Programmieren in C++

Überladen von Methoden und Operatoren

Inhalt

- Überladen von Methoden
- Überladen von Operatoren
- Überladbare Operatoren
- Operatoren als Instanzmethoden vs. freie Funktionen
- Indexoperatoren
- Zuweisungsoperator
- Inkrement-/Dekrement-Operatoren
- Typkonvertierungsoperator
- Literal-Operator
- Operatoren mit Move-Semantik

Überladen von Methoden

- Signatur einer Methode besteht aus
  - Namensraum, Klasse, Name, Parameterliste
    - Anzahl und Typen der Parameter (Parameterbezeichner sind irrelevant)
  - Rückgabetyp gehört nicht dazu
- alle Methoden müssen eine eindeutige Signatur haben
- Überladen von Methoden
  - wenn mehrere Methoden im selben Namensraum bzw. Klasse denselben Namen, aber dennoch nicht die gleiche Signatur haben
- Beispiel

```
class Point {
    double m_x, m_y, m_z;
public:
    Point& move(double x, double y = 0, double z = 0);
    Point& move(double delta[3]);
    Point& move(const Point& p);
};
```

Operatoren überladen

- Idee
  - nicht nur Methoden sondern auch Operatoren können überladen werden (bekanntes Beispiel: << für die Ausgabe)
  - ermöglicht schönere Syntax (infix anstatt präfix) als mit Methoden

```
Complex c1(2, 4), c2(2, -4);
Complex c = c1 + c2/10;
```

- Grundregeln
  - es können keine neuen Operatoren definiert werden
  - vorgegebene Vorrangregeln dürfen nicht verletzt werden
  - Überladen von && und || deaktiviert short-circuit-Evaluierung
  - mindestens ein Argument des Operators muss ein Objekt sein oder der Operator muss eine Instanzmethode sein
    → damit wird verhindert, dass die Operatoren der primitiven Datentypen verändert werden

Operatoren

- überladen erlaubt für

new	+	~	>	/=	|=	<<=	>=	++	->	%
delete	-	^	!	+=	%=	<<	==	--	()	[]
new[]	*	&	=	-=	^=	>>	!=	&&	->*	,
delete[]	/	|	<	*=	&=	>>=	<=	||	""	

- ab C++20
  - <=> *spaceship operator* entspricht dem compareTo aus Java
  - co_await gibt in Coroutine die Kontrolle an Aufrufer zurück

- überladen nicht erlaubt für

.	.*	::	? :

Operator als Funktionsaufruf

Element-Funktion	Syntax	Ersetzung durch
nein	`x ⊗ y`	`operator⊗(x,y)`
	`⊗ x`	`operator⊗(x)`
	`x ⊗`	`operator⊗(x,0)`
ja	`x ⊗ y`	`x.operator⊗(y)`
	`⊗ x`	`x.operator⊗()`
	`x ⊗`	`x.operator⊗(0)`
	`x = y`	`x.operator=(y)`
	`x(y)`	`x.operator()(y)`
	`x[y]`	`x.operator[](y)`
	`x->`	`(x.operator->())->`
	`(T)x`	`x.operator T()`

`T` ist Platzhalter für einen Datentyp

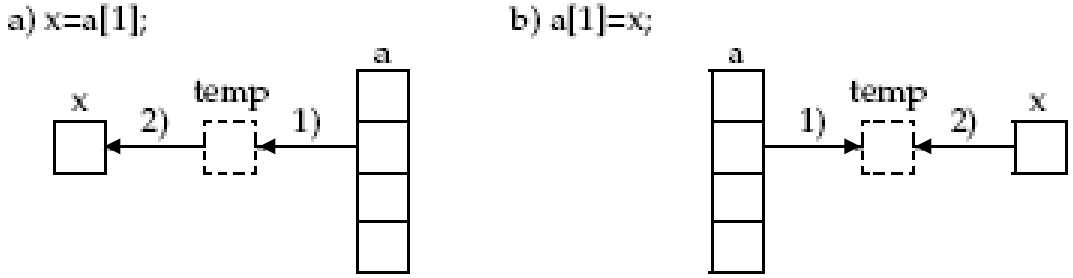
friend-Methoden

- Operatoren und Methoden können als freie Funktionen implementiert werden
  - haben keinen versteckten this-Parameter
  - haben standardmässig nur Zugriff auf öffentliche Attribute der Parameter
  - mittels friend kann der Zugriff auf alle Attribute erweitert werden
  - sollten primär für symmetrische Operatoren verwendet werden
  - ermöglicht dem Compiler mehr implizite Konvertierungen
- Beispiel

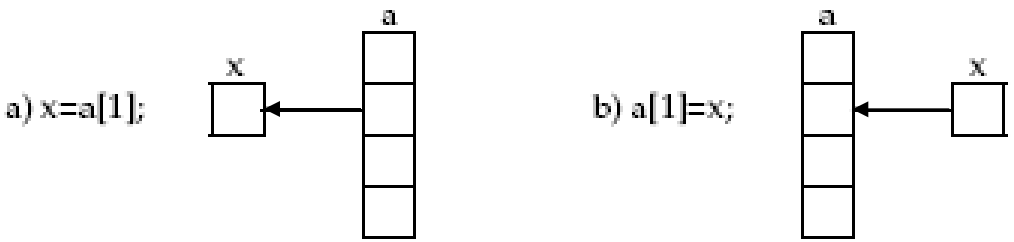
```
class Point {
public:
    friend bool operator<(const Point& lhs, const Point& rhs);
};

bool operator<(const Point& lhs, const Point& rhs) {
    return lhs.m_x < rhs.m_x || …;
}
```

Index-Operator []

- Rückgabe per Wert vs. Rückgabe per Referenz
  - T operator[](int index) const



  - T& operator[](int index)



- Indexbereich absichern
  - Range-Check nur in Form einer assert-Anweisung (#include <cassert>)!
  - gute Praktik: zusätzliche Methode at(size_t) mit Range-Check

Zuweisungsoperator =

- impliziter Zuweisungsoperator (flache Kopie)
  - wird kein eigener Zuweisungsoperator definiert → implizit generiert
  - der implizit generierte Zuweisungsoperator
    - ruft für jedes Attribut eines Objektes den passenden Zuweisungsoperator auf
    - bei einfachen Datentypen wird bitweise kopiert
- eigener Zuweisungsoperator (z.B. für tiefe Kopie)
  - in folgenden Situationen sinnvoll, wenn
    - Zeiger auf Objekte verändert werden müssen (z.B. tiefe anstatt flacher Kopie)
    - Konstanten angepasst werden müssen
  - Reihenfolge bei Erstellung von tiefen Kopien beachten
    - Speicherallokation
    - Inhalt in den neuen Speicher kopieren
    - Freigabe des vorher belegten Speichers
    - Aktualisieren der Verwaltungsinformation (z.B. Grösse des Speichers)

Inkrement-/Dekrement-Operatoren

- Unterscheidung zwischen Präfix- und Postfix-Version
  - T& operator++() // Einsatz: T x; ++x;
    - Präfix-Semantik: gibt den aktualisierten Wert zurück
  - T& operator++(int) // Einsatz: T y; y++;
    - Postfix-Semantik: gibt den alten Wert von y zurück

- Einsatzmöglichkeiten des Inkrement-Operators
  - im Zusammenhang mit einer Klasse Datum
    - korrektes Weiterschalten zum nächsten Datum
  - Iterator einer dynamischen Datenstruktur

- Dekrement-Operator
  - analog zum Inkrement-Operator

Typkonvertierungsoperator (T)

- Syntax
  - operator *T*(); // *T* steht für einen beliebigen Datentyp
  - kein Rückgabetyp und keine Parameter
- Wirkung
  - erlaubt die implizite und explizite Konvertierung in den Typ *T*
  - soll nur explizite Typumwandlung möglich sein, dann braucht es das Schlüsselwort explicit

    explicit operator *T*();
- Einsatzgebiete
  - wichtig: sehr sparsam einsetzen, um Mehrdeutigkeiten zu vermeiden
  - nur natürliche Typkonvertierungen anbieten
    - sinnvolles Beispiel: Vector::operator Matrix() const;
    - unsinniges Beispiel: Vector::operator Person() const;

Spaceship Operator <=> (1)

- Semantik
  - ähnlich zu Java's compareTo, aber Rückgabetyp ist entweder
    - auto oder
    - einer der nachfolgenden drei Vergleichsklassentypen

Rückgabetyp	als gleich bewertete Werte sind …	inkompatible Werte sind …
std::strong_ordering	ununterscheidbar	nicht erlaubt
std::weak_ordering	unterscheidbar	nicht erlaubt
std::partial_ordering	unterscheidbar	erlaubt

  - strong_ordering
    - equal, less, greater
  - weak_ordering
    - equivalent, less, greater
  - partial_ordering
    - equivalent, less, greater, unordered

Spaceship Operator <=> (2)

- Einsatz
  - anstatt mehrere relationale Operatoren zu implementieren, reicht der Spaceship Operator oft aus
    - früher mussten `operator<` und `operator==` implementiert werden

      um alle anderen relationalen Operatoren mithilfe von generischen Funktionen automatisch benutzen zu können
    - `heute werden operator<=> und operator== implementiert`

      dem Compiler ist erlaubt, relationale Operatoren im Code durch Einsatz des Spaceship-Operators zu ersetzen

      Beispiel

      ```
      if (a < b) {      ➔      if ((a <=> b) < 0) {
      ```
    - Default-Implementierung kann verwendet werden, wenn ein lexikografischer Vergleich passend ist (Reihenfolge der Attribute ist entscheidend)

      ```
      auto operator<=>(const Class& c) const = default;
      ```

Suffixe für Literale

- Zahlen-Suffixe für Literale

```
1.0     => double
1.0f    => float
1       => int
1U      => unsigned int
1L      => long
1UL     => unsigned long
1ULL    => unsigned long long
1LL     => long long
```

Literal-Operator (1)

- Benutzerdefinierte Literal-Operatoren
  - eigenes Suffix *_suffix* definieren, um Literale eines speziellen Typs zu markieren (vom Standard definierte Suffixe beginnen ohne _)
  - globale Operatoren
  - Ganzzahl
    - *type* operator"" *_suffix*(unsgined long long)
  - Fliesskommazahl
    - *type* operator"" *_suffix*(long double)
  - Zeichen
    - *type* operator"" *_suffix*(char)
  - Zeichenketten
    - *type* operator"" *_suffix*(const char*)
- Beispiel

```
std::complex c = 5i;              // c ist die komplexe Zahl (0,5)
constexpr auto duration = 10s;    // duration ist 10 Sekunden
```

Literal-Operator (2)

- Beispiele
  - *constexpr long double* operator"" _*km*(long double d) { return d*1000; }
  - *constexpr long double* operator"" _*deg*(long double d) { return d*pi/180; }
  - *Complex* operator"" _*i*(long double d) { return Complex(0,d); }
  - *Complex* operator"" _*i*(unsigned long long n) { return Complex(0,n); }

- Einsatz
  - std::complex c = 5i; // c ist die komplexe Zahl (0,5)
  - Complex c = 4_i; // eigene Klasse für komplexe Zahlen
  - auto dist = 10.0_km; // Distanz in Kilometer, dist in Meter
  - auto angle = 30_deg; // Winkel in Grad, angle in Radiant

Literal-Operator: Beispiel

```
class Distance {
    double m_meters;
public:
    Distance(double meters) : m_meters(meters) {}

    Distance operator+(const Distance& d)
    { return m_meters + d.m_meters; }
    friend ostream& operator<<(ostream& os, const Distance& d)
    { return os << d.m_meters << " m"; }
};

constexpr long double operator"" _cm(long double d) { return d/100; }
constexpr long double operator"" _km(long double d) { return d*1000; }

int main() {
    Distance d1 = 5._km;
    Distance d2 = 5._cm;

    cout << (d1 + d2) << endl;
}
```

Operatoren mit Move-Semantik

- Vermeidung unnötiger temporärer Objekte

```
class Matrix;

Matrix operator+(const Matrix& a, const Matrix& b);      // a + b
Matrix operator+(const Matrix& a, Matrix&& cd);          // a + c*d
Matrix operator+(Matrix&& ab, const Matrix& c);          // a*b + c
Matrix operator+(Matrix&& ab, Matrix&& cd);              // a*b + c*d

Matrix operator+(const Matrix& a, Matrix&& cd) {
        cd += a;
        return move(cd);
}
```