Clicker-Aufgaben zu Zeiger- und Wertesemantik

Im Folgenden finden Sie die Aufgaben aus der Vorlesung zur Übergabe einer Referenz, eines Zeigers oder eines Wertes an einen Funktionsparameter.

Drucken Sie die Aufgaben aus und verdecken jeweils zunächst den unteren Teil und versuchen die richtige Antwort durch Nachdenken und nicht durch Abgucken zu finden.
Alternativ können Sie die Aufgabe auch mit Power Point ablaufen lassen. Erst durch weiteres Clicken erscheint die Lösung.

Die grün markierten Teile sind die richtigen Antworten.

Fehlermeldungen bitte umgehend an Hartmut.Helmke@dlr.de.

Syntax / Semantik

Wert und Referenz (&) sind das gleiche (wenn es rein um die Syntax geht). Semantisch sind sie etwas völlig anderes

Ausgangstyp	Zieltyp	Operator/Zeichen
Wert/Referenz	Zeiger	&
Zeiger	Wert / Referenz	*
Sonst		Kein Operator, Passt schon

Zeiger und Referenz führen beide zum gleichen Assemblercode, aber mit unterschiedlicher Syntax im C++-Code. Deshalb sind beide im Sinne von Ausgangsparameter verwendbar. Das Original-Objekt wird manipuliert.

Merksätze

Syntaktisch werden Referenzen und Werte gleich verwendet.
Im einem Fall liegt jedoch nur eine Instanz vor (bei Referenzen), im anderen Fall liegen jedoch zwei Instanzen (zwei Werte) vor.

Semantisch führen dagegen Referenzen und Zeiger zum gleichen Ergebnis (sie erzeugen auch den gleichen Assembler-Code).

Wenn Quelle und Ziele beide nicht vom Typ Zeiger sind (also Referenz oder Wert), muss man nicht machen.

Clicker: Noch mal ganz langsam

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
Vorgang v;
// Übergabe eines Zeigers:
Vorgang* zeiger = new Vorgang();
// oder
Vorgang* zeiger = &v;
```

```
fzeiger(& zeiger);  // 1
fzeiger(   zeiger);  // 2
fzeiger(*  zeiger);  // 3
   //4 keine Ahnung
```

Noch mal ganz langsam (2)

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
Vorgang v;
// Übergabe eines Zeigers:
Vorgang* zeiger = new Vorgang;
// oder
Vorgang* zeiger = &v;
```

```
fref(& zeiger);  // 1
fref(   zeiger);  // 2
fref(*  zeiger);  // 3
   //4 keine Ahnung
```

Noch mal ganz langsam (3)

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
Vorgang v;
//Übergabe eines Zeigers:
Vorgang* zeiger = new Vorgang;
// oder
Vorgang* zeiger = &v;
```

```
fwert(& zeiger);  // 1
fwert(   zeiger);  // 2
fwert(*  zeiger);  // 3
   //4 keine Ahnung
```

Erklärungen

Syntaktisch werden Referenzen wie Werte behandelt.
Das Ergebnis ist jedoch wie bei Zeigern
Syntaktisch wie Werte, semantisch wie Zeiger.

```
void fref(Vorgang&  fref) {
   fref.dauer = 17;
   // nicht; denn syntaktisch wie Werte
   fref-> dauer = 17;  // Syntaxfehler
   }
```

```
void fref(Vorgang&  fref) {
   fref.dauer = 17;
}
int main() {
   Vorgang v;     v.dauer = 62;
   fref(v);  // nicht fref(&v);
   cout << v.dauer; // Ausgabe ist 17 nicht 62, nicht wie bei CallByValue
}
```

Noch mal ganz langsam (4)

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
Vorgang v;
```

```
fzeiger(& v);  // 1
fzeiger(   v);  // 2
fzeiger(*  v);  // 3
   //4 keine Ahnung
```

Noch mal ganz langsam (5)

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
Vorgang v;
```

```
fwert(& v);  // 1
fwert(   v);  // 2
fwert(*  v);  // 3
   //4 keine Ahnung
```

Es wird ein „ganzer“ Vorgang erwartet.
v ist bereits ein Vorgang, also keine Umwandlung, d.h. //2 ist richtig.

Ergebnis:
__ fwert(& v); //1 _-_ fwert(v); // 2
__ fwert(* v); // 3 __ //4 keine Ahnung

Noch mal ganz langsam (6)

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
Vorgang v;
```

```
fref(& v);  // 1
fref(   v);  // 2
fref(*  v);  // 3
   //4 keine Ahnung
```

fr ist Ausgangsparameter der Funktion fref (&).
fr ist eine Referenz, d.h. die Adresse eines Vorgangs.
Die Umwandlung eines Vorgangs in eine Adresse übernimmt jedoch der Compiler. Deshalb ist ein ganzer Vorgang zu übergeben,, d.h. //2 ist richtig.

Ergebnis:
__ fref(& v); //1 _-_ fref(v); // 2
__ fref(* v); // 3 __ //4 keine Ahnung

Noch mal ganz langsam (7): Übergabe von Referenzen

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
// Immer der gleiche Aufruf
fwert(& vref);  // 1
fwert(   vref);  // 2
fwert(*  vref);  // 3
  //4 keine Ahnung
```

```
void funk(Vorgang& vref) {
  fwert(??? vref);
}
Vorgang v;
Vorgang& vref = v;
fwert(? vref);

Vorgang* vz = &v;
Vorgang& vref = *(vz);
fwert(??? vref);
```

fw ist ein ganzer Vorgang.
vref ist zwar selbst eine Adresse, wird aber syntaktisch wie ein ganzer Vorgang verwendet, d.h. die Umwandlung übernimmt der Compiler, d.h. //2 ist richtig.

Ergebnis:

```
___  fwert(& vref); //1    _-_ fwert(   vref);  // 2
___  fwert(*  vref);  // 3 ___   //4 keine Ahnung
```

Noch mal ganz langsam (8): Übergabe von Referenzen

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
// Immer der gleiche Aufruf
fref(& vref);  // 1
fref(   vref);  // 2
fref(*  vref);  // 3
  //4 keine Ahnung
```

```
void funk(Vorgang& vref) {
  fref(??? vref);
}
Vorgang v;
Vorgang& vref = v;
fref(? vref);

Vorgang* vz = &v;
Vorgang& vref = *(vz);
fref(??? vref);
```

vref und fr sind beides Referenzen. Keine Umwandlung erforderlich, d.h. //2 ist richtig.

Ergebnis:

```
___  fref(& vref); //1    _-_ fref(   vref);  // 2
___  fref(*  vref);  // 3  ___   //4 keine Ahnung
```

Noch mal ganz langsam (9): Übergabe von Referenzen

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
// Immer der gleiche Aufruf
fzeiger(& vref);  // 1
fzeiger(   vref);  // 2
fzeiger(*  vref);  // 3
  //4 keine Ahnung
```

```
void funk(Vorgang& vref) {
  fzeiger(??? vref);
}
Vorgang v;
Vorgang& vref = v;
fzeiger(? vref);

Vorgang* vz = &v;
Vorgang& vref = *(vz);
fzeiger(??? vref);
```

fp ist ein Zeiger, d.h. es muss eine Adresse eines Vorgangs übergeben werden. vref ist zwar selbst eine Adresse, wird aber syntaktisch wie ein ganzer Vorgang verwendet, d.h. es ist der Adressoperator (&) erforderlich d.h. //1 ist richtig.

Ergebnis:

```
_-_  fzeiger(& vref); //1    __ fzeiger(   vref);  // 2
__  fzeiger(*  vref);  // 3  __   //4 keine Ahnung
```

Clicker-“Abstimmung“

```
void f(Vorgang* fp,  Vorgang fv)
{ … }
```

```
int main() {
   Vorgang* pv = new Vorgang();
   Vorgang v;

 f(*v, &pv);  // 1
 f(pv, v);      //  2
 f(&pv, v);    //3
 f(&v, pv);  // 4
```

Ergebnis:

__ f(*v, &pv); __ f(pv, v); __ f(&pv, v); __ f(&v, *pv);

Clicker-“Abstimmung“

```
void f(Vorgang* fp,  Vorgang fv)
{ … }
```

```
int main() {
   Vorgang* pv = new Vorgang();
   Vorgang v;

f(pv, &v);      // 1
f(&v, *pv);  // 2
Beides falsch // 3
Keine Ahnung  // 4
```

Ergebnis:

__ f(pv, &v); __ f(&v, *pv); __ Beides falsch __ Keine Ahnung

Clicker

Welche Ausgabe liefert der Aufruf des Hauptprogramms auf den Bildschirm?

```
#include <iostream>
#include <string>
using namespace std;
void g(string s1, string s2) {
   s1 = "Java ";
   s2 = "ist gut?";}
void h(string& s1, string& s2) {
   s1 = "C++ ";
   s2 = "ist schwierig?";
}
```

```
int main(){
   string m1("Horst ");
   string m2 = "Helmut";
   g(m1, m2);
   cout << m1 << m2 << endl;
   m1= "Anke ";
   m2= "Mei";
   h(m1, m2);
   cout << m1 << m2 << endl;
}
```

1. Horst HelmutAnke Mei
2. Horst HelmutC++ ist schwierig?
3. Java ist gut?C++ ist schwierig?
4. Java ist gut?AnkeMei

Ergebnis:

__ Horst HelmutAnke Mei __ Horst HelmutC++ ist schwierig?

__ Java ist gut?C++ ist schwierig? __Java ist gut?AnkeMei

Zeichnen

1

5000:	j	30
5004:	d	7.5
5008:		
5012:	pj	91

2

5000:	j	91
5004:	d	7.5
5008:		
5012:	pj	5000

3

5000:	j	30
5004:	d	7.5
5008:	pj	5000
5012:		

4

5000:	j	30
5004:	d	7.5
5008:		
5012:	pj	5000

```
void func1() {
   int j = 30;
   double d = 7.5;
   int* pj= &j;
   *pj = 91; /* 1 */
}
```

Zu verstehen ist:
pj enthält eine Adresse (z.B. 5000), aber nicht 91.
pj ist Adresse, aber j wird über *pj verändert:

Ergebnis:
__ 1 __ 2 __3 __ 4

Stack und Heapspeicher

```
Vorgang* v = new Vorgang();
```

Welche Aussage ist richtig?

1. v liegt auf dem Stack und verweist auf den Heap.
2. v liegt auf dem Heap und verweist auf den Heap.
3. v liegt auf dem Stack und verweist auf den Stack.
4. v liegt auf dem Heap und verweist auf den Stack.
5. Keine Ahnung

Ergebnis:

v liegt auf dem Stack und verweist auf den Heap.

v liegt auf dem Heap und verweist auf den Heap.

v liegt auf dem Stack und verweist auf den Stack.

v liegt auf dem Heap und verweist auf den Stack.

Keine Ahnung

Stack und Heapspeicher

```
Vorgang meinVorg;
Vorgang* v = & meinVorg;
```

Welche Aussage ist richtig?

1. v liegt auf dem Stack und verweist auf den Heap.
2. v liegt auf dem Heap und verweist auf den Heap.
3. v liegt auf dem Stack und verweist auf den Stack.
4. v liegt auf dem Heap und verweist auf den Stack.
5. Keine Ahnung

Ergebnis:
v liegt auf dem Stack und verweist auf den Heap.
v liegt auf dem Heap und verweist auf den Heap.
v liegt auf dem Stack und verweist auf den Stack.
v liegt auf dem Heap und verweist auf den Stack.
Keine Ahnung

Stack und Heapspeicher

```
void funk(Vorgang& v) {…}
…
Vorgang* v1 = new Vorgang();
funk(* v1); // das wo v1 hinzeigt
```

Welche Aussage ist richtig?

1. v (aus funk) liegt auf dem Stack und verweist auf den Heap.
2. v liegt auf dem Heap und verweist auf den Heap.
3. v liegt auf dem Stack und verweist auf den Stack.
4. v liegt auf dem Heap und verweist auf den Stack.
5. Keine Ahnung

Ergebnis:

v liegt auf dem Stack und verweist über v1 auf dem Stack auf den Heap.

v liegt auf dem Heap und verweist auf den Heap.

v liegt auf dem Stack und verweist auf den Stack.

v liegt auf dem Heap und verweist auf den Stack.

Keine Ahnung