3 Klassen

3.1 Grundprinzip

- *Klassen* sind verallgemeinerte Strukturtypen:
  - Elemente können *privat*, *halböffentlich* oder *öffentlich* (*private*, *protected*, *public*) sein.
  - Objekte können durch *Konstruktoren* initialisiert und durch *Destruktoren* „aufgeräumt“ werden.
  - Objekte können durch *Elementfunktionen* inspiziert und manipuliert werden.
  - Das Kopieren von Objekten kann durch spezielle Elementfunktionen (*Kopier-* und *Verschiebekonstruktor*, *Kopier-* und *Verschiebezuweisung*) kontrolliert werden.
- Formal ist eine Struktur (`struct`) eine Klasse (`class`), deren Elemente (und Basisklassen, vgl. § 4) standardmäßig öffentlich sind.
- Daher wird in den folgenden Beispielen der Einfachheit halber immer `struct` statt `class` verwendet.

3.2 Konstruktoren

3.2.1 Beispiel: Rationale Zahlen

```
// Definition:
struct Rational {
  // Datenelemente (numerator, denominator).
  int num, den;

  // Konstruktoren.
  Rational () { num = 0; den = 1; }
  Rational (int n) { num = n; den = 1; }
  Rational (int n, int d) { num = n; den = d; }
};

// Verwendung:
Rational r1 = Rational{};
Rational r2 = Rational{2};
Rational r3 = Rational{1, 2};

// Oder:
Rational r1;
Rational r2 = 2;
```

```
// Oder:
Rational r1{};
Rational r2{2};
Rational r3{1, 2};
```

3.2.2 Erläuterungen

- Ein Konstruktor besitzt denselben Namen wie seine Klasse bzw. Struktur.
- Ebenso wie (Element-)Funktionen, können Konstruktoren *überladen* werden.
- Ein Konstruktor „konstruiert“ ein Objekt seines Typs, indem er seine Datenelemente geeignet *initialisiert*. Er kümmert sich *nicht* um die eigentliche *Erzeugung* des Objekts, d. h. um die Bereitstellung *seines* Speicherplatzes.
- Unter Umständen erzeugt ein Konstruktor im Rahmen der Initialisierung *seiner* Datenelemente *weitere* Objekte, für deren Initialisierung *ihrerseits* Konstruktoren aufgerufen werden.
- Konstruktoren können *explizit* oder *implizit* aufgerufen werden:
  - Besitzt ein Typ `T` einen Konstruktor, der ohne Argumente aufgerufen werden kann (entweder, weil er keine Parameter besitzt, oder weil alle Parameter Default-Argumente besitzen), so wird er automatisch bei jeder Deklaration einer Variablen mit Typ `T` aufgerufen, sofern die Variable nicht explizit initialisiert wird.
  - Besitzt `T` einen Konstruktor, der mit einem Argument eines beliebigen Typs `U` aufgerufen werden kann, so wird er bei Bedarf automatisch aufgerufen, um ein Objekt mit Typ `U` in ein Objekt mit Typ `T` umzuwandeln (*implizite Typumwandlung*). Um dies zu verhindern, kann der Konstruktor mit dem Schlüsselwort `explicit` deklariert werden.

- Wenn ein Typ keine Konstruktordeklaration enthält, besitzt er automatisch einen leeren parameterlosen Konstruktor.
  Um dies zu verhindern, könnte man diesen Konstruktor explizit als gelöscht definieren:

```
struct X {
  X () = delete;
  ......
};
```

- Damit dieser parameterlose Standardkonstruktor trotz anderer Konstruktoren automatisch definiert wird, könnte man ihn wie folgt definieren:

```
struct Y {
  Y (int y) { ...... }
  Y () = default;
};
```

- Seit C++11 können für Konstruktoraufrufe, wie im Beispiel, geschweifte Klammern als einheitliche Syntax für Initialisierungen aller Art verwendet werden.

- Anstelle der blauen geschweiften Klammern können aber auch runde Klammern verwendet werden. (Ebenso in den nachfolgenden Beispielen.)
  Ausnahme: `Rational r1();` deklariert `r1` als parameterlose Funktion mit Resultattyp `Rational`.

3.2.3 Verwendung von Elementinitialisierern und Initialisierungsausdrücken

Beispiele

```
struct Rational {
  // Konstante Datenelemente.
  const int num, den;

  // Konstruktoren mit Elementinitialisierern.
  Rational () : num{0}, den{1} {}
  Rational (int n) : num{n}, den{1} {}
  Rational (int n, int d) : num{n}, den{d} {}
};

struct Rational {
  // Konstante Datenelemente mit Initialisierungsausdrücken.
  const int num = 0, den = 1;

  // Konstruktoren mit unvollständigen Elementinitialisierern.
  Rational () {}
  Rational (int n) : num{n} {}
  Rational (int n, int d) : num{n}, den{d} {}
};
```

Erläuterungen

- Elementinitialisierer stellen Konstruktoraufrufe für die Datenelemente des Typs dar, die vor der Ausführung des Konstruktorrumpfs ausgeführt werden (in der Reihenfolge, in der die Datenelemente deklariert wurden, die von der Reihenfolge der Elementinitialisierer abweichen könnte).

- Wenn ein Konstruktor keinen Elementinitialisierer für ein bestimmtes Datenelement aufruft, aber das Element einen Initialisierungsausdruck besitzt, wird es mit dem Wert dieses Ausdrucks initialisiert.
  Andernfalls wird für dieses Element je nach Typ entweder sein parameterloser Konstruktor ausgeführt (den es in diesem Fall geben muss!), oder das Element bleibt uninitialisiert.

- Durch die explizite Verwendung von Elementinitialisierern kann diese eventuelle Standardinitialisierung von Datenelementen vermieden werden.

- Falls ein Datenelement keinen parameterlosen Konstruktor besitzt, *muss* es entweder mit einem Elementinitialisierer initialisiert werden oder einen Initialisierungsausdruck besitzen.

- Dasselbe gilt für Referenzelemente und konstante Datenelemente, weil ihnen nur auf diese Weise Werte zugeordnet werden können.

Beispiel: Paare rationaler Zahlen

```
struct RationalPair {
  // Datenelemente.
  Rational x, y;

  // Konstruktoren.
  RationalPair (Rational x, Rational y) : x{x}, y{y} {}
  RationalPair (int a, int b, int c, int d) : x{a, b}, y{c, d} {}
};
```

3.2.4 Aufruf eines anderen Konstruktors

- Anstelle von Elementinitialisierern, kann ein Konstruktor einen Aufruf eines anderen Konstruktors desselben Typs enthalten.
- Die daraus resultierenden Aufrufbeziehungen zwischen den Konstruktoren eines Typs dürfen keinen Zyklus bilden.

Beispiel

```
struct Rational {
  // Konstante Datenelemente.
  const int num, den;

  // Konstruktoren.
  Rational () : Rational{0} {}
  Rational (int n) : Rational{n, 1} {}
  Rational (int n, int d) : num{n}, den{d} {}
};
```

3.2.5 Separate Deklaration und Definition von Konstruktoren

- Einfache Konstruktoren können direkt in ihrer Typdefinition definiert (d. h. implementiert) werden. Sie sind in diesem Fall automatisch `inline` deklariert (vgl. § 2.12.2).

- Kompliziertere Konstruktoren werden in der Typdefinition meist nur *deklariert* und später (unter Beachtung der Regeln in § 2.12.2) separat *definiert*.

- Dies ist insbesondere dann sinnvoll, wenn die Typdefinition in einer separaten Definitionsdatei (*header file*) steht.

Beispiel

```
// Typdefinition.
struct Rational {
  // Konstante Datenelemente.
  const int num, den;

  // Deklaration der Konstruktoren.
  Rational ();
  Rational (int n);
  Rational (int n, int d);
};

......

// Definition der Konstruktoren.
Rational::Rational () : num{0}, den{1} {}
Rational::Rational (int n) : num{n}, den{1} {}
Rational::Rational (int n, int d) : num{n}, den{d} {}
```

3.2.6 Beispiel: Dynamische Zeichenketten

```
struct String {
  // Datenelement: Dynamisch erzeugte Reihe von Zeichen
  // mit abschließendem Nullzeichen.
  char* str;

  // Hilfsfunktion: str mit der dynamisch erzeugten Verkettung
  // von s1 und ggf. s2 initialisieren.
  void init (const char* s1, const char* s2 = nullptr) {
    int len = strlen(s1);
    if (s2) len += strlen(s2);
    str = new char [len + 1];
    strcpy(str, s1);
    if (s2) strcat(str, s2);
  }

  // Konstruktor: Erzeugt und füllt die dynamische Reihe str.
  String (const char* s1 = "", const char* s2 = nullptr) {
    init(s1, s2);
  }
```

```
  // Elementfunktion:
  // Alle Kleinbuchstaben in str in Großbuchstaben umwandeln.
  void toupper () {
    // Achtung: In einer Funktion mit einem bestimmten Namen
    // sind andere Bedeutungen dieses Namens verdeckt.
    // Deshalb muss der Name der Bibliotheksfunktion toupper
    // mit std:: qualifiziert werden.
    for (char* s = str; *s; s++) *s = std::toupper(*s);
  }
};

// Globale Funktion: Verkettung der Zeichenketten s1 und s2.
String concat (const String& s1, const String& s2) {
  return String(s1.str, s2.str);
}

// Globale Funktion: Zeichenkette s ausgeben.
void print (const String& s) {
  cout << s.str << endl;
}
```

```
// Testprogramm.
int main (int argc, char* argv []) {
  // Initialisierung von s1 und s2 durch Aufrufe des Konstruktors
  // von String mit jeweils einem Argument.
  String s1 = argv[1];
  String s2 = argv[2];

  // Initialisierung des Resultats von concat durch Aufruf des
  // Konstruktors von String mit zwei Argumenten.
  String s = concat(s1, s2);

  // Kleinbuchstaben in s durch Großbuchstaben ersetzen.
  s.toupper();

  // s ausgeben.
  print(s);
}
```

3.3 Destruktoren

3.3.1 Problem

- Der Speicher, der vom Konstruktor von `String` beschafft wird, wird nicht wieder freigegeben.

3.3.2 Lösung

```
struct String {
  // Wie bisher.
  ......

  // Destruktor: Vernichtet die dynamische Reihe str,
  // die vom zugehörigen Konstruktoraufruf erzeugt wurde.
  ~String () {
    delete [] str;
  }
};
```

Erläuterungen

- Ein Destruktor ist eine parameterlose (Pseudo-)Elementfunktion, deren Name aus einer Tilde und dem Namen des Typs besteht.

- Ein Destruktor „zerstört“ ein Objekt seines Typs, indem er ggf. erforderliche *Aufräumarbeiten* ausführt. Er kümmert sich *nicht* um die eigentliche *Vernichtung* des Objekts, d. h. um die Freigabe *seines* Speicherplatzes.

- Unter Umständen vernichtet ein Destruktor im Rahmen *seiner* Aufräumarbeiten jedoch *andere* Objekte, die typischerweise von einem Konstruktor seines Typs dynamisch erzeugt wurden.

- Wenn ein Typ keine Destruktordeklaration enthält, besitzt er automatisch einen leeren Destruktor.

- Destruktoren werden in aller Regel *implizit* am Ende der Lebensdauer eines Objekts aufgerufen (vgl. § 3.3.3).

3.3.3 Ausführung von Konstruktoren und Destruktoren

- Variablen, die global oder in einem Namensbereich (namespace) definiert sind, sowie statische Elementvariablen von Klassen existieren während der gesamten Programmausführung.
  - Ihr Konstruktor wird entweder irgendwann vor der Ausführung von `main` ausgeführt oder irgendwann, bevor die Variable zum ersten Mal verwendet wird.
  - Für Variablen innerhalb derselben Übersetzungseinheit, die nicht `inline` definiert sind (vgl. § 2.12.2), werden die Konstruktoren in der Reihenfolge der Variablendefinitionen ausgeführt.
  - Für Variablen in verschiedenen Übersetzungseinheiten sowie `inline`-Variablen ist die Reihenfolge nicht festgelegt.
  - Wichtige Ausnahme seit C++20 für Variablen, die nicht `inline` sind:
    Wenn ein Modul ein anderes Modul importiert, werden die Variablen des importierten Moduls garantiert initialisiert, bevor die Variablen des Moduls selbst initialisiert werden, deren Definition sich nach der Import-Deklaration befindet.
  - Die Destruktoren solcher Variablen werden nach der Ausführung von `main` ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.
  - Die Destruktoren werden auch dann noch ausgeführt, wenn das Programm vorzeitig durch Aufruf der Bibliotheksfunktion `exit` beendet wird.

- Wenn derartige Variablen jedoch `thread_local` definiert sind, existieren sie je einmal während der Ausführung jedes Threads.
  - Ihr Konstruktor wird dann entweder vor der Ausführung der „Hauptfunktion“ des jeweiligen Threads ausgeführt oder bevor die Variable zum ersten Mal in diesem Thread verwendet wird.
  - Die Destruktoren solcher Variablen werden am Ende dieses Threads (nach der Ausführung seiner Hauptfunktion) ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.
  - Die Destruktoren werden auch dann noch ausgeführt, wenn dieser Thread `exit` ausführt. (Für die Variablen anderer Threads werden jedoch keine Destruktoren ausgeführt.)
- Variablen, die lokal in einem Block definiert sind, existieren während der Ausführung dieses Blocks.
  - Ihr Konstruktor wird bei der Ausführung ihrer Definition ausgeführt.
  - Die Destruktoren solcher Variablen werden am Ende des Blocks ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.
  - Die Destruktoren werden auch dann noch ausgeführt, wenn der Block vorzeitig mittels einer Sprunganweisung (`break`, `continue`, `return`, `goto`) oder aufgrund einer Ausnahme beendet wird, aber nicht, wenn das Programm mit `exit` beendet wird.

- Wenn derartige Variablen jedoch `static` oder `thread_local` definiert sind, existieren sie entweder während der gesamten Programmausführung oder je einmal während der Ausführung jedes Threads.
  - Ihr Konstruktor wird dann bei der ersten Ausführung ihrer Definition während der Programmausführung bzw. während der Ausführung des jeweiligen Threads ausgeführt.
  - Die Destruktoren solcher Variablen werden nach der Ausführung von `main` bzw. am Ende jedes Threads ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.
- Parameter einer Funktion existieren während der Ausführung dieser Funktion.
  - Ihre Konstruktoren werden beim Aufruf der Funktion (in einer nicht festgelegten Reihenfolge) ausgeführt.
  - Ihre Destruktoren werden am Ende der Funktion ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.

- Objekte, die dynamisch mittels `new` erzeugt werden, existieren bis zu ihrer Vernichtung mittels `delete`.
  - Ihr Konstruktor wird nach der Beschaffung ihres Speicherplatzes durch `new` ausgeführt.
  - Ihr Destruktor wird vor der Freigabe ihres Speicherplatzes durch `delete` ausgeführt.
- Elemente einer Reihe existieren genauso lange wie die Reihe.
  - Ihre Konstruktoren werden bei der Initialisierung der Reihe ausgeführt, und zwar in der Reihenfolge aufsteigender Indizes.
  - Ihre Destruktoren werden bei der Vernichtung der Reihe ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.

- Elementvariablen eines Klassenobjekts existieren genauso lange wie das Klassenobjekt.
  - Ihre Konstruktoren werden bei der Initialisierung des Klassenobjekts durch die Elementinitialisierer (oder die in § 3.2.3 genannten Alternativen) von dessen Konstruktor ausgeführt, und zwar in der Reihenfolge der Deklaration der Elementvariablen.
  - Ihre Destruktoren werden bei der Vernichtung des Klassenobjekts nach der Ausführung von dessen Destruktor ausgeführt, und zwar in umgekehrter Reihenfolge ihrer Konstruktoraufrufe.

- Damit die obigen Aussagen z. B. auch für elementare Typen wie `int` korrekt sind, besitzen auch solche Typen formal einen parameterlosen Konstruktor sowie einen Destruktor, die beide leer sind.

3.3.4 Ressourcenverwaltung durch Konstruktoren und Destruktoren

- Da Destruktoren immer automatisch „im richtigen Moment“ aufgerufen werden, können sie gezielt zur Freigabe von Ressourcen eingesetzt werden, die zuvor (meist in einem zugehörigen Konstruktor) beschafft wurden (resource acquisition/allocation is initialization, RAII).
- Da Anweisungsfolgen auf unterschiedliche Art – manchmal auch unerwartet – vorzeitig beendet werden können, ist dies einfacher und sicherer als eine manuelle Freigabe der Ressourcen am Ende einer Anweisungsfolge.

Beispiel

```
#include <mutex>
#include <fstream>
#include <memory>
using namespace std;

// Sperre für gegenseitigen Ausschluss (mutual exclusion).
mutex m;

void test () {
  // Der Konstruktor von lock_guard sperrt das übergebene Mutex.
  lock_guard<mutex> g {m};
```

```
    // Der Konstruktor von ifstream öffnet die Datei mit dem
    // übergebenen Namen.
    ifstream f {"input.txt"};

    // Der Konstruktor von unique_ptr speichert den übergebenen
    // Zeiger auf ein dynamisches Objekt oder eine dynamische Reihe.
    unique_ptr<char []> p {new char [100]};

    // Hier könnte die Funktion vorzeitig verlassen werden,
    // z. B. durch return oder eine unerwartete Ausnahme.
    ......

    // Der automatisch aufgerufene Destruktor von unique_ptr gibt
    // das dynamische Objekt oder die dynamische Reihe auf jeden Fall
    // wieder frei.

    // Der automatisch aufgerufene Destruktor von ifstream
    // schließt die Datei auf jeden Fall wieder.

    // Der automatisch aufgerufene Destruktor von lock_guard
    // gibt die Sperre auf jeden Fall wieder frei.
}
```

3.3.5 Sichere Initialisierung nicht-lokaler Variablen

- Gemäß § 3.3.3 ist die Initialisierungsreihenfolge von Variablen, die global oder in einem Namensbereich definiert sind, sowie von statischen Elementvariablen in verschiedenen Übersetzungseinheiten nicht festgelegt.

- Deshalb ist das Verhalten eines Programms undefiniert, in dem für die Initialisierung einer derartigen Variablen eine andere derartige Variable aus einer anderen Übersetzungseinheit benötigt wird.

- Dieses Problem kann umgangen werden, indem man anstelle einer Variablen eine Funktion definiert und verwendet, die eine Referenz auf eine lokale statische Variable liefert, zum Beispiel:

```
// Statt einer globalen Variablen ...
String path = "/bin:/usr/bin";

// ... lieber eine globale Funktion.
String& path () {
  static String path = "/bin:/usr/bin";
  return path;
}
```

- Die lokale statische Variable wird garantiert beim ersten Aufruf der Funktion initialisiert.

- Um die Variable `path` zu verwenden, muss dann jeweils `path()` geschrieben werden. Da die Funktion eine Referenz zurückliefert, sind auch Zuweisungen `path() = ...` möglich.

- Alternativ könnte man die Definitionen (nicht unbedingt die Deklarationen) aller nicht-lokalen Variablen in einer sinnvollen Reihenfolge in eine einzige Übersetzungseinheit schreiben.

- Wie bereits in § 3.3.3 erwähnt, lösen Module in C++20 das hier beschriebene Problem, indem sie garantieren, dass die aus anderen Modulen importierten Variablen bereits initialisiert sind, wenn die eigenen Variablen eines Moduls (deren Definition sich nach den Import-Deklarationen befindet) initialisiert werden.

3.4 Kopierfunktionen

3.4.1 Problem

```
int main (int argc, char* argv []) {
  // Initialisierung von s1 durch Aufruf des Konstruktors
  // von String.
  String s1 = argv[1];

  // Initialisierung von s2 als elementweise Kopie von s1.
  String s2 = s1;

  // Kleinbuchstaben in s1 in Großbuchstaben umwandlen.
  s1.toupper();

  // s1 und s2 ausgeben.
  print(s1);
  print(s2);

  // Automatischer Aufruf des Destruktors von String für s2 und s1.
}
```

- Weil das Objekt `s2` als elementweise „flache“ Kopie von `s1` konstruiert wird, zeigen `s1.str` und `s2.str` auf *dieselbe* dynamische Reihe.

- Damit werden durch den Aufruf `s1.toupper()` auch die Kleinbuchstaben in `s2` in Großbuchstaben umgewandelt, was vermutlich nicht beabsichtigt ist.

- Außerdem wird diese Reihe am Ende des Blocks fälschlicherweise *zweimal* freigegeben – was zu undefiniertem Programmverhalten führt –, weil sowohl für `s2` als auch für `s1` der Destruktor von `String` aufgerufen wird.

3.4.2 Lösung: Kopierkonstruktor

```
struct String {
  // Wie zuvor.
  ......

  // Kopierkonstruktor: Erzeugt eine "tiefe" Kopie von that.
  String (const String& that) {
    init(that.str);
  }
};
```

Erläuterungen

- Ein Kopierkonstruktor eines Typs `T` ist ein Konstruktor mit einem Parameter des Typs `const T&` oder `T&`. (Eventuell vorhandene weitere Parameter müssten Default-Argumente besitzen.)

- Er wird implizit aufgerufen, wenn ein Objekt des Typs `T` durch ein Objekt desselben Typs initialisiert wird, was auch bei der Übergabe eines Objekts als Parameter oder bei der Rückgabe eines Funktionsresultats der Fall ist. (Deshalb darf der Kopierkonstruktor selbst keinen Parameter mit Typ `T` besitzen, weil er sonst bei der Übergabe dieses Parameters ebenfalls aufgerufen werden müsste.)

- Vermeidbare Aufrufe des Kopierkonstruktors dürfen jedoch vom Übersetzer eliminiert werden.

- Außerdem kann man selbst Aufrufe des Kopierkonstruktors vermeiden, indem man Parameter als Referenzen (normalerweise auf Konstanten) deklariert (vgl. § 2.9.6).

- Wenn für einen Typ kein expliziter Kopierkonstruktor definiert ist, besitzt er automatisch einen impliziten Kopierkonstruktor, der alle Elementvariablen des Typs kopiert. Hierfür werden ggf. die Kopierkonstruktoren der Elementvariablen aufgerufen.

- Falls dies aus irgendeinem Grund nicht möglich ist (z. B. weil einer der benötigten Kopierkonstruktoren gelöscht ist), wird der implizite Kopierkonstruktor jedoch als gelöscht definiert. Dann können Objekte des Typs grundsätzlich nicht kopiert werden.

- Wenn für den Typ eine Verschiebefunktion definiert ist (vgl. § 3.5), wird der implizite Kopierkonstruktor ebenfalls als gelöscht definiert.

- Außerdem kann der Kopierkonstruktor auch explizit als gelöscht definiert werden, wenn Objekte des Typs aus irgendeinem Grund nicht kopiert werden sollen (vgl. § 3.4.4).

3.4.3 Weiteres Problem

```
int main (int argc, char* argv []) {
  // Initialisierung von s1 und s2 durch Aufrufe des Konstruktors
  // von String.
  String s1 = argv[1];
  String s2 = argv[2];

  // Elementweise Zuweisung von s1 an s2.
  s2 = s1;

  ......

  // Automatischer Aufruf des Destruktors von String für s2 und s1.
}
```

- Durch die elementweise „flache“ Zuweisung von `s1` an `s2` wird `s2.str` durch `s1.str` überschrieben.
- Am Ende des Blocks werden die Destruktoren für `s2` und `s1` aufgerufen, die jetzt beide dieselbe Reihe `s1.str` vernichten (wollen), während die ursprüngliche Reihe `s2.str` für immer als „Speicherleiche“ übrig bleibt.

3.4.4 Lösung: Kopierzuweisung

```
struct String {
  // Wie zuvor.
  ......

  // Kopierzuweisung: Vernichtet die eigene Reihe
  // und erstellt eine "tiefe" Kopie der Reihe von that.
  String& operator= (const String& that) {
    // Vorsicht: Die alte Reihe erst vernichten, wenn die neue
    // erfolgreich erzeugt und gefüllt wurde, weil das Erzeugen
    // prinzipiell fehlschlagen kann und weil die alte eventuell
    // noch zum Füllen der neuen gebraucht wird (wenn ein Objekt
    // an sich selbst zugewiesen wird).
    char* s = str;
    init(that.str);
    delete [] s;

    // Selbstreferenz zurückliefern.
    return *this;
  }
};
```

Erläuterungen

- Eine Kopierzuweisung (oder ein kopierender Zuweisungsoperator) eines Typs `T` ist ein überladener Operator = mit einem Parameter des Typs `const T&`, `T&` oder `T`, der anhand der üblichen Regeln für überladene Operatoren aufgerufen wird.

  (Daraus folgt nebenbei, dass bei Klassen auch R-Werte Ziel einer Zuweisung sein könnten, was eigentlich widersinnig ist! Um dies zu verhindern, kann der Zuweisungsoperator seit C++11 mit `&` qualifiziert werden; vgl. § 3.6.
  Selbst eine Zuweisung an ein konstantes Objekt wäre prinzipiell denkbar, wenn der Zuweisungsoperator mit `const` qualifiziert ist; vgl. ebenfalls § 3.6.)

- Im Gegensatz zu vielen anderen Programmiersprachen, besteht in C++ ein wesentlicher Unterschied zwischen der *Initialisierung* eines Objekts (durch einen Konstruktor) und der *Zuweisung* an ein Objekt (durch einen Zuweisungsoperator).

- Da das Zielobjekt im einen Fall uninitialisiert und im anderen Fall bereits initialisiert ist, müssen Konstruktoren und Zuweisungsoperatoren normalerweise unterschiedliche Anweisungen ausführen.

- Häufig enthält ein Zuweisungsoperator sowohl Teile eines Destruktors als auch Teile eines Konstruktors.

- Wenn die Zuweisung eines Objekts an sich selbst nicht korrekt funktionieren würde, muss dieser Sonderfall abgefangen werden: `if (this != &that) ......`
- Ein Zuweisungsoperator liefert üblicherweise eine Referenz auf das Zielobjekt zurück.
- Um das Kopieren von Objekten eines Typs `T` zu verbieten (z. B. bei Streams, Threads und Mutexes), kann man den Kopierkonstruktor und die Kopierzuweisung von `T` als gelöscht definieren.
- Wenn für einen Typ keine explizite Kopierzuweisung definiert ist, besitzt er automatisch einen impliziten Zuweisungsoperator, der für alle Elementvariablen des Typs eine Kopierzuweisung ausführt. Hierfür werden ggf. die entsprechenden Zuweisungsoperatoren der Elementvariablen aufgerufen.
- Falls dies aus irgendeinem Grund nicht möglich ist (z. B. weil einer der benötigten Zuweisungsoperatoren gelöscht ist oder weil der Typ konstante Elementvariablen enthält), wird der implizite Zuweisungsoperator jedoch als gelöscht definiert.
- Wenn für den Typ eine Verschiebefunktion definiert ist (vgl. § 3.5), wird der implizite Zuweisungsoperator ebenfalls als gelöscht definiert.

3.5 Verschiebefunktionen

3.5.1 Problem

```
// Alle Kleinbuchstaben in einer Kopie von s
// in Großbuchstaben umwandeln.
String toupper (String s) { // Init. von s durch Kopierkonstruktor.
  s.toupper();
  return s;
}                           // Automatischer Aufruf des Destruktors für s.

int main () {
  // Initialisierung von s durch Aufruf des Konstruktors von String.
  String s = argv[1];

  // Übergabe von s an toupper durch einen (unvermeidbaren) Aufruf
  // des Kopierkonstruktors.
  // Rückgabe des Resultats von toupper in ein temporäres Objekt
  // ebenfalls durch einen Aufruf des Kopierkonstruktors.
  // Zuweisung dieses temporären Objekts an s durch einen Aufruf
  // der Kopierzuweisung.
  s = toupper(s);
}
```

- Das von `toupper` gelieferte Objekt wird hier zweimal unnötig kopiert:
  - Zuerst wird es durch den Kopierkonstruktor in ein temporäres Objekt kopiert.
  - Anschließend wird dieses temporäre Objekt durch die Kopierzuweisung nach `s` kopiert.

3.5.2 Lösung: Verschiebekonstruktor und -zuweisung

```
struct String {
  // Wie zuvor.
  ......

  // Verschiebekonstruktor:
  // "Klaut" die Daten von that und hinterlässt that
  // in irgendeinem neuen wohldefinierten Zustand.
  String (String&& that) {
    str = that.str;
    that.init("");
  }

  // Verschiebezuweisung:
  // Vertauscht die Daten von that mit den eigenen und hinterlässt
  // that damit wiederum in einem neuen wohldefinierten Zustand.
  String& operator= (String&& that) {
    char* tmp = str; str = that.str; that.str = tmp;
    return *this;
  }
};
```

Erläuterungen

- Verschiebekonstruktor und -zuweisung eines Typs `T` sind analog zu Kopierkonstruktor und -zuweisung, allerdings mit einem Parameter des Typs `T&&`. (Alternativ könnte der Parametertyp auch `const T&&` sein, was aber meist nicht sinnvoll ist, weil das übergebene Objekt normalerweise verändert werden soll.)

- Wenn sie vorhanden sind, werden sie vom Übersetzer anstelle von Kopierkonstruktor bzw. -zuweisung verwendet, sofern das zu kopierende bzw. zuzuweisende Objekt ein R-Wert ist (weil ihr R-Wert-Referenz-Parameter nur mit R-Werten initialisiert werden kann, vgl. § 2.9.5; wenn es ein L-Wert ist, werden nach wie vor Kopierkonstruktor bzw. -zuweisung verwendet).

- Weil das an den Parameter gebundene R-Wert-Objekt (normalerweise) „im nächsten Moment“ nicht mehr existieren wird, ist es unnötig, seine Daten zu kopieren. Stattdessen kann man sie einfach „klauen“, d. h. in das Zielobjekt *verschieben*.

- Damit der später für das „bestohlene“ Objekt ausgeführte Destruktor korrekt funktioniert und keinen „Schaden anrichtet“, muss das Objekt aber in einem wohldefinierten neuen Zustand hinterlassen werden, der nichts mit seinem alten Zustand gemeinsam hat.
  Außerdem sollten anschließende (Kopier- und Verschiebe-) Zuweisungen an dieses Objekt noch korrekt funktionieren (vgl. das Beispiel in § 3.5.3).

- Bei der Verschiebezuweisung lässt sich das am einfachsten durch Vertauschen der Daten mit denen des Zielobjekts erreichen (was auch im Sonderfall einer Selbstzuweisung korrekt funktionieren würde).

- Beim Verschiebekonstruktor hinterlässt man das übergebene Objekt häufig im gleichen Zustand wie ein durch den parameterlosen Standardkonstruktor initialisiertes Objekt.

- Im Beispiel von § 3.5.1 wird das temporäre Objekt jetzt durch einen Aufruf des Verschiebekonstruktors anstelle des Kopierkonstruktors mit dem Resultat von `topupper` initialisiert (vgl. die Anmerkungen am Ende von § 3.5.3).
  Für die Zuweisung dieses temporären Objekts an `s` wird jetzt die Verschiebezuweisung anstelle der Kopierzuweisung verwendet, weil die rechte Seite `toupper(s)` ein R-Wert ist.
  Damit sind die beiden „teuren“ Kopieroperationen jeweils durch „billige“ Verschiebeoperationen ersetzt.

- Im Beispiel von § 3.4.3 wird bei der Zuweisung von `s1` an `s2` jedoch weiterhin die Kopierzuweisung aufgerufen – was auch sinnvoll ist –, weil das zuzuweisende Objekt `s2` ein L-Wert ist.
  Aus dem gleichen Grund wird im Beispiel von § 3.4.1 bei der Initialisierung von `s2` durch `s1` weiterhin der Kopierkonstruktor aufgerufen, was ebenfalls sinnvoll ist.

- Auch für Typen wie Streams, Threads und Mutexes, deren Objekte nicht sinnvoll kopiert werden können (vgl. § 3.4.4), lässt sich normalerweise eine sinnvolle Verschiebesemantik implementieren, die z. B. notwendig ist, um derartige Objekte in Containern wie z. B. `vector` speichern zu können.

- Wenn für einen Typ weder Verschiebekonstruktor und -zuweisung noch Kopierkonstruktor und -zuweisung noch Destruktor explizit definiert sind, werden Verschiebekonstruktor und -zuweisung implizit definiert.

- Analog zu implizit definierten Kopierfunktionen, führen diese impliziten Verschiebefunktionen die entsprechende Funktion jeweils für alle Elementvariablen des Typs aus.

- Wenn dies aus irgendeinem Grund nicht möglich ist, wird die entsprechende Funktion jedoch als gelöscht definiert.

3.5.3 Verschiebung von L-Werten

Beispiel

```
int main () {
  // Initialisierung von s1 durch Aufruf des Konstruktors.
  String s1 = argv[1];

  // Initialisierung von s2 durch Aufruf des Kopierkonstruktors.
  String s2 = s1;

  // Initialisierung von s3 durch Aufruf des Verschiebekonstruktors.
  String s3 = static_cast<String&&>(s1);

  // Oder besser so:
  String s3 = std::move(s1);

  // Aber nicht so:
  String s3 = static_cast<String>(s1);

  // Obwohl der Inhalt von s1 nach s3 verschoben wurde,
  // könnte s1 hier immer noch verwendet werden.
}
```

Erläuterungen

- Um eine Verschiebung anstelle einer Kopie eines L-Werts zu erzwingen – weil der Inhalt des Objekts anschließend nicht mehr benötigt wird –, kann der L-Wert in eine R-Wert-Referenz umgewandelt werden.

- Aus Gründen der Lesbarkeit wird hierfür üblicherweise die Bibliotheksfunktion `move` verwendet, die selbst keinerlei Daten verschiebt, sondern lediglich diese Typumwandlung vornimmt.

- Eine Umwandlung eines L-Werts in seinen eigenen Typ (ohne Referenz) liefert zwar auch einen R-Wert, der aber selbst durch einen Aufruf des Kopierkonstruktors erzeugt wird!

Beispiel: Vertauschen

```
void swap (T& x, T& y) {
  // Ungünstig wegen Verw.      // Besser wegen Verwendung von
  // von Kopierfunktionen:      // Verschiebefkt. (falls vorhd.):
  T z = x;                      T z = move(x);
  x = y;                        x = move(y);
  y = z;                        y = move(z);
}
```

Rückgabe lokaler Variablen und Parameter

- Wenn eine lokale Variable (aber kein Parameter) als Funktionsresultat geliefert wird (z. B. `return x`), kann der hierfür theoretisch erforderliche Aufruf des Kopierkonstruktors vom Übersetzer häufig eliminiert werden (sofern die Variable nicht `static` oder `thread_local` ist).

- Deshalb ist die Verwendung von `move` (d. h. `return move(x)`) in solchen Fällen unnötig und oft sogar kontraproduktiv, weil der Ausdruck `move(x)` keine lokale Variable mehr ist und der dann notwendige Aufruf des Verschiebekonstruktors deshalb *nicht* eliminiert werden darf.

- Außerdem werden sowohl lokale Variablen (sofern sie nicht `static` oder `thread_local` sind) als auch Parameter einer Funktion, die als Funktionsresultat geliefert werden, hier ausnahmsweise als R-Werte betrachtet, sodass ein vorhandener Verschiebekonstruktor sowieso automatisch anstelle des Kopierkonstruktors verwendet wird, wenn der Aufruf nicht eliminiert werden kann.

3.6 Elementfunktionen (member functions)

- Nichtstatische Elementfunktionen einschließlich Konstruktoren und Destruktor einer Klasse `C` besitzen einen impliziten Parameter `this` mit Typ `C*`, der auf das *Zielobjekt* des Funktionsaufrufs verweist.

- Die Verwendung eines Datenelements `m` der Klasse in einer Elementfunktion ist äquivalent zu `this->m`. Ebenso ist die Verwendung einer anderen Elementfunktion `f` der Klasse äquivalent zu `this->f`.

- Wenn die Deklaration einer Elementfunktion nach der Parameterliste das Schlüsselwort `const` enthält, ist `this` vom Typ `const C*`, d. h. die Funktion darf ihr Zielobjekt nicht verändern. (Analog für `volatile`.)

- Wenn das Zielobjekt eines Funktionsaufrufs konstant ist, *muss* die aufgerufene Elementfunktion `const` sein.
  Andererseits darf eine `const`-Elementfunktion auch für nicht-konstante Zielobjekte aufgerufen werden (vgl. § 2.4.2).
  Daher ist es ratsam, `const` bei der Deklaration von Elementfunktionen so oft wie möglich zu verwenden.

- Elementfunktionen einer Klasse können so überladen werden, dass sie sich nur durch die An- bzw. Abwesenheit von `const` unterscheiden.
  Bei einem Aufruf einer solchen Funktion wird für ein konstantes Zielobjekt die `const`-Funktion, für ein nicht-konstantes Zielobjekt die Nicht-`const`-Funktion verwendet.

- Seit C++11 kann nach der Parameterliste einer Elementfunktion auch `&` oder `&&` stehen, um anzuzeigen, dass das Zielobjekt ein L- bzw. R-Wert sein muss.

- Statische Elementfunktionen sind vergleichbar mit globalen Funktionen, müssen aber beim Aufruf außerhalb der Klasse mit dem Klassennamen qualifiziert werden.
  Sie besitzen keinen impliziten Parameter `this`.

- Datenelemente und Elementfunktionen müssen unterschiedliche Namen besitzen.
  (In Java darf eine Methode den gleichen Namen wie eine Objekt- oder Klassenvariable besitzen.)

Beispiel

```
// Klasse.
struct User {
  // Datenelemente.
  string username, password;

  // Konstruktor.
  User (const string& un) : username{un} {}

  // Konstante Elementfunktion.
  string get_username () const { return username; }

  // Nicht-konstante Elementfunktion.
  void set_password (const string& pw) { password = pw; }

  // Statische Elementfunktion.
  static bool equal (const User& u1, const User& u2) {
    // Da u1 und u2 konstant sind, muss get_username konstant sein.
    return u1.get_username() == u2.get_username();
  }
};
```

```
// Verwendung der unterschiedlichen Elementfunktionen.
User u{"Dummy"};
cout << u.get_username() << endl;
u.set_password("ymmuD");
if (User::equal(u, u)) ......
```

3.7 Statische Datenelemente

- Statische Datenelemente sind vergleichbar mit globalen Variablen, müssen aber außerhalb der Klasse ebenfalls mit dem Klassennamen qualifiziert werden.

- Statische Datenelemente können normalerweise in ihrer Klasse nur *deklariert* werden und müssen zusätzlich außerhalb der Klasse (unter Beachtung der Regeln für Variablen in § 2.12.2) *definiert* werden (sofern sie irgendwo im Programm verwendet werden). Ein eventueller Initialisierungsausdruck wird bei der Definition angegeben.

- Ausnahme: Wenn ein statisches Datenelement konstant ist und einen ganzzahligen oder Aufzählungstyp besitzt, kann es direkt bei seiner Deklaration in der Klasse mit einem *konstanten Ausdruck* initialisiert werden.
  Solange dann nur sein *Wert* (und nicht seine Adresse) verwendet wird, muss keine zusätzliche Definition angegeben werden. (In diesem Fall muss der Übersetzer auch keinen Speicherplatz anlegen.)
  Andernfalls darf bei der Definition kein Initialisierungsausdruck mehr angegeben werden.

- Anstelle von statischen Datenelementen können auch statische Elementfunktionen verwendet werden, die eine Referenz auf eine lokale statische Variable liefern (vgl. § 3.3.5).

Beispiel

```
struct X {
  // Deklaration statischer Datenelemente.
  static X* p;
  static X* q;
  static int x;
  static const int y = 2;
};

// Definition von X::p und X::x.
X* X::p = nullptr;
int X::x = 1;
```

```
int main () {
  // Verwendung von X::p und X::x.
  X::p = new X;
  cout << X::x << endl;
  cout << &X::x << endl;

  // X::q wird nirgends verwendet
  // und braucht deshalb keine Definition.

  // Der Wert von X::y kann ohne Definition verwendet werden.
  cout << X::y << endl;

  // Fehler: Die Adresse von X::y kann ohne Definition nicht
  // verwendet werden.
  cout << &X::y << endl;

  // Fehler: Um X::y an eine Referenz zu binden, wird implizit
  // ebenfalls seine Adresse verwendet.
  int& r = X::y;
}
```