Übungspaket 7

Konstruktoren in C++

Übungsziele:

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

Literatur:

C++-Skript[1], Kapitel 26

Semester:

Wintersemester 2021/22

Betreuer:

Theo und Ralf

Synopsis:

In C++ spielen Konstruktoren 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.

[1] www.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.

Die Aufgabe eines Konstruktors ist es, die einzelnen Attribute des Objektes (Komponenten des `structs`) mit sinnvollen Werten zu initialisieren. Dabei geht der Konstruktor davon aus, dass der Speicher für die Objekthülle, also dem `struct`, bereits angelegt ist. Weitere, (mittels Zeigern) dynamisch verknüpfte Objekte muss der Konstruktor eigenständig anlegen.

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

Die Aufgabe eines Destruktors ist es, ein Objekt zu „schließen“. Dazu gehören beispielsweise das Schließen einer offenen Datei (File Pointer) sowie das Freigeben verknüpfter, zuvor dynamisch angelegter Objekte. Die eigentliche Datenstruktur (`struct`, Objekthülle) wird vom Destruktor nicht freigegeben.

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 entsprecehnden Mechanismen?

1. Automatisch angelegte Objekte
2. Dynamisch angelegte Objekte

Aufgabe 3: Konstruktoraufrufe

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

Ein Konstruktor wird immer dann aufgerufen, wenn eine Variable (Objekt) oder ein Attribut angelegt wird, wird ein entsprechender Konstruktor aufgerufen. dann aufgerufen, wenn eingerichtet wird. Nehmen wir folgende Klasse an: `class X { int i; int j; }`. Bei der Anweisung `X x` werden *automatisch* drei Konstruktoraufrufe erzeugt, einer für das Objekt `x` und je einer für die Attribute `i` und `j`. Bei dynamsich erzeuten Objekten werden die Konstruktoren durch folgende Anweisung aktiviert: `X *x = new X( ... ).`

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. Erläuter dies anhand von Beispielen mit der gegebenen Klasse `C`.

	Automatische Objekte	Dynamische Objekte
In welchem Segment liegen sie?	Stack	Heap
Wie werden sie erzeugt?	`C objekt( ... )`	`C *ptr = new C( ... )`
Wie werden sie freigegeben?	Durch das Methoden- bzw. Blockende	`delete ptr`
Wer erzeugt die Konstruktoraufrufe?	Der Compiler	Der Programmierer
Wer erzeugt die Destruktoraufrufe?	Der Compiler	Der Programmierer

Aufgabe 5: Eigenschaften von Kon- und Destruktoren?

Welchen Rückgabetyp hat ein Konstruktor?	Keinen, gar keinen!
Wie viele Parameter kann ein Konstruktor haben?	Beliebig viele.
Welchen Rückgabetyp hat ein Destruktor?	Keinen, gar keinen!
Wie viele Parameter kann ein Destruktor haben?	Keinen!

Aufgabe 6: Aufbau von Konstruktoren

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

Ein Konstruktor besteht aus dem Klassennamen, einer Parameterliste und einem Anweisungsblock. Im Anweisungsblock können beliebige Anweisungen sehen. Zwischen Parameterliste und Anweisungsblock kann optional eine Initialisierungsliste stehen, die mit einem einzelnen Doppelpunkte eingeleitet wird. In dieser Initialisierungsliste können Konstruktoren für die einzelnen Attribute eines Objektes aufgerufen werden; in dieser Initialisierungsliste befindet man sich noch im Aufbau (dem Anlegen) des Objektes, das mit Beginn des Anweisungsblockes beendet ist.

Ein Destruktor besteht aus der Tilde, dem Klassenname und dem Anweisungsblock. Destruktoren haben *keine* Parameter.

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. `C obj_1( ... );`	Einer der „normalen“ Konstruktoren
2. `C obj_2 = obj_1;`	Der Kopierkonstruktor
3. `C obj_3; obj_3 = obj_1;`	Der Zuweisungsoperator „=“

Wer wählt den richtigen Konstruktor aus? Der Compiler bzw. die Sprachbeschreibung.

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	`CLASS( Parameterliste )`
Wie viel Parameter können sie haben?	Beliebig viele, je nach Anforderung.
Welche Signatur haben sie?	Je nach Parameterliste.

Kopiekonstruktoren:

Wie werden sie definiert	`CLASS( CLASS & Objekt )`
Welche Signatur haben sie?	`CLASS &`
Wie viel Parameter können sie haben?	*Nur* das Quellobjekt.
Können wir weitere Parameter angeben?	Nein.

Zuweisungsoperatoren:

Wie werden sie definiert	`CLASS & operator= ( CLASS & Objekt )`
Welche Signatur haben sie?	`CLASS &`
Wie viel Parameter können sie haben?	*Nur* das Quellobjekt.
Können wir weitere Parameter angeben?	Nein.

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

Zunächst entscheidet der Compiler anhand der jeweiligen Antwort. Sofern es sich nicht um das erstellen einer Kopie oder einer Zuweisung handelt, wählt er immer einen „normalen“ Konstruktor aus. Den konkreten Konstruktor wählt der anhand der aktuellen Parameter und den zur Verfügung stehenden Signaturen aus.

In den beiden Fällen Kopierkonstruktor und Zuweisungsoperator verlangt der Compiler *zwingend*, dass in der Klasse `CLASS` ein Konstruktor mit der Signatur `CLASS &` definiert ist. Ist dies der Fall, nimmt er selbigen. Ist dies nicht er Fall, stellt er für *diese beiden Fälle* einen Standardkonstruktor zur Verfügung, der das Qullobjekt byteweise in das Zielobjekt kopiert (erstellen einer flache Kopie).

Aufgabe 8: Konstruktoraufrufe

Welchen Namen hat der Konstruktor der Klasse `C`?	`C(... )`
Über welchen Namen wird der Konstruktor der Klasse `C` aufgerufen?	Ohne Name!
Was wird bei einem Konstruktoraufruf angegeben?	`( ... )`
Kann man den Zeitpunkt eines Konstruktoraufrufes selbst bestimmen?	Nein.

Begründe deine letzte Antwort:

Man kann den Zeitpunkt eines Konstruktoraufrufs *nicht* selbst bestimmen, da Konstruktoren immer beim Einrichten einer Variablen *automatisch* aufgerufen werden. Man kann durch Angabe der Parameter lediglich aussuchen, *welcher* der Konstruktoren aufgerufen wird. Dies gilt auch bei dynamisch eingerichteten Objekten. Der entsprechende Konstruktor wird unmittelbar nach dem Aufruf von `new` Klassen_Name ohne Angabe eines Namen ausgeführt. Bei der ublichen Anweisung `C *ptr = new C ( ... )` entspricht die Zeichenfolge `new C` einem Aufruf von `malloc( sizeof( C ) )` in Standard `C`. Die nachfolgenden Klammern nebst der eingeschlossenen Parameter `( ... )` entsprechen wieder dem Konstruktoraufruf, der ohne Angabe eines Namen erfolgt.

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?	i = 3, j = 7
In welcher Reihenfolge werden `j(b + 1)` und `i( a )` ausgeführt?	erst `i( a )` dann `j( b + 1 )`
Warum ist dies so?	Die Ausführungsreihenfolge richtet sich nach der Reihenfolge, in der die Attribute `i` und `j` in der Klasse definiert sind.

Aufgabe 2: Kopieren von Objekten

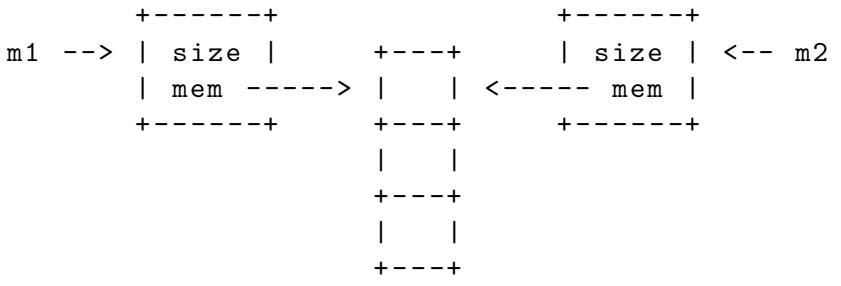
Nehmen wir an, wir hätten folgendes Programm:

```
 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? Beide Objekte teilen sich einen Speicherbereich (`mem`).

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

Am Ende von Zeile `18` haben wir die beiden Zeiger `m1` und `m2`, die jeweils auf ein eigenes Objekt zeigen. Da wir aber keinen Kopierkonstruktor definiert haben, wird das erste Objekt byteweise in das zweite kopiert. Dies hat zur folge, dass die Attribute `mem` auf den selben Arbeitsspeicher zeigen:



Dieser Effekt ist normalerweise nicht erwünscht. In den meisten Fällen sollten beide Objekte ihren eigenen (persönlichen) Speicherbereich `mem` haben.

Welche Signatur hat der Kopierkonstruktor der Klasse `MEM`? `MEM( MEM & source )`

Programmiere einen Kopierkonstruktor, der den eben besprochenen Nachteil eliminiert.

Zunächst müssen wir die Klassendefinition um einen Kopierkonstruktor erweitern:

```
 3  class MEM {
 4            private:
 5               int  size;
 6               int *mem;
 7            public:
 8               MEM( int size );
 9               MEM( MEM & source );
10         };
```

Ferner müssen wir den Kopierkonstruktor implementieren:

```
17  MEM::MEM( MEM & source )
18     {
19        mem = new int[ size = source.size ];
20     }
```

Dadurch erreichen wir, dass bei der Anweisung `m2 = m1` ein neuer Speicherbereich `mem` für `m2` dynamisch angelegt wird.

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.

Zeile	Fehler	Erläuterung	Korrektur
9	Signatur	Hier ist die Signatur falsch. Der Parameter `source` ist zwar als Zeiger definiert, muss aber als Referenz definiert sein. Sonst wird dieser Konstruktor *nicht* als Kopierkonstruktor erkannt und in Zeile 23 *nicht* aufgerufen.	`TEST & source`
16	Initialisierungsliste	Die Initialisierungsform `i( i )` darf nur im Konstruktorteil des Konstruktors stehen und *nicht* in seinem Anwendungsteil (-block)	`TRY( ... ) : i( i ) {}`
22	Parameterliste `t1()`	Der Konstruktor ist in Zeile 7 als parameterlose Methode definiert. Normalerweise müssen Methoden ohne Parameter mit runden Klammern „`()`“ aufgerufen werden. Für Konstruktoren gilt dies *nicht*; parameterlose Konstruktoren dürfen *keine* runden Klammern haben.	`TEST t1;`
23	Kopier konstruktor	Wie oben schon diskutiert, soll hier der Kopierkonstruktor aufgerufen werden. Da dieser aber in Zeile 9 falsch spezifiziert ist, wird vom Compiler ein Standard-Kopierkonstruktor verwendet, der das Quellobjekt (`t1`) byteweise in das Zielobjekt (`t2`) kopiert. Daher bekommt die Komponente `t2.i` den Wert `4711` und nicht 22.	
24	Konstruktoraufruf	Einen Konstruktor kann man nur im Rahmen einer Objektdefinition aufrufen aber *nicht* wie ein normaler Methodenaufruf.	
25		Diese Zeile ist völlig *korrekt*. Auf der rechten Seite `TEST( 42 )` wird ein anonymes Objekt kreiert, das im Rahmen der Zuweisung kopiert wird. Da die Klasse `TEST` keinen Zuweisungsoperator definiert hat, geschieht das Kopieren byteweise.	
26	Parameterliste `TRY xtry()`	Auch wenn der zugehörige Konstruktor einen Standardwert für seinen Parameter `i` hat (Zeile 16), dürfen bei Weglassen des Parameters *keine* runden Klammern „`()`“ bem Aufruf erscheinen.	`TRY xtry;`
27	Destruktor	Ein Destruktor kann nicht explizit aufgerufen werden. Er wird vom Compiler beim Abbau des Objektes (während der `return`-Anweisung in Zeile 28) automatisch initiiert.	

Mit unseren Korrekturen nimmt das Programm folgende Gestalt an:

```
 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; }
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 );      not allowed!
25        t2 = TEST( 42 );
26        TRY xtry;
27        // xyz.~TRY;      not allowed!
28        return 0;
29     }
```

Teil IV: Anwendungen

Aufgabe 1: Kopierkonstruktor und Zuweisungsoperator

Im Anwendungsteil von Übungspaket 5 haben wir einen Puffer entweickelt, 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.

Zunächst müssen wir die Klasse entsprechend erweitern, was wir mittels der beiden letzten Zeilen erledigen:

```
 1 class CBUF {
 2         public: //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
15            CBUF & operator= ( CBUF & from );    // assignment
16            CBUF ( CBUF & from );          // copy constructor
17       };
```

Anschließend können wir den Kopieroperator wie folgt implementieren:

```
45 CBUF & CBUF::operator= ( CBUF & from )
46       {
47          this->len  = from.len;
48          this->size = from.size;
49          this->buf  = new char [ from.size ];
50          test_memory( this->buf, "CBUF::=" );
51          memcpy( this->buf, from.buf, from.len );
52          return *this;
53       }
```

Als letztes könnten wir den Kopierkonstruktor mit Ausnamhe der `return`-Anweisung (Zeile `52`) in identischer Art und Weise realisieren. Da wir aber bereits einen Zuweisungsoperator haben, geht dies ganz einfach wie folgt:

```
55 CBUF::CBUF( CBUF & from )
56       {
57          *this = from;
58       }
```

Hinweis: In der Programmiersprache `C` können wir zwei Strukturen `a` und `b` einfach mittels `a = b` kopieren. Es wäre naheliegend, die ersten Zeilen des Kopierkonstruktors ebenfalls mittels `*this = from` zu ersetzen. Dies führt allerding zu einem Programmabsturz, da der Kopierkonstruktor ja nocht nicht fertig programmiert ist.