einem Extrablatt an.

Lösung:

```
f32
+T1
  3 on heap
+T2
  3 on heap
+T3
  3 on heap
+TC3
  3 on heap
+TC2
  3 on heap
+TC1
  3 on heap
Make1From3 called
+T0
  3 on heap
+TC6
  3 on heap
-T6 -T1 -T2 -T3 6 -T6 -T3 -T2 -T1
  Created int on heap: 24
```

f.) (5 P.) Sie haben gesehen, dass eine Menge teilweise *unnötiger* `int`-Instanzen auf dem Heap erzeugt wurden.

Wie erreicht man allein durch Anpassung der Schnittstelle von `Make1From3`, dass weniger `int`-Instanzen auf dem Heap erzeugt werden? Sie dürfen davon ausgehen, dass `Make1From3` mit geänderte Schnittstelle dann auch `friend` der Klasse `Task` ist.

Lösung:

Das Problem ist, dass die Instanzen von `Task` alle als Werteparameter übergeben wurden. Bei Übergabe als Referenzparameter werden keine Kopien erzeugt und damit werden keine Instanzen von `int` auf dem Heap erzeugt.

```
/* Kombination von 3 Tasks zu einer indem die
Attribute addiert werden.*/
Task Make1From3(const Task& t1,
const Task& t2, const Task& t3){
datei << "Make1From3 called\n";
Task hlp(0, -1, -1);
hlp.punkte = t1.punkte + t2.punkte + t3.punkte;
hlp.id = t1.id + t2.id + t3.id;
hlp.zeiten [0] = t1.zeiten [0] +
t2.zeiten [0] + t3.zeiten [0];
hlp.zeiten [1] = t1.zeiten [1] +
t2.zeiten [1] + t3.zeiten [1];
return hlp;
}
```

g.) (2 P.) Wie viele `int`-Instanzen werden nun insgesamt noch erzeugt?

Lösung:

Es werden nun nur noch 5 mal 3 `int`-Instanzen erzeugt, d.h. 15 Instanzen von `int`.

```
f33
+T1
  3 on heap
+T2
  3 on heap
+T3
  3 on heap
Make1From3 called
+T0
  3 on heap
+TC6
  3 on heap
-T6 6 -T6 -T3 -T2 -T1
  Created int on heap: 15
```

h.) (5 P.) Erweitern Sie nun zusätzlich die Klassenschnittstelle von `Task` um einen Kopierkonstruktor mit Move-Semantik (die *Dinger* mit `&&`-Parametern sind gemeint), sodass bei Aufruf von `f32` noch weniger `int`-Instanzen auf dem Heap erzeugt werden. Sie dürfen wiederum direkt in der Klassenschnittstelle von `Task` in der Header-Datei implementieren.

Lösung:

```
Task(Task&& t2) :
  id (t2.id),
  zeiten (t2.zeiten),
  punkte(t2.punkte) {
  datei << "+TM" << id << " ";
  t2.zeiten = nullptr;
}
```

Zwar nicht gefordert, aber es bietet sich an, auch gleich einen Zuweisungsoperator mit Move-Semantik zu implementieren:

```
Task& operator=(Task&& t2) {
  datei << "op=M" << t2.id << " ";
  if (this != &t2){
    zeiten = t2.zeiten;
    punkte = t2.punkte;
    id = t2.id;
    t2.zeiten = nullptr;
  }
  return *this;
}
```

i.) (2 P.) Wie viele `int`-Instanzen werden nun insgesamt noch erzeugt?

Lösung:

Es werden nun nur noch 4 mal 3 `int`-Instanzen erzeugt, d.h. 14 Instanzen von `int`.

```
f43
+T1
  3 on heap
+T2
  3 on heap
+T3
  3 on heap
Make1From3 called
+T0
  3 on heap
+TM6 -T6 6 -T6 -T3 -T2 -T1
  Created int on heap: 12
```

Eigentlich braucht man ja nur drei Instanzen von `Task`, die Heapspeicher belegen. Die anderen Objekte können daraus erzeugt werden. Wir kämen mit 3 mal 3 Instanzen von `int` auf dem Heap aus, wenn wir beim Aufruf von `Make1From3` nur temporäre Objekte erzeugen würden, da wir ja nur mit dem addierten Objekte weiter arbeiten wollen:

```
void f44() {
  Task t4 = Make1From3(Task(1, 1, 1),
    Task(2, 2, 2), Task(3, 3, 3));
  datei << t4.GetId() << " ";
}
```

`Make1From3` ist dann ebenfalls mit Move-Semantik zu implementieren:

```
Task Make1From3(Task&& t1,
  const Task&& t2, const Task&& t3){
  datei << "Make1From3 called\n";
  Task hlp(move(t1));
  hlp.punkte = hlp.punkte + t2.punkte + t3.punkte;
  hlp.id = t1.id + t2.id + t3.id;

  hlp.zeiten[0] = hlp.zeiten[0] +
    t2.zeiten[0] + t3.zeiten[0];
  hlp.zeiten[1] = hlp.zeiten[1] +
    t2.zeiten[1] + t3.zeiten[1];
  return hlp;
}
```

Ganz wichtig ist in diesem Zusammenhang aber der `move`-Befehl beim Aufruf des Kopierkonstruktors in `Task hlp(move(t1));`. Der Compiler weiß hier ja nicht, dass `t1` anschließend nicht mehr gebraucht wird. `t1` darf dann natürlich auch nicht mehr verwendet werden, sondern es muss über `hlp` auf die vorherigen Werte davon zugegriffen werden. In diesem Fall kommen wir also mit 3 mal 3 Instanzen von `int` auf dem Heap aus,

```
f44
+T3
  3 on heap
+T2
  3 on heap
+T1
  3 on heap
Make1From3 called
+TM1 +TM6 -T6 -T1 -T2 -T3 6 -T6
  Created int on heap: 9
```

Aufgabe 4 : Shared Pointer

ca. 26 Punkte

a.) (3 P.) Welche Ausgabe ergibt der Aufruf von f12?

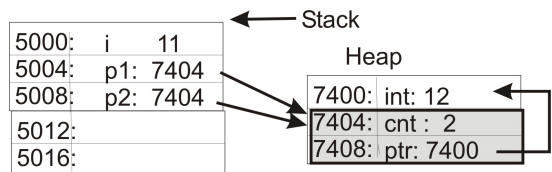
```
void f12() {
  shared_ptr<Task> p1(new Task(5, 6, 7)); /*1*/
  datei << p1->GetId() << " ";
}
```

Lösung:

```
f12
+T5 5 -T5
```

b.) (10 P.) Veranschaulichen Sie nun grafisch wiederum die Speicherbelegung zum Zeitpunkt /*1*/. Als Hilfe die Speicherbelegung bei Aufruf von `hilfe` zum Zeitpunkt /*2*/ kann die folgenden Grafik dienen:

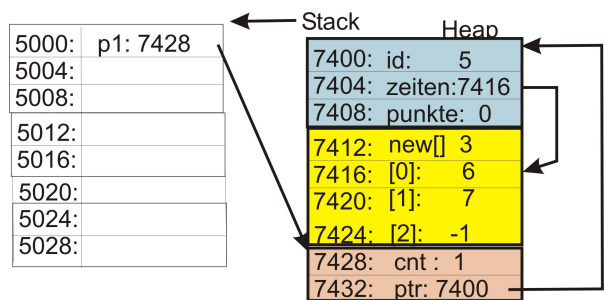
```
void hilfe () {
  int i(11);
  shared_ptr<int> p1(new int(12));
  shared_ptr<int> p2(p1); /*2*/
}
```



Als Hilfe sind schon einige Einträge vorgegeben.

Speicherbelegung zeichnen:(Zeitpunkt /*1*/ bei f12):

Lösung:



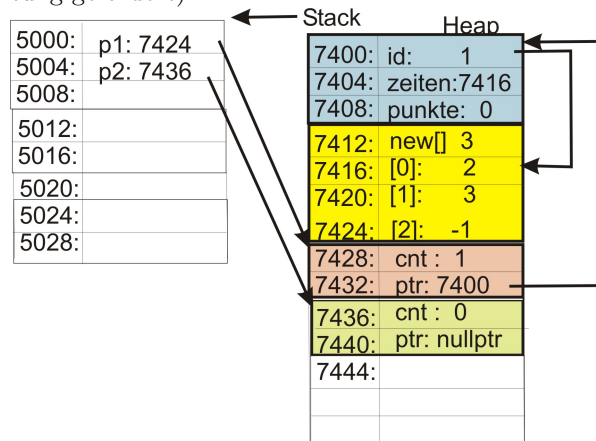
c.) (10 P.) Sie sehen hier die Funktion `f13`. Wie sehen die Speicherbelegungen zum Zeitpunkt /*2*/ und /*4*/ aus? Kennzeichnen Sie grafisch oder textlich, welche Speicherbereiche auf dem Heap bereits wieder freigegeben sind. Es bietet sich an, auch die Belegungen zu den Zeitpunkten /*1*/ und /*3*/ grob zu veranschaulichen (auch wenn es dafür keine Punkte gibt).

```
void f13() {
  shared_ptr<Task> p1 (new Task(1,2,3));
  shared_ptr<Task> p2; /*1*/
  if (p1->GetId() < 100) {
    shared_ptr<Task> p3(new Task(12));
    /*2*/
    p2 = p3; /*3*/
    p1 = p3; /*4*/
  }
  datei << p1->GetId() << " ";
}
```

Ganz wenige Einträge sind zur Orientierung, welches Objekt wohl zuerst erzeugt wird, sind schon vorgegeben.

Lösung:

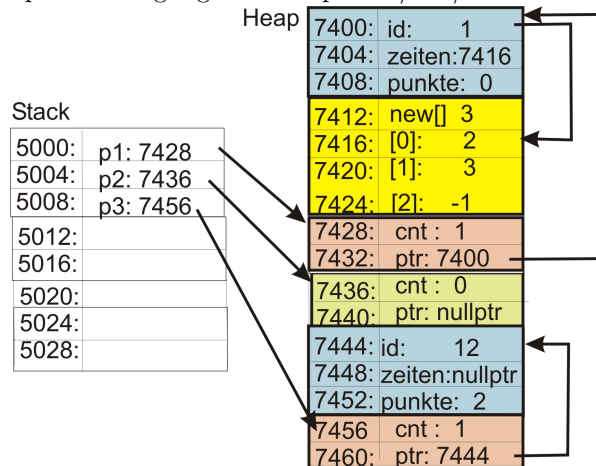
Speicherbelegung zum Zeitpunkt /*1*/ (nicht als Leistung gefordert).



Speicherbelegung zeichnen:zum Zeitpunkt /*2*/

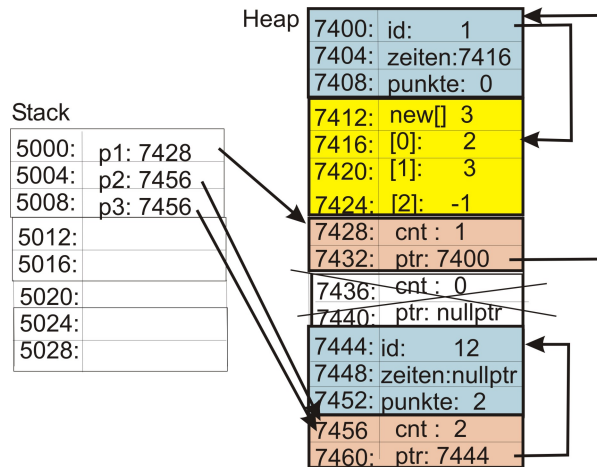
Lösung:

Speicherbelegung zum Zeitpunkt /*2*/.

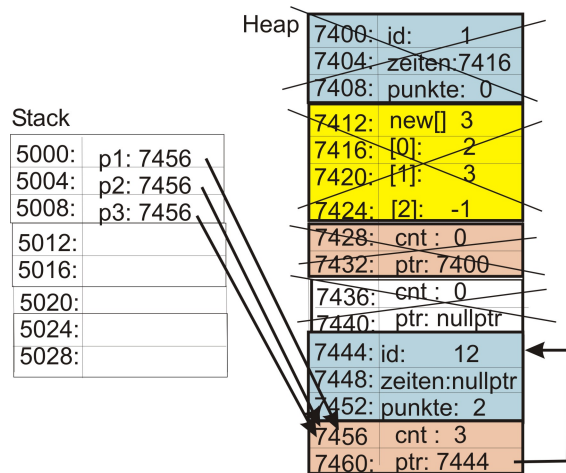


Lösung:

Speicherbelegung zum Zeitpunkt /*3*/ (nicht als Leistung gefordert).



Speicherbelegung zeichnen:zum Zeitpunkt /*4*/
Lösung:



d.) (3 P.) Mit diesen Vorüberlegungen sollte Ihnen nun die Ausgabe des Funktionsaufrufs von f13 in den Stream `datei` einfach gelingen. Wie sieht die Ausgabe aus?

Lösung:

```
f13
+T1 +T12 -T1 12 -T12
```