Fachhochschule Bingen

Programmieren

friend-Methoden, Dateien, Operatoren

Prof. Dr. Maximilian Mengel,
Professur Programmiermethodik,
Grundlagen der Informatik und Multimedia
Gebäude 1, Raum 212
Tel.: 06721-409 152
E-Mail: mengel@fh-bingen.de

Bereichsauflösungsoperator ::

- Der Bereichsauflösungsoperator :: wird verwendet um...
- ... Die Implementierung einer Methode einer bestimmten Klasse zuzuordnen
  - void Boote::drucke { cout << “Boot“;}
- ... Überdeckungen von Variablen und Funktionen aufzulösen
- ... Überdeckte Methoden einer Basisklasse aufzurufen

Überdeckung von Variablen und Funktionen

```
#include <iostream>
using namespace std;

int i;      // globale Variable

main()
{
  int i, cout;

  i = 5;
  cout = 7;
  // Globales i?: i = i * cout;
  // Ausgabe ?: cout << i << i(Global) << cout;
}
```


Überdeckung von Variablen und Funktionen

```
#include <iostream>
using namespace std;

int i;        // globale Variable

main()
{
  int i, cout;

  i = 5;
  cout = 7;
  ::i = i * cout;
  std::cout << i << ::i << cout;
  //  ::cout << i << ::i << cout;
  // wenn alte .h-include-Datei ohne using namespace verwendet wird

}
```

Bereichsauflösungsoperator zum Aufruf der Basisklassen-Methode und zum Verhindern der Polymorphie

- Man nutzt **in** einer Abgeleiteten Klasse den Bereichsauflösungsoperator um virtuelle Methoden der Basisklasse aufzurufen
  - Vor den Aufruf der Methode schreibt man den Klassennamen der Basisklasse sowie den :: und dann den Methodennamen
- Auch von „Außen“ kann man direkt eine virtuelle Methode der Basisklasse aufrufen:
  - Beim Aufruf einer entsprechenden Methode schreibt man vor den Methodennamen den entsprechenden Basisklassenname gefolgt von einem :: und dann dem Methodennamen

Aufruf einer Basisklassenmethode

```
class base {
  public:
    virtual void tutEtwas(){cout << “ich tue was...\n“;}
   };

class derived: public base {
  public:
    virtual void tutEtwas()
    {
      base::tutEtwas();
      cout << “und noch etwas anderes !!\n“;
    }
};

main()
{
  derived d;
  d.tutEtwas();
  d.base::tutEtwas();
}
```


friend

- Manchmal stehen Klassen in einem inhaltlich engen Zusammenhang und eine Klasse benötigt Zugriff auf private Eigenschaften oder Methoden einer anderen Klasse ohne das die beiden Klassen voneinander abgeleitet sind. In C++ kann man einen solchen Zugriff für eine Klasse, Methode oder Funktion erlauben, indem man diese Klasse, Methode oder Funktion als friend vereinbart

Friend-Klassen, Friend-Methoden, Friend-Funktionen

- Um den Zugriff auf private Eigenschaften und Methoden zu erlauben kann man innerhalb der Klasse
  - eine Klasse als friend vereinbaren, indem man im public-Bereich **`friend class <Klassenname>;`** aufführt
  - eine Methode als friend vereinbaren, indem man die komplette Methode mit Bereichsauflösung und vorangestelltem friend nennt.
  - eine Funktion als friend vereinbaren, indem man die komplette Funktion mit vorangestelltem friend benennt

Friend Klasse

```
class C2;   // Vorausdeklaration

class C1 {
  int i;
  public:
    void seti(int i) {this->i = i;}
    friend class C2;
   };

class C2{
  public:
    void showi(C1 c)
    {
      cout << c.i; // Klasse C2 darf das!!
    }
};
```


Friend Methode

```
class C2;   // Vorausdeklaration

class C1 {
  int i;
  public:
    void seti(int i) {this->i = i;}
    friend void C2::showi(C1 c);
   };

class C2{
  public:
    void showi(C1 c)
    {
      cout << c.i; // Methode C2::showi(C1 c) darf das!!
    }
};
```


Friend Funtion

```
class C1 {
  int i;
  public:
    void seti(int i) {this->i = i;}
    friend void showi(C1 c);
   };

void showi(C1 c)
{
  cout << c.i;     // Funktion showi darf das!!
}
```


Datei-I/O

- Die formatierte Dateiein- und -ausgabe funktioniert in C++ wie die Standartein- und -ausgabe über die Operatoren << und >>
- Mehre Klassen vereinheitlichen die Ein- und Ausgabe
- Einige wichtige Klassen (weitere Klassen existieren; z.B. für Unicode):
  - ios — Basisklasse für streams
  - ostream — Klasse für Output-Steams (u.a. cout)
  - istream — Klasse für Input-Streams (u.a. cin)
  - ofstream — Klasse für Datei-Output-Stream (Basisklasse: ostream)
  - ifstream — Klasse für Datei-Input-Stream (Basisklasse: istream)

Dateien öffnen / nutzen / schließen

- Zum Öffnen einer Datei bestehen im allg. mehrere Optionen:
  - Überladene Konstruktoren:
    - ofstream ausgabeDatei("Daten.out");
  - Methode open() (mit default-Parametern):
    - ifstream eingabeDatei;
      eingabeDatei.open("Daten.in");
- Überprüfen ob Öffnen funktioniert hat:
  - Methode is_open()
- Überprüfen ob Dateiende erreicht ist:
  - Methode eof()
- Zum Schließen:
  - Methode close()

Operatoren: Vereinbarung

- Operatoren werden ähnlich wie Funktionen bzw. Methoden, jedoch mit dem Schlüsselwort **operator** vereinbart
- Funktionsprototyp:
  - ***Datentyp*** **operator⊗(*Argumentliste*);**
- Methode:
  - **class *Name*{**
    ...
    Datentyp **operator⊗(*Argumentliste*);**
    }
  - ***Datentyp Name*****::operator⊗(*Argumentliste*)**
    {
    //Code
    }

Operatoren: Aufruf

- Der Aufruf eines Operators kann auf zwei verschiedene Weisen erfolgen
  - Wie eine Funktion/Methode; z.B.:
    - `s = operator+(x,y);`
    - `s = x.operator+(y);`
  - Wie ein „normaler" Operator; z.B.:
    - `s = x + y;`
- Bei der Operatorschreibweise kann man nicht mehr erkennen ob der Operator als „Funktion" oder als „Methode" vereinbart wurde.
- Intern wird die Operatorschreibweise in die Funktions- bzw. Methodenschreibweise umgewandelt.

Beispiel: Vereinbarung Operator / Methode

```
class feld {
  int wert[5];
  public:
    ...
    feld operator+(const feld& f);
   };

feld feld::operator+(const feld& f)
{
   feld temp;
   for(int i; i<5; ++i)
      temp.wert[i] = wert[i] + f.wert[i];
   return temp;
}

/* Variable:  feld f1, f2, summe;
   Aufruf:    summe = f1 + f2;  Zuweisung ???*/
```

Beispiel: Vereinbarung Operator / Funktion

```
class feld {
  int wert[5];
  public:
    ...
  friend feld operator-(const feld& f1, const feld& f2);
};

feld operator-(const feld& f1, const feld& f2)
{
   feld temp;
   for(int i; i<5; ++i)
      temp.wert[i] = f1.wert[i] – f2.wert[i];
   return temp;
}

/* Variable:  feld f1, f2, summe;
   Aufruf:    summe = f1 - f2;  Zuweisung ???*/
```


Unterschiede Operator-Funktion und -Methode

- Bei einer Methode ist der erste Parameter immer das entsprechende Objekt
  - feld f1, summe;
    summe = f1 + 3; // OK
    summe = 3 + f1; // nicht machbar
- Bei einer Funktion müssen alle Objekte als Parameter „übergeben“ werden und es muss der Zugriff auf die entsprechenden Objekt-Eigenschaften möglich sein
  - => im Allg. als friend-Funktionen

Beispiel: erster Parameter als Standarddatentyp

```
class feld {
    ...
    friend feld operator+(int i, const feld& f);
    friend feld operator+(const feld& f, int i);
   };

feld operator+(int i, const feld& f)
{
   feld temp;
   for(int k; k<5; ++k) temp.wert[k] = i + f.wert[k];
   return temp;
}

feld operator+(const feld& f, int i) { return i+f; }

/* Variable:  feld f1, f2, summe;
   Aufruf:    summe = 3 + f1 + f2 + 7 + 6 ; */
```


Spezielle Operatoren: =

- Die Zuweisung ist auch ein Operator (=)
- Wird keine Zuweisung vereinbart, wird (ähnlich wie beim default-Copy-Konstruktor) automatisch eine default-Zuweisung erzeugt
  - Kopie des Objekts wird erzeugt
  - Kein dynamischer Speicher => im Allg. OK
  - Dynamischer Speicher => im Allg. Fehlerhaft

Beispiel: Zuweisung 1

```
class feld {
    int* werte;
    int size;
    ...
    void operator=(const feld& f);
};

void feld::operator=(const feld& f)
{
   for(int i; i<size; ++i)
     wert[i] = f.wert[i];
}

/* Variable:  feld f1, f2, summe;
   f1 = f2; //OK
   summe = f1 + f2; // OK
   f1 = f2 = summe; // Fehler: f2 = summe liefert void!!!  */
```


Beispiel: Zuweisung 2 (besser)

```
class feld {
    int* werte;
    int size;
    ...
    feld& operator=(const feld& f);
};

feld& feld::operator=(const feld& f)
{
   for(int i; i<size; ++i)
     wert[i] = f.wert[i];
   return *this;
}

/* Variable:  feld f1, f2, summe;
   f1 = f2 = summe; // Jetzt OK!!!  */
```


Spezielle Operatoren: << und >>

- Auch die Ein- und Ausgabe kann man als Operatoren für seine eigenen Klassen verfügbar machen
- Da hierbei jeweils wieder entsprechende Streams zurückgegeben werden müssen können diese nur als friend Funktionen umgesetzt werden
  - `ostream& operator<<(ostream& s, const myclass m);`
  - `istream& operator>>(istream& s, myclass& m);`

Beispiel: Ausgabe

```
class feld {
    int* werte;
    int size;
    ...
    friend ostream& operator<<(ostream& s, const feld& f);
};

ostream& operator<<(ostream& s, const feld& f)
{
   s << "Feld: ";
   for(int i=0; i<size; ++i)
     s << f.wert[i] << " ";
   return s;
}

/* Variable:  feld f1, f2;
   cout << f1 << "\n" << f2;   */
```

Beispiel: Eingabe

```
class feld {
    ...
    friend istream& operator>>(istream& s, feld& f);
};

istream& operator>>(istream& s, feld& f)
{
   int newsize;
   cout << "Feldgröße?\n";
   s >> newsize;
   f.resize(newsize); // soll bereits implementiert sein
   cout << newsize << " Werte eingeben.\n"
   for(int i; i<newsize; ++i) s >> f.wert[i];
   return s;
}

/* Variable:  feld f1, f2;
   cin >> f1 >> f2;   */
```


C++: Was noch „fehlt“

- Templates
  - Beispiel: Stack für int / double / Klassen
- Ausnahmen
  - try catch und throw
- Run-Time-Type-ID (RTTI) und Cast
  - Zu welcher Klasse gehöre ich denn?
- Namensräume
  - Eigene Namensräume definieren und nutzen
- Standard-Template-Library (STL)
  - Klassen: Listen, Vektoren, ...
  - Algorithmen: find, sort, ...
- ...