§2 Überladen von Fkt./Op. – Grundlagen

Leitideen: *In diesem Abschnitt sollen Sprachelemente behandelt werden, die gleichnamige Funktionen oder Operatoren bereit stellen. Dabei sind sowohl unterschiedliche Parametertypen als auch unterschiedliche Parameteranzahlen möglich.*

Die Auswahl der jeweiligen (Operator)funktion wird nach bestmöglicher Übereinstimmung zwischen Argument- und Parametertypen vorgenommen, ggf. erfolgen automatische Typumwandlungen.

Die Regeln erlauben die Unterscheidung zwischen Zeigern/Referenzen auf Konstanten und Variablen.

Überladen von Operatoren ist auch mit Komponentenfunktionen möglich (manchmal nur damit)

Auf Objektcodeebene werden überladene Funktionen bzw. Operatoren typischerweise in Funktionen mit unterschiedlichen Namen abgebildet, die von den Parametertypen abgeleitet sind.

§2 Überladen von Fkt./Op. I – Themenübersicht

- Voreinstellungen für Parameter von Funktionen
- Auswahl gleichnamiger Funktionen I,II
- Überladen mit Nichtkomponentenfunktionen
- 3D-Vektorbeispiel - arithmetische Operatoren
- Hintergrund: Datum- und Zeitfunktionen I,II
- Besonderheiten bei Vergleichs- und Inkrementoperatoren
- Anmerkungen zum Datumbeispiel
- Einschub: Referenzen - Funktionswerte
- Überladen mit Komponentenfunktionen
- 3D-Vektorbeispiel - Indexoperator
- Beispiel: Vektoren mit Indexbereich m..n
- Überladen des Funktionsaufrufs
- Funktionsobjekte - Begriff und Beispiel
- Typumwandlungen mit Konstruktoren
- Typumwandlungen mit Operatorfunktionen

Funktionen - Voreinstellungen für Parameter

- Voreinstellungen für Funktionsparameter möglich

  Bsp.:
  ```
  int strtoint(string s, int basis=10)
  strtoint(s);     // Dezimalwert
  strtoint(s,16);  // Hexadezimalwert
  ```

- Voreinstellung eines Parameters $\Rightarrow$ Voreinstellung für nachfolgende Parameter erforderlich

  Bsp.:
  ```
  int strtoint(string s="0", int basis=10)
  // okay
  int strtoint(string s="0", int basis)
  // nicht okay
  ```

- Parameter als Voreinstellungen unzulässig

  Bsp.:
  ```
  double flaeche(double a, double b=a)
  // unzulaessig
  ```

  Grund: Auswertreihenfolge implementierungsabhängig

- Parametervoreinstellungen für Operatorfunktionen (z.B. `operator+`) unzulässig

Auswahl gleichnamiger Funktionen

- Auswahl derjenigen Funktion, deren Parametertypen am besten zu den Argumenttypen passen.

  Bsp.: `pow(-2.0,3)` $\rightarrow$ `double pow(double,int)`
  `pow(2.0,0.5)` $\rightarrow$ `double pow(double,double)`

- Jeder Argumenttyp muss implizit in den entsprechenden Parametertyp konvertierbar sein ($\rightarrow$ Inf.bl.3,S.2).
- Eindeutigkeit der am besten passenden Funktion erforderlich.
- Genaue Übereinstimmung am besten: T, $T\&$ und const T gelten dabei i. allg. als gleich.
  Jedoch: const $T*$ $\neq$ $T*$ und const $T\&$ $\neq$ $T\&$.
- Der kleine Unterschied zwischen const $T*$ und $T*$ bzw. $T\&$ und const $T\&$ ermöglicht unterschiedliche Komponentenfunktionen für konstante und nicht konstante Objekte.
- Zahlbereichserweiterungen vorrangig vor anderen arithmetischen Konversionen.

Auswahl gleichnamiger Funktionen - Fortsetzung

- Konstruktoren mit *einem* Parameter bewirken implizite Typumwandlung (falls nicht als `explicit` deklariert). Zusätzlich gibt es Typumwandlungsoperatoren. („Benutzerdefinierte Typumwandlungen“)
- Vorrang der arithmetischen vor benutzerdefinierten Typumwandlungen, weil Hinzufügen neuer Klassen vorhandene Typumwandlungen nicht ändern soll.
- Nur eine benutzerdefinierte Typumwandlung pro Argument.

- *Vorsicht:* Über Templates definierte Klassen können sich anders als explizit definierte Klassen gleicher Bauart verhalten, weil Typumwandlungen bei der *Erzeugung* konkreter Funktionen aus Funktionstemplates *nicht* erfolgen:

```
complex<double> i(0,1);
cout << 1+i;     // funktioniert nicht
cout << 1.0+i;   // funktioniert
```

Überladen von Operatoren mit Nichtkomp.fkt.

Allgemeines

- Überladen des Operators `@` durch Operatorfunktion
  x`@`*y* → `operator@(`*x*`,`*y*`)` binär
  `@`*x* → `operator@(`*x*`)` unär
- Überladen von Operatoren durch Nichtkomponentenfkt. erfordert keine befreundete Funktionen (außer im Fall privater Klassenkomponenten)
- Korrekte Behandlung von Rückgabewert und Seiteneffekten notwendig
  Fehlerträchtig: Vergessen des Rückgabewerts bei Ein/Ausgabeoperatoren und Zuweisungsoperatoren
- Überladen der Operatoren `= () [] ->` *nur* durch Komponentenfunktionen möglich
- *Kein* Überladen der Operatoren `. .* :: sizeof ?:`

3D-Vektorbeispiel - Arithmetische Operatoren

- Datenkomponenten *public* zu Demonstrationszwecken
- Standardkonstruktor muss Datenkomp. initialisieren
- Hier *keine* automatische Erzeugung des Standardkonstruktors („POD – plain old datatype“)
- Kopierkonstruktor und Zuweisungsoperator automatisch erzeugt
- Referenzrückgabe lokaler Variablen unzulässig
- Referenzrückgabe bei arithmetischen Zuweisungs-operatoren wie in der Standardbibliothek

Ausdruck	*Wert*	*Typ*	*Seiteneffekt*
`a+b`	$a + b$	`Vektor3d`	keiner
`a+=b`	$a + b$	`Vektor3d`	$a \rightarrow a + b$
`a*=lambda`	$\lambda \cdot a$	`Vektor3d`	$a \rightarrow \lambda \cdot a$
`a*lambda`	$\lambda \cdot a$	`Vektor3d`	keiner
`lambda*=a`	*nicht sinnvoll*		
`lambda*a`	$\lambda \cdot a$	`Vektor3d`	keiner
`a^b`	$a \times b$	`Vektor3d`	keiner

Hintergrund: Datum- und Zeitfunktionen

Unterschieden werden:

Prozessorzeit hohe Zeitauflösung (Linux amd64: Mikrosek.) zur Messung von Programmlaufzeiten

Kalenderzeit umspannt längere Zeiträume (Linux i386/amd64: Jahrzehnte/Jahrtausende), geringe Zeitauflösung (Sekunde)

Prozessorzeitfunktionen (`<ctime>`)

Datentyp: `clock_t` (Linux amd64: `long`)

- `clock` - Liefert Prozessorzeit ab impl.definiertem Zeitpunkt
  Messung von Programmlaufzeiten durch Differenzbildung
  Umrechnung in Sek. per Divison durch `CLOCKS_PER_SEC`
  Linux i386: Überlauf bereits nach kurzer Zeit (1 Std.)
- Linux/Unix: Systemaufruf `times` erlaubt feineren Zugriff
  Vorsicht: Umrechnung in Sek. *nicht* mit `CLOCKS_PER_SEC`
- C++11: Standardbibliothek (`<chrono>`) *evtl. später!*

Hintergrund: Datum- und Zeitfunktionen II

Kalenderzeitfunktionen (`<ctime>`)

- Interne Zeitdarstellung: `time_t` (Linux amd64: `long`)
  Linux/Unix: Sekunden seit 1.1.1970 GMT
- Kalenderdaten: `tm` - Komponenten für Datum, Uhrzeit sowie Wochentag, Tag im Jahr und Sommerzeit
  Vorsicht: Jahre ab 1900, Monate seit Januar (0-11)
  Bsp.: November 2022: `tm_mon`: 10, `tm_year`: 122
- Oft Zeiger als Argumente und Rückgabewerte
- Wichtigste Funktionen:

`time`	Aktueller Zeitpunkt in interner Darstellung
`localtime`	Interne Darstellung $\rightarrow$ Kalenderzeitrecord
`mktime`	Kalenderzeitrecord $\rightarrow$ Interne Darstellung Normalisiert ggf. die Komponenten des Kalenderzeitrecords
`strftime`	Formatierte Ausgabe der Kalenderdaten auf Zeichenkette

Vergleichs- und Inkrementoperatoren

Besonderheiten

- STL enthält Funktionstemplates für `!= > >= <=`
  daher nur Definitionen für $<$ und $==$ erforderlich!
  Aktivierung:
  ```
  #include <utility>
  using namespace rel_ops;
  ```
- `++`$x \rightarrow$ `operator++(`x`)`
  x`++` $\rightarrow$ `operator++(`x`,0)` zusätzlicher `int`-Parameter

Anmerkungen zum Datumbeispiel

- Verwendete Kalenderzeitfunktionen (`<ctime>`):
  `mktime` Liefert Zeit in Sek. seit 1970 *und* ergänzt bzw. korrigiert Kalenderdaten
  `strftime` Gibt Kalenderdaten entsprechend Formatangabe auf eine C-Zeichenkette aus
- Kalenderzeitrecord `tm`
  Komp.: `tm_mday tm_mon tm_year tm_hour ...`
  Vorsicht: Jahre ab 1900, Monate $\in \{0, \ldots 11\}$
- Nur Operatorfunktionen zu $<$ und $==$ notwendig
- Eingabefehlerbehandlung in `operator>>` mit `stream.setstate(ios::failbit)`
- `boolalpha` für Ausgabe von `true` bzw. `false` an Stelle von 1 und 0
- *Ausdruck* *Wert* *Seiteneffekt*
  `++tm` $\texttt{tm}_{neu}$ $\texttt{tm}_{neu} = \texttt{tm}_{alt} + 1\ \texttt{Tag}$
  `tm++` $\texttt{tm}_{alt}$ $\texttt{tm}_{neu} = \texttt{tm}_{alt} + 1\ \texttt{Tag}$
  Referenzparameter wegen Seiteneffekt
  Fkt.wert beim Postfixop. Kopie, beim Präfixop. Referenz

Einschub: Referenzen - Funktionswerte

Zweck

- Vermeidung unnötiger Kopien

  Beispiel:

  ```
  typedef complex<double> Complex;
  istream& operator>>(istream& stream, Complex& z)
  ```

  stream soll Ergebnis sein
  (keine Kopie, sondern *derselbe* Strom)

- Funktionsergebnis soll Schreibzugriff ermöglichen

  Beispiel:

  ```
  vector<double> a(3); // a[i]==0 (i=0,..,2)
  a.front() = 5;       // a[0]==5;
  ```

  Vereinbarung in `<vector>`: `T& front() { ... }`

Caveat:

- Bei Ergebnistyp Referenz: Rückgabe einer lokalen Variablen unzulässig

Überladen mit Komponentenfunktionen

- x`@`y $\rightarrow$ x`.operator@(`y`)` binär
  `@`x $\rightarrow$ x`.operator@()` unär
- Überladen der meisten Operatoren sowohl durch Komponentenfunktionen als auch durch Nichtkomponentenfunktionen möglich
- Operatoren `[]` `()` `=` `->` *nur* durch Komponentenfunktionen überladbar
- Auswahl der passenden Operatorfunktion gemäß Parametertypliste: Hypothetischer erster Parameter entsprechend Typ von x ergänzt
- `const-Attribut` bei Komp.fkt. erlaubt unterschiedliche Behandlung von konstanten und variablen Objekten
- `const` T x bzw. `const` T`&` x : Bei Aufruf von x`@`y bzw. `@`x Auswahl von `operator@ const` vorrangig vor `operator@`

3D-Vektorbeispiel - Indexoperator

Ziel: Zugriff auf Komponenten x, y, z des Vektors $a \in \mathbb{R}^3$ auch über $a[1], a[2], a[3]$

```
Vektor a; const Vektor c; double x;
```

- `a[1]=x` $\rightarrow$ `a.operator[](1)=x` $\hat{=}$ `a.x=x`
  (funktioniert nur, weil Referenz auf a.x geliefert wird)
- `x=a[1]` $\rightarrow$ `x=a.operator[](1)` $\hat{=}$ `x=a.x`
  (würde auch funktionieren, wenn Kopie von a.x geliefert wird)
- `c[1]=x`
  (nicht erlaubt; würde auch nicht funktionieren, weil `c.operator[] const` eine Kopie von c.x liefert)
- `x=c[1]` $\rightarrow$ `x=c.operator[](1)` $\hat{=}$ `x=` Kopie v. `c.x`
  (benutzt `operator[] const` und funktioniert)

Beispiel: Vektoren mit Indexbereich m..n

Ziele

1. `fvektor a(n),b(n,m),c(n,m,x);` Voreinst. m=1,x=0
2. Intern: C-Vektor-Komponente (Indizierung ab 0)
3. Variable Dimensionierung, deshalb `ap` Zeiger auf dynamisch allozierten Bereich
4. Indexoperator: `a[i]` $\rightarrow$ `ap[i-m]`

Anmerkungen

- Standardkonstruktor über Parametervoreinstellungen in allgemeinerem Konstruktor enthalten
- Konstruktor alloziert Speicherplatz mit `new`
- Destruktor deallociert Speicherplatz mit `delete[]`
- Indexoperator wie üblich als konstante und nicht konstante Operatorfunktion
- Komp.fkt. `ausgeben` statt überladenem Operator, um Name des fvektors zusätzlich auszugeben
- Kopierkonstruktor: alloz. zuerst Speicher und kopiert dann
- Zuweisungsoperator: deallociert zuerst Speicher, dann weiter wie beim Kopierkonstruktor, gibt Objekt zurück

Überladen des Funktionsaufrufs

Klasse F, Objekt f, Komp.fkt. `operator()` $(x_1, x_2, \ldots, x_k)$
Wirkung: $f(x_1, x_2, \ldots, x_k) \rightarrow f$`.operator()`$(x_1, x_2, \ldots, x_k)$

Beispiele

1. Polynombsp. – `p(x)` statt `p.value(x)`
   Änderungen:

```
class Polynom        double value(double x)
                  →  double operator()(double x)
main                 p.value(x)
                  →  p(x)
```

2. Matrixbsp. – `a(i,j)` statt `a[i*n+j]`
   Noch besser wäre: `a[i][j]` (schwerer realisierbar)
   Unterschiede zum Polynombsp.:
   Operatorfunktion hat 2 statt 1 Parameter
   Operator `()` überladen für konst. und nichtkonst. Matrixobj.
   Referenz als Rückgabewert, damit `a(i,j)=x` möglich

Funktionsobjekte - Begriff und Beispiel

Funktionsobjekt: Objekt einer Klasse („Funktorklasse“), bei der der Operator `()` überladen ist
(Verwendung des Begriffs, wenn funktionsartiger Aufruf Zweck der Klasse)

Beispiel

Normalverteilung $N(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma}}$

Implementierung als Funktionsobjekt:
Parameter μ, σ können beim Funktionsaufruf verborgen bleiben.
Statt globaler Variablen Initialisierung über Konstruktoraufruf oder Zuweisung mittels Komponentenfunktionen, z.B.

```
nv.mw()=5; nv.stdabw()=2;
```

(deshalb: Referenz als Rückgabewert)
Bem.: Angabe der Parameter beim Aufruf möglich, z.B

```
Normalverteilung(2,3)(x)
```

Nachteil: Bei jedem Aufruf Erzeugung eines Temporärobjekts

Typumwandlungen mit Konstruktoren

- Konstruktordef. `C(T t){...}` bewirkt Typumwandlung `T` → `C`, z.B. `int` → `fvektor`
- Diese Typumwandlung erfolgt auch implizit
  Problem: möglicherweise unerwünscht
  Bsp.:
  ```
  fvector a(3)
  a=2 // Schreibfehler
      // richtig waere a[1]=2
  ```
  `a=2` → `a.operator=(2)`
  Typumwandlung: `2` → `fvektor(2)`
  d.h. $a[1] = 0$, $a[2] = 0$ (Länge 2)
  statt $a[1] = 2$, $a[2] = 0$, $a[3] = 0$ (Länge 3)
- Attribut `explicit` in Konstruktordeklaration verhindert implizite Typumwandlung
- Wird häufig in der STL benutzt, z.B.

```
explicit vector(size_type n);
// Konstruktordeklaration in
// Template-Klasse vector
```

Typumwandlungen mit Operatorfunktionen

Situation: `C` Klasse, `c` Objekt, `T` eingebauter Datentyp

- Konstruktor: Typumwandlung `T` → `C` möglich,
  nicht aber `C` → `T`
- `operator T()` in Klasse `C` bewirkt Typumw. `C` → `T`
- *Kein* Ergebnistyp (auch nicht `void`), wie bei Konstruktoren.
- *Unterschied:* `return` mit passendem Argument nötig
  (Konstruktor: kein oder leeres `return`)
- Expliziter Aufruf z.B. durch Cast-Operator: `(T) c`
  oder `static_cast<T>(c)`
- *Vorsicht:* Nicht verwechseln mit `operator()`
- `operator void*()` in STL-Streamklassen bewirkt z.B.
  in `if (cin >> s)` die Umwandl. `istream` → `void*`
- Andere Anwendung: Def. von Typumwandlung `C` → `D`
  ohne Änderung der Klasse `D`
- *Bsp.:* Typumw. `fvektor` → `vector<double>`
  `fvektor a(n,m)` → `vector<double> b(n+1)`