Hochschule Esslingen
University of Applied Sciences

Objektorientierte Programmierung mit C++:

Initialisierung durch Konstruktoren

Profs. M. Dausmann, D. Schoop, A. Rößler

Fakultät Informationstechnik, Hochschule Esslingen

Agenda

- Initialisierung von Instanzen
- Default- oder Standardkonstruktor
- Überladen von Konstruktoren
- Konstruktoren mit Parametern
- Konstruktoren mir Defaultwerten
- Kopierkonstruktor
- Konvertierkonstruktor
- Vorwärtsdeklaration
- Konstruktoren und dynamische Objekte
- Destruktoren

Initialisierung von Instanzen

- Ein Objekt wird bei seiner Erzeugung normalerweise nicht initialisiert. Um eine automatische Initialisierung zu bewirken, stehen spezielle Methoden sogenannte **Konstruktoren** zur Verfügung.
- Konstruktoren sind Methoden ohne Rückgabetyp. Der Methodenname eines Konstruktors ist identisch mit dem Namen der Klasse.
- Konstruktoren können nicht explizit aufgerufen werden, sondern der Kompiler benutzt die Konstruktoren, wenn Objekte zu 'konstruieren' sind (Definition, new, ...).
- Der Kompiler stellt Konstruktoren zur Verfügung. Bei der Deklaration einer Klasse kann der Programmierer diese mit eigenen Konstruktoren redefinieren.

Standardkonstruktor des Kompilers

- Der Standardkonstruktor hat keine Parameter.
- Der vom Kompiler zur Verfügung gestellte Standardkonstruktor initialisiert die Datenelemente mit zufälligen Werten (also quasi gar nicht).
- Er heißt auch Defaultkonstruktor, weil er in manchen Fällen 'per default' benutzt wird, und es keine Möglichkeit gibt, den Aufruf eines anderen Konstruktors zu veranlassen (beispielsweise bei bestimmten Situationen bei der Vererbung).

Beispiel für Standardkonstruktor

```
class Bruch {
  int zaehler, nenner;
public:
  void print(){
   cout << "Quotient= " << zaehler
   << "/" << nenner << endl; }
};

void main(void){
  Bruch b;
  b.print();
}
```

```
Quotient= -858993460/-858993460
```

Selbstdefinierter Standardkonstruktor

```
class Bruch {
  int zaehler, nenner;
public:
  Bruch(){
   zaehler = 0; nenner = 1;
  }

  void print(){
   cout << "Quotient= " << zaehler
   << "/" << nenner << endl;
  }
};

void main(void) {
  Bruch b;
  b.print();
}
```

Konstruktor

```
Quotient= 0/1
```

Konstruktoren mit Parametern

- Wie eine Methode können Konstruktoren **Parameter** haben, um die Objekte mit individuellen Werten zu initialisieren.
- Bei bestimmten Parameterprofilen haben die Konstruktoren mit Parametern eigene Namen:
  - **Kopierkonstruktor**
    (1 Parameter vom Typ der eigenen Klasse)
  - **Konvertierkonstruktor**
    (1 Parameter von einem anderen Typ)
  - **Parametrisierter Konstruktor**
    (Mit beliebigen Parametern)

Überladen von Konstruktoren

```
class Bruch {
  int zaehler, nenner;
public:
  Bruch(){ zaehler = 0; nenner = 1;}

  Bruch(int zaehler){
   this->zaehler = zaehler;
   this->nenner = 1; }

  Bruch(int zaehler, int nenner){
   this->zaehler = zaehler;
   this->nenner = nenner; }

   void print(){.....}
};

Bruch b, c(12), d(7, -24);
```

Standard-konstruktor

Konvertier-konstruktor

parametrisierter Konstruktor

Konstruktor mit Defaultwerten

```
class Bruch {
  int zaehler, nenner;
public:
  Bruch(int zaehler = 0,
        int nenner = 1) {
  this->zaehler = zaehler;
  this->nenner = nenner; }
  void print(){.....}
};
.....
Bruch b, c(12), d(7, -24);
```

Standard-konstruktor

Überladener Standardkonstruktor

```
class Bruch {
  int zaehler, nenner;
public:
  Bruch(int zaehler, int nenner){
   this->zaehler = zaehler;
   this->nenner = nenner; }
  void print(){.....}
};

void main(void) {
  Bruch b;                          // Fehler !!!
  Bruch c(7,3);
  b.print();
}
```

Kopierkonstruktor

- Ein **Standard-Kopierkonstruktor** ist für jede Klasse automatisch definiert. Man kann jedoch auch einen Kopierkonstruktor selbst definieren:

```
Bruch(const Bruch & b){
this->zaehler = b.zaehler;
this->nenner = b.nenner; }
```

- Aufruf des **Kopierkonstruktors** bei der Objekt-Definition:

```
Bruch b, c(b);
Bruch d = b;
```

- Der **Standard-Kopierkonstruktor** kopiert **elementweise**, macht also genau das, was der selbstdefinierte Kopierkonstruktor im Beispiel oben auch macht.

Exkurs: Initialisierung vs. Zuweisung

- In C++ wird zwischen einer Zuweisung und einer Initialisierung strikt unterschieden:

  Zuweisung: **`bruch1 = bruch2;`**

  Initialisierung: **`Bruch bruch1 = bruch2;`**

- Eine Zuweisung erfolgt immer an ein bereits existierendes Objekt, verändert es und kann an beliebiger Stelle mehrmals auf dieses Objekt angewendet werden.
- Eine Initialisierung erfolgt immer mit der Definition, also zu Beginn der Lebenszeit eines Objekts; mit der Initialisierung erhält ein Objekt einen quasi voreingestellten inneren Zustand.

Exkurs: Initialisierung vs. Zuweisung

- Eine Initialisierung kann auf ein Objekt nur ein einziges Mal angewendet werden.
- Bei einer Initialisierung wird ein Konstruktor aufgerufen.
- Da im Beispiel auf der vorherigen Folie eine Kopie von `bruch2` erzeugt wird, nennt man den Konstruktor, der als Übergabeparameter eine Instanz der eigenen Klasse erhält, den **Kopierkonstruktor**.
- Eine Initialisierung wie im vorherigen Beispiel wird daher wie folgt präziser ausgedrückt:

```
Bruch bruch1(bruch2);
```

Kopierkonstruktor bei Funktionsaufruf

- Funktionsdeklaration mit **call-by-value**:

```
int fkt_cbv(Bruch o){
  return o.getZaehler()*o.getNenner(); }
```

- Zur Herstellung einer **lokalen Kopie** (also zum Initialisieren des Parameters **o** auf dem Stack) wird der **Kopierkonstruktor** aufgerufen.
- Der **Kopierkonstruktor** wird bei jedem Aufruf dieser Funktion verwendet.

Referenzparameter beim Kopierkonstruktor

- Zur Vermeidung der Kopie eines aktuellen Parameters kann in C++ ein Referenzparameter benutzt werden (**call-by-reference**):

```
int fkt_cbr(Bruch & o){
   return o.getZaehler()*o.getNenner(); }
```

- Würde man dies nicht beim Kopierkonstruktor selbst machen, also ohne **&**, dann müsste erneut der Konstruktor aufgerufen werden. Dies würde zu einer unendlichen Rekursion führen!

Rückgabewert aus einer Funktion

- Auch bei der Rückgabe eines Wertes aus einer Funktion kann zwischen call-by-value und call-by-reference unterschieden werden:

```
Bruch kuerzen_cbv (Bruch b, int k)
{
      b.zaehler/=k; b.nenner/=k;
      return b;
}

void main() {
      Bruch bruch1(12, 1000);
      bruch1 = kuerzen_cbv(bruch1, 4);
}
```

Rückgabewert aus einer Funktion

- Nun folgt das gleiche Programm mit call-by-reference:

```
Bruch & kuerzen_cbr (Bruch & b, int k)
{
      b.zaehler/=k; b.nenner/=k;
      return b;
}

void main() {
      Bruch bruch1(12, 1000);
      bruch1 = kuerzen_cbr(bruch1, 4);
}
```

Selbstdefinierter Kopierkonstruktor

- Wann braucht man einen eigenen Kopierkonstruktor? Häufig dann, wenn dynamische Elemente im Spiel sind (Zeiger)!
- Der Standard-Kopierkonstruktor kopiert nur den/die Pointer, z.B. nur den Anker einer verketteten Liste.
- Man spricht von einer flachen Kopie.
- Will man jedoch die komplette Struktur kopieren, z.B. die ganze verkettete Liste, dann spricht man von einer tiefen Kopie.
- Für eine tiefe Kopie benötigt man einen selbstdefinierten Kopierkonstruktor.

Beispiel eines Kopierkonstruktors

```
class Person {
  char * name;
public:
  Person(char * name){          // Konstruktor
     int len = (int)strlen(name)+1;
     this->name = new char[len];
     memcpy(this->name, name, len); }

  Person(const Person & p){  // Kopierkonstruktor
     int len = (int)strlen(p.name)+1;
     name = new char[len];
     memcpy(name, p.name, len); }

  ~Person(){                        // Destruktor
     delete [] name;}
   ...
};
```

Konvertierkonstruktor

- Ein Konvertierkonstruktor ist ein Konstruktor mit einem Parameter, der nicht ein Objekt der Klasse ist.

```
class Bruch {
  int zaehler, nenner;
public:
  Bruch(int p){
    zaehler = p; nenner = 1; }
  Bruch(char p){
    zaehler = int(p); nenner = 1; }
  Bruch(string s){
    // string in der Form "int/int" }

...
```

Konvertierkonstruktor

- Folgende Schreibweisen zum Aufruf des Konvertierkonstruktors sind möglich:

```
Bruch b(3);
Bruch b = 3;
Bruch b = (Bruch) 3;
Bruch b = Bruch (3);
```

- Der Konvertierkonstruktor kann auch für die Konvertierung einer eigenen Klasse in eine andere genutzt werden:

```
Bruch b(3,4);
Quotient q(b);
```

Konvertierung von einer Klasse in eine andere

```
class Quotient {
  int zaehler, nenner;
public:
  Quotient (Bruch & b){
    zaehler = b.zaehler;
    nenner = b.nenner; }
  ...
}
void main(void) {
  Bruch b(3,4);
  Quotient q(b); }
```

Vorwärtsdeklaration (1)

```
1 ...
2 class Quotient {
3   ...
4   Quotient (Bruch & b){
5     zaehler = b.zaehler; ... }
6   ... };
7 class Bruch { ... };
8 ...
```

- Führt zu Kompilationsfehler, da der Kompiler in Zeile 4 die Klasse **`Bruch`** nicht kennt.

Vorwärtsdeklaration (2)

```
1 class Bruch;   // Vorwärtsdeklaration
2 class Quotient {
3  ...
4  Quotient (Bruch & b){
5    zaehler = b.zaehler; ... }
6  ... };
7 class Bruch { ... };
8 ...
```

- Führt zu Kompilationsfehler, da der Kompiler in Zeile 5 **`b.zaehler`** nicht kennt.

Vorwärtsdeklaration (3)

```
1 class Bruch;   // Vorwärtsdeklaration
2 class Quotient {
3  ...
4  Quotient (Bruch & b);
5  ...
6  ... };
7 class Bruch { ... };
8 Quotient::Quotient (Bruch & b) {
9   zaehler = b.zaehler; ... }
```

- So geht es!

Konstruktoren und dynamische Objekte

- **`Bruch* a; a = new Bruch;`**
- **`Bruch* a = new Bruch;`** **`// abgekürzt`**
- Ablauf:
  1. Zeigervariable der Klasse **Bruch** erzeugen: **Bruch *a**
  2. Mit **new** Speicherbereich für ein Objekt der Klasse **Bruch** auf dem Heap allokieren: **new Bruch**
  3. Aufruf eines Konstruktors der Klasse **Bruch** um den allokierten Speicherbereich zu initialisieren
  4. Rückgabe der Adresse des Speicherbereichs an die Zeigervariable a

Destruktoren

- Ein **Destruktor** ist das Gegenstück zum Konstruktor.
- Er hat **keine Parameter** und kann **nicht überladen** werden.
- Er wird **automatisch aufgerufen**, wenn der Gültigkeits-bereich eines Objektes verlassen wird, oder **`delete`** aufgerufen wird.
- Name: **`~Bruch();`**
- Es gibt einen **Standarddestruktor** des Compilers; er gibt nur den Speicher frei.
- Mit einem **eigen geschriebenen Destruktor** kann man Aktionen veranlassen, die mit dem zu 'destruierenden' Objekt zusammenhängen (Speicher löschen, Datei freigeben, 'tiefe' Struktur löschen).

Beispiel für einen eigenen Destruktor

```
class Person {
  char * name;
public:
  ...
  ~Person ()
     { /* Jede Person hat ihren eigenen Namen,
          der nun separat gelöscht werden muss */
       delete [] name;
     }
  ...
};
```

Standardmethoden und -operatoren

- Standardkonstruktor

```
Bruch();
```

- Defaultkopierkonstruktor

```
Bruch (const Bruch & b);
```

- Defaultdestruktor

```
~Bruch();
```

- Defaultzuweisungsoperator

```
Bruch c(1,2), d;
d = c;
```

Zusammenfassung

- Die Initialisierung von Instanzen erfolgt durch Konstruktoren.
- Der Compiler stellt einen Defaultkonstruktor (Standard-konstruktor) zur Verfügung. Dieser hat keine Parameter.
- Eine Klasse kann eigene Konstruktoren haben, diese unterscheiden sich durch ein Parameterprofil wie beim Überladen.
- Spezielle Konstruktoren mit Parametern sind: Kopierkonstruktor, Konvertierkonstruktor(en).
- Bei Aufrufen mit call-by-value wird der Kopierkonstruktor benutzt, um die lokale Kopie herzustellen. Der Compiler stellt einen Standard-Kopierkonstruktor zur Verfügung.
- Konstruktoren werden auch aufgerufen, wenn dynamische Objekte im Heap mit **new** erzeugt werden.

Zusammenfassung

- Beim Löschen von Objekten wird der Destruktor der Klasse aufgerufen:
  - Dynamische Objekte werden mit **`delete`** gelöscht.
  - Lokale Objekte werden am Ende eines Blockes bzw. Funktionsrumpfes vom Stack abgeräumt.
  - Wenn ein Programm zu Ende ist, werden dessen globale Objekte gelöscht.
- Der Compiler stellt einen Defaultdestruktor (Standarddestruktor) zur Verfügung. Dieser hat keine Parameter.
- Eine Klasse kann einen eigenen Destruktor haben.