Programmierung für Fortgeschrittene - C++

Graz, SS 2005

Gundolf Haase

Inhaltsverzeichnis

Kapitel 1

Objektorientiertes Programmieren in C++

1.1 Klassen in C++

In §8 des Skriptes *Einführung in die Programmierung* [Haase, 2004], führten wir die Klasse `Studenten` ein. Hier zur Erinnerung das ensprechende Headerfile

A1/studs.hpp

```
1  //      studs.hpp
2  //      Class Studenten mit dynamischen Character-Array
3
4  #include <iostream>
5  using namespace std;
6
7  class Studenten
8  {
9  public:         //                      Data in Studenten
10   long int matrikel;
11        int skz;
12        char *pname, *pvorname;
13
14 public:         //                      Methods in Studenten
15 //                              Default constructor (no argument)
16  Studenten();
17
18 //                              Constructor with 4 arguments
19  Studenten(const char vorname[], const char name[] = "",
20        const long int  mat_nr =0 ,   const int  skz_nr = 0);
21
22 //                              Copy constructor
23  Studenten(const Studenten& orig);
24
25 //                              Destructor
26  ~Studenten();
27
28 //                              Assignment operator
29  Studenten& operator=(const Studenten & orig);
30
31 //                              Output operator
32  friend ostream & operator<<(ostream  & s, const Studenten & orig);
33
34  const int & GetSKZ() const
35    { return skz; };
36
37  void SetSKZ(const int & s_in)
38    { skz = s_in; };
```

```
39  };
40
```

Diese Klasse enthält dynamische Datenstrukturen und somit sind die folgenden Methoden der Klasse unbedingt erforderlich (und sollten auch bei anderen Klassen als allererstes deklariert und definiert werden). Die Implementierungsdetails sind im Quelltextfile zu finden . A1/studs.cpp

- Standardkonstruktor,
- Parameterkonstruktor - dieser kann den Standardkonstruktor via optionale Parameter enthalten,
- Copy-Konstruktor,
- Destruktor - räumt den dynamisch allokierten Speicher auf,
- Zuweisungsoperator, bestehend aus Test mit `this`, Destruktorteil und Copy-Konstruktorteil.

Wir verwenden nunmehr ausschließlich die neuen C++-Headerfiles des Systems (ohne Endung *.h*), was sich in Zeilen 4 und 5 von *A1/studs.hpp* in der Include-Anweisung `#include <iostream>` und der Nutzung des Namensraumes `std` äußert. Das Hauptprogramm *a1_1.cpp* ist von *Ex851.cpp* aus [Haase, 2004] abgeleitet und mit A1/a1_1.cpp

```
g++ -o a1_1 a1_1.cpp studs.cpp
```

wird das ausführbare Programm `./a1_1` erzeugt.

Die Implementierung der dynamischen Strings in den *studs.**-Files dient dem bewußtbarmachen, daß man mit dynamischem Speicher sorgfältig umgehen muß. Auf diese Problematik werden wir später zurückkommen. Die nunmehr elegante C++ Implementierung der Klasse `Studenten` nutzt die Standard-Klasse `string` deren Instanzen den Speicherplatz dynamisch verwalten, jedoch wie normale Variable verwendet werden können [Kirch-Prinz and Prinz, 2002, §18],[Schmaranz, 2002, §16.5]. Die Änderungen im Headerfile sind marginal, siehe Zeilen 4 und 12 des nachfolgenden Listings. A1/studs2.hpp

---

```
1   //      studs2.hpp
2   //      Class Studenten mit string-klasse
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   class Studenten
8   {
9   public:         //              Data in Studenten
10    long int matrikel;
11         int skz;
12         string name, vorname;
13   ...
14  };
```

Die entsprechenden Definitionen der Methoden im Quelltextfile sind einfacher als vorher, da eine Anweisung wie `vorname = orig.vorname;` nunmehr das dynamische Speichermanagement bereits beinhaltet.

A1/studs2.cpp

```
1   //      studs2.cpp
2   //      Class Studenten  mit string-klasse
3
4   #include <iostream>
5   #include <string>
6   #include "studs2.hpp"
7
8   using namespace std;
9
```

```
10  // ------------------------------------------------------------------
11
12  Studenten :: Studenten()
13  {
14   cout << "Standard-konstruktor" << endl;
15   matrikel = skz = 0;
16   name = vorname = string();
17  }
18
19  // ------------------------------------------------------------------
20
21  Studenten :: Studenten(const char vorname_c[], const char name_c[],
22                    const long int  mat_nr,    const int  skz_nr)
23  {
24   cout << "Parameter-konstruktor" << endl;
25   matrikel = mat_nr;
26   skz      = skz_nr;
27
28   name = string(name_c);
29   vorname = string(vorname_c);
30  }
31
32  // ------------------------------------------------------------------
33
34  Studenten :: Studenten(const Studenten& orig)
35  {
36   cout << "Copy-konstruktor" << endl;
37
38   matrikel = orig.matrikel;
39   skz      = orig.skz;
40   name     = orig.name;
41   vorname  = orig.vorname;
42  }
43
44  // ------------------------------------------------------------------
45
46  Studenten :: ~Studenten()
47  {
48   cout << "Destruktor " << endl;
49  }
50
51  // ------------------------------------------------------------------
52
53  Studenten& Studenten :: operator=(const Studenten & orig)
54  {
55   if ( &orig != this )
56    {
57      matrikel = orig.matrikel;
58      skz      = orig.skz;
59      name     = orig.name;
60      vorname  = orig.vorname;
61    }
62
63   return *this;
64  }
65
66  // ------------------------------------------------------------------
67
68  ostream & operator<<(ostream  & s, const Studenten & orig)
69  {
70   return s << orig.vorname << " "   << orig.name << " , "
71            << orig.matrikel << " , " << orig.skz;
72  }
73
```

Das Hauptprogramm *a1_2.cpp* ist analog zu *a1_1.cpp* und wird mit A1/a1_2.cpp
`g++ -o a1_2 a1_2.cpp studs2.cpp`
erzeugt.

1.2 Bessere Klassen in C++

Die Konstruktoren der Klasse `Studenten` in den Files *studs2.** sind noch nicht ganz so gestaltet, wie es der C++-Philosophie entspricht.

Bisher verwendeten wir zur gleichzeitigen Deklaration und Initialisierung einer Konstanten bzw. einer Variablen

```
const   int  N = 10;
      float  a = 5.98;
```

Jede Zeile obigen Codes entspräche bei Klassen formal dem Aufruf eines Standardkonstruktors und des Zuweisungsoperators. Genauso wie für Klassenvariablen könnte man (und sollte man in C++) die Konstante und die Variable über den zugehörigen Copy-Konstruktor initialisieren.

```
const   int  N(10);
      float  a(5.98);
```

Während diese Art der Initialisierung für einfache Datentypen in normalen Funktionen etwas gewöhnungsbedürftig ist, paßt diese Art der Initialisierung von Daten zu Konstruktoren von Klassen wesentlich besser. Da im Konstruktor Member der Klasse initialisiert werden, nennt man die nach dem Separator `:` folgenen Ausdrücke **Memberinitialisierer**. Der neue Copy-Konstruktor für `Studenten` sieht dann so aus:

```
1  //                                    andere Initialisierung im Copy-Konstruktor
2  Studenten :: Studenten(const Studenten& orig)
3          : matrikel(orig.matrikel), skz(orig.skz), name(orig.name), vorname(orig.vorname)
4  {
5   cout << "Copy-konstruktor" << endl;
6  }
```

Wir erweitern die Klasse `Studenten` um ein Konstante `max_terms`, welche die maximale Anzahl der Semester (also die Regelstudienzeit) für jeden Studenten individuell festlegt, d.h., jede Instanz der Klasse hat eine eigene (evtl. andere) Konstante. Gleichzeitig wird mit `static int anzahl` ein weiteres Member der Klasse deklariert, welches die Anzahl der Instanzen (=Variablen) der Klasse mitzählen soll. Durch das Schlüsselwort `static` wird angezeigt, daß es diese Membervariable nur **einmal für alle** Instanzen gibt. Man kann sich dies wie eine globale Variable `Studenten::anzahl` vorstellen, auf welche alle Instanzen der Klasse `Studenten` zugreifen können. Hier die Änderungen in der Deklaration der Klasse A1/studs3.hpp

```
1  //      studs3.hpp
2  //      Class Studenten mit const-Member und static-Member
3  //      Das static-Member ist  e i n e  Variable f"ur  a l l e  Instanzen der Klasse Studenten
4  ...
5  class Studenten
6  {
7  public:         //                    Data in Studenten
8    long int matrikel;
9         int skz;
10        string name, vorname;
11   static int anzahl;    // e i n e   Variable  f"ur  a l l e  Instanzen der Klasse
12   const int max_terms;  // j e d e  Instanz der Klasse hat ihre eigene Konstante
13
14 public:         //                    Methods in Studenten
15 ...
```

```
16  //                              Constructor with 5 arguments
17   Studenten(const char vorname_c[], const char name_c[] = "",
18         const long int  mat_nr =0 ,   const int  skz_nr = 0,
19         const int maximal_terms = 20                          );
20  ...
21  int AnzahlStudenten() const  { return anzahl; };
22  };
```

Nunmehr müssen auch die Definitionen der Methoden geändert werden . In nachfolgendem Listing wird in Zeile 5 die (für die Klasseninstanzen globale) Variable `anzahl` initialisiert und in den Konstruktoren inkrementiert bzw. im Destruktor dekrementiert. Die aktuelle Anzahl der Instanzen kann über die Methode `Studenten::AnzahlStudenten()` im Programm abgefragt werden, siehe Zeile 36 in *a1_3.cpp* .

A1/studs3.cpp

A1/a1_3.cpp

```
1   //      studs3.cpp
2   ...
3   #include "studs3.hpp"
4   ...
5   int Studenten :: anzahl = 0;    // Intialisierung des statischen Members der Klasse
6
7   // ------------------------------------------------------------------------
8
9   Studenten :: Studenten()
10          : matrikel(0), skz(0), name(), vorname(), max_terms(10)
11  {
12   anzahl++;
13   cout << "Standard-konstruktor" << endl;
14  }
15
16  // ------------------------------------------------------------------------
17
18  Studenten :: Studenten(const char vorname_c[], const char name_c[],
19                    const long int  mat_nr,    const int  skz_nr,
20                    const int maximal_terms                        )
21          : matrikel(mat_nr), skz(skz_nr), name(name_c), vorname(vorname_c),
22            max_terms(maximal_terms)
23  {
24   anzahl++;
25   cout << "Parameter-konstruktor" << endl;
26  }
27
28  // ------------------------------------------------------------------------
29  //      Copy constructor will be applied only to uninitialized data
30  //      ===> no deallocation of memory necessary
31
32  Studenten :: Studenten(const Studenten& orig):
33          matrikel(orig.matrikel), skz(orig.skz), name(orig.name), vorname(orig.vorname),
34          max_terms(orig.max_terms)
35  {
36   anzahl++;
37   cout << "Copy-konstruktor" << endl;
38  }
39
40  // ------------------------------------------------------------------------
41
42  Studenten :: ~Studenten(
43  {
44   anzahl--;
45   cout << "Destruktor " << endl;
46  }
```

Die Initialisierung des konstanten Klassenmembers `max_anzahl` **muß** , wie in Zeile 10, über den Memberinitialisierer im Konstruktor erfolgen. Eine spätere Initialisierung ist nicht mehr möglich.

Insbesondere kann dieses konstante Member auch im Zuweisungsoperator nicht mehr überschrieben werden - Man sollte sich also genau überlegen, welche Klassenmember konstant sein müssen.

Spezifische Konstanten, welche für alle Instanzen einer Klasse gelten sollen, sind in der Klassendeklaration (also im Headefile) mit `static const double Gravity = 9.806;` anzugeben. Diese Konstante wird nur einmal abgespeichert (wir sparen Speicherplatz) und sie ist gleichzeitig nur innerhalb der Klasse gültig.

Operatorüberladung

`MyVector`

1.3 Ableitungen von Klassen

Dieser Abschnitt der Vorlesung orientiert sich an [Corp., 1993, §7] und wir beginnen mit den objektorientierten Sprachunterstützungen von C++.

1.3.1 Design einer Klassenhierarchie

Wir stellen uns ein Möbelhaus mit drei Arten von Angestellten vor: den normalen Angestellten, den Verkäufern und den Managern. Von allen benötigt die Buchhaltung den Namen, die Lohnverrechnung ist jedoch unterschiedlich. So wird der Angestellte (`WageEmployee`) auf Stundenbasis bezahlt, der Verkäufer (`SalesPerson`) auf Stundenbasis und Verkaufsprovision, der Manager (`Manager`) erhält ein wöchentliches Gehalt.

Natürlich könnte man nun 3 Klassen/Strukturen konventionell programmieren und dann mit `switch`-Anweisungen immer unterscheiden, welche Variante der Lohnverrechnung nun benutzt werden soll. Dieses Konzept ist aber sehr fehleranfällig bzgl. der Erweiterbarkeit unserer Angstelltenklassen (Verkaufsmanager, Aktienindexmanager), da dann **immer alle** `switch`-Anweisungen geändert werden müssen - und irgendetwas vergißt man immer.

Der objektorientierte Ansatz stellt erstmal die Frage nach den gemeinsamen Eigenschaften unserer Angestellten und dann erst nach den Unterschieden. Die Gemeinsamkeiten bei allen sind der Name und die notwendige Lohnberechnung. Unterschiede bestehen in der Art und Weise der Lohnberechnung und der dafür notwendigen Daten. Zusätzlich könnte `SalesPerson` die stundenbasierte Lohnberechnung von `WageEmployee` nutzen.

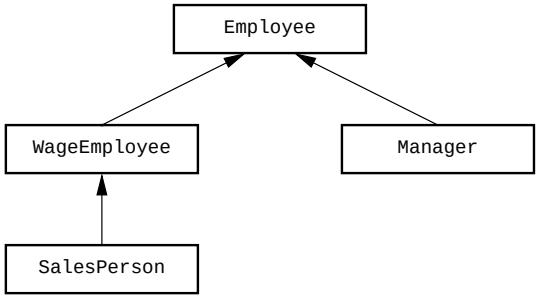
Diese Betrachtungen stehen in enger Verbindung mit dem Vererbungskonzept im objektorientierten Programmieren. Dort deklariert man Basisklassen, von denen weitere Klassen, mit zusätzlichen Eigenschaften ableitet werden. In [Schmaranz, 2002, §8.2] ist dieses Vererbungskonzept sehr schön erläutert:

- Eine Ableitung einer Klasse repräsentiert eine **IS-A**-Relation.
- Eine Membervariable repräsentiert eine **HAS A**-Relation

In unserem Falle wären alle Angestellen erstmal Beschäftige, d.h., wir benötigen eine Basisklasse `Employee`. Dann können wir sagen:

- `Employee` **HAS-A** Name.
- `WageEmployee` **IS-A**n `Employee` und `WageEmployee` **HAS-A** (zusätzlich) einen Stundenlohn und eine Arbeitszeit.
- `SalesPerson` **IS-A** `WageEmployee` und `SalesPerson` **HAS-A** (zusätzlich) eine Umsatzbeteiligung am erbrachten Umsatz.
- `Manager` **IS-A**n `Employee` und `Manager` **HAS-A** Wochengehalt (zusätzlich).

Dies ergibt folgende Hierarchy von Klassen. Man beachte, daß wir bis jetzt noch überhaupt keine konkrete Implementierung angesprochen haben. Vielmehr betrachten wir nur Eigenschaften der zu handhabenden Objekte. Dieses **Design** der Klassenhierarchie muß immer zu Beginn eines OO-Programmes stehen. Ein gründlich erarbeitetes Design erspart zeitraubende und fehleranfällige Umstrukturierungen der Klassenhierarchie in einem späteren Projektstadium.



1.3.2 Die Basisklasse

Unser Entwurf für die Basisklasse `Employee` sieht so aus :

A3/employ.hpp

```
1   //              employ.hpp
2   #include <string>
3   using namespace std;                    // std::string
4
5   class Employee
6   {
7    public:
8     Employee();
9     Employee(const char *name);
10    const string& getname() const;        // Namen holen
11    void info();                          // Namen ausgeben
12    void payment();                       // Gehalt ausgeben
13   private:
14    string name;
15  };
16  ...
```

Man beachte, daß die Methode für die Gehaltsausgabe, `payment()` in der Basisklasse keine sinnvolle Berechnung ausführen kann, da keinerlei Daten zur Gehaltsberechnung vorhanden sind. An dieser Stelle wird hier erstmal eine Dummy-Methode implementiert . Diese unsaubere Lösung kann erst in §1.4 richtig ersetzt werden.

A3/employ.cpp

1.3.3 Abgeleitete Klassen

Unser `WageEmployee` **IS-A**n `Employee` and **HAS-A** Stundenlohn und Arbeitszeit. Diese Zusatzeigenschaften erfordern zusatzliche Methoden zur ihrer Handhabung und erlauben nun eine sinnvolle Methode `payment()`.

```
1   //              employ.hpp
2   ...
3   class WageEmployee : public Employee
4   {
5    public:
6      WageEmployee();
7      WageEmployee(const char *nm);        // Konstructor
8      void setWage(const float lohn);      // Uebergabe Stundenlohn
9      void setHours(const float std);      // Uebergabe geleisteter Arbeitsstunden
10     float getWage();                     // Stundenlohn
11     float getHours();                    // geleistete Arbeitsstunden
12     float computePay();                  // Lohnberechnung
13     void payment();                      // Ausgabe Lohnberechnung
14    private:
```

```
15      float wage;                          // Stundenlohn
16      float hours;                         // Arbeitszeit in Stunden
17    };
18    ...
```

Die neuen Eigenschaften (**HAS-A**) sind in Zeilen 15,16 obigen Codefragmentes deklariert. Zeile 3 beinhaltet die Ableitung der neuen Klasse von der Basisklasse (**IS-A**). Das Schlüsselwort `public` in Zeile 3 erlaubt den Zugriff auf Basisklassenmethoden wie `getname()` über Instanzen (Variablen) der Klasse `WageEmployee`. Dies erlaubt dann folgendes Codefragment:

```
1   WageEmployee ab("Ritchie Valens");
2     cout << ab.getname() << endl;
```

Eine Klassenableitung der Form `class WageEmployee : private Employee` oder `class WageEmployee : protected Employee` verbietet die Benutzung der Methode `getname()` in obiger Form. Während die Verwendung von `protected` wenigstens die Nutzung der Methode innerhalb der Klasse `WageEmployee` erlaubt, ist selbst dies bei der Ableitung als `private` nicht möglich.

In analoger Weise leiten wir die Klasse `Manager` ab.

```
1   //              employ.hpp
2   ...
3   class Manager : public Employee
4   {
5     public:
6      Manager();
7      Manager(const char *nm);             // Konstructor
8      void setSalary(const float salary);  // Uebergabe Wochengehalt
9      float computePay();                  // Lohnberechnung
10     void payment();
11    private:
12     float weeklySalary;                  // Wochengehalt
13  };
```

Die noch fehlende Klasse `SalesPerson` benötigt die Klasse `WageEmployee` als Basisklasse, da die dortige, stundenweise Lohnverrechnung auch hier wieder gebraucht wird.

```
1   //              employ.hpp
2   ...
3   class SalesPerson : public WageEmployee
4   {
5     public:
6      SalesPerson();
7      SalesPerson(const char *nm);         // Konstructor
8      void setComission(const float comm); // Uebergabe Umsatzbeteiligung
9      void setSales(const float sales);    // Uebergabe des Umsatzes
10     float computePay();                  // Lohnberechnung
11     void payment();                      // Ausgabe Lohnberechnung
12    private:
13     float comission;             // Umsatzbeteiligung
14     float SalesMade;             // Umsatz
15  };
16  ...
```

Die Implementierungen sämtlicher Methoden der vier Klassen ist im Quelltextfile zu finden. Hervorzuheben ist an dieser Stelle die Methode `computePay` der Klasse `SalesPerson`:

A3/employ.cpp

```
1  float SalesPerson::computePay()
2  {
3    return WageEmployee::computePay() + comission*SalesMade;
4  }
```

Hier benötigen wir explizit den Scope-operator `::` um die `computePay`-Methode der Basisklasse `WageEmployee` zu benutzen. Was würde passieren, falls wir stattdessen `return computePay() + comission*SalesMade;` programmieren würden?

Eine einfache Demonstration unserer neuen Klassen ist in *A3/a3_1.cpp* zu finden . A3/a3_1.cpp

1.3.4 Konvertierungen abgeleiteter Klassen

Konvertierungen waren bislang stillschweigend in unseren Programmen enthalten, wie in

```
{
 double  dd = 15.373;
 int     ii = 5, kk;

 dd = ii;
 kk = dd;
}
```

Die erste Konvertierung (`int` ⟶ `double`) ist problemlos, da die Nachkommastellen der Gleitkommazahl mit Nullen gefüllt werden. Im Gegensatz dazu gehen bei der zweiten Konvertierung (`double` ⟶ `int`) sämtliche Nachkommastellen verloren und bei ensprechenden Optionen (`-Wall`) gibt der Compiler eine Warnung aus. Beide Konvertierungen sind versteckt im Code enthalten, daher nennt man sie *implizit*. Mit einer *explizit*en Konvertierung (Cast-operator) der zweiten Zuweisung `kk = (int)dd;` läßt sich die Warnung (beim `g++`) abschalten.

Implizite Konvertierung

Bei Klassen gibt es eine implizite Typkonvertierung nur in Richtung abgeleitete Klasse zu Basisklasse vor dem Hintergrund, daß eine abgeleitete Klasse ein *spezieller Typ* der Basisklasse ist (also gemeinhin mehr Daten beinhaltet als diese). Daher sind alle Member der Basisklasse in der abgeleiteten Klasse enthalten. Das folgende Beispiel veranschaulicht dies. A3/a3_2.cpp

```
1  //                          a3_2.cpp
2  ...
3   WageEmployee aWage;
4   SalesPerson  aSale("Buddy Holly");
5
6    aWage = aSale;             //      erlaubte Konvertierung
7    aWage.payment();           //        (Provision ist futsch)
8
9    SalesPerson aSale2;
10 //  aSale2 = aWage;           //      nicht erlaubte Konvertierung
11 ...
```

In Zeile 6 sind Name, Stundenlohn und Stundenanzahl (Members) des Angestellten `aWage` als entsprechende Einträge des Verkäufers `aSale` vorhanden. Die Verkaufsprovison (spezielles Member) wird dabei nicht übernommen. Umgekehrt müßte der Compiler eine unkommentierte Zeile 10 ablehnen, denn woher soll er (implizit!!) eine vernünftige Verkaufsprovision aus den vorhandenen Daten des Angestellten ableiten (die Klasse `WageEmployee` besitzt weniger Member als `SalesPerson`)? Näheres zu Konvertierungskonstruktoren und -funktionen zwischen Klassen steht in [Kirch-Prinz and Prinz, 2002, §9 und §19], [Schmaranz, 2002, §9.4.4], [Schader and Kuhlins, 1998, §16.5].

Konvertierung von Klassenpointern

Die demonstrierte, implizite Typkonvertierung erlaubt uns, daß ein mit einer Basisklasse typisierter Pointer sowohl auf Instanzen der Basisklasse als auch auf Instanzen davon abgeleiteter Klassen zeigt.

Dies, und mißbräuchliche Benutzung demonstrieren wir wiederum an einem Beispiel : A3/a3_2.cpp

```
1   //                              a3_2.cpp
2   ...
3    SalesPerson  aSale("Buddy Holly");
4   ...
5    SalesPerson *salePtr;          //      nun das ganze mit Pointern
6    WageEmployee *wagePtr;
7
8    salePtr = & aSale;
9    wagePtr = & aSale;             //      Basisklassenpointer
10
11   salePtr->setSales(1000);       // SalesPerson :: setSales
12  //  wagePtr->setSales(1000);    // WagePerson  :: setSales  gibt es nicht !!
13
14   salePtr->payment();            // SalesPerson ::computePay
15   wagePtr->payment();            // WageEmployee::computePay
16
17  //      und nun schmutzig mit Pointern
18    cout << endl << "         schmutzig mit Pointern" << endl;
19
20    //salePtr = wagePrt;                  // geht nicht
21    salePtr = (SalesPerson *) wagePtr;    // expizite Typkonvertierung, gefrlich !!
22    salePtr->payment();
23
24  //      und nun  g a n z  schmutzig mit Pointern
25    cout << endl << "         g a n z   schmutzig mit Pointern" << endl;
26
27    Employee *empPtr = &simple;   // --> einfacher Employee
28    SalesPerson *sale_Ptr;
29
30    sale_Ptr = (SalesPerson *)empPtr;     // erlaubt, aber inkorrekt
31    sale_Ptr->setComission(0.06);         // nunmehr erlaubt, aber katastrophal
32    sale_Ptr->payment();
33  ...
```

Wie in Zeile 12 zu sehen ist, können nur die Methoden der Klasse aufgerufen werden, für welche der Pointer typisiert ist - obwohl er auf eine Instanz zeigt, für welche die Methode `setSales` deklariert ist. Heikel (und verboten gehört) ist die Pointerkonvertierung in Zeile 21 da die Berechnungsmethode in Zeile 22 auf nicht vorhandene Member zugreifen könnte. Hier funktioniert es nur deshalb, da `wagePtr` auf eine Instanz der Klasse `SalesPerson` verweist (Zeile 9).

Die Katastrophe tritt dann aber in den Zeilen 30-32 ein, da hier wirklich auf nicht vorhandene Member zugegriffen wird. Im günstigsten Fall ist die Berechnung in Zeile 32 falsch, wenn man Pech hat, dann stürzt das Programm unkontrolliert ab.

Solche Pointerkonvertierungen von Basisklasse zu abgeleiteter Klasse sind unnötig, gefährlich und zeugen von einem schlechten Klassendesign.

Nutzung von Basisklassenpointern

Natürlich stellt sich die Frage, wozu man Basisklasenpointer gebrauchen kann. Eine Teilantwort ist, daß es damit z.B., möglich ist alle verschiedenen Angestellten (`Employee`) in einem Array zu speichern und dann Aktionen nur noch auf die einzelnen Feldelemente anzuwenden. Dieser Ansatz wird in im nachfolgenden Beispiel demonstriert : A3/a3_3.cpp

```
1  //              a3_3.cpp
2  #include <iostream>
3  #include "employ.hpp"
4  using namespace std;
5
6  int main()
7  {
8   Employee  simple("Josef Gruber");
9
10  WageEmployee hilf("Heiko");
11  hilf.setWage(20); hilf.setHours(7.8);
12
13  SalesPerson emp("Gundolf Haase");
14  emp.setWage(hilf.getWage());  emp.setHours(hilf.getHours());
15  emp.setComission(0.05); emp.setSales(10000.0);
16
17  Manager man("Max Planck");
18  man.setSalary(1000.0);
19
20  const int N=4;
21  Employee* liste[N];            // array von Pointern auf Employee
22  int i;
23
24  liste[0] = &simple;
25  liste[1] = &hilf;
26  liste[2] = &emp;
27  liste[3] = &man;
28
29  cout << endl << "   Nur die Namen ausgeben" << endl;
30  for (i=0; i<N; i++)
31   {
32     liste[i]->info();
33   }
34
35  cout << endl << endl << "   Name und (spezifisches) Gehalt klappen nicht" << endl;
36  for (i=0; i<N; i++)
37   {
38     liste[i]->payment();        //      ==> Ausweg: Virtuelle Methoden
39   }
40
41  return 0;
42 }
```

Die Namensausgabe in Zeile 32 funktioniert so wie geplant, jedoch wird in Zeile 38 stets die Methode `Employee :: payment()` aufgerufen, welche jedoch keine geeignete Gehaltsberechnungsroutine zur Verfügung hat. Die Definition einer Methode `computePay()` für die Basisklasse `Employee` löst das Problem auch nicht, da in Zeile 38 die verschiedenen Beschäftigungsverhältnisse *a priori* nicht bekannt sind. Aber gerade diese Unterschiede sind das Spezielle in den abgeleiteten Klassen.

Natürlich bietet C++ einen Ausweg aus dem Dilemma - die *virtuellen Methoden* mit ihren *dynamischen Bindungen*.

Einige Bemerkungen zum Casting

An Stelle des C-Casts `(Typ) Ausdruck` wie in `(double) idx` sollte man die vier in C++ enthaltenen Casts benutzen, welche ein spezifischeres Casting erlauben, leichter im Programmcode auffindbar sind und es dem Compiler auch erlauben, bestimmte Casts abzulehnen.

- `static_cast<Typ>(Ausdruck)` ersetzt das bekannte C-Cast, also würde obiges Beispiel zu `static_cast<double>(idx)`
- `const_cast<Typ>(Ausdruck)` erlaubt die Beseitigung der Konstantheit eines Objektes. Eine Anwendung dafür ist der Aufruf einer Funktion, welche nichtkonstante Objekte in der

Parameterliste erwartet, diese aber nicht verändert.

```
void Print(Studenten&);
...
int main()
{
 const Studenten arni("Arni","Schwarz",89989, 787);
 Print(const_cast<Studenten>(arni));
}
```

- `dynamic_cast<Typ>(Ausdruck)` dient der sicheren Umformung von Pointern und Referenzen in einer Vererbungshierarchie und zwar nach unten (abgeleitete Klasse) oder zwischen benachbarten Typen. So wäre das exakte Casting in Zeile 21 des Codes auf Seite 10: `salePtr = dynamic_cast<SalesPerson*>(wagePtr);` . Falls das Casting (zur Laufzeit!!) nicht erfolgreich ist wird ein Nullzeiger zurückgegeben bzw. eine Exception ausgeworfen bei Referenzen.
- `reinterpret_cast<Typ>(Ausdruck)` wird für Umwandlungen benutzt deren Ergebnis fast immer implementationsabhängig ist. Meist wird es zur Umwandlung von Funktionspointern benutzt.

Obige Erläuterungen und weitere Beispiele zu diesen Casts sind in [Meyers, 1997, §1.2] zu finden. Mehr Beispiele und Bemerkungen zur Typüberprüfung der Casts, siehe [Schmaranz, 2002, p.246f].

1.4 Virtuelle Methoden

Der Grund, daß der Code in §1.3.4 nicht wie gewünscht die spezifischen Gehälter ausgerechnet hat liegt darin, daß die, schon in der Basisklasse vorhande Methode `payment()` (und auch `computePay()`) in den abgeleiteten Klassen redefiniert wurde. Dies hat natürlich auf die Basisklasse keine Auswirkung, sodaß eine über einen Basisklassenpointer adressierte Instanz konsequenterweise immer die in der Basisklasse deklarierte Methode aufruft.

Die Alternative zur Redefinition ist ein Ersatz: Soll eine Methode der Basisklasse durch eine Methode der abgeleiteten Klasse *komplett ersetzt* werden, deklariert man diese Methode in der Basisklasse als *virtuell*. Das entsprechende Schlüsselwort ist `virtual`. Ist eine Methode einer Basisklasse virtuell, dann sind alle gleichnamigen Methoden in abgeleiteten Klassen automatisch virtuell, ohne daß das Schlüsselwort angegeben werden muß.

1.4.1 Nutzung virtueller Methoden

Mit solchen virtuellen Methoden läßt sich erreichen, daß der Code von Seite 11 so funktioniert, daß immer die richtige Gehaltsberechnungsmethode in Zeile 38 aufgerufen wird. Dazu müssen wir nur die Basisklasse leicht abändern :

A4/employ.hpp

```
1  //                      employ.hpp
2  ...
3  class Employee
4  {
5   public:
6    Employee();
7    Employee(const char *name);
8    const string& getname() const;
9    void info();                          // Namen ausgeben
10   virtual void payment();       //  N E W,  virtuell
11   virtual ~Employee()           //  N E W,  jetzt notwendig
12                 {};
13  private:
14   string name;
15 };
16 ...
```

Im Verzeichnis *A4* kann jetzt der Code mit A4/a4_1.cpp
`g++ -Wall -o a4_1 a4_1.cpp employ.cpp`
übersetzt und gelinkt werden. Die Änderung in Zeile 10 erlaubt es, zur Laufzeit zu entscheiden, welche Methode `payment()` aufgerufen werden soll. In unserem Code bedeutet dies für den letzten Zählzyklus in den Zeilen 36-39,

```
for (i=0; i<N; i++)
 { liste[i]->payment(); }
```

daß folgende Methoden gerufen werden:

$i = 0$ `Employee :: payment()`
$i = 1$ `WageEmployee :: payment()`
$i = 2$ `SalesPerson :: payment()`
$i = 3$ `Manager :: payment()`

Diese Möglichkeit eine Methode für eine Instanz aufzurufen, ohne dessen Typ genau zu kennen, nennt man *Polymorphie*. Dieser griechische Begriff bezeichnet die Fähigkeit sich von verschiedenen Seiten zu zeigen oder verschiedene Formen anzunehmen [Corp., 1993, p.161].

Besitzt eine Klasse mind. eine virtuelle Methode, dann muß der Destrukor ebenfalls virtuell sein, was in unserer Basisklasse `Employee` der Fall ist (Der Destruktor ist dort auch gleich definiert.). Damit man die Notwendigkeit hierfür einsieht, betrachten wir das Ende des Gültigkeitsbereiches von `liste`. Für jedes Element des Arrays von Basisklassenzeigern wird dann der Destruktor aufgerufen. Ohne einen virtuellen Destruktor in der Basisklasse würde dann stets der Destruktor der Basisklasse aufgerufen. Dies bewirkt bei abgeleiteten Klassen mit dynamisch allokiertem Speicher, daß dieser Speicher nicht freigegeben wird. Bei einem virtuellen Destruktor der Basisklasse wird wiederum erst zur Laufzeit entscheiden, welcher Destruktor aufgerufen wird sodaß ein sauberes Speichermanagement möglich ist (es wird erst der Destruktor der abgeleiteten Klasse aufgerufen welcher seinerseits den Destruktor der Basisklasse implizit aufruft).

1.4.2 Rein virtuelle Methoden

Die Methode `payment()` der Basisklasse `Employee` ist nur als Platzhalter implementiert und führt keine sinnvollen Berechnungen aus. Eigentlich benötigen wir diese Methode nur, um anzuzeigen, daß eine Methode `payment()` aus abgeleiteten Klassen verwendet werden soll. Dies ist nicht sonderlich elegant, denn der einzige Zweck von `Employee :: payment` besteht darin, *nie* aufgerufen zu werden.

Eine Alternative dazu besteht in einer Methode, die weder semantisch noch physisch existiert. Eine derartige Methode ist *rein virtuell* und sie wird definiert, indem man der Deklaration ein `=0` anhängt. Die neue Deklaration der Klasse `Employee` ist dann : A4/employ2.hpp

```
1  //                      employ2.hpp
2  class Employee
3  {
4   public:
5    Employee();
6    Employee(const char *name);
7    const string& getname() const;
8    void info();
9    virtual void payment() = 0;          //  N E W,  r e i n   virtuell
10   virtual ~Employee() {};
11  private:
12   string name;
13 };
```

Im Definitionsfile muß natürlich die Definion von `Employee :: payment` entfernt werden. A4/employ2.hpp

Die rein virtuelle Deklaration von `Employee :: payment` hat weitere Konsequenzen:

- Es läßt sich keine Instanz (Variable) der Klasse `Employee` mehr erzeugen. Es können jedoch nach wie vor Basisklassenpointer auf die Klasse `Employee` deklariert werden, um auf Instanzen abgeleiteter Klassen zu zeigen. Deshalb muß auch Zeile 24 des Hauptprogramm von Seite 11 in
  `liste[0] = &hilf`
  geändert und die Definition in Zeile 8 gestrichen werden . A4/a4_2.cpp
- Jetzt *muß* `payment` in den abgeleiteten Klassen definiert werden.
- `Employee` ist eine *abstrakte Klasse*, da sie eine rein virtuelle Funktion enthält und somit von ihr keine Instanzen deklariert werden können.
- Klassen, von welchen Instanzen deklariert werden können heißen *konkrete Klassen*. Somit ist, z.B., `WageEmployee` eine konkrete Klasse.

Falls eine Klasse von einer abstrakten Basisklasse abgeleitet wird und die darin enthaltenen rein virtuellen Methoden nicht definiert, dann erbt die neue Klasse auch diese reine virtuellen Funktionen und wird damit selbst eine abstrakte Klasse. Bei einer normalen virtuellen Methode in der (dann konkreten) Basisklasse würde in diesem Falle einfach die Methode der Basisklasse verwendet.

1.4.3 Dynamische Bindung

In unserem Beispiel fungieren die virtuelle Methode `payment` und der virtuelle Destruktor als dynamische Methoden, im Gegensatz den bislang verwendeten statischen Funktionsaufrufen. Diese dynamische Bindung zum Programm wird über die Virtual Method Table (VMT) realisiert.

Eine besondere Eigenschaft der dynamischen Bindung stellt die Möglichkeit dar, das Verhalten bereits existierenden Codes nachträglich zu verändern, ohne daß die bereits existierenden Teile neu compiliert werden müssen. Bereits übersetzte Module können so ohne Veränderung des Codes oder einer Neukompilierung *nachträglich* um neue Datentypen erweitert werden. Dies wollen wir an einem Beispiel demonstrieren.

Zuerst separieren wir den Teil des Hauptprogrammes, welcher die Polymorphie ausnutzt, in eine extra Funktion `PrintListe` welche im File *liste.cpp* definiert ist.

A4/liste.cpp

```
1   #include <iostream>
2
3   #include "employ2.hpp"  // Nur die Basisklasse ist notwendig!!
4
5   using namespace std;
6
7   void PrintListe(const int n, const Employee* const liste[] )
8   {
9    int i;
10
11   cout << endl << "   Nur die Namen ausgeben" << endl;
12   for (i=0; i<n; i++)
13    {
14      liste[i]->info();
15    }
16
17  //              NEU !!
18     cout << endl << endl << "   Name und (spezifisches) Gehalt klappt  j e t z t" << endl;
19    for (i=0; i<n; i++)
20    {
21      liste[i]->payment();
22    }
23
24  }
25
```

Die Files *liste.cpp* und *employ2.cpp* werden compliert
`g++ -c liste.cpp employ2.cpp`
und im folgenden benutzen wir nur noch die beiden Objektfiles *liste.o* und *employ2.o*.

Von der Klasse `Manager` leiten wir eine Klasse `BoxPromoter` mit der neuen Eigenschaft der Bestechlichkeit ab. Also: `BoxPromoter` **IS-A** `Manager` und `BoxPromoter` **HAS-A** Eigenschaft der Bestechlichkeit.

A4/bestech.hpp

```
1   //                      bestech.hpp
2   // Demonstration, damit virtuellen Funktionen ein bersetzter
3   // Programmteil nachtrlich verdert werden kann.
4
5   #include "employ2.hpp"
6
7   class BoxPromoter : public Manager
8   {
9     public:
10     BoxPromoter();
11     BoxPromoter(const char *nm);                 // Konstructor
12     //void setSalary(const float salary);        // --> Manager::setSalery
13     float computePay() const;                    // virtuell
14     void payment() const;                        //  (auch virtual lbar)
15     void setBestechung(const float bestechung);  // N E W
16    private:
17     float bestechung_;                           // Bestechungsgeld
18  };
19
```

Die Zeile 13 ist auskommentiert, da die Klasse `BoxPromoter` die `setSalary` ihrer Basisklasse `Manager` benutzt. Das neue Hauptprogramm sieht dann recht kurz aus:

```
1   //                      a4_3.cpp
2   #include <iostream>
3
4   #include "employ2.hpp"
5   #include "bestech.hpp"          // NEW
6   #include "liste.hpp"            // !! employ2.hpp wird 2-mal eingebunden!!
7   using namespace std;
8
9   int main()
10  {
11   WageEmployee hilf("Heiko");
12   hilf.setWage(20); hilf.setHours(7.8);
13
14   SalesPerson emp("Gundolf Haase");
15   emp.setWage(hilf.getWage());  emp.setHours(hilf.getHours());
16   emp.setComission(0.05); emp.setSales(10000.0);
17
18   Manager man("Max Planck");
19   man.setSalary(1000.0);
20
21   //     Basisklassenzeiger
22   cout << endl << "           Basisklassenzeiger" << endl;
23
24   const int N=4;
25   Employee* liste[N];            // array von Pointern auf Employee
26
27    // NEW
28   BoxPromoter boxer("Larry King");
29   boxer.setBestechung(19300.0);
30
```

```
31    liste[0] = &boxer;
32    liste[1] = &hilf;
33    liste[2] = &emp;
34    liste[3] = &man;
35
36    PrintListe(N, liste);
37
38    return 0;
39   }
```

In Zeile 31 setzen wir den Basisklassenpointer `liste[0]` auf die Instanz der neuen Klasse und in Zeile 36 wird die bereits als Objektfile vorliegende Funktion `PrintListe` angewandt. In den Zeilen 4-6 wird das Headerfile *employ2.hpp* einmal direkt und zweimal indirekt über *bestech.hpp* und *kiste.hpp* eingebunden. Solange nur Deklarationen in *employ2.hpp* stehen bleibt dies folgenlos. Da wir aber z.B., den Destruktor der Klasse `Employee` auch gleich im Headerfile definiert (=implementiert) haben, würde diese Methode dreimal definiert, was eine Fehlermeldung nach sich zieht. Dies hätte zur Folge, daß beim Compilieren von *a4_3.cpp* die Klasse `Employee` dreimal deklariert wird. Eine elegante Lösung des Problems besteht darin, den Quelltext des Headerfiles nur dann einzubinden, wenn eine .

```
1   //
2   #ifndef FILE_EMPLOY2
3   #define FILE_EMPLOY2
4   ...                              // Quelltext von employ2.hpp
5   #endif
```

Dieses Vorgehen garantiert, daß die Deklarationen und Definitionen von *employ2.hpp* genau einmal pro Compiliervorgang eingebunden werden.

Wir compilieren nun die beiden neuen Files und linken sie mit den bereits vorhandenen Objektfiles (Bibliotheken, bei größeren Projekten) zusammen.

```
g++ -o a4_3 a4_3.cpp bestech.cpp liste.o employ2.o
```

Die dynamische Bindung ermöglicht es, Bibliotheken mit Klassen und Methoden zu erstellen, die von anderen Programmierern erweitert werden können. Sie müssen dafür lediglich die Include-Dateien (**.h*, **.hpp*) und den compilierten Code (**.o*, *lib*.a*) bereitstellen, welche die Hiererachie der Klassen und die Methoden enthalten. Andere Programmierer können damit von Ihren Klassen eigene Klassen ableiten und die von Ihnen deklarierten virtuellen Methoden neu definieren. Methoden, die ursprünglich nur Ihre Klassen verwendet haben, arbeiten dann auch mit den neuen Klassen [Corp., 1993, p.164].

1.4.4 Nochmals zu Copy-Konstruktor und Zuweisungsoperator

Sie werden bislang die Copy-Konstruktoren und Zuweisungsoperatoren für die Klassen unserer Hierarchy vermißt haben - oder auch nicht. Das Fehlen derselben ist zum einen der Übersichtlichkeit geschuldet und zum anderen, daß für unsere, sehr einfachen, Klassen mit einfachen Datentypen die vom Compiler automatisch eingefügten Standardmethoden ausreichend sind.

Sobald wir aber kompliziertere Datenstrukturen haben, siehe §1.1, sind diese beiden Methoden **unbedingt notwendig** [Meyers, 1998, §16]. Ein erster, aber *falscher* Ansatz des Zuweisungsoperators für `WageEmployee` sähe so aus:

```
1   //                     f a l s c h e r  Zuweisungsoperator
2   WageEmployee & WageEmployee :: operator=(const WageEmployee & orig)
3   {
4    if ( this != &orig )
5     {
6       age   = orig.wage;
```

```
7       hours = orig.wage;
8     }
9    return *this
10  }
```

Der Fehler besteht darin, daß `Employee::name` nicht mit den Daten des Originals belegt wurde. Eine direkte Zuweisung ist nicht möglich, da `name` als `private` deklariert wurde und somit nur innerhalb der Basisklasse darauf zugegriffen werden kann. Die einzig saubere Lösung ist der folgende Code.

```
1   //                      r i c h t i g e r  Zuweisungsoperator
2   WageEmployee & WageEmployee :: operator=(const WageEmployee & orig)
3   {
4    if ( this != &orig )
5     {
6       Employee::operator=(orig);
7       age   = orig.wage;
8       hours = orig.wage;
9     }
10   return *this
11  }
```

Die neue Anweisung in Zeile 6 ruft den entsprechenden Zuweisungsoperator der Basisklasse auf, in diesem Falle die Methode `this->Employee::operator=`. Zwar erwartet diese Methode ein Argument vom Typ `Employee` aber da `WageEmployee` von dieser Klasse abgeleitet ist, wird eine implizite Typkonvertierung durchgeführt (die Basisklasse holt sich was sie braucht aus der abgeleiteten Klasse).

Beim Copykonstruktor muß man die entsprechenden Basisklasseninitialisierer aufrufen. In unserem Falle wäre dies.

```
1   //                      richtiger Copy-Konstruktor
2   WageEmployee & WageEmployee :: WageEmployee(const WageEmployee & orig)
3           : Employee(orig), wage(orig.wage), hours(orig.hours)
4   {}
```

1.5 Mehrfachvererbung und virtuelle Basisklassen

1.5.1 Mehrfachvererbung

Bislang haben wir nur einfache Vererbungen betrachtet. Es ist aber auch möglich, eine neue Klasse von mehreren Basisklassen abzuleiten. So ist zum Beispiel ein Amphibienfahrzeug sowohl ein Auto als auch ein Schiff und wir könnten eine entsprechende Klasse ableiten. Diese Mehrfachvererbung sieht dann wie folgt aus:

```
1   class Amphibienfahrzeug : public Schiff, public Auto
2    {
3      ...
4    };
```

Die Basisklassen können wiederum auch als `protected` oder `private` eingebunden werden. Mehrfachnennungen von Basisklassen sind nicht zulässig, jedoch gilt dies nur für direkte Vererbung. Wurden vorher sowohl `Auto` als auch `Schiff` von einer gemeinsamen Basisklasse `Transportmittel`

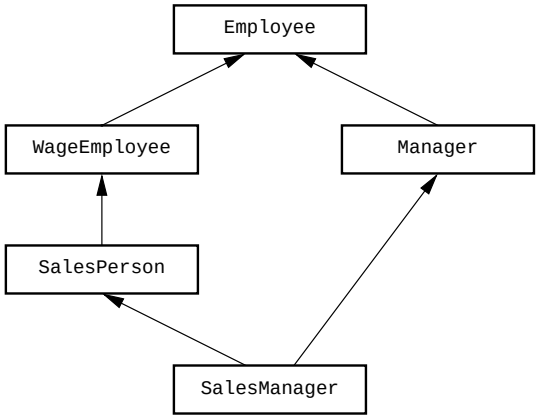
abgeleitet, dann ist diese Klasse eine indirekte Basisklasse von `Amphibienfahrzeug`. Indirekte Basisklassen dürfen mehrfach eingebunden werden.

Dies erlaubt es uns, aus unseren Beschäftigtenklassen eine neue Klasse abzuleiten : A5/employ3.hpp

```
1   //                      employ3.hpp
2   class SalesManager : public SalesPerson, public Manager
3   {
4     public:
5      SalesManager();
6      SalesManager(const char *nm);                // Konstructor
7     // void setSalary(const float salary);       //  <-- Manager
8     // void setComission(const float comm);      //  <-- SalesPerson
9     // void setSales(const float sales);         //  <-- SalesPerson
10     float computePay();
11     void payment();
12  };
```

Die neuen Methoden werden zum File *employ2.cpp* hinzuprogrammiert und das neue File in *employ3.cpp* umbenannt . A5/employ3.cpp

Der `SalesManager` **IS-A** `Manager` und **IS-A** `SalesPerson` und hat darüber hinausgehend keine weiteren Eigenschaften (Member) - obwohl er welche haben dürfte. Er nutzt sämtliche Member und Methoden der über ihm in der Hierarchie stehenden Klassen (da alle Ableitungen via `public` erfolgten).



1.5.2 Probleme bei Mehrfachvererbung

Die restlichen Teile von *employ3.cpp* und *employ3.hpp* übernehmen wir aus *employ2.cpp* und *employ2.hpp*. Das Compilieren
`g++ -c employ3.cpp`
endet in der Fehlermeldung

```
employ3.cpp: In member function ‘virtual void SalesManager::payment()’:
employ3.cpp:145: request for member ‘info’ is ambiguous in multiple inheritance
   lattice
employ3.cpp:24: candidates are: void Employee::info()
employ3.cpp:24:                 void Employee::info()
```

Ein Blick den Quelltext der Klasse `SalesManager`

```
133   //                      employ3.cpp   - inkorrekt
134   SalesManager::SalesManager() : SalesPerson(), Manager()
135   {}
136
137   SalesManager::SalesManager(const char *nm) : SalesPerson(nm), Manager(nm)
138   {}
139
140   float SalesManager::computePay()
141   { return SalesPerson::computePay() + Manager::computePay(); }
142
143   void SalesManager::payment()
144    {
```

```
145     info();
146     cout << "\n Gehalt : " << computePay() << " DM pro Woche\n";
147   }
```

zeigt in Zeile 145 die Probleme aufwerfende Funktion `info()`, welche nur einmal deklariert und definiert ist. Wo also ist das Problem??

Grund dafür ist, daß `SalesManager` sowohl von `SalesPerson` als auch von `Manager` abgeleitet wurde. Der Compiler weiß nun nicht, über welche der beiden Klassen die Methode `info()` aufgerufen werden soll. Hintergrund dessen ist, daß die Methode `info()` ihrerseits die virtuelle Methode `payment` aufruft und dann macht es schon einen Unterschied, ob nun `SalesPerson :: payment()` oder `Manager :: payment()` verwendet werden soll. Zur Lösung des Problems gibt es zwei Möglichkeiten:

a) Explizite Angabe der Methode mit Scope, z.B.: `Manager:payment()`, oder

b) Deklaration der Klasse `Employee` als *virtuelle Klasse* wenn andere Klassen von ihr abgeleitet werden.

Variante a) ist eine Lösung bei einfachen Anwendungen von Klassen, produziert jedoch beim Übersetzen des Hauptprogrammes

A5/a5_1.cpp

```
48  //                      a5_1.cpp
49   ...
50   const int N=4;
51   Employee* liste[N];
52   ...
53   SalesManager mana("Larry King");
54   mana.setWage(200); mana.setHours(20.4);
55   mana.setComission(0.05); mana.setSales(10000.0);
56   mana.setSalary(1000.0);
57   liste[0] = &mana;
58   ...
```

leider eine Fehlermeldung für Zeile 57.

```
a5_1.cpp: In function ‘int main()’:
a5_1.cpp:57: ‘Employee’ is an ambiguous base of ‘SalesManager’
```

Diese Fehlermeldung moniert zunächst, daß da etwas mehrdeutig (engl.: ambiguous) ist. Die Problematik ist, daß `Employee` zweimal als indirekte Basisklasse von `SalesManager` auftaucht. Der Compiler kann nicht wissen, über welchen der beiden Vererbungspfade der Basisklassenpointer `liste[0]` auf die Instanz `mana` der Klasse `SalesManager` verweisen soll. Da hilft nur noch Variante b).

1.5.3 Virtuelle Basisklassen

Wie im vorigen Abschnitt gezeigt, ist es nicht wünschenswert, daß eine indirekte Basisklasse mehrfach in einer anderen Klasse enthalten ist. Es wäre sinnvoll, wenn dies nur einmal geschehen würde. Dazu gibt es in C++ *virtuelle Basisklassen* [Kirch-Prinz and Prinz, 2002, p.324-329].

Unsere Basisklasse `Employee` müßte also als virtuelle Basisklasse deklariert werden. Dies geschieht durch Hinzufügen des Schlüsselwortes `virtual` in die Liste der Basisklassen einer Klassendeklaration.

```
1   //                      Aenderungen in employ3.hpp
2   class WageEmployee : public virtual Employee            // N E W
3   {
4    ...
5   };
```

```
6   ...
7   class Manager : public virtual Employee                 // N E W
8   {
9    ...
10  };
11  ...
```

Damit läßt sich erstmal der gesamte Code compilieren und linken.
`g++ -o a5_1 a5_1.cpp employ3.c liste.cpp`

Betrachtet man die Ausgabe des Testlaufes, so fällt auf, daß kein Name mehr ausgegeben wird für die Instanzen, welche von den Klassen `SalesPerson` und `SalesManager` erzeugt wurden. Dies ist (leider) kein Zufall.

```
1   //                              employ3.cpp - noch inkorrekt
2   ...
3   Employee::Employee(const char *inname)
4           : name(inname)
5   {}
6   ...
7   WageEmployee::WageEmployee()
8           : Employee(), wage(0.0), hours(0.0)
9   {}
10  ...
11  SalesPerson::SalesPerson(const char *nm)
12          : WageEmployee(nm), comission(0.0), SalesMade(0.0)
13  {}
14  ...
15  Manager::Manager(const char *nm)
16          : Employee(nm), weeklySalary(0.0)
17  {}
18  ...
19  SalesManager::SalesManager(const char *nm)
20          : SalesPerson(nm), Manager(nm)
21  {}
22  ...
```

Betrachten wir die (Parameter-)Konstruktoren unserer Hierarchy in *employ3.cpp*, so sieht formal alles korrekt aus.

ABER Basisinitialisierer (Aufruf des entprechenden Konstruktors) für virtuelle Basisklassen, die im Konstruktor einer darüberliegenden Klasse stehen, werden ignoriert [Kirch-Prinz and Prinz, 2002, p.327].

Normalerweise müßte in Zeile 20 der Basisklasseninitialisierer `Manager(nm)` wiederum (u.a.) seinen Basisklasseninitialisierer `Employee(nm)` in Zeile 16 aufrufen. Da letztere Basisklasse aber virtuell ist, unterbleibt dieser Aufruf. Als Ergebnis dessen ist der String `Employee::name` nach dem Konstruktoraufruf nicht initialisiert!!

Hier hilft leider nur der explizite Aufruf des Basisklasseninitialisierers in den entsprechenden Konstruktoren, d.h., wir erhalten :

A5/employ3.cpp

```
1   //                              employ3.cpp - korrekt
2   ...
3   Employee::Employee(const char *inname)
4           : name(inname)
5   {}
6   ...
7   WageEmployee::WageEmployee()
8           : Employee(), wage(0.0), hours(0.0)
9   {}
```

```
10  ...
11  SalesPerson::SalesPerson(const char *nm)
12          : Employee(nm), WageEmployee(nm), comission(0.0), SalesMade(0.0)        // NEW
13  {}
14  ...
15  Manager::Manager(const char *nm)
16          : Employee(nm), weeklySalary(0.0)
17  {}
18  ...
19  SalesManager::SalesManager(const char *nm)
20          : Employee(nm), SalesPerson(nm), Manager(nm)                            // NEW
21  {}
22  ...
```

Nunmehr liefert compilieren und linken ein korrektes Programm.
`g++ -o a5_1 a5_1.cpp employ3.c liste.cpp`

1.6 Exception-Handling an Hand von `new`

C++ erlaubt dem Programmierer auf Ausnahmefälle des Programmes, wie zuwenig Speicher, korruptes Einlesefile, Indexüberschreitung, im Definitionsbereich nicht enthaltene Argumente (`log(-1.5)`), usw. mit eigenen Methoden zu reagieren. Wir betrachten in diesem Abschnitt eine Vektorklasse `MyVector`, in welcher wir 3 Möglichkeiten des Exception-Handlings demonstrieren. Danach leiten wir eine Klasse `MySaveVector`, in welcher zusätzlich (**HAS-A**) eine Indexüberprüfung stattfindet.

Die Deklaration von `MyVector` ist

A6/myvector.hpp

```
1   //                              myvector.hpp
2   class MyVector
3   {
4    public:
5     MyVector(const int n = 0);            // Standard-, Parameterkonstruktor
6     MyVector(const MyVector& orig);       // Copykonstruktor
7     virtual ~MyVector();                            // Destruktor
8     MyVector& operator=(const MyVector& orig);    // Zuweisungsoperator
9
10    friend std::ostream & operator<<(std::ostream & s, const MyVector& orig);
11
12    const int& Length()                   const { return length_; };
13          double& operator[](const int i)       { return val_[i]; };
14    const double& operator[](const int i) const { return val_[i]; };
15    void  Put(const int i, const double a)      { val_[i] = a; };
16
17   private:
18    int      length_;
19    double * val_;
20  };
```

Der Indexoperator `operator[]` ist sowohl als `const` als auch als nicht-`const` deklariert um sowohl konstante als auch nicht-konstante Instanzen vom Typ `MyVector` zu unterstützen [Meyers, 1998, p.110]. Das Hauptprogramm *A6/main.cpp* enthält zu Beginn folgende Teile:

```
1   //                      A6/main.cpp
2   #include <iostream>
3   #include "myvector.hpp"
4   using namespace std;
5
6   int main()
```

```
 7  {
 8   int n,i;
 9   cout << " n = "; cin >> n;
10
11   MyVector aa(n);
12
13   for (i=0; i<n; i++) { aa.Put(i,(i+1)*3.14); }
14
15   aa[3] = 4.5;
16   const MyVector bb( aa );
17   out << bb[13] << endl;
18
19   return 0;
20  }
```

Darin wird in Zeile 9 der Parameterkonstruktor, in Zeile 16 der Copy-Konstruktor gerufen, sowie in Zeile 15 der Index(`[]`)-operator und in Teile 17 der konstante Indexoperator gerufen (siehe Zeilen 13 bzw. 14 im Headerfile)

1.6.1 Klassische (C-) Fehlerbehandlung bei Speichermangel

Wir betrachten die Implementation des Parameterkonstruktors (=Standardkonstruktor wegen des optionalen Parameters) in anfänglichen File *A6/mayvector.cpp*:

```
1  //                      Ausgangscode
2  MyVector :: MyVector(const int n )              // Standard-, Parameterkonstruktor
3          : length_( n>0 ? n : 0), val_(0)
4  {
5    if ( length_>0 )
6     {
7       val_    = new double [length_];
8     }
9  }
```

Kompilieren und Linken
`g++ -o main main.cpp myvector.cpp`
erzeugt ein Programm, welches die Feldlänge `n` abfragt und danach den Speicher allokiert. Auf meinem Laptop mit 256MB Speicher und der Eingabe von $n = 90.000.000$ beendet das Programm die Abarbeitung mit einem Fehlercode, da nicht genügend Speicher vorhanden ist. Dies ist ärgerlich, denn zum einen weiß man nicht, wo der Fehler auftritt und zum anderen könnte man mit einer eigenen Fehlerbehandlung evtl. noch etwas retten.

Ein Abtesten des Pointers `val_` auf den Nullpointer `0`, wie in C, führt erstmal zu keinem Erfolg, da C++ schon bei der fehlgeschlagenen Speicherallokierung abbricht. Jedoch kann man die Methode `new` mit dem Parameter `(nothrow)` benutzen, um ein Nullsetzen des Pointers bei fehlgeschlagener Speicherallokierung zu erzwingen.

```
1  //                      Code mit nothrow
2  #include <iostream>
3  #include <new>                  //      NEW: new(nothrow)
4  #include <cassert>              //      NEW: assert()
5  #include "myvector.hpp"
6
7  MyVector :: MyVector(const int n )
8          : length_( n>0 ? n : 0), val_(0)
9  {
10   if ( length_>0 )
11    {
12      val_    = new(nothrow) double [length_];          // NEW
13      assert(val_ != 0);                                // NEW
```

```
14      }
15  }
```

Das Headerfile *new* in Zeile 3 stellt die Version von `new` in Zeile 12 zur Verfügung, welche keine Exeption auswirft (`nothrow`). Das Headerfile *cassert* ist das C++-File des C-Headers *assert.h* und stellt die gleichnamige Funktion in Zeile 13 zur Verfügung, welche ausgibt, in welcher Quelltextzeile `val_` gleich dem Nullpointer ist. Der Test und der Abbruch in `assert` lassen sich (falls eh nichts passieren kann) mit der Compileroption `-DNDEBUG` eliminieren. Allerdings landet das Programm dann im Falle von zu wenig Speicher im Nirvana, d.h., der beliebte Fehler `Segmentation fault` wird ausgegeben.

Mein Laptop bricht das Programm `main` kontrolliert ab bei $n = 90.000.000$. Bei $n = 50.000.000$ läuft alles erstmal normal bis das Programm plötzlich abbricht. Nach einigem Suchen stellt man fest, daß die Speicheranforderung im Copy-Konstruktor schuld ist. Zwar könnte man dies wieder mit `new(nothrow)` und `assert` abfangen, jedoch gibt es zwei bessere Lösungen in C++.

1.6.2 Ein eigener Exception-Handler für `new`

Der normale Exception-Handler für `new` beendet das Programm ohne irgendeine Meldung oder Aktion. Es kann aber notwendig sein, daß das Programm noch einige Aktionen ausführen soll bevor es terminiert wird, z.B. eine email an den Programmierer schicken. Eine einfache Möglichkeit dies zu realisieren ist das Setzen des new-Handlers auf eine eigene Funktion.

```
1   //                      A6/main.cpp  mit new-handler
2   #include <iostream>
3   #include "myvector.hpp"
4   using namespace std;
5
6   void my_new_handler(void)       // Eigener exception-Handler
7   {
8     cerr << " Nicht genuegend Speicher vorhanden" << endl;
9     cerr << " Beendigung des Programmes" << endl;
10    cout << "Bitte teilen Sie dem Programmierer Ihre Meinung mit:" << endl;
11    system("mail gundolf.haase@uni-graz.at");
12    exit (-1);
13  }
14
15  int main()
16  {
17   int n,i;
18   cout << " n = "; cin >> n;
19
20   set_new_handler( my_new_handler );     // Meinen Handler benutzen
21   MyVector aa(n);
22
23   for (i=0; i<n; i++) { aa.Put(i,(i+1)*3.14); }
24
25   aa[3] = 4.5;
26   const MyVector bb( aa );
27   out << bb[13] << endl;
28
29   return 0;
30  }
```

Der eigene Exception-Handler für `new` wird in den Zeilen 6-13 definiert (und deklariert). In Zeile 30 wird dann dieser neue Handler als Exception-Handler für `new` gesetzt. Er wird dann jedesmal aufgerufen, wenn bei einer `new`-Anweisung etwas schief geht (aber nicht bei `new(nothrow)` !). Dieser neue Exception-Handler fängt nun den Speichermangel im Copy-Konstruktor ab (Laptop: $n = 50.000.000$).

1.6.3 Der try-und-catch Mechanismus mit `new`

Die in den beiden vorigen Abschnitten behandelten Ausnahmebehandlungen für die `new`-Funktion haben den Nachteil, daß sie zwingend einen Programmabbruch nach sich ziehen. Will man den Speichermangel abfangen, indem man z.B., lieber eine langsamere aber speicherschonendere mathematische Methode benutzt, dann bietet sich der folgende Mechanismus an.

```
1   //                      A6/main.cpp  mit try-und-catch
2   #include <iostream>
3   #include "myvector.hpp"
4   using namespace std;
5
6   int main()
7   {
8    int n,i;
9    cout << " n = "; cin >> n;
10
11   try{
12     MyVector aa(n);
13
14     for (i=0; i<n; i++) { aa.Put(i,(i+1)*3.14); }
15
16     aa[3] = 4.5;
17     const MyVector bb( aa );
18     out << bb[13] << endl;
19   }
20   catch (std::bad_alloc)
21   {
22     cout << "Hey, kein Speicher mehr frei." << endl;
23   }
24   cout << "Catch beendet" << endl;
25
26   return 0;
27  }
```

Falls im `try`-Block in den Zeilen 11-19 eine Exception auftritt, dann wird nachgeschaut, ob die entsprechende Exception in den nachfolgend `catch`-Anweisungen enthalten ist. Wenn ja, so wird der entsprechende `catch`-Block als Exception-Handler ausgeführt. Die Exception `std::bad_alloc` in Zeile 20 ist in der Standardbibliothek definiert [Yang, 2001, p.329] und steht für Fehler bei der Speicherallokierung (also bei `new`) sodaß Speichermangel im `catch`-Block der Zeilen 21-23 abgefangen wird. Danach fährt das Programm mit der nächsten Anweisung nach dem `catch`-Block fort.

Die vordefinierten Exception-Handler der Standardbibliothek sind im Headerfile *stdexcept* enthalten.

1.6.4 Eigene Exceptions schreiben

Wir wollen von unserer Klasse `MyVector` eine Klasse `MySaveVector` ableiten, welche bei Anwendung der Indexoperatoren den übergebenen Index auf Gültigkeit überprüft und im Fehlerfalle eine Exception auswirft. Dazu erweitern wir das File *myvector.hpp*

A6/myvector.hpp

```
1   //                      A6/myvector.hpp
2   ...
3   #include "myexceptions.hpp"            // Exeption handlers and classes
4   ...
5   class MySaveVector:  public MyVector
6   {
7    public:
8     MySaveVector(const int n = 0)        // Standard-, Parameterkonstruktor
```

```
9           : MyVector(n) {};
10
11          double& operator[](const int i)       { IndexCheck(i); return val_[i]; };
12    const double& operator[](const int i) const { IndexCheck(i); return val_[i]; };
13
14   private:
15    void IndexCheck(const int &i) const
16     { if ( i<0 || i>=Length() )
17          throw OutOfRange(i,Length()-1);         // Indexfehler
18     };
19  };
```

Der Indextest wird in der `private`-Methode `IndexCheck` erledigt, welche in Fehlerfall eine Exception `OutOfRange` in Zeile 17 auswirft (engl.: throw). Unsere neue Exception-Klasse besteht nur aus dem Konstruktor und den Zugriffsmethoden auf die Member .

A6/myexception.hpp

```
1   //                      A6/myexception.hpp
2   class OutOfRange
3   {
4    public:
5     OutOfRange(const int &i, const int &l);       // Konstruktor
6
7     const int& Index()        const { return index; };
8     const int& Interval_End() const { return end_interval; };
9
10   private:
11    int index, end_interval;
12  };
```

Der Konstruktor muß nur die Member initialisieren und gibt zur Kontrolle eine Fehlermeldung aus (was eigentlich bei uns nicht nötig sein wird) .

A6/myexception.cpp

```
1   //                      A6/myexception.cpp
2   #include <iostream>
3   #include "myexceptions.hpp"
4
5   OutOfRange :: OutOfRange(const int &i, const int &l)    // Konstruktor
6           : index(i), end_interval(l)
7   {
8     cerr << "Index  " << i << "  is not in interval  [ 0, " << l << " ]" << endl;
9     return;
10  }
```

Nunmehr können wir im Hauptprogramm unsere neue Exception-Klasse benutzen.

```
1   //                              main.cpp mit zwei Exceptions
2   #include <iostream>
3   #include "myvector.hpp"
4   #include "myexceptions.hpp"
5
6   using namespace std;
7
8   int main()
9   {
10   int n,i;
11
12   cout << " n = "; cin >> n;
13
14   try{
15     MySaveVector aa(n);
```

```
16
17      for (i=0; i<n; i++) { aa.Put(i,(i+1)*3.14); }
18      aa[3] = 4.5;
19      cout << aa[3] << endl;
20
21      const MySaveVector bb( aa );
22
23      cout << bb[13] << endl;
24    }
25    catch (std::bad_alloc)
26    {
27     cout << "Hey, kein Speicher" << endl;
28     system("mail gundolf.haase@uni-graz.at");
29    }
30    catch (OutOfRange &err)
31    {
32      cerr << "Indexerror: " << err.Index() << "   " << err.Interval_End() << endl;
33    }
34    catch ( ... )
35    {
36     cerr << "Alle restlichen Exception" << endl;
37    }
38
39    cout << "nach catch " << endl;
40    return 0;
41   }
```

Der Code wird mit
`g++ main.cpp myvector.cpp myexceptions.cpp`
erzeugt. In Zeile 34 werden alle anderen Exceptions abgefangen.

Das Verhalten bei Speicherüberlauf ist wie vorher. Jedoch wird jetzt noch zusätzlich die `OutOfRange`-Exception abgefangen. Unsere Implementierung stützt sich auf[Yang, 2001, p.329]. Weitere Hinweise zum Anlegen eigener Exception-Klasses sind in [Kirch-Prinz and Prinz, 2002, §16] zu finden.

1.7 Funktions-Templates

1.7.1 Mehrfache Implementierungen

In irgendeinem Stadium der Implementierung stellt man plötzlich fest, daß die gleichen Funktionen für verschiedene Datentypen gebraucht werden. In nachfolgendem, einfachen Beispiel sind dies das Maximium zweier Zahlen und das Maximum eines Feldes .

A7/fkt.hpp

```
1   //                      A7/fkt.hpp
2   int    max (const int    &a, const int    &b);
3   float  max (const float  &a, const float  &b);
4   double max (const double &a, const double &b);
5
6   int    max_elem(const int n, const int    x[]);
7   float  max_elem(const int n, const float  x[]);
8   double max_elem(const int n, const double x[]);
```

Die Implementierungen beider Funktionsgruppen unterscheiden sich nur im jeweiligen Datentyp, die Struktur der Implementierung ist ansonsten komplett identisch .

A7/fkt.cpp

```
1   //                      A7/fkt.cpp
2   int    max (const int    &a, const int    &b)
3    {
```

```
4     return  a > b  ?  a  : b;
5    }
6
7    float    max (const float    &a, const float    &b)
8    {
9     return  a > b  ?  a  : b;
10   }
11
12   double    max (const double    &a, const double    &b)
13   {
14    return  a > b  ?  a  : b;
15   }
16
17   float  max_elem(const int n, const float  x[])
18   {
19    int    i;
20    float  vmax;
21
22    assert(n>0);          // --> richtiges Exeption-handling muss rein
23
24    vmax = x[0];
25    for (i=1; i<n; i++)
26     {
27       vmax = max(vmax,x[i]);
28     }
29
30    return vmax;
31   }
32  //                      usw.
33  ...
```

Neben dem unguten Gefühl, sich ungeschickt anzustellen, müssen algorithmische Verbesserungen in jeder Funktion einer Funktionsgruppe implementiert werden. Die ist wiederum fehleranfällig und der Gesamtcode schwerer zu warten.

1.7.2 Implementierung eines Funktions-Templates

C++ bietet die Möglichkeit, mit Hilfe von *Templates* (engl.: Schablonen) eine parametrisierte Familie verwandter Funktionen zu definieren. Ein Funktions-Template legt die Anweisungen einer Funktion fest, wobei statt eines konkreten Typs ein Parameter verwendet wird [Kirch-Prinz and Prinz, 2002, p.365]. Vorteile dieser Templates sind:

- Ein Funktionen-Template muß nur einmal kodiert werden.
- Einzelne Funktionen zu einem konkreten Parameter werden anhand des Templates automatisch erzeugt.
- Typunabhängige Bestandteile (der Algorithmus) können ausgetestet werden und funktionieren dann für die anderen Paramatertypen.
- Es besteht immer noch die Möglichkiet, für bestimmte Typen spezielle Lösungen anzugeben.

Einem Funktions-Template wird das Präfix

`template <class T>`

vorangestellt. Der Parameter `T` ist hierbei der Typname welcher in der nachfolgenden Definition benutzt wird. Dabei schließt das Schlüsselwort `class` auch einfache Datentypen wie `int` oder `double` ein.

Unsere Funktionsfamilien werden nunmehr mit Templates so definiert . A7/tfkt.cpp

---

```
1  //                              tfkt.cpp
```

```
2   #include <cassert>
3   //#include "tfkt.hpp"
4
5   export template <class T>
6   T  max (T  &a, T  &b)
7    {
8     return  a > b  ?  a  : b;
9    }
10
11  export template <class T>
12  T  max_elem(const int n, const T  x[])
13   {
14    int i;
15    T vmax;
16
17    assert(n>0);          // --> richtiges Exeption-handling murein
18
19    vmax = x[0];
20    for (i=1; i<n; i++)
21     {
22       vmax = max(vmax,x[i]);
23     }
24
25    return vmax;
26   }
```

Die Deklaration ist dann .

A7/tfkt.hpp

```
1   //                      tfkt.hpp
2   #ifndef FILE_TFKT
3   #define FILE_TFKT
4
5   template <class T>
6   T  max (T  &a, T  &b);
7
8   template <class T>
9   T  max_elem(const int n, const T  x[]);
10
11  //              Irgendwann wird "export" untersttzt,
12  //              dann ist nachfolgendes inkludieren nicht mehr notwendig
13  #include "tfkt.cpp"
14  #endif
```

Wie Sie bemerkt haben werden, enthalten die beiden Files *tfkt.cpp* und *tfkt.hpp* in Zeilen 5, 11 bzw. Zeile 13 einige Merkwürdigkeiten. Mehr dazu in §1.7.3.

Die Anwendung unserer Funktionstemplates im Hauptprogramm ist relativ einfach.

A7/main.cpp

```
1   //                      A7/main.cpp
2   #include <cstdlib>                      //      rand()
3   #include <iostream>
4   #include "tfkt.hpp"                     //      mit Templates
5   using namespace std;
6
7   int main()
8   {
9    const int N = 10;
10   int    i;
11   int    ia[N], im;
12   float  fa[N], fm;
13   double da[N], dm;
14  //                      Felder Initialisieren
```

```
15    for (i=0; i<N; i++)
16     { ia[i] =         i;
17       fa[i] = (float) rand();
18       da[i] = (double)rand();
19     }
20
21    im = max_elem( N, ia);          // int
22    cout << "   int-Feld(max):  " << im << endl;
23
24    fm = max_elem( N, fa);          // float
25    cout << " float-Feld(max):  " << fm << endl;
26
27    dm = max_elem( N, da);          // double
28    cout << "double-Feld(max):  " << dm << endl;
29
30    return 0;
31   }
```

Ein Funktions-Template kann auch mit mehreren Typparametern definiert werden.

```
1   template<class A, class B>
2   B& func(const int n, const A &mm, B v[])
3   {
4   ...
5   }
```

1.7.3 Das Schlüsselwort `export`

Zeile 13 im File *tfkt.hpp* inkludiert ungewöhnlicherweise die Definitionen unserer Funktions-Templates. Kommentiert man diese Zeilen aus, so kann der `g++`-Compiler (und ähnlich der `icc`) zwar compileren, jedoch nicht linken.

```
LINUX> g++ -c tfkt.cpp
LINUX> g++ -c main.cpp
LINUX> g++ -o main main.o tfkt.o

main.o(.text+0x78): In function ‘main’:
: undefined reference to ‘int max_elem<int>(int, int const*)’
main.o(.text+0xc6): In function ‘main’:
: undefined reference to ‘float max_elem<float>(int, float const*)’
main.o(.text+0x11a): In function ‘main’:
: undefined reference to ‘double max_elem<double>(int, double const*)’
main.o(.text+0x17a): In function ‘main’:
: undefined reference to ‘float max<float>(float&, float&)’
```

Der Grund dafür ist, daß der Compiler den Code nur dann compilieren kann, wenn er ihn vollständig sieht. Beim Compilieren von *tfkt.cpp* konnte er noch nicht wissen, für welchen Typ die Funktionen definiert werden sollen und beim Compilieren von *main.cpp* ist die Implementierung der Funktions-Templates unbekannt. Es wurde also überhaupt keine Funktionen `max` oder `max_elem` compiliert. Aus diesem Grunde ist die Includedeklaration in Zeile 13 des Files *tfkt.hpp* notwendig.

Wünschenswert wäre natürlich auch bei Templates eine saubere Trennung zwischen Header- und Quelltextfile, bei dem ein Funktions-Template trotzdem global verfügbar ist. Laut C++-Standard wird genau dies durch das Schlüsselwort `export` bewirkt [Yang, 2001, p.248f]. Soweit die Theorie - in der Praxis erhält man allerdings die Compilerwarnung
`tfkt.cpp:11: warning: keyword ‘export’ not implemented, and will be ignored`
d.h., diese wünschenswerte Funktionalität von C++ ist derzeit noch nicht verfügbar (Mai 2004,

nach meinem Wissen). Da diese Funktionalität aber früher oder später verfügbar sein wird, sollte man erstmal die Quelltextfiles in die Headerfiles inkludieren und entsprechende Funktions-Templates jetzt schon mit `export` im Quelltextfile kennzeichnen (Zeilen 5 und 11 im Code auf Seite 27). Falls das `export`- Schlüsselwort eines Tages korrekt unterstützt wird, dann braucht man nur im Headerfile das Inkludieren des Quelltextfiles auszukommentieren [Schmaranz, 2002, p.447f].

1.7.4 Implizite und explizite Templateargumente

Im Hauptprogramm auf Seite 28 wurde das richtige Templateargument `T` , und damit die konkrete Funktion, anhand der Funktionsparameter bestimmt. Falls dies nicht eindeutig ist, bzw. das Templateargument nicht in der Parameterliste der Funktion vorkommt, dann muß das Templateargument beim Funktionsaufruf explizit angegeben werden. Das folgene Codefragment demonstriert implizite und explizite Templateargumente in den Zeilen 11 und 12.

```
1   //                      A7/main.cpp
2   #include "tfkt.hpp"                    //      mit Templates
3   ...
4   int main()
5   {
6    const int N = 10;
7    int    i;
8    float  a=5.455, b=-3.344;
9    double da[N], dm;
10  ...
11   dm = max_elem( N, da);         // double, implicit instantiation
12   fm = max<float>(a,b);          // float,  explicit instantiation
13  ...
14  }
```

Weitere Hinweise zu Typanpassungen bei Funktions-Templates sind in [Kirch-Prinz and Prinz, 2002, p.371, p.377] zu finden.

1.7.5 Spezialisierung

So schön unsere allgemeinen Funktions-Templates sind, sie sind nicht für alle denkbaren Typen einsetzbar:

- Das Funktions-Template enthält Anweisungen, die für bestimmte Typen nicht ausgeführt werden können.
- Die allgemeine Lösung, die das Template bereitstellt, liefert kein sinnvolles Ergebnis.
- Für bestimmte Typen gibt es bessere (schnellere, speicherschonendere) Lösungen.

Für solche Fälle läßt sich z.B., für unsere Funktion `max` eine Spezialisierung für C-Zeichenketten implementieren.

```
1   // template<>                  // ANSI
2   const char* max( const char* s1, const char* s2 )
3   {
4     if ( strcmp(s1,s2) > 0 )     return s1;
5     else                         return s2;
6   }
```

Die auskommentierte Zeile 1 ist notwendig, falls der Compiler den ANSI-Standard einfordert (nicht bei `g++`).

Diese Funktion überlädt, bei (genau!) passenden Parametern das entsprechende Funktions-Template von `max`. Dazu werfen wir einen Blick in das folgende Programm.

```
1   ...
2   #include <cstring>
3   #include "tfkt.hpp"                    //      mit Templates
4   int main()
5   {
6    const int N = 10, M=20;
7          int i;
8          char *a[N];
9    const char *name1 = "Richard",  *name2 = "Anton";
10   const char *max_str;
11  //                      Feldelemente allokieren und initialisieren
12   for (i=0; i<N; i++) { a[i] = new char [M]; }
13   strcpy(a[0],"Heidi");
14   ...
15   strcpy(a[9],"Lilli");
16
17    max_str = max(name1,name2);
18    cout << max_str << endl;
19
20    //max_str = max(a[5],a[9]);
21    //max_str = max<const char*>(a[5],a[9]);
22    max_str = max((const char*)a[5], static_cast<const char*>( a[9] ));
23    cout << max_str << endl;
24
25    max_str = max_elem<const char*>(N, a);
26    cout << max_str << endl;
27
28    for (i=N-1; i>=0; i--)
29     { delete [] a[i]; }
30   return 0;
31  }
```

In Zeile 17 wird `max ( const char *, const char * )` aufgerufen, d.h., unsere Spezialisierung von `max` wird genutzt. Versucht man das gleiche in Zeile 20 dann wird eine Funktion `max ( char *, char * )` gesucht und diese gibt es nur in den Funktions-Templates (zumindest beim `g++`). Diese Funktion würde in diesem Falle nur die Zeiger, aber nicht die Zeichenketten vergleichen. Der Lösungsversuch in Zeile 21 wird wiederum die Funktions-Templates benutzen, da unsere Spezialisierung keinen Templateparameter hat. Erst die explizite Typkonvertierung (cast) in Zeile 22 schafft es, unsere Spezialisierung aufzurufen. Bei diesem Cast sollte man die C++-Funktion `static_cast<typ>(var)` dem älteren C-Cast des ersten Argumentes vorziehen.

Das Funktions-Template `max_elem` kann nun sogar mit unserer Spezialisierung arbeiten. Dazu ist aber eine explizite (und exakte!) Typangabe des Templateparameters notwendig. Ansonsten würde wiederum das Funktions-Template für `max` genommen.

Das Spezialisierungsbeispiel aus [Kirch-Prinz and Prinz, 2002, p.372f] konnte ich nicht in der originalen Form zum korrekten Programmlauf überreden. Vielleicht interpretieren andere Compiler die impliziten Typkonvertierungen etwas anders als der `g++`-3.2.3.

1.8 Klassen-Templates

1.8.1 Ein Klassen-Template für `MyVector`

Analog zu den Funktions-Templates bietet C++ die Möglichkeit, Klassen-Templates zu definieren. Die Klassen-Templates werden in Anhängigkeit von einem noch festzulegendem Typ (oder mehreren Typen) konstruiert. Diese Klassen-Templates werden häufig bei der Erstellung von Klassenbibliotheken eingesetzt, wir werden im Zusammenhang mit Standardklassen in §1.10 und Containern

in §1.11 auf solche Klassenbibliotheken zugreifen.

Einem Klassen-Template ist der Präfix `template<class T>` vorangestellt, dem die eigentliche Klassendefinition folgt.

```
template<class T>
class X
{
... // Definition der Klasse X<T>
};
```

Der Name des Template für Klassen ist `X<T>`. Der Parameter `T` steht wieder für einen beliebigen (auch einfachen) Datentyp. Sowohl `T` als auch `X<T>` werden in der Klassendefinition wie normale Datentypen verwendet.

Zur Demonstration leiten wir aus der Klasse `MyVector` von Seite 21 das Klassen-Template `MyVector<T>` ab indem wir den original verwandten Datentyp `double` für die Vektorelemente durch `T` und `MyVector` durch `MyVector<T>` ersetzen.

A8/tmyvector.hpp

```
1   //                                      A8/tmyvector.hpp
2   template<class T>
3   class MyVector
4   {
5     public:
6     MyVector<T>(const int n = 0);                 // Standard-, Parameterkonstruktor
7     MyVector<T>(const MyVector<T>& orig);         // Copykonstruktor
8     virtual ~MyVector<T>();                               // Destruktor
9     MyVector<T>& operator=(const MyVector<T>& orig);      // Zuweisungsoperator
10
11    //friend std::ostream & operator<< <>(std::ostream & s, const MyVector<T>& orig);
12    friend std::ostream & operator<<(std::ostream & s, const MyVector<T>& orig);
13
14    const int& Length() const { return length_; };
15
16    const T& Get(const int i)      const { return val_[i]; };
17        void Put(const int i, const T a) { val_[i] = a; };
18
19    virtual       T& operator[](const int i)       { return val_[i]; };
20    virtual const T& operator[](const int i) const { return val_[i]; };
21
22    protected:
23    int      length_;
24    T      * val_;
25  };
```

In den Methodennamen der Konstruktoren und des Destruktors könnte der Templateparameter `<T>` weggelassen werden. Aus Gründen der Konsistenz sollte man dies nicht tun. Zum Problem von `friend`-Funktionen in Klassen-Templates siehe die Zeilen 11 und 12 in obigem Programm.

Die Deklaration der Methoden erfolgt dann analog, hier sei exemplarisch der Copy-Konstruktor präsentiert .

A8/tmyvector.cpp

```
1   //                                      A8/tmyvector.cpp
2   ...
3   template<class T>
4   MyVector<T> :: MyVector(const MyVector<T>& orig)        // Copykonstruktor
5           : length_(orig.length_), val_(0)
6   {
7     if (length_>0)
8      {
9        val_ = new  T [length_];
10       for (int i=0; i<length_; i++)
11        {
```

```
12            val_[i] = orig.val_[i];
13          }
14      }
15  }
16  ...
```

Wichtig ist, daß vor jeder Methodendefinition der Präfix `template<class T>` steht. Während der `g++` sowohl `MyVector<T> :: MyVector<T>(...)` als auch `MyVector<T> :: MyVector(...)` als Beginn der Methodendefinition erlaubt, kann der `icc`(7.0) nur die letztere Variante compilieren, jedoch compiliert Version 8.0 dieses Compilers jetzt auch die erstere (und korrekte) Variante.

Das neue Klassen-Template kann mit dem Hauptprogramm übersetzt A8/main.cpp
`LINUX> g++ -Wall -o main main.cpp myexeptions.cpp`
gestestet werden. Das File *tmyvector.cpp* mit den Definitionen für die Methoden der Klassen-Templates muß wiederum im Headerfile *tmyvector.hpp* inkludiert werden, siehe §1.7.3.

```
1   //                      A8/main.cpp
2   #include "tmyvector.hpp"
3   ...
4   int main()
5   {
6    int n;
7    ...
8    cout << " n = "; cin >> n;
9
10   MyVector<float> aa(n);
11
12   for (i=0; i<n; i++)   aa.Put(i,(i+1)*3.14);
13   ...
14   const MyVector<float> bb( aa );
15
16   cout << bb[13] << endl;
17   ...
18  }
```

1.8.2 Mehrere Parameter

Analog zu den Funktions-Templates können auch Klassen-Templates mit mehreren Typparametern definiert werden.

```
template<class T1, class T2>
class X
{
... // Definition der Klasse X<T1,T2>
};
```

Desweiteren ist ein Template mit einem festgelegtem Parameter, einem Argument, möglich [Kirch-Prinz and Prinz, 2002, p.397ff],

```
template<class T, int n>
class Queue {...};
```

mit welchem `n`, z.B., die Größe eines immer wieder gebrauchten, internen temporären Feldes bezeichnet. So deklariert dann

```
Queue<double, 100> db;
```

eine Instanz der Klasse `Queue<double, 100>`, welche mit `double`-Zahlen arbeitet und intern, z.B., ein temp. Feld `double tmp[100]` verwaltet.

Das Template-Argument kann mit einem Default-Parameter definiert werden,

```
template<class T, int n=255>
class Queue {...};
```

wodurch `Queue<double> db;` ein temp. Feld der Länge 255 intern verwalten würde.

Gleitkommazahlen sind nicht als Template-Argumente erlaubt, wohl aber Referenzen auf Gleitkommazahlen.

1.8.3 Explizite Instantiierung

Oft benötigt man in einem Programm nur eine bestimmte Implementierung eines Templates, also dessen Spezialisierung. Dort wird in jedem Modul dieses Programmes immer wieder die konkrete Klasse aus dem Klassen-Template erzeugt werden, d.h., die Methoden der Klasse werden in jedem Modul übersetzt. Der Linker muß dann diese redundant erzeugten Methoden bis auf eine alle wieder entfernen. Für Klassenbibliotheken bietet sich da ein Ausweg an, welchen wir für unseren Klassen-Template demonstrieren.

Zuerst kommentieren wir die Zeile `#include "tmyvector.cpp"` im Headerfile *tmyvector.hpp* aus und schreiben ein File

A8/vecfloat.cpp

```
1  //                      A8/vecfloat.cpp
2  #include "tmyvector.hpp"
3  #include "tmyvector.cpp"
4
5  template class MyVector<float>;
```

Die letzte Zeile in obigem Code ist die explizite Instantiierung des Klasssen-Template `MyVector<class T>` zur Klasse `MyVector<float>`. Dieses Definitionsfile der neuen Klasse läßt sich übersetzten

```
LINUX>  g++ -c vecfloat.cpp
```

und zum (neu compilierten!) Hauptprogramm linken.

```
LINUX>  g++ -o main main.cpp myexceptions.cpp vecfloat.o
```

Ohne das Objektfile *vecfloat.o* würde der Linker einige Fehler vermelden.

1.8.4 Ableitung von Klassen-Templates

Von Klassen-Templates kann man wiederum abgeleitete Klassen-Templates bilden, wie nachfolgendes Beispiel zeigt .

A8/tmyvector.hpp

```
1   ...
2   template<class T>
3   class MySaveVector:  public MyVector<T>
4   {
5    public:
6    MySaveVector(const int n = 0)          // Standard-, Parameterkonstruktor
7     : MyVector<T>(n) {};
8
9          T& operator[](const int i)        { IndexCheck(i); return val_[i]; };
10   const T& operator[](const int i) const  { IndexCheck(i); return val_[i]; };
11
12   private:
13   void IndexCheck(const int &i) const
14    { if ( i<0 || i>=Length() )
15         throw OutOfRange(i,Length()-1);          // Indexfehler
16    };
17  };
```

Hätten wir Zeile 2 weggelassen und anstelle des Template-Parameters `T`, z.B., ein `double` eingesetzt,

so hätten wir aus dem Klassen-Template eine ganz normale Klasse explizit erzeugt.

1.9 Expressiontemplates

1.10 Allgemeine Bemerkungen zur Standardbibliothek

Die Standardbibliothek enhält in der STL (Standard Template Library) schon viele brauchbare Klassen und Methoden, welche man nicht immer wieder neu implementieren muß. Wir gehen nur auf einige wenige Möglichkeiten der STL ein, es sei auf [Meyers, 1998, §49], [Meyers, 1997, §6.4.1],[Yang, 2001, §10], [Kirch-Prinz and Prinz, 2002, 567-711], [Kuhlins and Schader, 2002] verwiesen. Die Klassen und Methoden der Standardbibliothek sind im Namensraum `std` untergebracht, sodaß, z.B., `cout` via den Namespace `std::cout` aufgerufen werden muß (nicht zu Verwechseln mit dem Scope-Operator) oder man gibt einzelne Komponenten des Namensraumes `using std::cout` bzw. den gesamten Namensraum frei `using namespace` .

A1/a1_1.cpp

Die STL basiert auf drei grundsätzlichen Konzepten: Containern, Iteratoren und Algorithmen. Die Container beinhalten (mehrere) Objekte deren Templates konkretisiert werden, und in welchen mit Hilfe von Iteratoren navigiert werden kann. In Fortführung dessen sind Algorithmen Funktionen welche auf Containern (mitttles der Iteratoren) bestimmte Operationen durchführen.

Zur String- (`<string>`) und die I/O-Stream-Bibliothek (`<iostream>`) in [Kirch-Prinz and Prinz, 2002, 567-612] gut erläuterte, kompakte Informationen verfügbar.

1.10.1 Ein kleines Beispiel

In unserer Funktion `max_elem` aus §1.7.1

```
float  max_elem(const int n, const float  x[])
 {
  float  vmax = x[0];

  for (int i=1; i<n; i++)
   {  vmax = max(vmax, x[i]); }

  return vmax;
 }
```

wäre `float  x[]` der Container, der Zeiger `x+i` ist dann der Iterator und die gesamte Funktion wäre ein Algorithmus auf dem Container mit dem entsprechenden Iterator. In der originalen Version benutzen wir das eigene Funktions-Template `max` aus *A7/tfkt.hpp*. Diese eigene Version ist aber gar nicht nötig, da wir in den ALgorithmen der STL u.a., schon ein entsprechendes Template verfügbar ist, siehe auch [Kirch-Prinz and Prinz, 2002, p.657-673]

Für obige Maximumbestimmung werden wir jetzt nacheinander Container, Iteratoren und Algorithmen (zusätzlich zu `max`) der STL benutzen.

Zunächst benutzen wir die **Container**-Klasse `vector<T>` wobei der Parameter `T` wiederum eine Klasse (oder einfacher Datentyp) ist [Kirch-Prinz and Prinz, 2002, p.622-626].

```
1   //                              STL/stl1.cpp
2   #include <iostream>
3   #include <vector>                       // NEW
4   using namespace std;
5
6   int main()
7   {
8    int n; cout << " n = "; cin >> n;
9    vector<float> x(n);                    // NEW
10  //             Vector wird mit Zufallszahlen initialisieren
```

```
11    for (int i=0; i<n; i++) { x[i] = rand(); }
12   //              Maximumbestimmung - vector (klassisch)
13    float vmax(x[0]);                       // NEW:
14
15    for (unsigned int i=1; i<x.size(); i++)// NEW: size()
16      {  vmax = max(vmax,x[i]);  }          // NEW:
17    ...
18   }
```

Der Speicher wird in Zeile 9 durch den Konstruktor von `vector` allokiert.Beim Verlassen des Gültigkeitsbereiches von `x` in Zeile 18 wird dieser Speicher durch den entsprechenden Destruktor deallokiert. Zeilen 11, 13 und 16 demonstrieren mit dem Indexoperator `[]`, daß Instanzen von `vector` wie Arrays behandelt werden können. Gleichzeitig ist Zeile 13 guter C++-Stil, da Variablen erst definiert werden, wenn sie gebraucht werden und hier wird diese auch gleich über einen Parameterkonstruktor initialisiert (Einsparung gegenüber `float max; max=x[0];`). Die Länge des Vektors wird in Zeile 16 abgefragt.

Auf die Elemente eines normalen Arrays kann man neben dem Indexoperator auch über einen Pointer zugreifen, siehe *Ex630.cpp* in [Haase, 2004, §6.3]. Das ensprechende (weiterführende) Äquivalent dazu für Container sind die **Iteratoren**. Wir benutzen diese nun in der Maximumbildung :

STL/stl1.cpp

```
17   ...
18   //                      Maximumbestimmung - Iterator
19    float vmax( x.front() );               // NEW: front()
20    vector<float>::iterator xp;            // NEW: iterator
21
22    for (xp=x.begin(); xp<x.end(); xp++)   // NEW: begin(), end()
23      {  vmax = max(vmax,*xp);  }          // (NEW): *xp
24   ...
```

Die Definition des Iterators in Zeile 20 entspräche einem `float *xp;` für ein konventionelles Array. Die Methoden `begin()` und `end()` sind in diesem Sinne Pointer auf das erste (`xp+0`) bzw. das hinterletzte Datenelement (`xp+n`) von `x`. Die Methode `front()` liefert die Referenz auf das erste Element.

Eigentlich können wir uns die ganze eigene Arbeit bei der Maximumbestimmung sparen. Diese Funktion ist bereits in den **Algorithmen** definiert. Damit verkürzt sich der Code zu:

```
24   ...
25   //                      Maximumbestimmung - Algorithmus
26    vector<float>::iterator xp;
27    xp = max_element(x.begin(), x.end());  // NEW: Iterator zeigt auf Max.
28    vmax = *xp;
29   ...
```

Obiger Code ließe sich auch in einer Zeile ausdrücken aber das wesentliche ist, daß die Funktion `max_element` die Iteratoren benötigt und einen Iterator auf das größte Element liefert.

Damit wissen Sie eigentlich schon fast alles über Container, Iteratoren und Algorithmen. Eine schöne Übersicht über Container-Klassen ist in [Kirch-Prinz and Prinz, 2002, p.620ff] zu finden. Ebendort, auf Seite 618, ist eine kurze Einführung zu Iteratoren wobei diese in den Methoden der Container-Klassen immer wieder auftauchen. Ein guter Überblick zu verfügbaren Algorithmen (Header `<algorithm>`) steht dann in [Kirch-Prinz and Prinz, 2002, p.657ff], welcher sich in kompakter Form auch in [Yang, 2001, §10.2] findet.

1.10.2 Nützliche Container und Algorithmen der STL

Um Container effizient einzusetzen sollte man sich genau anschauen wozu man diese benötigt. Die nachfolgenden Container benötigen für bestimmte Funktionalitäten Algorithmen unterschiedlicher

Komplexität. Eine genauere Information über die speziellen Container ist daher unerläßlich!

Prinzipiell besitzen fast alle Containerklassen eine variable Länge, es können Elemente eingefügt, gelöscht, gesucht werden (und anderes aus den Algorithmen). Ich werde jeweils auf die Spezialilitäten vom Blickpunkt der Numerischen Mathematik eingehen.

- `vector<T>` mit Header `<vector>`
  Dieses 1D-Array hat eine variable Länge, es eignet sich insbesondere für Fälle in denen ein Vektor succesive im (num.) Algorithmus wachsen muß [Yang, 2001, §10.1.1].
  Bsp: Matrixzeilen bei Galerkin-Vergröberung von Matrizen.

- `list<T>` mit Header `<list>`
  Listenelemente haben, im Gegensatz zu `vector` keine fixe Position (also auch kein Indexoperator `[]`). Listen (und Vektoren) können geordnet (`sort()`), doppelte Einträge gelöscht (`unique()`) und vereinigt werden (`merge()`) [Yang, 2001, §10.1.2].
  Bsp.: Bestimmung des Matrixmusters bei einer FE-Vernetzung.

- `map<T>` mit Header `<map>`
  Die Elemente des Containers `map` bestehen paarweise aus *key* und *value*. Der Container kann als eine Verallgemeinerung von `vector` aufgefaßt werden [Yang, 2001, §10.1.3].

- `multimap<T>` mit Header `<multimap>`
  Wie `map` mit mehreren *values* pro *key*.

- `set<T>` mit Header `<set>`
  Enspricht einem `map` ohne *values* und hat eindeutige Elemente.

- `multiset<T>` mit Header `<multiset>`
  Wie ein `set`, aber die *keys* können mehrfach auftauchen.

- `bitset<T>` mit Header `<bitset>`
  Menge von Bits, welche ansonsten wie `set` behandelt werden. Auf jedes Bit kann einzeln zugegriffen werden.
  Bsp.: Das Markieren von ne Kanten eines FE-Netzes würde konventionell, mit `short int`-Markern, $2 * ne$ Byte erfordern. Mit einem `bitset` ist nur 1/16-tel dessen notwendig.

- `stack<T>` mit Header `<stack>`
  `queue<T>` mit Header `<queue>`
  `dqueue<T>` mit Header `<dqueue>` Effizente Stack-Operationen [Yang, 2001, §10.1.4]

Die Algorithmen für die Container unterscheidet man nichtmodifizierende und modifizierende. Für einen knappen Überblick zu verfügbaren Algorithmen sei jeweils auf den Beginn der Kapitel §10.2.1–§10.2.3 in [Yang, 2001] verwiesen. Eine detaillierte Beschreibung der Parameter und Ergebnisse ist in [Kirch-Prinz and Prinz, 2002, p.657ff] dargestellt. Eine kompakte und gut strukturierte Beschreibung der Container und Algoritmen ist in [Schildt, 2003, §16]. Die Algorithmen lassen sich grob in 3 Mengen einteilen:

- Sortieren (`sort()`), Kopieren (`copy()`), Mischen (`merge()`), Entfernen (`remove()` via Value, `erase()` via Iterator) mehrfacher Elemente (`unique()`), Ersetzen (`replace()`) ...

- Suchen (`find()`, `search()`) sowohl von einzelnen Elementen als auch von Elementen einer Menge in einer Menge (`find_first_of()`), Nichtübereinstimmung (`mismatch()`). Minimum, Maximum (`max()`, `max_element()`), Operationen auf Elementen (`for_each()`, `transform()`), Bestimmung von Anzahl des Auftretens, erstes/letztes Auftreten eines Elements (`count()`, `*bound()`, `range()`).

- Mengenoperationen wie `includes()`, `set_union()`, `set_intersection()`, `set_symmetric_difference()` und Permutationen (`*permutation()`) sowie Heapoperationen (`*heap()`).

Zur Demonstration bestimmen wir für die Container-Klasse `vector<float>` das Maximum (Zeile 14), sortieren den Vektor (Zeile 19), suchen (Zeile 21) und entfernen (Zeile 24) den Maximalwert im sortierten Vektor, und bestimmen das Maximum (Zeile 25) des verkürzten Vektors erneut . STL/stl2.cpp

Falls die Suche erfolglos ist, dann steht der Iterator auf dem hinterletzten Element, darauf wird in Zeile 24 getestet.

```
1  //                              STL/stl2.cpp
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8   ...
9   vector<float> x(n);
10 //                      Feld initialisieren
11  ...
12 //              Maximumbestimmung - max_element aus Algorithmen
13  vector<float>::iterator xp;
14  xp         = max_element(x.begin(),x.end());   // Position des Max.
15  float vmax = *xp;                              // Wert des Max.
16
17 //              Sortieren - sort aus Algorithmen
18 //              Finden    - find aus Algorithmen
19  sort(x.begin(), x.end());
20  vector<float>::iterator xs;
21  xs = find(x.begin(), x.end(), vmax);
22
23 //              Entfernen des (umsortierten) Maximums - erase aus <vector>
24  if ( xs != x.end() )  x.erase(xs);             // Vektor wird verk"urzt
25  cout << "neues Max.: " << *max_element(x.begin(),x.end()) << endl;
26  ...
27 }
```

Im Beispielcode wird zusätzlich auf die Längenangaben zu Vektor eingegangen. So liefert `x.size()` immer die **aktuelle** Länge des Vektors, ist also nach Zeile 24 um 1 vermindert. Im Gegensatz dazu bleibt `x.capacity()` auf dem alten Wert, da die **allokierte** Länge gleich bleibt. Diese Länge kann sich ändern, falls Elemente hinzugefügt werden oder ein explizites `resize()` bzw. `reserve()` benutzt werden. Die **max. mögliche** Länge eines Vektors wird mit `x.max_size()` abgefragt.

1.11 Bibliotheken für Numerische Berechnungen

1.11.1 Numerische Grenzwerte

In C war es immer ein Krux Informationen über darstellbare Zahlbereiche und Genauigkeiten aus den Headerfiles herauszuklauben (ich konnte mir nie merken, welcher Header nun was für welche Konstanten beherbergt). Daher bietet die Bibliothek zur Sprachuntzerstützung (engl.: Language Support Library), u.a., den Klassen-Template `numeric_limits` für welchen Spezialisierungen (§1.7.5, §1.8.3) für alle elementaren Datentypen definiert sind. Das nachfolgende Beipiel demonstriert einige numerische Grenzwerte.

STL/stl3.cpp

```
1  //                              STL/stl3.cpp
2  #include <iostream>
3  #include <limits>       //      numeric_limits
4  using namespace std;
5
6  int main()
7  {
8   cout << "max(double)  " << std::numeric_limits<double>::max()  << endl;
9   cout << "min(float)   "  << std::numeric_limits<float> ::min()  << endl;
10  cout << "min(int)   "    << std::numeric_limits<int>   ::min()  << endl;
11  cout << "eps(float)   "  << std::numeric_limits<float> ::epsilon() << endl;
```

```
12  }
```

So kompliziert die Bezeichner aussehen, so einfach sind sie letztendlich. Der Namensbereich `std::` ist hier nur der Vollständigkeit halber angegeben, die Anweisung in Zeile 4 gibt den Namensbereich bereits zur Nutzung frei. Die Spezifikation des Klassen-Templates ergibt die Klassenspezifikation, sodaß `numeric_limits<float>::min()` die kleinste **positive** Zahl ist, welche sich in einfacher Genauigkeit darstellen läßt. Achtung, `numeric_limits<int>::min()` liefert tatsächlich die kleinste darstellbare ganze Zahl, und diese ist **negtiv**! Die Methode `epsilon` retourniert die kleinste postive `float`-Zahl, sodaß $1 + \varepsilon \neq 1$ ist. Für weitere Methoden siehe [Kirch-Prinz and Prinz, 2002, p.710] und [Kuhlins and Schader, 2002, §13.5].

1.11.2 Komplexe Zahlen

Komplexe Zahlen sind in der STL als Klassen-Template verfügbar `complex<T>` wobei `<T>` nur für Gleitkommazahlen einen Sinn macht. Die Umwandlungen der Darstellung in Polarkoordinaten und zurück ist darin genauso enthalten wie arithm. Grundoperationen, trigonometrische und Potenzfunktionen [Yang, 2001, §7.4], [Kirch-Prinz and Prinz, 2002, p.677ff].

STL/stl4.cpp

```
1   //                              STL/stl4.cpp
2   #include <iostream>
3   #include <complex>      //      complexe Zahlen [Prinz, p678ff]
4   using namespace std;
5
6   int main()
7   {
8    complex<double> a, b(3.0,-1.0);
9    complex<float>  c(-0.876, 2.765), d(c);
10   complex<int> gg;
11
12   cout << "a = " << a << "   b = " << b << "   c = " << c << endl;
13   cout << "real b = " << real(b) << "   imag b = " << imag(b) << endl;
14
15   b /= 3.0;
16   a += c;
17   cout << "c+b/3 = " << a+b << "   c/b = " <<  a/b << endl;
18
19   cout << " d = " << d << endl;
20   cout << "Polarkoord:  betrag= " << abs(d) << "   phi= " << arg(d) << endl;
21   cout << " und wieder zurueck:  " << polar(abs(d), arg(d)) << endl;
22
23   cout << "sqrt(d) = " << sqrt(d) << endl;
24   return 0;
25  }
26
27
```

1.11.3 Das Klassen-Template `valarray<T>`

Natürlich lassen sich die Container aus §1.10.2 auch in numerischen Berechnungen einsetzen. Allerdings enthalten diese Container zu viel Funktionalität um im Hochleistungsrechnen wirklich effizient zu sein. Daher verwendet hier man anstelle der Container-Klasse `vector<T>` den Klassen-Template `valarray<T>` [Yang, 2001, §7.5], [Kirch-Prinz and Prinz, 2002, p.689ff] ind insbesondere [Kuhlins and Schader, 2002, §13.6]

Dieses Klassen-Template erledigt intern die gesamte dynamische Speicherverwaltung kann größenmäßig auch via `resize()` geändert werden. Die aktuelle Feldlänge kann via `size()` abgefragt werden. Die Konstruktion und Initialisierung über ein C-Array ist möglich (Zeile 21). Die

valarrays können miteinander arithmetisch verknüpft werden (Zeile 23), es können elementweise (auch eigene) Funktionen (Potenz, trig.) angewendet werden (Zeile 24) und valarrays können elementweise verglichen werden(Zeile 36). Darüber hinaus ist der Indexoperator [] derart überladen, daß mit dem verallg. Indexoperator Teilvektoren über Schrittweiten (Zeile 33), oder indirekt adressierte Vektoren extrahiert werden können (Zeile 33) . Zusätzlich stehen max/min/sum und mehr zur Verfügung.

STL/stl5.cpp

```
1  //                              STL/stl5.cpp
2  #include <iostream>
3  #include <valarray>     //      valaray
4  using namespace std;
5
6  double func1(const double& x)  //       Function fuer jedes Vektorelement
7  { return 1./x; }
8
9  int main()
10 {
11  int n=10,i;
12
13  double *x = new double[2*n];   //                       C-Array
14  for (i=0; i<2*n; i++)  { x[i] = (i+1.0)/n; }
15
16 //                      valarray (ohne init)
17  valarray<double> a(n);
18 //                      valarray (const. init)
19  valarray<double> b(3.14579, n);
20 //                      valarray (c-vektor init)
21  valarray<double> c(x+n/2, n);
22 //                      Multiplikation / Addition
23  a += b*c;
24  a = a.apply(func1);    // Funktion  func1 anwenden
25
26 //                              indirekte Adressierung
27  valarray<size_t> idx(5);
28  idx[0] = 3; idx[1] = 1; idx[2] = 4; idx[3] = 3; idx[4] = 0;
29
30  valarray<double> d( a[idx] );
31
32 //      mit stride: holt a[1], a[3], a[5]
33  valarray<double> asl( a[ slice(1,3,2) ] );
34
35 //      Finde alle Elemente in a, die kleiner als 0.3 sind.
36  valarray<bool>  bb( a < 0.3 );  // boolean vector
37
38
39 //      Finde alle Elemente in a, die groesser als 0.3 sind
40 //              und speichere diese in einem neuen Vektor
41  valarray<double>  abc( a[a > 0.3] );
42 }
```

Das innere Produkt zweier Vektoren v und w könnte über (v * w).sum() berechnet werden. Hierbei wird allerdings erst ein temporärer Vektor erzeugt und hinterher addiert. Eine bessere Lösung (aber nur ca. 2-4% schneller) besteht in der Nutzung des entsprechenden numerischen Algorithums aus der STL [Kuhlins and Schader, 2002, §9.11] was zum Aufruf inner_product(&v[0], &v[v.size()], &w[0], 0.0) führt [Kuhlins and Schader, 2002, p.391 unten]. Das explizite Ausprogrammieren des inneren Produktes (wie beim C-Array) ist hingegen um 2-4% schneller (Pentium4 2.8GHz).

Kapitel 2

Werkzeuge zur Programmentwicklung

2.1 Einführung in die Shell-Programmierung: `bash`

2.2 Arbeiten unter UNIX/LINUX

2.2.1 Nützliche Programme unter UNIX/LINUX

Die folgende, nichtrepräsentative Auswahl kleiner Programme ist unter jedem UNIX/LINUX-System verfügbar. Die Auswahl erfolgte unter dem Gesichtspunkt der Nützlichkeit für das Programmieren. Eine komplette Beschreibung der Programme und deren Optionen ist mittels
`LINUX> man <prog_name>`
abrufbar.

- `touch` Änderung von Zeitmarken eines Files.
  Bsp.:
  `LINUX> touch *.cpp`
  Ändert den Modifikationszeitpunkt aller Quelltextfiles (im aktuellen Verzeichnis) auf die aktuelle Systemzeit.
  `LINUX> touch file.cpp`
  Erzeugt ein File *file.cpp* falls dieses noch nicht existiert. Andernfalls, wie oben.

- `basename` - Entfernen des Verzeichnisnamens und Suffixes aus einem Filenamen.
  Bsp.:
  `LINUX> basename vektor.cpp .cpp`
  Liefert *vektor* als Ergebnis.

- `grep` - Suche von Zeichenketten (korrekt: regulären Ausdrücken) in einem druckbaren ASCII-File.
  Bsp.:
  `LINUX> grep -i "studenten" */*.hpp`
  Ausgabe sämtlicher Zeilen der Files **.hpp* in sämtlichen Unterverzeichnissen **/* welche die Zeichenkette „`studenten`“ enthalten. Die Option `-i` ignoriert Groß-/Kleinschreibung, sodaß, z.B., auch „stuDeNten“ usw. gefunden wird. Die Option `-n` gibt zusätzlich zu den Filenamen noch die Zeilennummer der Fundstelle an. Will man nur wissen, in welchen Files die Zeichenkette enthalten ist, dann ist die Option `-l` zu verwenden.

- `strings` - Ausgabe von druckbaren Zeichenketten in einem, insbesondere nicht druckbaren, File.
  Bsp.:
  `LINUX> strings vektor.o`
  u.a., werden die Namen der im (nicht so leicht lesbaren) Objektfile *vektor.o* enthaltenen

Funktionen und Methoden ausgegeben. Sehr nützlich in Verbindung mit `grep`, siehe die Suche in Bibliotheken auf Seite 43.

- `find` - Auffinden von Files in einer Verzeichnisstruktur.
  Bsp.:
  `LINUX> find /usr/lib -name "*g2c*.a" -print`
  Findet alle Bibliotheken unterhalb des Verzeichnisses `/usr/lib`, welche die Zeichenkette `g2c` im Namen enthalten. Unter LINUX kann die Option `-print` weggelassen werden. Siehe auch das Skript *B1/FindLarge* . B1/FindLarge

- `awk` - Komplexere Mustersuche und Weiterverarbeitung dieser Muster. Siehe dessen Anwendung im bash-Skript *B1/FindLarge*. Darin wird `awk` über ein awk-Skript gesteuert . Large.awk

- `sed` Komplexer Streameditor, welcher einen Inputdatenstrom nach gegebenen Regeln verändert. Siehe auch das Skript *B1/konvert* in welchem `sed` B1/konvert
  `sed -f ./konvert.sed filename >file.temp`
  mit dem Regelfile *konvert.sed* auf das Inputfile *filename* angewandt wird. Die Ausgabe wird auf ein temp. File umgeleitet. Im Regelfile bedeutet z.B., die Zeile B1/konvert.sed
  `s/Ansatz/Basis/g`
  daß die Zeichenkette „Ansatz“ durch „Basis“ **g**lobal **s**ubstituiert werden soll.

- `‘<command>‘` - liefert die Ausgabe des enthaltenen Kommandos als Zeichenkette zur weiteren Verarbeitung.
  Bsp.:
  `LINUX> xdvi ‘basename main.tex .tex‘.dvi`
  Hier wird zuerst von `basename` der Filename ohne Suffix erzeugt (*main*) um gleich danach den Suffix *.dvi* zu erhalten. Danach wird `xdvi` mit dem File *main.dvi* aufgerufen.

- `diff` - Findet die Unterschiede zweier ASCII-Files oder Verzeichnisse. Es gibt (zusätzlich zu installierende) Vergleichsprogramme wie `xdiff` welche das Vergleichsergebnis graphisch aufbereiten.
  Bsp.:
  `LINUX> diff studs.hpp studs2.hpp > diff.out`
  Die Unterschiede von *studs.hpp* und *studs2.hpp* werden im File *diff.out* gespeichert.
  Das Kommando `patch` kann umgekehrt dazu genutzt werden, um aus dem File *studs.hpp* eine neuere Version zu erzeugen, welche identisch zu *studs2.hpp* ist (in einem extra Verzeichnis probieren!!):
  `patch studs.hpp < diff.out`
  Dieses Patchen ist sehr sinnvoll verwendbar bei Updates von größeren ASCII-Files, da die zu übertragende Informationsmenge in der regel wesentlich kleiner als das gesamte File ist.

- `mgdiff` - Farbliches Markieren von Unterscheiden zweier Files, ähnliche Programme sind `tkdiff` und `tkxcd`

- `more` - Seitenweises Anzeigen von ASCII-Files im terminal.
  Bsp.:
  `LINUX> more studs.hpp`

2.2.2 Nützliche Tricks

- Selbstgestaltung des `bash`-Promptes im File *$(HOME)/.bashrc*:

```
export PROMPT_COMMAND=
export PS1='\u@\h: \W> '
```

  Nach einem `source ~/.bashrc` sieht mein Prompt dann so aus:

```
ghaase@mephisto: latex>
```

- Ausdrucken einer man-page, z.B. des Programmes `grep`:
  `LINUX> man -t grep | lp` bzw. `LINUX> man -t grep | lpr`
  Alternativ kann die Ausgabe auch in ein Postscript-File umgeleitet werden:
  `LINUX> man -t grep > p.ps`
  Falls das Programm `a2ps` installiert ist, dann erzeugt
  `LINUX> man -t grep | a2ps`
  einen Ausdruck der man-pages, sodaß 2 Seiten auf einer A4-Seite Platz haben.

- Umlenkung der Compilerwarnungen (`2>&1`) in ein File *out.txt*
  `LINUX> g++ -c -Wall studs.cpp >> out.txt 2>&1`
  Dieses Vorgehen ist sinnvoll beim automatischen Übersetzen, oder falls viele Warnungen über den Terminal scrollen, oder falls eine langsame Netzverbindung nicht mit den Ausgaben belastet werden soll. Die Warnungen werden immer wieder an das File *out.txt* angehängt (`>>`), sodaß dieses File immer größer wird.

- Suche, in welcher Bibliothek (`*.a`) des aktuellen `g++`-Compilers eine bestimmte Funktion, z.B., `norm2_f77__` enthalten ist:

  ```
  LINUX> cd /usr/lib/gcc-lib/i686-pc-linux-gnu/3.2.3
  LINUX> strings -f *.a | grep norm2_f77__
  ```

  Das Pipe-Kommando `|` bewirkt, daß die Ausgabe von `strings` als Eingabe für `grep` benutzt wird.

2.3 `make` und *Makefile*

2.3.1 Grundlagen

etwas zu `make` in [Schmaranz, 2002].

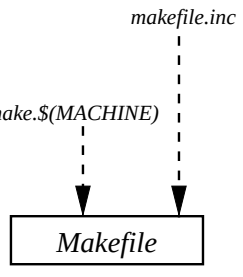
2.3.2 Abhängigkeiten erzeugen

2.3.3 Arbeiten mit mehreren Compilern und Betriebssystemen

Solange Compileroptionen wie `-O` oder `-Wall` ausreichend sind, besteht die Hauptarbeit bei Einsatz eines neuen Compilers im Umdefinieren des Macros für den Compiler im *makefile*. Schon prozessor- oder algorithmusspezifische Optimierungen (`-march=i486` bzw. `-funroll-loops`) oder nützliche Compilerwarnungen (`-Weffc++`) sind nicht mehr einheitlich verfügbar. Sobald compilerspezifische Bibliotheken noch mit verwendet werden müssen, siehe §2.6.1, werden die ständigen Änderungen im *Makefile* unübersichtlich.

In solchen Fällen hilft die Trennung des *Makefile*s in 3 Teile:

- allg. Abhängigkeiten und Transformationsregeln $\Longrightarrow$ *makefile.inc*
- compilerspezifischer Teil $\Longrightarrow$ *make.$(MACHINE)*
- verzeichnisspezifischer Teil $\Longrightarrow$ *Makefile*



wobei die ersten beiden in das letztere per `include <filename>` eingebunden werden. Da diesen beiden Files keine verzeichnisspezifischen Informationen enthalten, können sie auch in einem zentralen Verzeichnis liegen (relative oder absolute Pfadangabe beim Einbinden nicht vergessen!). Dies ist insbesondere für größere Projekte sinnvoll.

Die Umgebungsvariable `MACHINE` muß in der aufrufenden Shell gesetzt werden, z.B., `LINUX> export MACHINE=LINUX_GCC` für die Gnu-Compiler unter LINUX, bzw., `LINUX_GCC` für die 32-bit Compiler von Intel.

Ein Beispiel für ein solches Makefile ist

MixCF/Makefile.2

```
1  #                                MixCF/Makefile.2
2  include make.$(MACHINE)
3  include makefile.inc
4
5  PROG1 = main
6  SRC1  = main.cpp fkt1.cpp fkt2.c fkt3.f
7  #                        Objektfiles von verschiedene Programmiersprachen
8  OBJ1  = $(addsuffix .o,  $(basename $(SRC1)) )
9  ...
10 $(PROG1): $(OBJ1)
11         @echo "Linke aus  " $(OBJ1) "  das Programm  " $@
12         $(CPP) -o $@ $(OBJ1) $(LIB_F77_IO)
13 ...
```

Beim Aufruf von `LINUX> make -f Makefile.2` File mit den Regeln für `make` eingebunden.

MixCF/makefile.inc

```
1  #                        MixCF/makefile.inc
2  .PRECIOUS: .cpp .hpp .c .f
3  .SUFFIXES: .cpp .c .f .o
4  #       Abhgigkeit des Objektfiles vom Sourcefile
5  .cpp.o:
6          $(CPP) -c $(CPPFLAGS)  $<
7
8  .c.o:
9          $(CC) -c $(CCFLAGS)  $<
10
11 .f.o:
12         $(F77) -c $(F77FLAGS)  $<
```

Das Gnu-spezifische File ist.

MixCF/make.LINUX_G

```
1  #                                MixCF/make.LINUX_GCC
2  CPP = g++
3  CPPFLAGS = -Wall
4
5  CC = gcc
6  CCFLAGS = -Wall
7
8  F77 = g77
9  FFLAGS = -Wall
10
11 LIB_F77_IO = -lg2c
```

Das entsprechende Intel-spezifische File ist:

MixCF/make.LINUX_IC

```
1  #                                MixCF/make.LINUX_ICC
2  CPP = icc
3  CPPFLAGS = -Wall
4
5  CC = icc
6  CCFLAGS = -Wall -wd1418
7
8  F77 = ifc
9  FFLAGS =
10
11 LIB_F77_IO = -lIEPCF90 -lF90
```

2.4 Nutzung von Profilern und Debuggern

2.4.1 Compileroptionen

`-O`

2.4.2 Profiler

gprof

2.4.3 Speicherüberprüfungen

mpr[1],

valgrind ⟶ gA8, sehr gut

memprof naja

2.4.4 Debugger

`insight` Gut auch für ASM, Anzeige Klassenvariablen, Funktionenbrowser

`ddd` ahnliche Funktionalität wie `insight`

2.5 Dokumentationstools

kdoc, doxygen

doc++[2],

2.6 Nutzung von C- und Fortranquellen in C++-programmen

Effizienter als die Neuimplementierung von altbekannten oder neueren Algorithmen ist es, bereits vorhandene Bibliotheken zu nutzen. Insbesondere im Bereich des wissenschaftlichen Rechnens gibt es viele gute Bibliotheken, die teilweise schon seit Jahrzehnten weiterentwickelt werden, wie z.B.: ARPACK[3], BLAS[4], LAPACK[5], SUPERLU[6]. Aus historischen (und teilweise aus religiös-programmtechnischen) Gründen sind viele dieser Bibliotheken in Fortran oder C geschrieben, bei einigen wird mittlerweile ein C++-Interface bereitgestellt. Andererseits kann es passieren, daß ein Kooperationspartner darauf besteht, daß Ihr Code als Fortran-Subroutine funktionieren soll. Schon Lenin kannte dieses Problem und schrieb daraufhin einen Aufsatz mit dem Titel: Was tun?

2.6.1 Einbinden von C- und Fortran-Code in C++

Die Quelltexte in C, C++, F77

Zunächst definieren wir die Funktionen `int<LANG>` und `norm_2<LANG>` welche einen Array initialisieren bzw. die L_2-Norm berechnen. Der Suffix `<LANG>` steht für eine der Sprachen {`Cpp, C, F77`}, algorithmisch sind die jeweiligen drei Funktionen identisch. Die C++-Implementierung sieht

[1] http://www3.telus.net/taj_khattra/mpr.html
[2] http://www.zib.de/Visual/software/doc++
[3] http://www.caam.rice.edu/software/ARPACK
[4] http://www.cs.utexas.edu/users/flame/goto
[5] http://www.netlib.org/lapack/
[6] http://crd.lbl.gov/ xiaoye/SuperLU

vertraut aus.

MixCF/fkt1.cpp

```
1  //                      fkt1.cpp - Funktionen in C++
2  #include <cmath>
3  #include "fkt.hpp"
4
5  void initCpp(const int n, double x[], const double & s)
6  {
7    for (int i=0; i<n; i++ )
8     {
9       x[i] = s*(i+1);
10    }
11 }
12
13 double norm2_Cpp(const int n, const double x[])
14 {
15   double sum = 0.0;
16   for (int i=0; i<n; i++ )
17    {
18      sum += x[i]*x[i];
19    }
20   return sqrt(sum);
21 }
22
```

Die C-Version unterscheidet sich nur marginal in der Parameterliste.

MixCF/fkt2.c

```
1  //                      fkt2.c - Funktionen in C
2  #include <math.h>
3
4  void initC(const int n, double x[], const double s)
5  {
6    int i;
7    for (i=0; i<n; i++ )
8     {
9       x[i] = s*(i+1);
10    }
11 }
12
13 double norm2_C(const int n, const double x[])
14 {
15   int i;
16   double sum = 0.0;
17   printf(" In C\n");
18   for (i=0; i<n; i++ )
19    {
20      sum += x[i]*x[i];
21    }
22   return sqrt(sum);
23 }
24
```

Der Fortran77-Code ist da etwas gewöhnungsbedürftiger.

MixCF/fkt3.f

```
1  c                       fkt1.f - Funktionen in F77
2        SUBROUTINE initf77(n, x, s)
3
4        INTEGER n,i
5        DOUBLE PRECISION x(*),s
6  c                       Achtung: Felder beginnen in F77  mit Index 1   !!
7        DO  i = 1, n
```

```
8            x(i) = s*i
9        END DO
10
11       RETURN
12       END
13
14
15       DOUBLE PRECISION FUNCTION norm2_f77(n,x)
16
17       INTEGER n,i
18       DOUBLE PRECISION x(*),sum
19
20       WRITE(*,*) "In F77"
21       sum = 0.0
22       DO  i = 1, n
23           sum = sum + x(i)*x(i)
24       END DO
25
26       norm2_f77 = SQRT(sum)
27       RETURN
28       END
29
```

Die 3 Files werden (mit den gcc-Compilern) mit

```
LINUX>  g++ -c fkt1.cpp
LINUX>  gcc -c fkt2.c
LINUX>  g77 -c fkt3.f
```

compiliert und sollen zum Hauptprogramm *main.cpp* gelinkt werden.

```
LINUX>  g++ -o main main.cpp fkt1.o fkt2.o fkt3.o
```

Unser Haupprogramm ist in C++ geschrieben und kann natürlich problemlos mit den C++-Funktionen arbeiten, deren Deklarationen im Headerfile *fkt.hpp* stehen

MixCF/main.cpp

```
1   //                      MixCF/main.cpp
2   #include "fkt.hpp"
3   ...
4   int main()
5   {
6    const int    N = 10;
7          double x[10], s=3.2, t;
8
9    initCpp(N, x, s);
10   t = norm2_Cpp(N, x);
11
12   cout << " C++  : " << t << endl;
13  ...
14  }
```

C-Funktionen in C++ aufrufen

Ein analoger Versuch mit den C-Funktionen

```
1   ...
2    initC(N, x, s);
3    t = norm2_C(N, x);
4   ...
```

```
5  }
```

scheitert beim Linken (`g++ -o main main.cpp fkt1.o fkt2.o`), da die Funktionen `initC` und `norm2_C` nicht gefunden werden. Grund dafür ist, daß C- und C++-Objektfiles unterschiedliche Informationen zu den enthaltenen Funktionen speichern. Mit einem

```
1  //                      fkt.hpp
2  ...
3  extern "C"
4  {
5  //              C-Funktionen
6   extern void initC(const int n, double x[], const double s);
7   extern double norm2_C(const int n, const double x[]);
8  ...
9  }
```

wird dem C++-Compiler mitgeteilt, daß die im Block einhaltenen Funktionen nach den C-Konventionen behandelt werden müssen. Das Schlüsselwort `extern` ist notwendig, um dem Compiler mitzuteilen, daß die Funktionen nicht in der aktuellen (C++-)Übersetzungseinheit, sondern in einer anderen, einer externen Quelle vorliegen.

In C können Parameter nur als Wert (call by value) oder als Zeiger (call by address) übergeben werden, nicht als Refernenz. Natürlich können auch keine Klassen in C auftauchen.

F77-Funktionen in C++ aufrufen

F77-Funktionen werden compilertechnisch im Objektfile als C-Funktionen gehandhabt. Damit benötigen diese ebenfalls die Einbindung via `extern "C"`. F77-Funktionen werden intern stets mit Kleinbuchstaben bezeichnet (nur die Compiler der Firma nCube benutzten Großbuchstaben) und erhalten zur Unterscheidung von den C-Funktionen einen Underscore `"_"` angehängt. Leider gibt es von dieser Regel eine Ausnahme: Enthält der F77-Funktionsname schon einen Underscore, so hängen einige Compiler (z.B., der `g77`, nicht aber der `icc`) dem Funktionsnamen einen Double-Underscore `"__"` an. Ich nutze die Präprozessoranweisung `#define` um mir handliche F77-Funktionsnamen zum Aufruf in C++ (ohne angehängte Underscores) zu erzeugen. Dort fange ich auch das unterschiedliche Verhalten bzgl. der Underscores durch abtesten auf die Präprozessorvariable `__GNUG__` ab, welche von den Gnu-Compilern gesetzt wird.

Die Übergabe der Funktionsparameter in F77 erfolgt stets als call by reference. Falls das C-Interface eine Übergabe als Referenz erlaubt, so kann diese genutzt werden. Ansonsten muß die die Parameterübergabe über Zeiger erfolgen. Wir nutzen in unserem Beispielcode die Präprozessorvariable `F77_BY_REFERENCE` zum Umschalten zwischen beiden Möglichkeiten. Alles zusammen sieht die Schnittstelle zu den F77-Funktionen im Headerfile *fkt.hpp* dann so aus:

```
1  extern "C"
2  {
3  //              F77-Funktionen
4  #define initF77         initf77_        // _ fr F77
5
6  #ifdef __GNUG__                         //  auch Complilerflag -Ddoubleunderscore  mlich
7                                          // __ fr F77_gnu,
8  #define norm2_F77       norm2_f77__     // falls Underscore im Fkt.-namen vorkommt
9
10 #else                                   // sonst (z.B.:  __INTEL_COMPILER )
11
12 #define norm2_F77       norm2_f77_
13
14 #endif
15
16 //                      F77-Parameteruebergabe via Reference oder Adddress
17 #ifdef F77_BY_REFERENCE
```

```
18    extern void initF77(const int &n, const double x[], const double &s);
19    extern double norm2_F77(const int &n, const double x[]);
20   #else
21    extern void initF77(const int *n, const double *x, const double *s);
22    extern double norm2_F77(const int *n, const double *x);
23   #endif
24   }
```

Im Hauptprogramm wird der Schalter `F77_BY_REFERENCE` definiert. Man beachte, daß im Falle der Parameterübergabe per Address auch wirklich die Adressen der Variablen übergeben werden.

```
1    //                       MixCF/main.cpp
2    #define F77_BY_REFERENCE
3    #include "fkt.hpp"
4    ...
5    int main()
6    {
7     const int    N = 10;
8           double x[10], s=3.2, t;
9    ...
10   #ifdef F77_BY_REFERENCE
11    initF77(N, x, s);              // by reference
12    t = norm2_F77(N, x);
13   #else
14    initF77(&N, x, &s);            // by address
15    t = norm2_F77(&N, x);
16   #endif
17    cout << " F77  : " << t << endl;
18   ...
19   }
```

I/O-Bibliotheken von F77 mit C++ linken

Leider liefert der Linkprozess Fehlermeldungen wie

```
fkt3.o(.text+0x72): In function ‘norm2_f77__’:
: undefined reference to ‘do_lio’
```

sobald F77-I/O-Operationen im Fortran-Quelltext auftauchen (hier: Zeile 20 in *fkt3.f*). Auskommentieren dieser Zeilen geht nicht immer, sodaß die entsprechende Bibliothek gesucht werden muß.

Und hier fängt das Compilerspezifische an. An Hand des `g++-3.2.3` unter LINUX zeigen wir die prinzipielle Vorgehensweise.

i) Linkversuch mit dem C++-Compiler
`LINUX>  g77 -o main main.o fkt1.o fkt2.o fkt3.o`
u.a., wird die Funktion `do_lio` nicht gefunden.

ii) (erfolgloser) Linkversuch mit dem F77-Compiler und der Option `-v`
`LINUX>  g77 -v  fkt3.o`
In der Augabe sucht man nach Optionen `-L`, welche Verzeichnispfade enthalten, in denen nach Bibliotheken (**.a*) zu suchen ist. Wir wählen `<lib-Pfad> = /usr/lib/gcc-lib/i686-pc-linux-gnu/3.2.3` im weiteren.

iii) Wechsel in eines der Bibliotheksverzeichnisse und Suche, in welcher Bibliothek die gesuchte Funktion `do_lio` enthalten ist.
`LINUX>  pushd <lib-Pfad>`
`LINUX>  strings -f *.a | grep do_lio`

`LINUX> popd`
Die gesuchte Bibliothek ist *lib**g2c**.a* .

iv) Erfolgreiches Linken mit der zusätzlichen Bibliothek.
`LINUX> g77 -o main main.o fkt1.o fkt2.o fkt3.o -lg2c`
Falls die zusätzliche (F77-)Bibliothek nicht standardmäßig vom (C++-)Linker gefunden wird, dann muß der gefundene (F77-)Bibliothekspfad beim Linken mit `-L<lib-Pfad>` angegeben werden.

Beim Intel-Compiler `icc`-7.0 ist das Vorgehen analog, mit `ifc` als Fortran-Compiler. Hier fehlen die Fortran-Funktionen `f_iod` und `a_qtoi` welche sich in den Bibliotheken *lib**IEPCF90**.a* und *lib**F90**.a* im Verzeichnis */opt/intel/compiler70/ia32/lib* befinden. Das erfolgreiche Linken erfordert hier also

```
LINUX> icc -o main main.o fkt1.o fkt2.o fkt3.o -lIEPCF90 -lF90
```

Wie man mit zwei (und mehr) Compilern sinnvoll abwechselnd arbeiten kann wird in §2.3.3 erläutert.

2.7 Verteiltes Programmieren mit Versionsverwaltung: `cvs`

Solange man allein an seinem Programm schreibt, weiß man meist, wo sich die aktuelle Version der Quelltexte befindet. Schon bei der Arbeit mit Laptop und Arbeitsplatzrechner wird es etwas komplizierter. Sobald noch andere Leute am Programm herumwerkeln ist die Wahrscheinlichkeit recht groß, daß aktuellere Versionen unbeabsichtigt überschrieben werden bzw. keiner mehr weiß wer denn nun die aktuelle Version hat.

Diese Probleme werden beseitigt, wenn man Software zur Versionsverwaltung benutzt. Wir konzentrieren uns konkret auf `cvs` (Concurrent Version System) welches sowohl unter Linux als auch unter Windows verfügbar ist. Ein gutes, preisgünstiges und kleines Nachschlagwerk dazu ist [Purdy, 2004], für kompliziertere Fälle sei auf [Fogel and Bar, 2004] verwiesen.

2.7.1 Einige Begriffe

Es gibt in der UNIX-Welt schon von Anbeginn Versionskontrollsysteme, insbesondere `sccs` und `rcs`. Beide erlauben die zentrale Abspeicherung der Dateien in einem sogenannten *Repository* und die lokale Bearbeitung in der sogennnten *Sandbox* (worin man nach Herzenslust spielen kann). Beide Systeme protokollieren alle dem Repository bekanntgegebenen Änderungen und erlauben es damit, auch ältere Versionen der Files wiederherzustellen. Der grundlegende Unterschied zwischen `sccs` und `rcs` besteht darin daß ersteres Programm das Originalfile speichert und dann sämtliche Änderungen dazu protokolliert, während letzteres das aktuelle File speichert und Änderungen zu älteren Versionen speichert. Damit ist `rcs` schneller, da man meist mit aktuellen Fileversionen arbeitet. Daher baut `cvs` auf `rcs` und es bietet zusätzlich mehr Möglichkeiten für eine bequeme Arbeit in größeren Projektgruppen.

2.7.2 Checkout der Vorlesungsmaterialien

Als Fingerübung und zur Überprüfung, ob alles korrekt funktioniert werden wir das Vorlesungsskript und die Beispieldateien aus-checken.

Wir nehmen an, Sie arbeiten unter LINUX und haben eine `bash`-Shell zum Arbeiten geöffnet. Unser Repository ist das Verzeichnis `/local/home/cvsroot` auf Rechner `mephisto.uni-graz.at`. Es gibt (wie immer) mehrere Möglichkeiten, wie wir an die Dateien kommen, wir werden nur die einfachste behandeln.

- Login am Repository-Rechner überprüfen
`ssh mephisto.uni-graz.at -l login_name`
Die Option mit `login_name` ist nur notwendig, falls man auf dem Repository-Rechner einen anderen login-Namen hat.

- Umgebungsvariable setzen:
  Das Setzen der Umgebungsvariablen kann jedesmal in der `bash` erfolgen, oder wir schreiben die entsprechenden Befehle einmal in das File *~/.bashrc* (beim ersten mal danach `source ~/.bashrc` nicht vergessen).

  ```
  export CVS_RSH=ssh
  export CVSROOT=mephisto.uni-graz.at:/local/home/cvsroot
  ```

  bzw., falls der login-Name ein anderer ist:

  ```
  export CVSROOT=login_name@mephisto.uni-graz.at:/local/home/cvsroot
  ```

  Zusätzlich gebe ich noch den Editor für Meldungen an das CVS-System an (Standard ist der `vi`) und eine Liste von Dateien, welche vom CVS-System ignoriert werden sollen.

  ```
  export CVSEDITOR='nedit -bg grey'
  input* output* *.aux *.blg *.log *.o *.dvi *.bbl'
  ```

  Wenn bis jetzt alles gut ging, dann

- Aus-checken der Vorlesung

  ```
  cvs checkout advProg
  ```

  erzeugt ein neues Unterverzeichnis *advProg* in welchem die Programmierbeispiele und die LaTeX-Quellen des Skriptes zu finden sind.

- Aktualisieren der Sandbox (ihrer Version der Quellen)

  ```
  cd advProg
  cvs update
  ```

  Danach haben Sie die aktuellste Version der Vorlesung. Sie können auch nur ein File *file_name* aktualisieren.

  ```
  cvs update file_name
  ```

  Das Update-Kommando kann auch in Unterverzeichnissen ausgeführt werden. Allerdings wird dann auch nur dieses Unterverzeichnis (und untergeordnete Verzeichnisse) aktualisiert.

- Aktualisieren des Repository (globale Quellen)

  ```
  cvs commit
  ```

  Wenn Sie Fehler im Skriptum entdecken, dann können Sie das File editieren und anschließend mit obigem Kommando dem Repository mitteilen. Für Unterverzeichnisse und einzelne Files gilt die Bemerkung vom vorigen Punkt.
  Beim Aktualisieren des Repository kann es passieren, daß jemand anderes schon aktuellere Files eingespielt hat. Daher ist ein `cvs update` vor dem `cvs commit` empfehlenswert. Achten Sie in diesem Falle beim `cvs update` auf evtl. Konflikte, falls Sie beide in gleichen Textregionen Änderungen vorgenommen haben.

2.7.3 Ein eigenes Projekt im Repository verwalten

- Ein Projekt ins Repository bringen
  Zuerst sollten Sie das Projektverzeichnis *proj_dir* sicherheitshalber kopieren, z.B., `cp -r proj_dir proj_dir.bak` . Dann löschen Sie alle nicht benötigten Files im Projektverzeichnis *proj_dir* (alle übersetzten Objektfiles, Bibliotheken, temp. Files etc). Einzige Bedingung - es dürfen nur Files entfernt werden, die man sich wieder generieren kann.

  ```
  cp -r proj_dir proj_dir.bak
  cd proj_dir ;   rm del_files; cd ..
  cvs import progs/proj_dir user_name start
  ```

- Files im Repository hinzufügen oder löschen
  Ein neues Files *new_file* wird hinzugefügt mit:

  ```
  cvs add new_file
  cvs commit
  ```

  Das Löschen eines Files *rem_file* (bitte nur mit eigenen, neuen Files!!) geschieht mittles:

  ```
  rm  rem_file
  ```

```
cvs remove rem_file
cvs commit
```

Download für Windows: www.wincvs.org

cvs add projdir
cvs add projdir/*

cvs checkout proj
cd proj
cvs commit

Zeilenendezeichen DOS/UNIX beachten!

Literaturverzeichnis

[Capper, 2001] Capper, D. (2001). *Introducing C++ for Scientists, Engineers and Mathematicians*. Springer.

[Clauß and Fischer, 1988] Clauß, M. and Fischer, G. (1988). *Programmieren mit C*. Verlag Technik.

[Corp., 1993] Corp., M. (1993). *Richtig einsteigen in C++*. Microsoft Press.

[Davis, 2000] Davis, S. R. (2000). *C++ für Dummies*. Internat. Thomson Publ., Bonn/Albany/Attenkirchen, 2. edition.

[Erlenkötter, 1999] Erlenkötter, H. (1999). *C Programmieren von Anfang an*. Rowohlt.

[Fogel and Bar, 2004] Fogel, K. and Bar, M. (2004). *Open Source-Projekte mit CVS*. mitp-Verlag, Bonn, 3. edition.

[Gode, 1998] Gode, E. (1998). *ANSI C++: kurz & gut*. O'Reilly.

[Haase, 2004] Haase, G. (2004). Einführung in die Programmierung - C/C++. Vorlesungsskript, Institut für Numerische Mathematik, Uni Linz.

[Herrmann, 2000] Herrmann, D. (2000). *C++ für Naturwissenschaftler*. Addison-Wesley, Bonn, 4. edition.

[Herrmann, 2001] Herrmann, D. (2001). *Effektiv Programmieren in C und C++*. Vieweg, 5. edition.

[Hunt and Thomas, 2003] Hunt, A. and Thomas, D. (2003). *Der Pragmatische Programmierer*. Hanser Fachbuch.

[Josuttis, 1994] Josuttis, N. (1994). *Objektorientiertes Programmieren in C++: von der Klasse zur Klassenbibliothek*. Addison-Wesley, Bonn/Paris/Reading, 3. edition.

[Kirch-Prinz and Prinz, 2002] Kirch-Prinz, U. and Prinz, P. (2002). *OOP mit C++*. Galileo Press, limitierte studentenausgabe edition.

[Kuhlins and Schader, 2002] Kuhlins, S. and Schader, M. (2002). *Die C++ Standardbibliothek*. Springer, Berlin, Heidelberg, 3. edition.

[Meyers, 1997] Meyers, S. (1997). *Mehr effektiv C++ programmieren*. Addison-Wesley.

[Meyers, 1998] Meyers, S. (1998). *Effektiv C++ programmieren*. Addison-Wesley, 3., aktualisierte edition.

[Oram and Talbott, 1993] Oram, A. and Talbott, S. (1993). *Managing Projects with make*. O'Reilly.

[Purdy, 2004] Purdy, G. N. (2004). *CVS kurz & gut*. O'Reilly, deutsche edition.

[Satir and Brown, 1995] Satir, G. and Brown, D. (1995). *C++: The Core Language*. O'Reilly.

[Schader and Kuhlins, 1998] Schader, M. and Kuhlins, S. (1998). *Programmieren in C++*. Springer, Heidelberg, 5. neubearbeitete edition.

[Schildt, 2003] Schildt, H. (2003). *C/C++ ge-packt*. mitp-Verlag, Berlin, 2. edition.

[Schmaranz, 2002] Schmaranz, K. (2002). *Softwareentwicklung in C++*. Springer, Heidelberg.

[Strasser, 2003] Strasser, T. (2003). *Programmieren mit Stil. Eine systematische Einführung*. dpunkt.

[Stroustrup, 2000] Stroustrup, B. (2000). *Die C++ Programmiersprache*. Addison-Wesley, 4. aktualisierte edition.

[Yang, 2001] Yang, D. (2001). *C++ and object-oriented numeric computing for Scientists and Engineers*. Springer, New York.

Index