

Programmierkurs

Steffen Müthing

Interdisciplinary Center for Scientific Computing, Heidelberg University

January 25, 2019

Inhalt

Operator Overloading

- Motivation

- Syntax

- Unterstützung für I/O

Funktoren

- Beispiel

Lambda-Funktionen

- Beispiele aus der STL

Motivation (I)

```
class Vector {  
public:  
    Vector();  
    Vector(double x, double y);  
    Vector(const Vector& v);  
    void add(const Vector& v);  
    void subtract(const Vector& v);  
    void scale(double s);  
}
```

Gegeben $\mathbf{u}, \mathbf{v} \in \mathbb{R}^2, a \in \mathbb{R}$: Berechne $\mathbf{u} = \mathbf{u} + a \cdot \mathbf{v}$

```
Vector av(v);  
av.scale(a);  
u.add(av);
```

Lesbarkeit??

Motivation (II)

```
class Vector {
public:
    double x() const;
    double y() const;
    ...
    void print(std::ostream& os) const {
        os << "(" << x() << ", " << y() << ")";
    }
};

std::cout << "Result: ";
u.print(std::cout);
std::cout << std::endl;
```

Lösung: Operator Overloading

In C++ können fast alle Operatoren überladen werden, wenn **mindestens** ein Operand eine Klasse oder eine Enumeration ist.

- ▶ Überladene Operatoren gegeben durch spezielle Funktionen mit festem Namensschema: Fiktiver Operator ? durch Funktion `operator?()`
- ▶ Verschiedene Typen von Operatoren:
 - ▶ Unäre Operatoren (ein Operand), z.B. `-x`, `!x`
 - ▶ Binäre Operatoren (zwei Operanden), z.B. `a + b`, `a << b`
 - ▶ Spezielle Operatoren, z.B. `a[b]`, `a(b,c)`
- ▶ Operatorfunktionen entweder als Memberfunktionen von Klassen oder als freistehende Funktionen
- ▶ Bei Operatoraufruf keine Namespace-Angabe möglich \Rightarrow freistehende Funktionen immer in Namespace des eigenen Objekts, damit der Compiler sie findet (ADL: argument-dependent lookup)

Unäre Operatoren

▶ Member-Funktion:

```
class Vector {  
public:  
    Vector operator-() const {  
        return Vector(-x(),-y());  
    }  
};
```

▶ Freistehende Funktion:

```
Vector operator-(const Vector& v) {  
    return Vector(-v.x(),-v.y())  
}
```

Binäre Operatoren

▶ Member-Funktion:

```
class Vector {  
public:  
    Vector operator+(const Vector& b) const {  
        return Vector(x() + b.x(),y() + b.y());  
    }  
};
```

▶ Freistehende Funktion:

```
Vector operator+(const Vector& a, const Vector& b) {  
    return Vector(a.x() + b.x(),a.y() + b.y());  
}
```

Binäre Operatoren mit unterschiedlichen Typen

- ▶ Reihenfolge der Argumente ist wichtig!
- ▶ Member-Funktion:

```
class Vector {  
    public:  
    Vector operator*(double s) const {  
        return Vector(s * x(), s * y());  
    }  
};
```

- ▶ Objekt ist immer der linke Operand!
- ▶ Funktioniert nur für $v * s$, nicht für $s * v$

Binäre Operatoren mit unterschiedlichen Typen

- ▶ Reihenfolge der Argumente ist wichtig!
- ▶ Member-Funktion:

```
class Vector {  
    public:  
    Vector operator*(double s) const {  
        return Vector(s * x(), s * y());  
    }  
};
```

- ▶ Objekt ist immer der linke Operand!
- ▶ Funktioniert nur für $v * s$, nicht für $s * v$
- ▶ Zwei freistehende Funktionen:

```
Vector operator*(const Vector& v, double s) {  
    return Vector(s * v.x(), s * v.y());  
}  
  
Vector operator*(double s, const Vector& v) {  
    return v * s; // forward to other implementation  
}
```

Freistehende Funktionen als **friends**

Freistehende Operator-Funktionen oft erforderlich, aber

- ▶ haben keinen Zugriff auf private Variablen / Methoden
- ▶ nicht innerhalb der Klassendeklaration, schwer zu finden

Freistehende Funktionen als **friends**

Freistehende Operator-Funktionen oft erforderlich, aber

- ▶ haben keinen Zugriff auf private Variablen / Methoden
- ▶ nicht innerhalb der Klassendeklaration, schwer zu finden

Freistehende Funktionen können mit einer Klasse **befreundet** sein

- ▶ Deklaration (und möglicherweise Definition) innerhalb der Klasse
- ▶ Kennzeichnung durch Voranstellen von **friend**
- ▶ Voller Zugriff auf alle privaten Variablen und Methoden
- ▶ **Keine** Member-Funktion!

```
class Vector {  
public:  
...  
    friend Vector operator*(double s, const Vector& v) {  
        return Vector(s * v.x(), s * v.y());  
    }  
};
```

Klassen mit Unterstützung für Ein- / Ausgabe

Um Eingabe und Ausgabe mit C++-Streams zu unterstützen, muss eine Klasse die passenden Operatoren überladen:

```
class Vector {
public:
...
    friend std::ostream& operator<<(
        std::ostream& os, const Vector& v) {
        os << "(" << x() << ", " << y() << " ";
        return os;
    }

    friend std::istream& operator>>(
        std::istream& is, Vector& v) {
        ...
        return is;
    }
};
```

- ▶ Syntax exakt wie hier
- ▶ Nicht vergessen, den Stream zurückzugeben

Funktoren: Funktionen mit Gedächtnis

Manchmal braucht man eine Funktion, die sich Informationen zwischen den Aufrufen merken kann:

- ▶ Eine Funktion, die zu einem Argument immer eine feste, aber zur Laufzeit bestimmte Zahl hinzuaddiert
- ▶ Eine Funktion, die weiss, wie oft sie schon aufgerufen wurde

Funktoren: Funktionen mit Gedächtnis

Manchmal braucht man eine Funktion, die sich Informationen zwischen den Aufrufen merken kann:

- ▶ Eine Funktion, die zu einem Argument immer eine feste, aber zur Laufzeit bestimmte Zahl hinzuaddiert
- ▶ Eine Funktion, die weiss, wie oft sie schon aufgerufen wurde

Lösung: Funktoren: Objekte, die man wie eine Funktion aufrufen kann.

```
class Funktor {  
public:  
  
    T operator()(A a, B b, C c) const {  
        ...  
    }  
};
```

Funktoren: Beispiel

```
template<typename T>
class add {
    T _number;
    int _calls = 0;
public:

    add(T number)
        : _number(number)
    {}

    template<typename U>
    auto operator()(const U& u) const {
        ++_calls;
        return u + _number;
    }
};
```

Funktoren in der Standardbibliothek

Viele Algorithmen in der STL akzeptieren Funktoren:

- ▶ `std::sort`
- ▶ `std::find_if`
- ▶ `std::copy_if`
- ▶ `std::transform`
- ▶ `std::generate`
- ▶ ...

Algorithmen können so angepasst werden:

- ▶ Sortiere Berge absteigend nach Höhe
- ▶ Kopiere alle Berge höher als 8.000m
- ▶ Extrahiere Liste von Erstbesteigern aus Liste von Bergen
- ▶ ...

STL-Funktoren: Beispiel

Im folgenden operieren wir mit folgender Klasse:

```
struct Mountain {  
    std::string name;  
    int height;  
    int first_ascent;  
    std::vector<std::string> first_ascenders;  
};
```

- ▶ Die Klasse ist mit als **struct** definiert, alle Member sind also **public**.
- ▶ Aus Lesbarkeitsgründen greifen wir im folgenden direkt auf die Member-Variablen zu.

Ausserdem haben wir eine Liste von Bergen:

```
std::vector<Mountain> mountains = ...;
```

Berge sortieren

Wir können `mountains` nicht mit `std::sort()` sortieren, weil der Compiler nicht weiss, welcher Berg zuerst kommen soll.

- ▶ Wir können einen Vergleichsoperator für die Relation `<` definieren. Dieser wird von `std::sort()` verwendet:

```
bool operator<(const Mountain& m1, const Mountain& m2) {  
    return m1.name < m2.name; // Sort mountains alphabetically  
}
```

- ▶ Falls wir die Berge anders sortieren wollen, können wir `std::sort()` dies explizit sagen:

```
std::sort(  
    mountains.begin(),  
    mountains.end(),  
    sort_mountains_by_descending_height()  
);
```

Die Definition von `sort_mountains_by_descending_height` folgt auf der nächsten Folie.

Berge sortieren: Funktor

Bevor wir die Berge nach Höhe sortieren können, müssen wir ausserhalb der Funktion, in der wir sortieren wollen, den passenden Funktor definieren:

```
struct sort_mountains_by_descending_height {  
  
    bool operator()(  
        const Mountain& m1,  
        const Mountain& m2  
    ) const  
    {  
        // the functor returns whether the first  
        // argument is smaller than the second  
        return m1.height > m2.height;  
    }  
  
};
```

Lambda-Funktionen: Motivation

Oft wird ein Funktor nur einmal beim Anruf eines Algorithmus benötigt

- ▶ Definition als Klasse weit weg von Verwendung
- ▶ Viel Boilerplate

Lambda-Funktionen erlauben `inline`-Definition von Funktoren als Variablen

```
auto sort_height = [](auto& m1, auto& m2) {  
    return m1.height > m2.height;  
};
```

Lambda-Funktionen: Syntax

```
[capture-list](arg-list) -> return-type {  
    body  
};
```

Die Definition einer Lambda-Funktion besteht immer aus

- ▶ einer **capture list** in [], die steuert, welche Variablen aus dem aktuellen Scope im body der Lambda-Funktion verfügbar sind.
- ▶ einer Liste von **Funktions-Argumenten** in ().
- ▶ dem eigentlichen **Funktionscode** in {}.

Meistens kann der Compiler den Rückgabebetyp der Lambda-Funktion erraten, ansonsten kann er optional mit `-> returntype` angegeben werden.

Lambda-Funktionen: Implementierung

Intern sind Lambda-Funktionen bis auf Details nur eine Kurzschreibweise für die Definition eines Funktors und die Erzeugung einer Variable vom Typ des Funktors:

```
auto squared = [](double i) {  
    return i * i;  
};
```

ist äquivalent zu

```
struct unknowable_type {  
    auto operator()(double i) const {  
        return i * i;  
    }  
};  
...  
{  
    auto squared = unknowable_type();  
}
```

Lambda-Funktionen: Capture-Spezifikation (I)

- ▶ Standardmässig kann man in einer Lambda-Funktion nicht auf die Variablen des umgebenden Scopes zugreifen.
- ▶ Um diese Variablen verfügbar zu machen, kann man sie in der Capture-Spezifikation auflisten:
 - ▶ Variablen-Name: `var`: Die Variable im Lambda ist eine Kopie.
 - ▶ Variablen-Name mit `&`: `&var`: Die Variable im Lambda ist eine Referenz auf die Original-Variable.
 - ▶ Ein einzelnes `&`: Im Lambda verwendete Variablen werden automatisch per Referenz verfügbar gemacht.
 - ▶ Ein einzelnes `=`: Im Lambda verwendete Variablen werden automatisch als Kopie verfügbar gemacht.

```
double a = 2.0, y = 3.0;
auto axpy = [=,&y](double x) {
    // default capture: by copy, b captured by reference
    return a * x + y;
}
a = 4.0; // does not change the lambda
y = 3.0; // changes the lambda
```

Generische Lambda-Funktionen

- ▶ Oft ist es praktisch, wenn eine Lambda-Funktion für verschiedene Typen von Argumenten funktioniert (wie eine Template-Funktion).
- ▶ Bei Verwendung von `auto` in der Parameter-Liste wird der `operator()` im Funktor eine Template und jeder `auto`-Parameter ein Template-Argument:

```
auto plus = [](auto a, auto b) {  
    return a + b;  
};
```

erzeugt den Funktor

```
struct unknown_plus_type {  
    template<typename T1, typename T2>  
    auto operator()(T1 a, T2 b) const {  
        return a + b;  
    }  
};
```


Lambda-Funktionen: Hinweise (I)

- ▶ Der genaue Typ des Funktors wird vom Compiler festgelegt und kann nicht aufgeschrieben werden.
- ▶ Lambda-Funktionen kann man daher nur in einer `auto`-Variablen speichern.
- ▶ Man kann Lambda-Funktionen auch an Template-Parameter binden:

```
template<typename Functor, typename Arg>
auto call(Functor f, Arg arg)
{
    return f(arg);
}
...
call(axpy, 3.0);
```

Lambda-Funktionen: Hinweise (II)

- ▶ Captures werden intern zu Member-Variablen des Funktors.
- ▶ Der Compiler generiert einen passenden Konstruktor und den zugehörigen Aufruf.
- ▶ Vorsicht mit der Lebenszeit von Variablen bei Capture by reference, wenn das Lambda von der Funktion zurückgegeben wird:

```
auto makeLambda(int add) {  
    return [&](int i) {  
        return i + add;  
    };  
} // boom!
```

In diesem Beispiel speichert die Lambda-Funktion eine Referenz auf die Variable `add`, die aber nach dem Verlassen von `makeLambda()` nicht mehr gültig ist!

Lambda-Funktionen: Hinweise (II)

- ▶ Captures werden intern zu Member-Variablen des Funktors.
- ▶ Der Compiler generiert einen passenden Konstruktor und den zugehörigen Aufruf.
- ▶ Vorsicht mit der Lebenszeit von Variablen bei Capture by reference, wenn das Lambda von der Funktion zurückgegeben wird:

```
auto makeLambda(int add) {  
    return [&](int i) {  
        return i + add;  
    };  
} // boom!
```

In diesem Beispiel speichert die Lambda-Funktion eine Referenz auf die Variable `add`, die aber nach dem Verlassen von `makeLambda()` nicht mehr gültig ist!

Bedingtes Kopieren

- ▶ Die Funktion `std::copy_if` kopiert die Werte aus einer Iterator-Range, für die das Predicate (ein Funktor, der einen `bool` zurückgibt) `true` ist:

```
template<typename InIt, typename OutIt, typename Pred>
OutIt copy_if(
    InIt first, InIt last,
    OutIt out_first,
    Pred predicate);
```

- ▶ Wir wollen eine Liste mit hohen Bergen:

```
std::vector<Mountain> high;
int min_height = 5000;
std::copy_if(
    mountain.begin(), mountain.end(), // source
    std::back_inserter(high), // append copied values to high
    [=](auto& mountain) {
        return mountain.height >= min_height;
    }
);
```

Zählen von Elementen

- ▶ Die Funktion `std::count_if` zählt, für wie viele Elemente das Predicate `true` ist:

```
template<typename It, typename Pred>
std::size_t count_if(It first, It last, Pred predicate);
```

- ▶ Wir wollen wissen, wie viele Berge erst nach 1900 bestiegen wurden:

```
auto late_ascents = std::count_if(
    mountain.begin(), mountain.end(), // source
    [=](auto& mountain) {
        return mountain.first_ascent >= 1900;
    }
);
```