Praxisorientierte Einführung in C++
Lektion: "Operatoren"

Christof Elbrechter

Neuroinformatics Group, CITEC

April 24, 2014

Table of Contents

Motivation

- Funktionsbasierte Programmierung in den meisten Fällen angebracht
- In einigen Sonderfällen allerdings sehr Umständlich
- Insbesondere für mathematische Datentypen wie Matritzen und Vektoren
- C++ bietet eine einfache Möglichkeit, um fast alle Operatoren für einen Datentyp frei zu implementieren
- Operatoren sind in C++ einfache Funktionen, welche mit dem Schlüsselwort `operator` gekennzeichnet werden
- Es können allerdings nur die schon vorhandenen Operatoren verwendet werden

Beispiel für den Einsatz von Operatoren

function_based_mathematics.cpp

```
struct Vec3D{
  Vec3D(float x=0,float y=0, float z=0):x(x),y(y),z(z){}
  float x,y,z;
};
Vec3D add(const Vec3D &a, const Vec3D &b){
  return Vec3D(a.x+b.x,a.y+b.y,a.z+b.z);
}
Vec3D sub(const Vec3D &a, const Vec3D &b){
  return Vec3D(a.x-b.x,a.y-b.y,a.z-b.z);
}
Vec3D mul(const Vec3D &a, float k){
  return Vec3D(a.x*k,a.y*k,a.z*k);
}
int main(){
  Vec3D a(1,2,3);
  Vec3D b(0,0,7);
  a = sub(add(a,b),add(a,mul(b,2))); // ???
}
```

Beispiel für den Einsatz von Operatoren

operator_based_mathematics.cpp

```
struct Vec3D{
  Vec3D(float x=0,float y=0, float z=0):x(x),y(y),z(z){}
  float x,y,z;
};
Vec3D operator+(const Vec3D &a, const Vec3D &b){
  return Vec3D(a.x+b.x,a.y+b.y,a.z+b.z);
}
Vec3D operator-(const Vec3D &a, const Vec3D &b){
  return Vec3D(a.x-b.x,a.y-b.y,a.z-b.z);
}
Vec3D operator*(float k, const Vec3D &a){
  return Vec3D(a.x*k,a.y*k,a.z*k);
}

int main(){
  Vec3D a(1,2,3);
  Vec3D b(0,0,7);
  a = (a+b)-(a+2*b);
  // nun sieht man auch, dass es eigentlich -b ist
}
```

Vorteile von Operatoren

- Code wird kürzer
  - besser zu lesen
  - besser zu schreiben
  - leichter zu warten
- Code wird **selbsterklärender**
- Dadurch kann deutlich leichter von Hand optimiert werden (letztes Beispiel)
- Sehr intuitive Verwendung insbes. für mathematische Datentypen

Operatoren

Durch das Anbieten von Operatoren wird erreicht, dass sich eigene Datentypen wie built-in Datentypen *anfühlen*

Welche Operatoren gibt es in C++?

- Grobe Unterteilung in **unär** und **binär**

Beispiele für **unäre** Operatoren (ein Operand)

- `++, --` (post- und pre-Inkrement-Operatoren)
- `!, ~` Negations-Operator (logisch und binär)
- `-, ^` unäres Minus und bitweise XOR

Beispiele für **binäre** Operatoren (zwei Operanden)

- `=` Zuweisungs-Operator
- `==, !=, >=, ...` Vergleichs-Operatoren
- `[]` Array-Index-Operator
- `+, -, *, /, ||, &&, +=, ...` logische und arithmetische Op.
- `<<` und `>>` Stream-Opeatoren

Welche Operatoren gibt es in C++?

Weitere Beispiele für sonstige Operatoren

`()` Funktions-Operator (n-när) `,`,`?:`,`::`,`.`,`->`,`throw` , `sizeof`, `new`, `delete`
`(type)` `cast-Operator` `*`,`&` (De)Referenz-Operator

Wichtig

- Die Operandenzahl der Operatoren ist in C++ festgelegt und kann nicht verändert werden (Ausnahme Funktions-Operator)
- Einige Operatoren können Methoden oder globale Funktionen sein
- Einige Operatoren müssen als Methoden implementiert werden
- Einige Operatoren müssen als globale Funktionen implementiert werden

Syntax

Syntax Operator-Definition **operator**

```
RueckgabeTyp operator OP(OperandenListe){
   // Implementation
}
```

Beispiel vector_test.cpp

```
struct Vector3D{
   float data[3];
   float &operator[](int idx){ return data[idx]; }
   const float &operator[](int idx) const { return data[idx]; }
};
float sprod(const Vector3D &a, const Vector3D &b){
   int sum = 0;
   for(int i=0;i<3;++i) sum += a[i]*b[i];
   return sum;
}
```

Zuweisungs-Operator

- **I.d.R. gilt:**
  Musste ein Copy-Konstruktor implementiert werden, so ist auch die Implementation des Zuweisungs-Operators notwendig

Empfohlene Signatur

```
struct MyClass{
  MyClass &operator=(const MyClass &other);
};
```

- Rückgabe sollte immer eine **Referenz des lvalues** sein
- Dies ermöglicht u.A. gestaffelte Zuweisungen wie a=b=c;
- Abweichung: Möglich, führt aber oft zu Problemen

Überladen des Zuweisungs-Operators

```
class Vector{
   int dim;
   float *data;
 public:
   Vector(int dim) : dim(dim),data(new float[dim]){}

   ~Vector() { delete [] data; }

   Vector &operator=(const Vector &other){
      delete[] data;
      data = new float[other.dim];
      dim = other.dim;
      for(int i=0;i<dim;++i) data[i] = other.data[i];
      return *this;
   }

   Vector &operator=(float value){
      for(int i=0;i<dim;++i) data[i] = value;
      return *this;
   }
};
```

Operatoren und Konventionen

- Es existieren Konventionen, über Signatur und Semantik von Operatoren
- Diese sollten eingehalten werden
- Grundlage: Was passiert bei den Standard-Datentypen?

Wichtig!

Abweichungen und Erweiterungen immer gut dokumentieren!

Beispiel: Eine Klasse für Komplexe Zahlen

- Gibt's auch schon (`std::complex`) – aber egal!

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0):
   re(re),im(im){}
};
```

- Real- und Imaginärteil können `public` sein (da keine Nebeneffekte zu erwarten sind)
- Konstruktor mit Standardargumenten bietet viele Möglichkeiten:
  - Default (`0+0i`)
  - `Complex(5)` erzeugt reelle Zahl (mit Imaginärteil = 0)

Complex: Copy-Konstruktor und Zuweisungsoperator

- Sind nicht notwending, da Default-Verhalten genau richtig ist
- Hier nur der Vollständigkeit halber

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0):
      re(re),im(im){}
   inline Complex(const Complex &c):
      re(c.re),im(c.im){
      // Alternativ: *this = c;
   }
   Complex &operator=(const Complex &c){
     re = c.re;
     im = c.im;
     return *this;
   }
};
```

- Kein dynamischer Speicher $\Rightarrow$ kein Destruktor notwendig

Complex: Operatoren '+' und '-'

- Die Operatoren '+' und '-' können in der Klasse (als Methoden) oder außerhalb (als globale Operatoren) implementiert werden

Operator-Methoden

Bei Operator-Methoden ist der linke Operand immer automatisch eine Instanze der Klasse (in der Implementation ist dieser `this`)

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0) { ... }
   Complex operator+(const Complex &c){
      return Complex(re+c.re,im+c.im);  // linker Operand ist this
   }
   Complex operator-(const Complex &c){
      return Complex(re-c.re,im-c.im);  // linker Operand ist this
   }
};
```

Complex: Operatoren '*' und '/'

- Der Vollständigkeit halber noch zwei weitere Operatoren

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0) { ... }
   Complex operator+(const Complex &c){ ... }
   Complex operator-(const Complex &c){ ... }
   inline Complex operator*(const Complex &other) const{
     float newRe = re*other.re - im*other.im;
     float newIm = re*other.im + im*other.re;
     return Complex(newRe,newIm);
   }
   inline Complex operator/(const Complex &other) const{
     const float &a=re, &b=im, &c=other.re, &d=other.im;
     float denom = c*c+d*d;
     return Complex((a*c+b*d)/denom, (b*c-a*d)/denom);
   }
};
```

Complex: Operatoren '+=', '-=', '*=' und '/='

- Diese Operatoren sind nicht const und geben ein Referenz auf den **lvalue** zurück

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0) { ... }
   Complex operator+(const Complex &c){ ... }
   Complex operator-(const Complex &c){ ... }
   inline Complex operator*(const Complex &other) const{ ... }
   inline Complex operator/(const Complex &other) const{ ... }
   inline Complex &operator+=(const Complex &other){
      return (*this = *this + other); // nicht 100%ig effizient!
   }
   // -=, *= und /= analog!
};
```

Inkrement- und Dekrement-Operatoren ++ und –

- Pre- und Post-(In/De)krement Operator werden durch ein nicht verwendetes `int`-Argument voneinander unterschieden

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0) { ... }

   inline Complex &operator++(){ // pre-inkrement -> ++c
      ++re;
      return *this;
   }
   inline Complex operator++(int){ // post-inkrement -> c++
      Complex tmp = *this;
      ++re;
      return tmp;
   }
   // --c und c-- analog!
};
```

Überladen von Operatoren

- Evtl. auch noch sinnvoll:
  - `Complex + float` ⇒ addiert `float` zum Realteil
  - `Complex * float` ⇒ multipliziert Real- und Imaginärteil mit `float`

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0) { ... }

   inline Complex operator+(float f) const{
      return Complex(re+f,im);
   }

   inline Complex &operator*=(float f){
     re *= f;
     im *= f;
     return *this;
   }
   // und Kombinationen wie /=(float), -=float, ... !
};
```

Unäres '+' und unäres '-'

- **Achtung:** Auf die richtige Signatur achten

complex.h

```
struct Complex{
   float re,im;
   inline Complex(float re=0,float im=0){...}
   inline Complex &operator+() {
     return *this;
   }
   inline const Complex &operator+() const{
     return *this;
   }
   inline Complex &operator-() { //Achtung Falsch !!
     re *= -1;
     im *= -1;
     return *this;
   }
   inline Complex operator-() const {   // Richtig!!
     return this->operator*(-1.0f);
   }
};
```

Vergleichsoperatoren '==' und '!='

- Vergleichsoperatoren sind `const` und können ebenfalls überladen werden

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0) { ... }

   inline bool operator==(const Complex &other) const{
      return other.re == re && other.im == im;
   }
   inline bool operator!=(const Complex &other) const{
      return ! (*this == other);
   }
   bool operator==(float f) const{
     return re==f && im == 0;
   }
};
```

Impliziter Cast-Operator nach bool

- Implizite Cast-Operatoren können für beliebige Typen Implementiert werden
- Damit: Impliziter Cast und Zuweisung zu diesem Typ möglich
- Sehr gebräuchlich: `operator bool(){ ... }`
- Syntax etwas seltsam (z.B., kein Rückgabetyp)
- Umwandlung ist i.d.R. `const`

Syntax: Impliziter Cast-Operator

```
operator ZielTyp() const { Implementation }
```

complex.h

```
struct Complex{
   // ...
   operator bool() const{
     return !(re == 0.0f && im == 0.0f); // einfacher: return re || im
   }
};
```

Anwendung und Probleme Impliziten Cast-Operators

Anwendung

```
#include <complex.h>

int main(){
   Complex c     = ...;
   if(c){ // operator bool()
      c = 4;
   }
}
```

- **Aber Achtung:** Es können schnell nicht direkt ersichtliche Mehrdeutigkeiten Entstehen

Beispiel für Mehrdeutigkeiten

```
int main(){
   Complex c = ...;
   if(c==4){ // mehrdeutig
      ...
   }
}
```

Möglichkeiten

- `((int)(bool)c) == 4`
- `c == (Complex)4`
- `c == (float)4`

ostream-Operator (falsch)

- Naiver Ansatz: Als Methode

complex.h

```
struct Complex{
   float re;
   float im;
   inline Complex(float re=0,float im=0) { ... }

   std::ostream &operator<<(std::ostream &s){ // Achtung: Falsch!
      return s << re << '+' << im << 'i';
   }
};
```

- Das geht zwar so, ist aber nich das, was wir wollen

Anwendung wäre dann so

```
int main(){
  Complex b(3,4);

  b << std::cout << std::endl; // Ooops!
}
```

ostream-Operator (richtig)

- Der ostream-Operator << kann nur richtig als globale Funktion implementiert werden
- Erinnerung: Bei Operator-Methoden ist der lvalue immer eine Instanz der Klasse
- Hier soll das aber der `std::ostream` sein

complex.h

```
struct Complex{
   // ...
};

std::ostream &operator<<(std::ostream &s,const Complex &c){
   return s << c.re << '+' << c.im << 'i';
}

int main(){ // nun: Anwendung korrekt
  Complex b(3,4);
  std::cout << b << std::endl;
}
```

Der ostream-operator

- Der ostream-operator unterstreicht eines der Grundkonzepte von C++
- Streams stellen ein äußerst nützliches Konzept dar
- Falls möglich: Stream-basierte Ein- und Ausgabe verwenden
- STL bietet eine Vielzahl von Stream-Implementationen, die alle zu diesem einen Operator kompatibel sind
- Z.B.:
  - File-Streams
  - String-Streams

Syntax: ostream-Operator

```
std::ostream &operator<<(std::ostream &, const ClassName &);
```

Operatoren als globale Funktionen

- Viele Operatoren können auch außerhalb der Klasse definiert werden
- Es gibt Ausnahmen (z.B. Zuweisungs-Operator und Cast-Operatoren)
- **Vorteil:** Typ kann auch linker Operand sein (siehe Operator <<)
- **Nachteil:** Da Implementation außerhalb der Klasse $\Rightarrow$ kein direkter Zugriff auf private Variablen[1]
- **Ausweg:** Operator als `friend` deklarieren ($\Rightarrow$ kommt bald)

[1] es sei denn, es gibt für alles `inline` getter methoden

Der istream-operator

- Gegenstück zu ostream-Operator
- Er ist dazu da, um Daten zu de-serialisieren / zu parsen
- Datentypen, für die der istream-Operator implementiert wurde, können problemlos aus einem Stream gelesen werden

Wichtig!

- Die Implementation des istream-Operators sollte möglichts analog – also invers – zu der des ostream-Operators erfolgen

Syntax: istream-Operator

```
std::istream &operator>>(std::istream &, ClassName &);
```

- **Beachte:** Hier wird *in den rvalue hinein* gelesen, also ist die rvalue Referenz nicht `const`

Der istream-Operator für den Complex-Typen

- Complex serialisiert: Form `A+Bi` (`A` und `B` `float`-Zahlen)
- Der istream-Operator ist i.d.R. etwas komplizierter, da er mit Lesefehlern umgehen können muss

```
#include <iostream>
#include <sstream>
struct Complex { ... };
std::istream &operator>>(std::istream &s, Complex &c){
  char ch[2]={0};
  s >> c.re >> ch[0];  // lese A+
  if(ch[0] != '+')
    throw std::runtime_error("expected '+' not '"+std::string(ch)+"'");
  s >> c.im >> ch[0];  // lese Bi
  if(ch[0] != 'i')
    throw std::runtime_error("expected 'i' not '"+std::string(ch)+"'");
  return s;
}
int main(){
  Complex a(2,1);
  std::ostringstream ostr;  ostr << a << a << a;
  Complex x,y,z;
  std::istringstream istr(ostr.str());  istr >> x  >> y >> z;
}
```

Funktoren

- Instanzen von Klassen, die den Funktions-Operator implementieren, nennt man **Funktoren**
- Klammer-Operator ist besonders, da er beliebig viele Argumente haben darf (N-när)

Syntax: Funktions-Operator

```
RueckgabeTyp operator()(ArgumentListe);
```

- Der Vorteil von Funktoren gegenüber Funktionen ist die Tatsache, dass Funktoren **State** besitzen können
- Insb. in Kombination mit STL-Algorithmen (Header `<algorithm>` ⇒ später) sehr mächtiges Konzept

Beispiel für Funktoren

- Betrachten wir folgende Funktion (mit besserer Implementation auch in `algorithm` zu finden)

```
template<class T, class UnaryFunc>
void for_each(T *begin, T *end, UnaryFunc f){
   while(begin!=end) f(*begin++);
}
```

- Wie alle Templates funktioniert auch `for_each` mit *Duck-Typing*
- Also auch gültig für Typen `UnaryFunc`, die einen entsprechenden Funktions-Operator anbieten
- *State* des Funktors kann verwendet werden, um Closure-Verhalten zu Implementieren

Beispiel: Aufsummieren aller Elemente mittels for_each

```
#include <iostream>

template<class T, class UnaryFunc>
void for_each(T *begin, T *end, UnaryFunc f){
   while(begin!=end) f(*begin++);
}

struct Count{
  int &n;
  Count(int &n):n(n){}
  void operator()(int &i) { n += i; }
};

int main(){
  int *pi = new int[100];
  for(int i=0;i<100;++i) pi[i] = 2;
  int n=0;
  for_each(pi,pi+100,Count(n));
  std::cout << n << std::endl;
}
```

- Vorteil: Optimierung in `for_each` nur einmal implementieren
- STL-`for_each` ist deutlich schneller als eine `for`-Schleife

Anmerkungen zum Funktions-Operator

- Kann verwendet werden, um Funktionsverhalten in Klassen zu Implementiern
- Im Gegensatz zum Index-Operator ([]) beliebige Argumentzahl (auch 0)
- Kann selbstverständlich auch überladen werden
- Weitere prominiente Einsatzgebiete:
  - Elementzugriff für Matrix-Klassen
  - Pixelzugriff für Bild-Klassen

Anmerkung zum C++-11 Standard

- Im neuen Standard gibt es sog. Lambda-Expressions
- Diese bieten die Möglichkeit, anonyme Funktionen zu schreiben
- Mittels sog. *Closures* kann *State* aus dem Caller-Scope auch innerhalb der Lokalen Funktion verwendet werden

Beispiel: Lambda Expressions (C++-11)

```
#include <algorithm>
#include <iostream>
int main(){
  int is[5] = { 1, 2 ,3, 4, 5};
  std::for_each(is,is+5,[](int &x){x*=2;});
  std::for_each(is,is+5,[](int x){std::cout << x << " ";});

  int c = 0;
  std::for_each(is,is+5,[&c](int &x){c+=x; });
  std::cout << "sum is " << c << std::endl;
}
```

Pointer- und Dereferenz-Operator

- In seltenen Fällen sollen Klassen ein Pointer-Verhalten aufweisen
- Pointer-Operator -> wird speziell behandelt
  - Ruft sich automatisch *gestaffelt* auf ($\Rightarrow$ später)
- Bitte nur verwenden, wenn analog zu Standardsemantik
- Prominentes Beispiel: Sog. *SmartPointer*
- Hierzu werden wir noch ein Beispiel implementieren
- Dafür brauchen wir allerdings Klassen Templates
- $\Rightarrow$ siehe Lektion *Smart-Pointer*