Einführung in die Programmierung mit C++

Überladung

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Funktionen

- Intrinsische Funktionen
- Nutzerdefinierte Funktionen

Operatoren

- Zuweisung
- Feldelementzugriff
- Unäre Operationen
- Binäre Operationen
- In-/Dekrementierung als Prä-/Postfixoperatoren
- Relationale Operationen
- Arithmetische Zuweisungen
- Ein- und Ausgabeströme

Überladung ermöglicht Polymorphie von Funktionen und Operatoren.

- Funktionen, z.B. **void** print(**const** T &x);

```
1 print(42); // int
2 print("Hi!"); // char*
3 print(MyClass(3.1415)); // MyClass
```

- Operatoren z.B. std::ostream& **operator**<<(ostream& s, **const** T &x);

```
1 std::cout << 42.42 // double
2     << std::string("Hi!") // std::string
3     << MyOtherClass(1,2,3) // MyOtherClass
4     << std::endl; // std::endl
```

Unterschieden werden die Varianten mittels ihrer Signatur (nicht Rückgabetyp).

- Die Bedeutung der intrinsischen Funktionen / Operatoren ist für die Standarddatentypen von C++ vordefiniert. Durch Definition einer Klasse wird ein neuer Datentyp geschaffen, für den die Semantik der intrinischen Funktionen / Operatoren (falls es Sinn macht) durch Überladung neu bestimmt werden kann.

- Was in den überladenen Funktionen / Operatoren getan wird, ist recht beliebig:
  - Sorge für Konsistenz mit der vordefinierten Bedeutung der intrinsischen Funktionen / Operatoren.
  - Sorge für Konsistenz zwischen den verschiedenen überladenen intrinsischen Funktionen / Operatoren.

Überladung

Fallstudie


```
1  /// static vector type and arithmetic
2  template<int N, typename T>
3  class vector {
4    T vals[N];
5  public:
6    /// constructor initializes to zero
7    vector() {
8      for (size_t i=0;i<N;i++) vals[i]=T(0);
9    }
10   /// constructor initializes to val
11   vector(const T& val) {
12     for (size_t i=0;i<N;i++) vals[i]=T(val);
13   }
14   /// copy constructor
15   vector(const vector &v) {
16     for (size_t i=0;i<N;i++) vals[i]=v.vals[i];
17   }
18   /// print vector to screen
19   void print() const { ... }
20   // ... to be extended
21 };
```

- Wir definieren eine Klasse vector zu Illustration der Überladung.
- Zwei Konstruktoren unterstützen die Initialisierung zum Zeitpunkt der Allozierung.
- Der (hier triviale, da vom automatisch generierten nicht unterscheidbare) → Kopierkonstruktor wird im Kontext der dynamischen Speicherverwaltung genauer betrachtet.
- Eine Objektfunktion zur Ausgabe auf den Bildschirm wird zur Verfügung gestellt.

Fallstudie

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y=4.2,z(y);
4
5    x.print(); y.print(); z.print();
6
7    return 0;
8  }
```

generiert die Ausgabe

```
[ 0 0 ]
[ 4.2 4.2 ]
[ 4.2 4.2 ]
```

Outline

Überladung von Funktionen

Intrinsische Funktionen


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6  /// non-member friends
7  template<int M, typename S>
8  friend vector<M,S> sin(const vector<M,S>&);
9  };
10
11 /// sine as non-member function
12 template<int N, typename T>
13 vector<N,T> sin(const vector<N,T>& v) {
14   vector<N,T> w;
15   for (size_t i=0;i<N;i++) w.vals[i]=sin(v.vals[i]);
16   return w;
17 }
```

- Überladung von, z.B. sin für Argumente vom Typ vector als externe Funktion;
- **friend** für effizienteren Elementzugriff ohne Indirektion über spezielle Zugriffsfunktion;
- Beachte verschiedene template-Argumentnamen (M,S vs. N,T) zur Vermeidung von Namenskonflikten.

Intrinsische Funktionen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y=4.2,z(y);
4
5    sin(x).print(); sin(y).print(); sin(z).print();
6
7    return 0;
8  }
```

resultiert in der Ausgabe

```
[ 0 0 ]
[ -0.871576 -0.871576 ]
[ -0.871576 -0.871576 ]
```

Überladung von Funktionen

Nutzerdefinierte Funktionen


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6  /// non-member friends
7  template<int M, typename S>
8  friend S l2_norm(const vector<M,S>&);
9  };
10
11 /// l2-norm of vector as non-member
12 template<int N, typename T>
13 T l2_norm(const vector<N,T> &v) {
14   T n=0;
15   for (int i=0;i<N;i++)
16     n+=pow(v.vals[i],2); // pow, += and ...
17   return sqrt(n); // ... sqrt to be defined for T
18 }
```

- Der Compiler generiert Überladungen von `l2_norm` für Varianten von `vector` für verschiedene `N` und `T`.
- Die verwendeten intrinsischen Funktionen und Operatoren (hier: `pow`, `+=`, `sqrt` müssen für Argumente vom Typ `T` definiert sein.

Nutzerdefinierte Externe Funktionen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y=4.2,z(y);
4
5    std::cout << l2_norm(x) << std::endl
6              << l2_norm(y) << std::endl
7              << l2_norm(z) << std::endl;
8
9    return 0;
10 }
```

resultiert in der Ausgabe

```
0
5.9397
5.9397
```

Überladung von Funktionen

Nutzerdefinierte Objekt-[und Klassen-]funktionen


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    /// l1_norm as member function
7    T l1_norm() const {
8      T n=0;
9      for (int i=0;i<N;i++) n+=fabs(vals[i]);
10     return n;
11   }
12 };
```

- Objekt-[und Klassen-]funktionen können nach dem selben Prinzip überladen werden.
- hier: Varianten von l1_norm für verschiedene N und T sind keine Überladungen, da sie zu verschiedenen Klassen gehören.
- Die verwendeten intrinsischen Funktionen und Operatoren (hier: fabs, += müssen für Argumente vom Typ T definiert sein.

Nutzerdefinierte Objektfunktionen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y=4.2,z(y);
4
5    std::cout << x.l1_norm() << std::endl
6              << y.l1_norm() << std::endl
7              << z.l1_norm() << std::endl;
8
9    return 0;
10 }
```

resultiert in der Ausgabe

```
0
8.4
8.4
```

AUFWACHEN!

Was wird nach erfolgreicher Übersetzung und Ausführung des folgenden C++ Programms auf den Bildschirm ausgegeben?

```
1  #include<cmath>
2  #include<iostream>
3
4  template<typename T>
5  struct A { T v; };
6
7  template<typename T>
8  A<T> sin(const A<T> &x) { A<T> y; y.v=cos(x.v); return y; }
9
10 int main() {
11   const double pi=4*atan(1); A<double> x; x.v=pi;
12   std::cout << sin(x).v << std::endl;
13   return 0;
14 }
```

Wach?

```
1  #include<cmath>
2  #include<iostream>
3
4  template<typename T>
5  struct A { T v; };
6
7  template<typename T>
8  A<T> sin(const A<T> &x) { A<T> y; y.v=cos(x.v); return y; }
9
10 int main() {
11   const double pi=4*atan(1); A<double> x; x.v=pi;
12   std::cout << sin(x).v << std::endl;
13   return 0;
14 }
```

-1

Überladung von Funktionen

Generalisierung

Dasselbe Prinzip gilt analog für binäre, ternäre, etc. intrinsische sowie nutzerdefinierte Funktionen, z.B.

```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6  // non-member friends
7  template<int M, typename S>
8  friend const vector<M,S>& max(const vector<M,S>&, const vector<M,S>&);
9  };
10
11 // non-member overloads for binary intrinsic function
12 template<int N, typename T>
13 const vector<N,T>& max(const vector<N,T>& u, const vector<N,T>& v) {
14   if (u.l1_norm()>=v.l1_norm()) return u;
15   return v;
16 }
```

Nutzerdefinierte Objektfunktionen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x=-4.2,y=4.2;
4
5    max(x,y).print();
6    max(y,x).print();
7
8    return 0;
9  }
```

resultiert in der Ausgabe

```
[ -4.2 -4.2 ]
[ 4.2 4.2 ]
```

Outline

Überladung von Operatoren

Überblick

Priorität	Operator	Beschreibung
1	`::`	scope
2	`() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`	postfix
3	`++ -- ! sizeof new delete`	unary (prefix)
	`* &`	indirection and reference (pointers)
	`+ -`	unary sign operator
4	`(type)`	type casting
5	`.* ->*`	pointer-to-member
6	`* / %`	multiplicative
7	`+ -`	additive
8	`<< >>`	shift
9	`< > <= >=`	relational
10	`== !=`	equality
11	`&`	bitwise AND
12	`^`	bitwise XOR
13	`|`	bitwise OR
14	`&&`	logical AND
15	`||`	logical OR
16	`?:`	conditional
17	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment
18	`,`	comma

Nur fünf der insgesamt 49 in C++ definierten Operatoren dürfen **nicht** überladen werden:

`.`	Elementauswahl (`X.a`)
`.*`	Elementauswahl durch Zeiger (`X.*a`)
`::`	Bereichsauflösung (`X::a`)
`?:`	Tenärer Operator (bedingte Auswertung) (`x = y<0?-y:y;`)
`sizeof`	Größe von Objekten

- Unäre Operatoren können durch Objektfunktionen ohne Parameter oder durch externe Funktionen mit einem Parameter definiert werden, z.B.

```
1 vector<N,T> vector<N,T>::operator-() const; // elementwise unary minus
2 vector<N,T> operator+(const vector<N,T>&); // elementwise unary plus
```

- Binäre Operatoren können durch Objektfunktionen mit einem Parameter oder durch externe Funktionen mit zwei Parametern definiert werden, z.B.

```
1 vector<N,T> operator+(const vector<N,T>&) const; // elementwise binary
    addition
2 vector<N,T> operator-(const vector<N,T>&, const vector<N,T>&); //
    elementwise binary subtraction
```

- Ausnahmen: Die Operatoren =, [], (), -> müssen Objektfunktionen sein, damit garantiert ist, dass ihre erster Operand ein *lvalue* ist, d.h. ein Ausdruck, der ein Objekt referenziert und damit z.B. auf der linken Seite einer Zuweisung stehen darf (aber z.B. keine Konstante).

... liegt prinzipiell im in ihrem Ermessen (außer =, [], (), ->).

Was sind die Unterschiede?

- Objektfunktionen
  - Definiert innerhalb der Klasse (oder mit class_name:: vor dem Funktionsnamen auch außerhalb der Klasse)
  - Erster Operand ist Aufrufer
  - Hier: Erster Operand ist das "Heimatobjekt" ***this**
- externe Funktionen
  - Definiert außerhalb der Klasse
  - Aufruf vom Compiler generiert, d.h. Operator hat kein aufrufendes Objekt
  - Damit: Erster Operand ist erstes Argument
  - Operator existiert ausserhalb der Klasse
    - Kein Zugriff auf **private** - Daten der Klasse
    - Abhilfe: Operator als **friend** deklarieren

Welche Auswirkung haben diese Unterschiede?

- externe Funktionen
  - Alle Operanden sind Argumente eines Funktionsaufrufs.
  - Per Überladung möglich: v=v+1 und v=1+v.
  - Implizite Typkonvertierung durch Compiler für beide Operanden.
- Objektfunktionen
  - Erster Operand ist aufrufendes Objekt, d.h. Entscheidung welche Operator-Implementierung benutzt wird, wird vom Compiler nach Auswertung des ersten Operanden getroffen.
  - Per Überladung möglich: v=v+1, aber nicht v = 1+v.
  - Keine implizite Typkonvertierung durch Compiler für ersten Operanden, wohl aber für den zweiten.

Faustregel für die Entscheidung:

- Unäre Operatoren als Objektfunktion.
- Binäre Operatoren, die beide Operanden gleichartig behandeln (z.B. unverändert lassen), als externe Funktionen.
- Bei binären Operatoren, die beide Operanden nicht gleichartig behandeln (z.B. wird der linke Operand verändert, aber nicht der Rechte), kann es nützlich sein, sie als Objektfunktionen des linken Operanden zu definieren, falls sie Zugriff auf die **private** Elemente des Operanden benötigen.

Überladung von Operatoren

Zuweisung


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    vector& operator=(const vector &v) {
7      if (&v!=this)
8        for (size_t i=0;i<N;i++) vals[i]=v.vals[i];
9      return *this;
10   }
11 };
```

- Ein Standardzuweisungsoperator wird durch den Compiler automatisch generiert. Er kopiert den belegten statischen Speicher.
- Nutzerdefinierte Zuweisungen sollten bei Selbstzuweisungen der Form `x=x;` das Kopieren der Daten für Effizienz (bzw. später – im Kontext dynamischer Speicherverwaltung – auch Korrektheit) verweiden.

Nutzerdefinierte Objektfunktionen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y=4.2,z;
4
5    y.print(); z.print();
6    z=y;
7    y.print(); z.print();
8
9    return 0;
10 }
```

resultiert in der Ausgabe

```
[ 4.2 4.2 ]
[ 0 0 ]
[ 4.2 4.2 ]
[ 4.2 4.2 ]
```

AUFWACHEN!

Was wird nach erfolgreicher Übersetzung und Ausführung des folgenden C++ Programms auf den Bildschirm ausgegeben?

```
1  class A {
2    int i=0;
3    A& operator=(A& a) {
4      for (;&a==this;) return a;
5      i=42; return *this;
6    }
7    friend int main();
8  };
9
10 #include<iostream>
11
12 int main() {
13   A x,y; x=x; y=x;
14   std::cout << y.i << std::endl;
15   return 0;
16 }
```

Wach?

```
1  class A {
2    int i=0;
3    A& operator=(A& a) {
4      for (;&a==this;) return a;
5      i=42; return *this;
6    }
7    friend int main();
8  };
9
10 #include<iostream>
11
12 int main() {
13   A x,y; x=x; y=x;
14   std::cout << y.i << std::endl;
15   return 0;
16 }
```

42 (Klar, aber warum?)

Überladung von Operatoren

Feldelementzugriff


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    // read/write element access
7    T& operator[](const size_t i) {
8      assert(i>=0&&i<N);
9      return vals[i];
10   }
11
12   // read-only element access
13   const T& operator[](const size_t i) const {
14     assert(i>=0&&i<N);
15     return vals[i];
16   }
17 };
```

- Der Dereferenzierungsoperator [] für Felder sollte (muss) sowohl für Schreib-/Lese- als auch exklusiven Lesezugriff implementiert werden.
- Letzterer modifziert sein "Heimatobjekt" (∗**this**) nicht (⇒ 2× **const**).
- Die Zulässigkeit des Elementzugriffs sollte sichergestellt werden ⇒ z.B. assert.

Nutzerdefinierte Objektfunktionen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x;
4
5    x.print();
6    std::cin >> x[0];
7    std::cout << x[0] << std::endl;
8    x.print();
9
10   return 0;
11 }
```

resultiert in der Ausgabe

```
[ 0 0 ]
4.2
4.2
[ 4.2 0 ]
```

Überladung von Operatoren

Unäre Operationen als Objektfunktionen


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    /// unary minus as member function
7    vector operator-() const {
8      vector<N,T> v;
9      for (size_t i=0;i<N;i++) v.vals[i]=-vals[i];
10     return v;
11   }
12
13 };
```

- Unäre Objektfunktionen benötigen kein explizites Argument. Sie agieren implizit auf ihrem "Heimatobjekt" ***this**.
- Hier wird eine Kopie mit negierten vals Werten des "Heimatobjekts" zurückgegeben.
- Die unäre Negation muss für Objekte vom Typ T definiert sein.

Überladung von Operatoren

Unäre Operationen als Externe Funktionen

Unäre externe Funktionen benötigen ein explizites Argument vom Typ des Zielobjekts (hier: **const** vector<M,S>&).

```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5  };
6
7  /// unary plus as non-member function
8  template<int N, typename T>
9  vector<N,T> operator+(const vector<N,T>& u) {
10   return u;
11 }
```

- Hier wird eine Kopie des "Heimatobjekts" zurückgegeben.
- Es wird kein Zugriff auf die **private** Variablen des Zielobjekts benötigt. Wäre das der Fall (z.B. bei Negation als externe Funktion), dann muss der Operator zum **friend** gemacht werden.
- Das unäre Plus muss für Objekte vom Typ T definiert sein.

Unäre Operationen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> y=4.2,z(y);
4
5    (-+y).print();
6    (+-z).print();
7
8    return 0;
9  }
```

resultiert in der Ausgabe

```
[ -4.2 -4.2 ]
[ -4.2 -4.2 ]
```

Überladung von Operatoren

Binäre Operationen als Objektfunktionen


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    /// binary multiplication as member function
7    vector operator*(const vector& v) const {
8      vector<N,T> w;
9      for (size_t i=0;i<N;i++) w.vals[i]=vals[i]*v.vals[i];
10     return w;
11   }
12
13 };
```

- Binäre Objektfunktionen benötigen ein explizites Argument für den zweiten Operand. Der erste Operand ist das "Heimatobjekt" ***this**.
- Hier wird eine Kopie mit elementweisen Produkten der `vals` Werte der Operanden zurückgegeben.
- Die binäre Multiplikation muss für Objekte vom Typ `T` definiert sein.

Überladung von Operatoren

Binäre Operationen als Externe Funktionen

```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    /// non-member friends
7    template<int M, typename S>
8    friend vector<M,S> operator/(
9      const vector<M,S>&,
10     const vector<M,S>&);
11 };
12
13 /// division as non-member function
14 template<int N, typename T>
15 vector<N,T> operator/(
16       const vector<N,T>& u,
17       const vector<N,T>& v) {
18   vector<N,T> w;
19   for (size_t i=0;i<N;i++)
20     w.vals[i]=u.vals[i]/v.vals[i];
21   return w;
22 }
```

- Binäre externe Funktionen benötigen zwei explizite Argumente für die jeweiligen Operanden.
- Hier wird eine Kopie mit elementweisen Quotienten der `vals` Werte der Operanden zurückgegeben.
- Der benötigte Zugriff auf **private** Variablen der Operanden (hier `vals`) erfordert die Deklaration als **friend**.
- Die binäre Division muss für Objekte vom Typ `T` definiert sein.

Binäre Operationen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y=4.2,z(y);
4
5    (y*z).print();
6    x=y/z;
7    x.print();
8    return 0;
9  }
```

resultiert in der Ausgabe

```
[ 17.64 17.64 ]
[ 1 1 ]
```

Überladung von Operatoren

In-/Dekrementierung als Präfixoperatoren


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    /// prefix increment as member function
7    vector& operator++() {
8      for (size_t i=0;i<N;i++) vals[i]++;
9      return *this;
10   }
11
12   template<int M, typename S>
13   friend vector<M,S>& operator--(vector<M,S>&);
14 };
15
16 /// prefix decrement as non-member function
17 template<int N, typename T>
18 vector<N,T>& operator--(vector<N,T>& u) {
19   for (size_t i=0;i<N;i++) --u.vals[i];
20   return u;
21 }
```

- vals Werte des "Heimat-" bzw. Zielobjekts werden vor Verwendung im aktuellen Ausdruck in- bzw. dekrementiert.
- Erhalte die Semantik!
- Eine Referenz auf das "Heimat-" bzw. Zielobjekt wird zurückgegeben.
- Modifikation des Objektzustands verbietet Verwendung von **const**.

In-/Dekrementierung als Präfixoperatoren

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> y=4.2,z(y);
4
5    (--y).print(); (++z).print();
6
7    return 0;
8  }
```

resultiert in der Ausgabe

```
[ 3.2 3.2 ]
[ 5.2 5.2 ]
```

Überladung von Operatoren

In-/Dekrementierung als Postfixoperatoren


```
1  template<int N, typename T>
2  class vector { T vals[N]; public: // ...
3
4    /// postfix decrement as member function
5    vector operator--(int) {
6      vector w(*this);
7      for (size_t i=0;i<N;i++) vals[i]--;
8      return w;
9    }
10
11   /// non-member friends
12   template<int M, typename S>
13   friend vector<M,S> operator++(vector<M,S>&,int);
14 };
15
16 /// postfix increment as non-member function
17 template<int N, typename T>
18 vector<N,T> operator++(vector<N,T>& u,int) {
19   vector<N,T> w(u);
20   for (size_t i=0;i<N;i++) u.vals[i]++;
21   return w;
22 }
```

- vals Werte des "Heimat-" bzw. Zielobjekts werden nach Verwendung im aktuellen Ausdruck de- bzw. inkrementiert.
- *Dummy* (Syntaxkonvention) **int** Argument markiert postfix-Charakter.
- Erhalte die Semantik!

In-/Dekrementierung als Postfixoperatoren

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> y=4.2,z(y);
4
5    (y++).print(); (z--).print();
6    y.print(); z.print();
7
8    return 0;
9  }
```

resultiert in der Ausgabe

```
[ 4.2 4.2 ]
[ 4.2 4.2 ]
[ 5.2 5.2 ]
[ 3.2 3.2 ]
```

AUFWACHEN!

Welche Probleme (falls überhaupt) treten bei der Übersetzung folgender C++ Quelltexte auf?

```
1 class A { A operator++(); };
2 int main() { A a; a++; return 0; }
```

```
1 class A { A operator++(); };
2 int main() { A a; ++a; return 0; }
```

```
1 struct A { A operator++(); };
2 int main() { A a; ++a; return 0; }
```

```
1 struct A { A* operator++(){ return this;} };
2 int main() { A a; ++a; return 0; }
```

Wach?

```
1 class A { A operator++(); };
2 int main() { A a; a++; return 0; }
```

```
:2:20: error: no ‘operator++(int)’ declared for postfix ‘++’ [-fpermissive]
    2 | int main() { A a; a++;  return 0; }
      |                   ~~~
```

```
1 class A { A operator++(); };
2 int main() { A a; ++a; return 0; }
```

```
:2:21: error: ‘A A::operator++()’ is private within this context
    2 | int main() { A a; ++a;  return 0; }
      |                     ^
```

```
1 struct A { A operator++(); };
2 int main() { A a; ++a; return 0; }
```

```
:(.text+0x23): undefined reference to ‘A::operator++()’
```

```
1 struct A { A* operator++(){ return this;} };
2 int main() { A a; ++a; return 0; }
```

Keine Fehler.

Überladung von Operatoren

Relationale Operatoren


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6  };
7
8  /// relational operator as non-member function
9  template<int N, typename T>
10 bool operator>=(const vector<N,T>& u,
11                 const vector<N,T>& v) {
12   if (u.l1_norm()>=v.l1_norm()) return true;
13   return false;
14 }
```

- Zugriff auf **public** Objektfunktion l1_norm verlangt nicht nach Deklaration als **friend**.
- Implementierung als Objektfunktion ist analog möglich.

Relationale Operatoren

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y=4.2;
4
5    if (y>=x) y.print(); else x.print();
6
7    return 0;
8  }
```

resultiert in der Ausgabe

```
[ 4.2 4.2 ]
```

Überladung von Operatoren

Arithmetische Zuweisungen als Objektfunktionen


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    /// incremental assignment as member function
7    vector& operator+=(const vector& v) {
8      for (size_t i=0;i<N;i++) vals[i]+=v.vals[i];
9      return *this;
10   }
11 };
```

- Arithmetische Zuweisung (z.B.) inkremetiert vals-Werte des Heimatobjekts und gibt Referenz auf dieses zurück.
- Implementierung als externe Funktion ist analog möglich und verlangt nach Deklaration als **friend** aufgrund Zugriffs auf **private** Variable vals.

Zu (z.B.) dekremetierender erster Operand ist linke Seite der Zuweisung.

```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    // non-member friends
7    template<int M, typename S>
8    friend vector<M,S>& operator-=(vector<M,S>&, const vector<M,S>&);
9  };
10
11 /// arithmetic assignment as non-member function
12 template<int N, typename T>
13 vector<N,T>& operator-=(vector<N,T>& u, const vector<N,T>& v) {
14   for (size_t i=0;i<N;i++) u.vals[i]-=v.vals[i];
15   return u;
16 }
```

Arithmetische Zuweisungen

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y=4.2,z(y);
4
5    x+=y; y-=z;
6    x.print(); y.print();
7
8    return 0;
9  }
```

resultiert in der Ausgabe

```
[ 4.2 4.2 ]
[ 0 0 ]
```

Überladung von Operatoren

Ausgabeströme


```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    /// non-member friends
7    template<int M, typename S>
8    friend std::ostream& operator<<(std::ostream&, const
         vector<M,S>&);
9  };
10
11 /// output stream only as non-member function
12 template<int N, typename T>
13 std::ostream& operator<<(std::ostream& s, const vector
     <N,T>& v) {
14   s << "[ ";
15   for (size_t i=0;i<N;i++) s << v.vals[i] << " ";
16   s << "]";
17   return s;
18 }
```

- Die Überladung ist nur durch eine externe Funktione möglich.
- Der **operator** << gehört zur Klasse std::ostream. Nur dort könnte er mit als Objektfunktion überladen werden.

Überladung mit externer Funktion erfolgt analog zum Ausgabestrom.

```
1  template<int N, typename T>
2  class vector {
3    T vals[N];
4  public: // ...
5
6    /// non-member friends
7    template<int M, typename S>
8    friend std::istream& operator>>(std::istream&, vector<M,S>&);
9  };
10
11 /// input stream only as non-member function
12 template<int N, typename T>
13 std::istream& operator>>(std::istream& s, vector<N,T>& u) {
14   for (size_t i=0;i<N;i++) s >> u.vals[i];
15   return s;
16 }
```

Ein- und Ausgabeströme

Beispiel


```
1  int main() {
2    const int n=2;
3    vector<n,float> x,y;
4
5    std::cin >> x >> y;
6    std::cout << x << y << std::endl;
7
8    return 0;
9  }
```

resultiert bei Eingabe von

```
1 2 3 4
```

in der Ausgabe

```
[ 1 2 ][ 3 4 ]
```

```
1 // read from file
2 std::ifstream in("v2.in");
3 in >> x;
4 in.close();
5
6 // write to file
7 std::ofstream out("v2.out");
8 out << x;
9 out.close();
```

- Datei-In- und Ausgabeströme sind jeweils Spezialisierungen von std::istream und std::ostream.
- Daher funktioniert die Ein- und Ausgabe mit Dateien ohne weitere Modifikationen der Klasse vector und ohne zusätzliche externe Funktionen.

Zusammenfassung

Überladung von ...

Funktionen

- Intrinsische Funktionen
- Nutzerdefinierte Funktionen

Operatoren

- Zuweisung
- Feldelementzugriff
- Unäre Operationen
- Binäre Operationen
- In-/Dekrementierung als Prä-/Postfixoperatoren
- Relationale Operationen
- Arithmetische Zuweisungen
- Ein- und Ausgabeströme