

Übungspaket 7

Konstrukturen in C++

Übungsziele:

1. Wie Konstrukturen aufgebaut sind
2. Initialisierungsliste
3. Wann und in welchem Konstrukturen aufgerufen werden
4. Welche Konstrukturen es gibt
5. Kopieroperator

Literatur:

C++-Skript¹, Kapitel 26

Semester:

Wintersemester 2020/21

Betreuer:

Theo und Ralf

Synopsis:

In C++ spielen Konstrukturen eine zentrale Rolle. Sie werden an vielen Stellen, oft automatisch, aufgerufen und können uns in unterschiedlicher Gestalt begegnen. In diesem Übungspaket wiederholen wir die einzelnen Aspekte und üben sie mittels einiger praktischer Beispiele ein.

¹www.amd.e-technik.uni-rostock.de/ma/rs/lv/hoqt/script.pdf

Teil I: Stoffwiederholung

Aufgabe 1: Aufgabe eines Konstruktors/Destruktors

Erkläre mit eigenen Worten, was die Aufgabe eines Konstruktors ist.

Erkläre mit eigenen Worten, was die Aufgabe eines Destruktors ist.

Aufgabe 2: Unterschiedliche Objekt-/Variablenarten

In Programmiersprachen wie C und C++ kann man Objekte und Variablen auf zwei unterschiedliche Arten anlegen. Wie nennt man die entsprechenden Mechanismen?

1. 2.

Aufgabe 3: Konstruktoraufrufe

Beschreibe mit eigenen Worten, wann und wie ein Konstruktor aufgerufen wird. Unterscheide dabei die beiden in der vorherigen Aufgabe erwähnten Objektarten.

Aufgabe 4: Objekt erzeugen und freigeben

In [2. Aufgabe](#) haben wir besprochen, welche Arten von Objekten angelegt werden können. Trage die entsprechenden Begriffe in den Kopf der folgenden Tabelle ein und vervollständige diese. Erläutere dies anhand von Beispielen mit der gegebenen Klasse C.

In welchem Segment liegen sie?
Wie werden sie erzeugt?
Wie werden sie freigegeben?
Wer erzeugt die Konstruktoraufrufe?
Wer erzeugt die Destruktoraufrufe?

Aufgabe 5: Eigenschaften von Kon- und Destruktoren?

Welchen Rückgabebetyp hat ein Konstruktor?	<input type="text"/>
Wie viele Parameter kann ein Konstruktor haben?	<input type="text"/>
Welchen Rückgabebetyp hat ein Destruktor?	<input type="text"/>
Wie viele Parameter kann ein Destruktor haben?	<input type="text"/>

Aufgabe 6: Aufbau von Konstruktoren

Beschreibe kurz mit eigenen Worten, aus welchen Bestandteilen Konstruktoren und Destruktoren bestehen können.

Aufgabe 7: Typen von Konstruktoren

Welcher Konstruktor wird in den folgenden Anweisungen aufgerufen, wenn `C` eine bereits definierte Klasse und `obj_1`, `obj_2` und `obj_3` entsprechende Objekte sein sollen.

- | | |
|---|----------------------|
| 1. <code>C obj_1(...);</code> | <input type="text"/> |
| 2. <code>C obj_2 = obj_1;</code> | <input type="text"/> |
| 3. <code>C obj_3; obj_3 = obj_1;</code> | <input type="text"/> |

Wer wählt den richtigen Konstruktor aus?

Im Folgenden stellen wir einige Fragen zu den jeweiligen Konstruktortypen. Beantworte diese und nimm **CLASS** als Name einer Beispielklasse.

„Normale“ Konstruktoren:

Wie werden sie definiert	<input type="text"/>
Wie viel Parameter können sie haben?	<input type="text"/>
Welche Signatur haben sie?	<input type="text"/>

Kopiekonstruktoren:

Wie werden sie definiert	<input type="text"/>
Welche Signatur haben sie?	<input type="text"/>
Wie viel Parameter können sie haben?	<input type="text"/>
Können wir weitere Parameter angeben?	<input type="text"/>

Zuweisungsoperatoren:

Wie werden sie definiert	<input type="text"/>
Welche Signatur haben sie?	<input type="text"/>
Wie viel Parameter können sie haben?	<input type="text"/>
Können wir weitere Parameter angeben?	<input type="text"/>

Woher weiß der Compiler, welchen Konstruktor er in einem konkreten Fall aufrufen soll?

Aufgabe 8: Konstruktoraufrufe

Welchen Namen hat der Konstruktor der Klasse C ?	<input type="text"/>
Über welchen Namen wird der Konstruktor der Klasse C aufgerufen?	<input type="text"/>
Was wird bei einem Konstruktoraufruf angegeben?	<input type="text"/>
Kann man den Zeitpunkt eines Konstruktoraufrufes selbst bestimmen?	<input type="text"/>

Begründe deine letzte Antwort:

Teil II: Quiz

Aufgabe 1: Initialisierungslisten

Betrachte folgendes Programm

```
1  #include <stdio.h>
2
3  class C {
4      private:
5          int i;
6          int j;
7      public:
8          C( int a, int b );
9          void print();
10 };
11
12 C::C( int a, int b )
13     : j( b + i ), i( a )
14     {
15     }
16
17 void C::print()
18     {
19         printf( "i = %d, j = %d\n", i, j );
20     }
21
22 int main( int argc, char **argv )
23     {
24         C c( 3, 4 );
25         c.print();
26         return 0;
27     }
```

Nehmen wir an, dass der Arbeitsspeicher jungfräulich sei, sodass der gesamte Stack mit Nullen initialisiert ist.

Wie lautet die Ausgabe des Programms?

In welcher Reihenfolge werden `j(b + 1)` und `i(a)` ausgeführt?

Warum ist dies so?

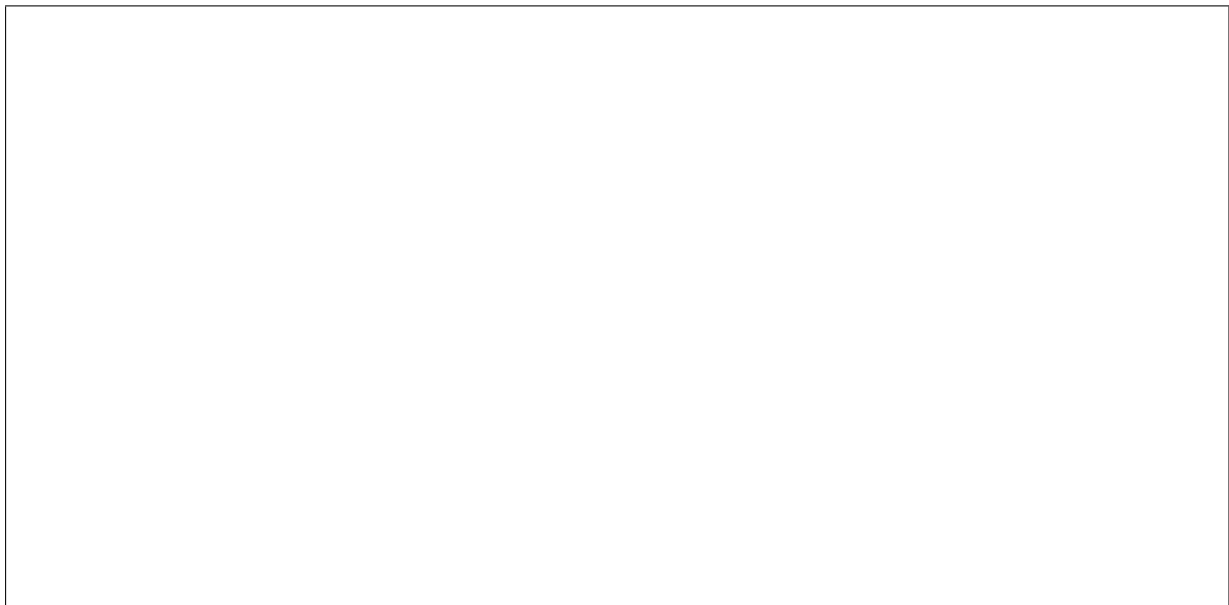
Aufgabe 2: Kopieren von Objekten

Nehmen wir an, wir hätten folgendes Programm:

```
1  #include <stdio.h>
2
3  class MEM {
4      private:
5          int  size;
6          int *mem;
7      public:
8          MEM( int size );
9      };
10
11 MEM::MEM( int size )
12 {
13     mem = new int[ this->size = size ];
14 }
15
16 int main( int argc, char **argv )
17 {
18     MEM *m1 =new MEM( 3 );
19     MEM *m2 = m1;
20     return 0;
21 }
```

Welches Problem entsteht hier?

Illustriere die Speicherbelegung, die sich am Ende von Zeile 18 ergibt.



Welche Signatur hat der Kopierkonstruktor der Klasse `MEM`?

Programmiere einen Kopierkonstruktor, der den eben besprochenen Nachteil eliminiert.



Teil III: Fehlersuche

Aufgabe 1: Konstruktoren im fehlerhaften Gebrauch

DR. HOCHBAU ist eigentlich Bauingenieur und beschäftigt sich beruflich mit der Konstruktion von Häusern. Als er in einem Beitrag laß, dass C++ auch Konstruktoren hat, fühlte er sich angesprochen und probierte ein paar Sachen aus. Hier sein aktueller Programmentwurf:

```
1  #include <stdio.h>
2
3  class TEST {
4      private:
5          int i;
6      public:
7          TEST() : i( 4711 ) {}
8          TEST( int i ) : i( i ) {}
9          TEST( TEST * source ){ this->i = 22; }    // copy
10     };
11
12  class TRY {
13      private:
14          int i;
15      public:
16          TRY( int i = 4711 ) { i( i ); }
17          ~TRY(){ printf( "destructor\n" ); }
18     };
19
20  int main( int argc, char **argv )
21  {
22      TEST t1();
23      TEST t2 = t1;
24      t2( 24 );
25      t2 = TEST( 42 );
26      TRY xtry();
27      xtry.~TRY;
28      return 0;
29  }
```

In seinem Testprogramm hat DR. HOCHBAU die beiden Klassen `TEST` und `Try` definiert, die drei bzw. einen Konstruktor haben. Zusätzlich die Klasse `TRY` noch einen Destruktor. Der Konstruktor in Zeile 9 soll ein Kopierkonstruktor sein. Aus Platzgründen haben wir alle Konstruktoren als inline-Code gesetzt. Da DR. HOCHBAU noch nicht alles richtig verstanden hat, schüttet der Compiler diverse Fehlermeldungen aus, die ihr hier finden und korrigieren sollt.

Teil IV: Anwendungen

Aufgabe 1: Kopierkonstruktor und Zuweisungsoperator

Im Anwendungsteil von Übungspaket 5 haben wir einen Puffer entwickelt, der auf Anforderung wachsen kann, sodass er niemals überläuft. Im Rahmen dieser Aufgabe haben als Lehrteam folgende Lösung entwickelt:

cbuf.h:

```
1  class CBUF {
2      private:
3          int len;           // length incl. '\0'
4          int size;         // total size of buf
5          char *buf;        // the acutal buffer
6
7      public:
8          CBUF ();           // constructor
9          ~CBUF ();         // destructor
10         void  cb_reset ();
11         char *cb_addc ( char c );
12         char *cb_addstr( const char *str );
13         char *cb_buf      ();
14     };
```

Im Kern hat jeder Puffer ein Array `buf`, das dynamisch angelegt wird. In der Methode `cb_addc` wird zunächst überprüft, ob noch Platz in diesem Array ist. Sollte dies nicht der Fall sein, wird zunächst ein neues, etwas größeres Array dynamisch angelegt und der bereits existierende Inhalt von `buf` dorthin kopiert. Anschließend werden die Arrays umgehängt und das alte wieder freigegeben.

Ein Blick auf die Klassendefinition zeigt, dass wir weder einen Kopierkonstruktor noch einen Zuweisungsoperator haben. Dies bedeutet, dass in den folgenden drei Objektdefinitionen, alle Objekte auf dem gleichen Puffer (Array `cbuf`) arbeiten:

```
CBUF buf_1;
CBUF buf_2 = buf_1;           // copy constructor
CBUF buf_3; buf_3 = buf_1;    // assign operator
```

Dieser Effekt ist aus unserer Sicht nicht erwünscht. Wir wollen, dass beim Kopieren bzw. Zuweise der aktuelle Stand des Puffers zunächst kopiert, dann aber unabhängig vom Original weiter verwendet werden kann. Daher besteht eure Aufgabe darin, einen entsprechenden Kopierkonstruktor bzw. Zuweisungsoperator zu implementieren.

