Notizen AIK ProgTech 3: Operator Overloading

Klaus Kusche

Grundidee

- Man will die bekannten Operatoren **+**, **=**, **<=**, **<<** usw.
  auch für Operanden verwenden können, für die sie nicht vordefiniert sind
  (z.B. für die Objekte eigener Klassen).
- Man möchte alle Möglichkeiten, die C++ für Funktionen erlaubt
  (Vererbung, Overloading, d.h. mehrere Funktionen gleichen Namens
  für verschiedene Argument-Typen, ...) auch für Operatoren nutzen.
- Man macht das, indem man im Prototyp / in der Funktionsdefinition **`operator`**
  und das Rechenzeichen als *Funktions-Name* verwendet (also z.B. "**`operator+`**").
- Anzahl der Operanden (1 oder 2) und bestehende Vorrangregeln (z.B. ***** vor **+**)
  lassen sich dabei aber *nicht* ändern.

Methode oder Funktion?

- Wenn ein Operator eine *Methode* sein soll, so muss er
  in derjenigen Klasse definiert werden, zu der der ***linke** Operand* gehört:
  Ein Operator kann *nicht* in der Klasse des rechten Operanden definiert werden!

  ==› Steht links *kein Objekt* (sondern z.B. ein **`int`**) oder
  ein Objekt einer vordefinierten oder *fremden Klasse*, die man nicht ändern will
  (z.B. bei **<<** und **>>**: Links steht das **`iostream`**-Objekt, nicht das eigene Objekt!),
  so muss man den Operator als *globale Funktion* programmieren
  (außerhalb der Klasse)!
- Wird ein Operator *als Methode* definiert,
  so hat die Methode *einen Parameter weniger* als der Operator:
  Unäre Operatoren haben *gar keinen* Parameter (der *einzige* Operand ist das **`this`**),
  binäre Operatoren haben nur *einen* Parameter, nämlich den *rechten* Operanden
  (der *linke* ist immer das **`this`**).
- Wird der Code einer Operator-Methode *außerhalb* des **`class`** definiert, steht
  wie bei anderen Funktionsnamen der Klassenname mit **`::`** vor dem **`operator`** :

  ```
  myClass myClass::operator+(const myClass &b) const { ...
  ```
- Einige Operatoren (**=**, **`[ ]`**, Typumwandlungsoperator) können *nur als Methode*
  und nicht als Funktion definiert werden.
- Wird ein Operator als Funktion implementiert,
  so hat die Funktion so viele Parameter wie der Operator Operanden hat
  (also 1 bei unären und 2 bei binären Operatoren).
- Damit die Funktion trotzdem auf die **`private`**- und **`protected`**-Member
  des Operanden zugreifen kann, wird sie im **`class`** normalerweise als **`friend`**
  deklariert (üblich z.B. für **<<** und **>>**).

Prototyp und Returnwert

- Berechnet der Operator ein ***neues Ergebnis***, so gilt:
  - *Der Returnwert ist* ***keine Referenz***: Das in der Funktion berechnete, lokale Ergebnis-Objekt muss in den Aufrufer ***kopiert*** werden!

    Würde man eine Referenz auf das lokal berechnete Ergebnis zurückgeben, würde diese nach dem **`return`** ins Leere zeigen, da das lokale Objekt freigegeben wird!
  - *Die Operanden sind* **`const`***-Referenzen*, die *Methode als Ganzes* ist auch **`const`** (weil sie ja das **`this`**, nämlich den linken Operanden, nicht ändern darf!)

    ```
    myClass operator+(const myClass &b) const;
    ```
  - Im Code der Methode bzw. der Funktion wird ein *neues, lokales Objekt* (d.h. eine lokale Variable) *für das Ergebnis* angelegt und im **`return`** zurückgegeben.
- ***Ändert*** *der Operator einen der Operanden*, so gilt:
  - *Der Returnwert muss eine* ***Referenz*** *sein*: Er verweist *auf den geänderten Operanden* (das Original selbst, es darf *keine Kopie* davon sein!).
  - Bei einer *Funktion*:
    - Dieser Operand (meist der linke) *muss eine Referenz* sein, aber er darf *nicht* **`const`** sein.

      Beispiele:

      ```
      ostream &operator<<(ostream &outFile,
                          const myClass &val);
      ```

      ```
      istream &operator>>(istream &inFile, myClass &var);
      ```
    - Dieser Operand *muss* im **`return`** *zurückgegeben* werden:

      ```
      return outFile;
      ```
  - Bei einer *Methode* (wenn der geänderte Operand der *linke* bzw. einzige ist):
    - Das Ergebnis wird *im eigenen Objekt* (**`this`**) gespeichert, und es wird eine *Referenz auf das eigene Objekt zurückgegeben*:

      ```
      return *this;
      ```
    - Die Methode als Ganzes ist daher *nicht* **`const`**.

      Beispiel:

      ```
      myClass& myClass::operator=(const myClass& value);
      ```

Sonderfall `operator=`

- **=** ist *immer eine Methode*, eine Implementierung als Funktion ist nicht möglich!
- Als Erstes sollte immer auf die Selbst-Zuweisung **`a = a;`** geprüft werden.
  In diesem Fall kehrt das **=** sofort zurück, ohne etwas zu tun (weil man sonst bei den nachfolgenden Schritten den eigenen Wert verlieren könnte!):

  ```
  if (this == &value) return *this;
  ```
- Wenn die Objekte Pointer auf mit **`new`** angelegte Arrays enthalten und das bisher im **`this`** gespeicherte Array nicht für den neuen Wert passt:

  Zuerst altes Array mit **`delete`** *freigeben*, dann neues Array *anlegen*.
- Werte kopieren.
- Am Ende muss immer **`return *this;`** stehen.

Sonderfall Increment- und Decrement-Operator

- Zur Unterscheidung zwischen Prefix-**++** (**`++i`**) und Postfix-**++** (**`i++`**) wird beim Postfix-**++** im Prototyp ein zusätzlicher **`int`**-Parameter angegeben, der ignoriert wird. Dasselbe gilt für **−−**.
- Die Präfix-Operatoren geben eine Referenz auf das geänderte eigene Objekt als Returnwert zurück, die Postfix-Operatoren liefern einen neuen Wert “by value” (da ja der Returnwert vom Wert des eigenen Objektes verschieden ist).

Anmerkungen

- Weitere Operatoren mit Spezialbedeutungen sind nicht Stoff (z.B. **`[ ]`**, **`( )`** , Pointer-*****, Pointer-**&**, **−>** oder der Typumwandlungsoperator).
- Virtuelle Operatoren in abgeleiteten Klassen sind trickreich und bei den Schülern nicht Stoff.