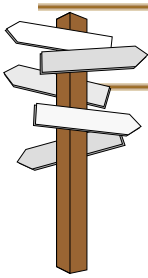


Abschnitt III
Ableiten von Klassen & Templates

- Dynamische Strukturen
- Programmierparadigmen
- Typen und Deklarationen
- Zeiger und Strukturen
- Funktionen
- Klassen
- Überladen von Operatoren
- ☞ Abgeleitete Klassen
- Klassenhierarchien
- Namensbereiche
- Templates
- Ausnahmebehandlung

Einschübe:

- Einstieg "Graphikprogrammierung"
- Quelldateien und Programme
- Umgang mit größeren Programmierprojekten (make)
- Versionskontrollsysteme (Beispiel SCCS)
- Standardbibliothek
- Ereignisgesteuerte Programmierung / MFC

Ableiten von Klassen
- Vererbung -

Ableiten von Klassen
Warum?

- **Klasse entspricht einem Konzept (wie wird etwas realisiert?)**
  - Konzept existiert nicht isoliert, es **koexistiert** mit anderen Konzepten
  - Beispiel: Konzept Auto (Bestandteile: Konzepte Räder, Motor, Fahrer, Straßen, ...)
- **Verwandtschaft zwischen Konzepten; z.B. durch ...**
  - sie bauen aufeinander auf (Hierarchie, s.o.)
  - sie bauen auf demselben Basiskonzept auf
    Beispiel: Konzepte "Kreis" und "Dreieck" bauen auf "Form" auf, stammen von Form ab
- **Ausdrücken dieser Verwandtschaft (in C), Beispiel:**

```
struct Angestellter {
   string  vorname, nachname;
   Datum   Einstellungsdatum;
};
struct Manager {
   Angestellter        ang;
   set<Angestellter*>  gruppe;
};
```

  - Ist ein Manager (für den Compiler) auch ein Angestellter?
  - Kann ein Manager auch auf einer Liste von Angestellten geführt werden?

Lösungsansätze für die Darstellung von Verwandtschaftsbeziehungen

```
struct Angestellter {
   string   vorname, nachname;
   Datum    Einstellungsdatum;
};
```

```
struct Manager {
    Angestellter ang;
    set<Angestellter*>  gruppe;
};
```

- **Lösungsansätze für Verwaltung des Managers als Angestellter**
  - explizite Typumwandlung auf `Manager*` anwenden
  - Nur Verwenden des `ang-Teils` von Manager auf der Angestellenliste
- **Bessere Form der Verwandtschaftsrepräsentation (C++)**

```
struct Manager : public Angestellter {
   set <Angestellter*> gruppe;
   // ...
};
```

  - Der Manager ist von Angestellter abgeleitet (d.h. Angestellter + gruppe + etc.)
  - Angestellter ist die Basisklasse von Manager
    Er "erbt" Eigenschaften seiner Basisklasse (--> Vererbung)
  - Ein Manager ist (auch) ein Angestellter, deshalb kann ein Manager* als Angestellter* benutzt werden. Allerdings ist ein Angestellter nicht notwendig Manager, deshalb gilt der umgekehrte Fall nicht!

Ableiten von Klassen
Zugriffskontrolle

- **Zugriffsspezifizierer public - protected - private**
  - `private` Zugriff nur über Elementfunktionen der Klasse selbst
  - `protected` ... über Elementfunktionen der Klasse selbst und der abgeleiteten Klassen
  - `public` ... über alle Funktionen

- **Ableitungskonsequenzen:** `class A : {public,protected,private} B`
  - `B-private:` Die public- und protected-Elemente von B können nur von Elementfunktionen von A benutzt werden. Nur Elementfunktionen von A können einen A* in einen B* konvertieren
  - `B-protected:` Die public- und protected-Elemente von B können nur von Elementfunktionen von A sowie der von A abgeleiteten Klassen benutzt werden. Nur diese und die von A abgeleiteten Klassen können einen A* in einen B* konvertieren
  - `B-public:` Ihre public und protected Elemente können von allen Funktionen (innerhalb und außerhalb! von A) benutzt werden. Jede Funktion kann einen A* in einen B* konvertieren

Zugriffskontrolle (§C.11)
(public, protected, private)

```
class X {
private:
    int  priv;
protected:
    int  prot;
public:
    int  publ;
    void m();
}

void X::m()
{
   priv = 1; // ok
   prot = 2; // ok

   publ = 3; // ok
}
```

```
class Y : public X {
   void mAbgeleitet();
}




void Y::mAbgeleitet()
{
   priv = 1; // Fehler: privat!
   prot = 2; // ok. prot ist protected und mAbgeleitet() ist
             ein Element der abgeleiteten Klasse Y
   publ = 3; // ok. publ ist öffentlich
}
```

Ableiten von Klassen §15.3, §C.11
Zugriffskontrolle - Beispiel: Zugriff von außerhalb

```
class X {
   public:
      int a;
      // ...
};

class Y1 : public    X { };
class Y2 : protected X { };
class Y3 : private   X { };

void f(Y1* py1, Y2* py2, Y3* py3) // Achtung: f außerhalb einer Klasse!
{
   X* px = py1;    // 0K: X ist eine öffentliche Basisklasse von Y1
   py1->a = 7;     // 0K

   px = py2;       // Fehler: X ist eine protected Basisklasse von Y2
   py2->a = 7;     // Fehler

   px = py3;       // Fehler: X ist eine private Basisklasse von Y3
   py3->a = 7;     // Fehler
}
```


Ableiten von Klassen - Zugriffskontrolle
Beispiel: Zugriff aus einer weiter abgeleiteten Klasse

```
class Y2  : protected X  { };
class Z2  : public    Y2 { void f(y1*, Y2*, Y3*); };

void Z2::f(Y1* py1, Y2* py2, y3* py3) // Achtung: f innerhalb von Z2 !
{
    X* px = pyl;     // OK: X ist eine öffentliche Basisklasse von Y1
    py1->a = 7;      //  OK

    px = py2;        // OK: X ist eine protected Basisklasse von Y2 und Z2 ist
                     //       davon abgeleitet
    py2->a = 7;      // OK

    px = py3;        // Fehler: X ist eine private Basisklasse von Y3
    py3->a = 7;      // Fehler
}
```

Ableiten von Klassen - Zugriffskontrolle
Beispiel: Exklusiver Zugriff aus der abgeleiteten Klasse

```
class Y3 : private X { void f(Y1*, Y2*, Y3*); };


void Y3::f(Y1* pyl, Y2* py2, Y3* py3) // Achtung: f innerhalb von Y3 !
{
    X* px  = py1;       // OK: X ist eine öffentliche Basisklasse von Y1
    py1->a = 7;         // OK

    px     = py2;       // Fehler: X ist eine protected Basisklasse von Y2
    py2->a = 7;         // Fehler

    px     = py3;       // OK: X ist eine private Basisklasse von Y3 und Y3::f()
                           ist ein Element von Y3
    py3->a = 7;         // OK
}
```


virtuelle Funktionen

- `Virtual` bedeutet: "Kann später in einer abgeleiteten Klasse überschrieben werden"
- `virtual ... =0`: Reine abstrakte Funktion: muß später in einer abgeleiteten Klasse überschrieben werden

- Explizite Bereichsauflösung z.T. notwendig `::`

```
void CPKW::Print()
{
  static char *Klassen[3] = {"Sauber", "Dreckschleuder",
  "Diesel"};

  CFZbase::Print(); // Ausgeben der Infos der Basisklasse
                    // CFZbase:: ist unbedingt notwendig!
  cout << "Typ:\t\t\tPKW\n";
  cout << "Hubraum:\t\t" << m_hubraum << endl;
  cout << "Leistung:\t\t" << m_leistung << endl;
  cout << "Schadstoffklasse:\t" <<
            Klassen[m_schadstofftyp] << endl;
}
```

- virtuelle Basisklassenoperation, z.B. virtual CFZbase *Copy()=0

Basisklassenzeiger

- **Basisklassenzeiger (Bsp.: CFZbase *)**
  - C++-Compiler sorgt bei Zugriff auf ein Objekt über den Basisklassenzeiger dafür, daß der richtige Objekttyp (hier: PKW, LKW, ...) gewählt wird!

```
void CNode::Print()
{
  if (m_left != NULL)
       m_left->Print();   // Daten des li. Teilbaums ausgeben
  m_object->Print();      // Ausgeben der Objektdaten
  if (m_right != NULL)
       m_right->Print(); // Daten des re. Teilbaums ausgeben
}
```

- **Funktion liefert Basisklassenzeiger zurück, NICHT Zeiger auf eigene (abgeleitete) Klasse; automatische Konvertierung**

```
CFZbase* CLKW::Copy()
{
   return new CLKW(*this);
}
```

(Beachte: wird eine Klasse A von einer Basis B (public) abgeleitet, so kann jederzeit ein A* in einen B* konvertiert werden - jedoch nicht umgekehrt!)

Abstraktion durch Basisklasse, Verwaltung der Objekte

- **Abstraktion ermöglicht, abgeleitete Klassen bzw. deren Instanzen als "Objekte" zu betrachten**
  - Erzeugen

```
root = root->AddObject(new CPKW);
root = root->AddObject(new CMotorrad);
```

  - Kopieren [obj->Copy()]

```
CNode *CNode::AddObject(CFZbase *obj)
{
  // ...
  CNode    *newnode = new CNode(obj);
}
```

  - Suchen

```
CFZbase *FZ;
FZ = root->Suchen(kennz); // Suchen
if (FZ != NULL)           // Ausgeben der betreffenden FZdaten
     FZ->Print();
```

  - Wichtig! Diese Abstraktion funktioniert in C++ **NUR** über den Zugriff durch Zeiger !!!

Abstrakter Datentyp / Elementfunktion / abstrakte Klasse (= 0)

- **Auswirkungen**
  - Von dieser Klasse können keine Instanzen angelegt werden
  - Diese Klasse kann nur in abgeleiteter Form genutzt werden
- **Weitere Verwendungsmöglichkeit der Abstraktion**
  - Entkoppelung der Klassenschnittstelle von der Repräsentation der Klasse; Beispiel:

```
class Stack {
  public:
    class Underflow{};
    class Overflow{};
    virtual void push (int i) = 0;
    virtual int  pop() = 0;
};
```

- **Verwendung als reine Schnittstellendefinition**
  - Keine Festlegung einer bestimmten Stack-Implementierung, wie z.B. Liste oder Feld
  - Keine Offenlegung von implementierungsspezifischen Konstanten
- **Jedoch ...**
  So (abstrakt) noch nicht verwendbar; es muß erst eine von Stack abgeleitete Klasse geben, welche eine entsprechende Implementierung hierzu anbietet!

Konstruktion/Konstruktoren bei abgeleiteten Klassen

- Initialisierung durch ':' Notation bei Konstruktor der abgeleiteten Klasse
- Automatischer Defaultkonstruktoraufruf, falls nicht angegeben

```
CPKW(char *Knz, int j, int h, int l, short s=0):
                                  // Variante mit Parametern
    CFZbase(Knz, j),              // Konstruktor der Basisklasse aufrufen
    m_hubraum(h),
    m_leistung(l),
    m_schadstofftyp(s) {};

CPKW(): CFZbase() { /* ... Code ... */ };
                                  // Variante ohne Parameter
```

Konstruktion / Destruktion

```
CNode::CNode(CFZbase *obj)      // Konstruktor
{
   m_object = obj->Copy();      // Aufruf der virtuellen Copy-Funktion
                                // Copy fordert jeweils neuen Speicher an
   m_left = m_right = NULL;
}


CNode::~CNode()                 // Destruktor
{
   if (m_left)
     delete m_left;
   if (m_right)
     delete m_right;
   cout << "Destruktor: " << m_object->Kennzeichen() << endl;
   delete m_object;             // Aufruf des (virtuellen) Destruktors
}
```

Virtuelle Destruktoren

- **Wie bei (üblichen) Elementfunktionen sollte auch der Destruktor in abgeleiteten Klassen 'virtual' sein**
  - D.h. der Destruktor wird in der abgeleiteten Klasse "überdefiniert"
- **Ablaufreihenfolge ist umgekehrt wie bei der Konstruktion**
  - Konstruktion: Erst Basisklasse, dann abgeleitete Klasse
  - Destruktion: Erst abgeleitete Klasse (d.h. neue Membervariablen), dann Basisklasse

delete m_object - ohne 'virtual':

```
Destruktion von Objekt A
   Basis von A zerstoert
Destruktion von Objekt Z
   Basis von Z zerstoert
Destruktion von Objekt D
   Basis von D zerstoert
```

delete m_object - mit 'virtual'

```
Destruktion von Objekt A
   Zerstoere LKW A
   Basis von A zerstoert
Destruktion von Objekt Z
   Zerstoere Motorrad Z
   Basis von Z zerstoert
Destruktion von Objekt D
   Zerstoere PKW D
   Basis von D zerstoert
```

Weitere Verwendungsmöglichkeit des Ableitens: Hierarchische Ausnahmebehandlung

- **Hierarchische Anordnung von Fehlern**

```
class MathError { };
class Overflow : public MathError { };
class Underflow : public MathError { };
```

- **Fehlerbehandlung**

```
void f()
{
    try {
        // ...
   }
   catch (Overflow) {
       // Behandle Overflow und alles davon abgeleitete
   }
   catch (MathError) {
       // behandle jeden MathError, der kein Oerflow ist
   }
   catch ( ... ) {
       // behandle alle anderen Ausnahmen
   }
}
```

Fehlerbehandlung mit Parametern

- **Ausnahme-Typen können sich auch Fehlerzustände merken:**

```
class     Range_error {
public:
          int    i;
          Range_error(int ii) { i = ii; } // Konstruktor
};
```

- **Verwendung: throw Range_error(i);**

```
// ...
if (x > 100)
    throw Range_error(x);        // Es wird ein Fehler(zustands-)objekt erzeugt
// ...                          // und geworfen
```

- **Fangen:**

```
try { ... }
catch (Range_error RErr) {
    cerr << "Range Error(" << RErr.i << ")\n";
}
```

Hierarchische Fehlerbehandlung mit Parametern

- **Ausnahme-Typen können sich auch Fehlerzustände merken:**

```
class   Range_error { /* ... wie oben ... */ }
class   Range_error_spez : public Range_error {
public: int   j;
        Range_error_spez(int ii, int jj) :
              Range_error(ii) { j = jj; }     // Konstruktor
};
```

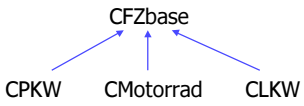
- **Verwendung**
  - `throw Range_error(x);` // Werfen des allgemeinen Fehlerobjekts
  - `throw Range_error_spez(x,y);` // Werfen des spezifischeren Fehlerobjekts
- **Fangen:**

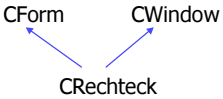
```
try { ... }
catch (Range_error RErr) { /* ... */ }
```

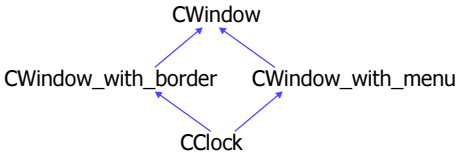
oder ... *(nicht beide zusammen! Sonst verdeckt der allgem. den spez. Fall!)*

```
catch (Range_error_spez RErrSp) {
    cerr << "Range Error(" << RErrSp.i << "," << RErrSp.j << ")\n";
}
```

Klassenhierarchien







- **Einfachvererbung**

```
class CPKW : public CFZbase { ... };
```

- **Mehrfachvererbung, §12.4**

```
class CRechteck : public CForm,
                  public CWindow { ... };
```

- **Mehrfachvererbung mit virtuellen Basisklassen, §15 insbes. §15.2.3 ff**

```
class CWindow_with_border :
    public virtual CWindow { ... };
class CClock : public CWindow_with_border,
    public CWindow_with_menu { ... };
```

Friends, §11.5

- **Deklaration einer Elementfunktion spezifiziert ...**
  1. Die Funktion kann auf private Teile der Klassendeklaration zugreifen
  2. Die Funktion ist im Sichtbarkeitsbereich der Klasse
  3. Die Funktion muß über ein Objekt aufgerufen werden (d.h. hat einen this-Zeiger)
- **Statische Elementfunktionen --> Funktion besitzt Eigenschaften (1)+(2)**
- **Deklariert man sie als 'friend' --> Funktion besitzt nur Eigenschaft (1)**

- **Anwendung**
  - Funktionen, die auf die privaten Eigenschaften zweier Klassen zugreifen müssen, ohne daß jedoch der private Zugriffsschutz ganz aufgehoben werden soll!
- **Positionierung**
  - Eine friend-Deklaration kann sowohl im privaten als auch im öffentlichen Teil einer Klassendeklaration stehen, wo spielt keine Rolle
  - Wie eine Elementfunktion wird eine friend-Funktion explizit in der Deklaration der Klasse, deren friend sie ist, deklariert. Sie ist daher genauso Teil der Schnittstelle wie eine Elementfunktion

friends, Beispiel

```
class Matrix;

class Vektor {
   float v[4];
   // ...
   friend Vektor operator*(const Matrix&, const Vektor&);
};

class Matrix {
   Vektor v[4];
   // ...
   friend Vektor operator*(const Matrix&, const Vektor&);
};

Vektor operator*(const Matrix& m, const Vektor& v)
{
   Vektor r;
   for (int i=0; i<4; i++) { // r[i] = m[i]*v
      r.v[i] = 0;
      for (int j=0; j<4; j++)
            r.v[i] += m.v[i].v[j] * v.v[j];
   }
   return r;
}
```

Templates

Templates §13

- **Stacks, Fifos, Listen, Bäume, ... d.h. allg. Verwaltungsdatenstrukturen ...**
  - Implementierung ist bis auf die **Objekttypen** identisch!
  - Beispiel: integer-, float-, double-, string-, etc. - Stack, BTree, etc ???
- **Template-Konzept**
  - Typinformationen werden bei der Definition des DT als **Parameter**(!) mitangegebenen
  - Der Compiler **generiert** dann jeweils den benötigten Stacktyp

```
template <class T> class Stack
{                  // T: Stacktyp!
   T              *buf;
   int            top;
   int            max_size;

public:
   Stack(int s = 10);
   ~Stack();
   void push(T data);
   T    pop();
};
```

Verwendungsbeispiele:

```
Stack<char>           sc;
Stack<complex>        scompl;
Stack< list<int> >    sli;
```

Templates
Organisation des Quellcodes

- **Wege, Template benutzenden Code zu organisieren**
  - Füge Template-Definitionen vor ihrer Benutzung in die Übersetzungseinheit ein
    --> Das ganze Template in einer Header-Datei ablegen
    --> Compiler-Aufgaben: Code nur bei Bedarf erzeugen, Redundanzen entfernen
  - Füge (nur) Template-Deklarationen vor ihrer Benutzung in die Übersetzungseinheit ein und übersetze ihre Definitionen separat
    --> Aufspalten des Templates in eine Header- und eine C-Datei, Beispiel:

```
// out.h:
   template <class T> void out(const T &t);

// out.c:
   #include <iostream>
   #include "out.h"
   export template <class T> void out(const T &t) {cerr << t ; }
```

  - Compiler und Binder müssen für richtige Instanziierung sorgen
- **Entscheidungsaspekte**
  - Güte von Compiler & Binder
  - Vorteil der getrennten Übersetzung: geringerer Übersetzungsaufwand bei mehrfachem Einbinden, Modularisierung, Reinheit von Headerdateien

Template-Parameter

- **Parameter**
  - Typparameter (wie im Beispiel: fifo<int>)
  - Parameter mit einfachen Typen wie int (z.B. um Größen oder Grenzen zu übergeben)
  - Template-Parameter (um der Umgebung, Einfluß auf die Instanziierung zu gewähren)
- **Ein Template kann mehrere Parameter haben**

```
template <class T, int i> class Buffer {
   T  v[i];
   int gr;
public:
   Buffer() : gr(i) {}
   // ...
}
// Verwendung:
Buffer<char,127>  cbuf;
Buffer<Record,8>  rbuf;
```

- **Anforderungen an Template-Argumente**
  - konstante Ausdrücke, z.B.: `Buffer<int, i>` // Fehler: i nicht konstant!
  - Objektadresse, Nicht-überladener Zeiger auf Objekte oder Funktionen, ...

Templates
Typprüfung und Fehlererkennungsmöglichkeiten

- **Prüfung der T.-Definition auf Syntaxfehler und andere Fehler, die sich isoliert von den Template-Argumenten bestimmen lassen**

```
template <class T> class List {
   struct Link {
      Link *pre, *suc;
      T    val;
      Link(Link *p, Link *s, const T& v) : pre(p), suc(s), val(v) {}
   }                                    // Syntaxfehler: Semikolon fehlt!
   Link *head;
public:
   List() : head(7) {}                  // Fehler: Zeiger mit int initialisiert
   List(const T &t) :
      head(new Link(0,o,t)) {}          // Fehler: undefinierter Bezeichner 'o'
   // ...
   void print_all() { for (Link *p=head; p; p=p->suc)
      cout << p->val << endl; }
};
```

- **Wo könnten weitere Fehlerquellen stecken???**
  - Fehler, die sich erst bei der Instanziierung, d.h. bei der Verwendung eines konkreten Typs, ergeben

Funktions-Templates

- **Generische Funktionen: Funkt. mit einem weiten Typparameterumfang**
  - Beispiel: sortieren

```
template <class T> void sort(vector<T> &);    // Deklaration

void f(vector<int> &vi, vector<string> &vs)   // Verwendung
{
   sort(vi);    // sort(vector<int> &);
   sort(vs);    // sort(vector<string> &);
}

template <class T> void sort(vector<T> &v)    // Definition
{
   const size_t n = v.size();
   for (...) for (...) {
      if (v[j+gap] < v[j]) {                  // Vertausche v[j] mit v[j+gap]
           T temp   = v[j];
           v[j]     = v[j+gap];
           v[j+gap] = temp;
      }
   }       // Wichtig: Wertzuweisung und Vergleich '<' müssen für diesen
}          // Typ definiert sein!!!
```

Überladen von Funktions-Templates (1)

- **Es kann mehrere Funktions-Templates mit demselben Namen und sogar eine Kombination aus Funkions-Templates und normalen Funktionen geben, Beispiel:**

```
template<class T> T sqrt(T);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{
   sqrt(2);         // sqrt<int>(int)
   sqrt(2.0);       // sqrt(double)
   sqrt(z);         // sqrt<complex<double>>(complex<double>)
}
```

- **Regeln für die Spezialisierung des Funktionstemplates analog zu den bekannten Regeln bei der Überladung von Funktionsnamen**
  - Falls keine oder mehrere gleich gute Spezialisierungen gefunden werden --> Fehler / Mehrdeutigkeit

Überladen von Funktions-Templates (2)

- **Beispiele für das Spezialisieren von Funktions-Templates**

```
template<class T> T max(T,T);

const int s=7;

void k()
{
   max(1,2);              // max<int>(1,2)
   max('a','b');          // max<char>('a','b')
   max(2.7, 4.9);         // max<double>(2.7, 4.9)
   max(s,7);              // max<int>(int(s), 7) (einfache Konvertierung)
   max('a', 1);           // Fehler: mehrdeutig (keine Standard-Konvertierung)
   max<int>('a',1);       // Auflösung: max<int>(int('a'),1)
   max(2.7, 4);           // Fehler: mehrdeutig (keine Standard-Konvertierung)
   max<double>(2.7,4);    // Auflösung: max<double>(2.7, double(4))
}
```

- **Auflösung durch Zufügen einer passenden Deklaration:**

```
inline int max(int i, int j) { return max<int>(i,j); }
```

Modularisierung durch Namespaces

- **Ein Satz verwandter Prozeduren zusammen mit den von ihnen manipulierten Daten wird häufig als Modul bezeichnet (-> Datenkapselung)**
- **Beispiel: Definition eines Stacks (Wesentliche Probleme):**
  - Bereitstellung einer **Schnittstelle** zur Benutzung des Stacks (z.B. Funktionen wie push() und pop())
  - Sicherstellen, daß auf die **Repräsentation** des Stacks (z.B. ein Feld aus Elementen) nur über die **definierte Schnittstelle** zugegriffen werden kann.
  - Sicherstellen, daß der Stack for der ersten Benutzung **initialisiert** wird
- **Daten, Strukturen und Funktionen werden in einem gemeinsamen Namensbereich zusammengefaßt**
- **Es kann mehrere Schnittstellen geben, z.B. Anwenderschnittstelle und Implementierungsschnittstelle**

Namespaces (2)

- **Anwenderschnittstelle:**

```
namespace Stack   // Anwenderschnittstelle zeigt nur für Anwender notw. Informat.
{
   void    push(int);
   int     pop();
}
```

- **Implementierungsschnittstelle:**

```
namespace Stack // Implementierungsschnittstelle zeigt Implementierungsdetails
{               // auch doppelte Deklaration erlaubt, solange identisch!!!
   const int    max_size = 200;
   int          v[max_size];
   int          top = 0;

   void    push(int);      // Implementierung: Stack::push()
   int     pop();          // Aufruf analog!
}
```

- **Zugriff**

```
Stack::push();   Stack::pop(); ...
using Stack::push;   // Ermöglicht, die Namensqualifizierung wegzulassen
```

Namespaces, Using

- **Aufwendig und schwer lesbar, wenn häufig benötigte Funktionen, etc. immer wieder durch 'Namespace::' qualifiziert werden müssen**
  - using - Direktive, um Namen aus einem Namensbereich verfügbar zu machen

```
double Parser::prim(bool get)
{
    using Lexer::get_token;     // Benutze Lexers get_token
    using Lexer::cur_token;     // Benutze Lexers curr_token
    using Error::error;         // Benutze Errors error
    // ...
 }
```

- **Globaler Import von allen Namen eines anderen Namensbereichs**
  - `using namespace ...`

```
double Parser::prim(bool get)
{
    using namespace Lexer;      // Importiere alle Namen aus Lexer
    using Error::error;         // Benutze Errors error
    // ...
}
```

Zusammensetzen von Namensbereichen

- **Zusammenbinden aller Funktionen, etc., die zu einem Modul / Namensbereich gehören könnten**

```
namespace MyRationals {
    class rational { /* ... */ };
    rational operator+(rational,rational);
    rational operator-(rational,rational);
    // ...
}
```

  - Einbinden über explizite Qualifizierung oder 'using'

- **Komposition und Auswahl**

```
namespace Meine_bib {
    using namespace Seine_bib;  // Alles aus Seine_bib
    using namespace Ihre_bib;   // Alles aus Ihre_bib

    using Seine_bib::String;    // Löse potentielle Kollision zugunsten Seine_bib
    using Ihre_bib::Vektor;     // Löse potentielle Kollision zugunsten Ihre_bib
    // ...
}
```

Standardbibliothek

Standardbibliothek / Container

- **Klassen mit der Hauptaufgabe, Objekte zu verwalten: Container**
  - vector<T> variabel großer Vektor
  - list<T> doppelt verkettete Liste
  - queue<T> Warteschlange
  - stack<T> Stack
  - deque<T> double ended queue
  - priority_queue<T> eine nach Wert sortierte Queue
  - set<T> eine Menge
  - multiset<T> eine Menge, in der ein Wert mehrfach enthalten sein kann
  - map<key,wert> ein assoziatives Feld
  - multimap<key,wert> Map, in der ein Schlüssel mehrfach enthalten sein kann

Standardbibliothek: Variabel große Vektoren

- **vector - Felder mit dynamisch veränderlicher Größe**
  - Zugriff mit u. ohne Bereichsüberprüfung
  - Bestunterstützte Zugriffsart: Indexzugriff

```
vector<Eintrag> telbuch(1000); // Achtung! ( ) nicht [ ]
```

- **Zugriff über [ ]**

```
void print_eintrag(int i)
{
   cout << telbuch[i].name << ' ' << telbuch[i].nummer << endl;
}
```

- **Verlängern des Vektors:**

```
void eintraege_zufuegen(int n) // um n vergroessern!
{
   telbuch.resize(telbuch.size()+n);
}
```


Standardbibliothek: Listen

- **list - doppelt verkettete Liste**
  - Bestunterstützte Zugriffsart: Einfügen und Löschen

```
list<Eintrag> telbuch;

void print_eintrag(const string &s)
{
   typedef list<Eintrag>const_iterator  LI;
   for (LI i=telbuch.begin(); i != telbuch.end(); ++i) {
      if (s == i->name)
           cout << i->name << ' ' << i->nummer << endl;
   }
}
```

- **Hinzufügen von Elementen**

```
void eintrag_zufuegen(Eintrag &e, list<Eintrag>::iterator i)
{
   telbuch.push_front(e);      // Hinzufuegen am Anfang
   telbuch.push_back(e);       // Hinzufuegen am Ende
   telbuch.insert(i,e);        // Vor dem Element, das i referenziert, hinzufügen
}
```

Standardbibliothek: Map

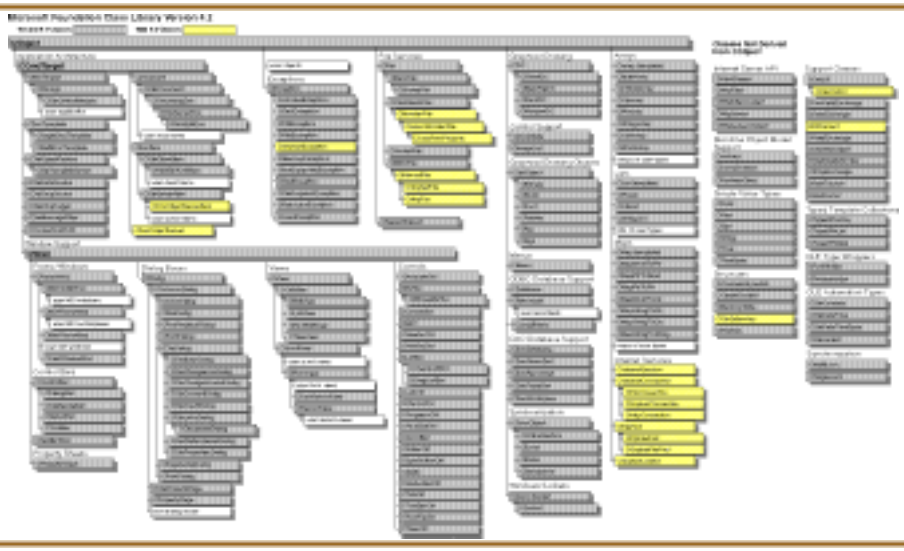
- **Map (Abbildung, assoziatives Feld, Dictionary) - ein Container, der aus Wertepaaren (Schlüssel & Wert) besteht**
  - Bestunterstützte Zugriffsart: Suchen nach einem Schlüssel

```
map<string, int> telbuch;

void print_eintrag(const string &s)
{
   if (int i = telbuch[s])
      cout << s << ' ' << i << endl;
}
```

- **Wenn eine map mit einem Wert vom Typ ihres Schlüssels indiziert wird, liefert sie den zugehörigen Wert ihres zweiten Typs zurück**
- **Setzen eines Eintrags**
  - `telbuch["Wienkop"] = 614;`

Microsoft Foundation Classes (MFC)



Was ist die MFC?

- **Neue Programmierschnittstelle, die die BS-Aufrufe in Form von Klassenmethoden gekapselt zur Verfügung stellt.**
- **MFC: Bibliothek von Klassen, in afxwin.h enthalten**
- **Objektorientierter Ansatz: Verwenden der von Microsoft gelieferten Funktionalität (der Klassen) und Erweitern dieser Klasse um eigene Aspekte (Ableiten der MFC-Klassen)**
- **Grundlegende Klassen: CWinApp und CFrameWnd**
  - CWinApp repräsentiert die Applikation als solche oder anders gesprochen den physikalisch vorhandenen Prozeß
  - CFrameWnd repräsentiert das Hauptfenster, das zu jeder Windows-Applikation gehört (~ Standard Ein-/Ausgabe-Kanal der Applikation)
  - Alle Ereignisse, die in diesem Fenster passieren (Tastatureingaben, Mausbewegungen, Mausklicks, ...) werden von CFrameWnd verwaltet
  - Die Applikation kann natürlich weitere Fenster enthalten, diese sind dann von CWnd abgeleitet

Hello World

```
class CMyApp : public CWinApp {          // Applikationsklasse
public:
      virtual BOOL InitInstance();
};
class CMyFrame : public CFrameWnd {      // Rahmenfenster-Klasse
public:
      CMyFrame();
};
```

myapp.h

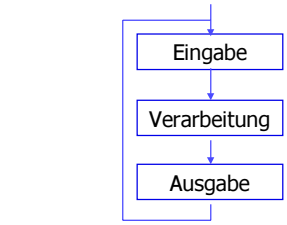
```
#include <afxwin.h>       // Klassenbibliothek-include-Datei
#include "myapp.h"
BOOL CMyApp::InitInstance()
{
   m_pMainWnd = new CMyFrame();
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}
CMyFrame::CMyFrame()
{
   Create("MyAppFrame", "Hello World");
}
CMyApp NEAR theApp;
```

myapp.cpp

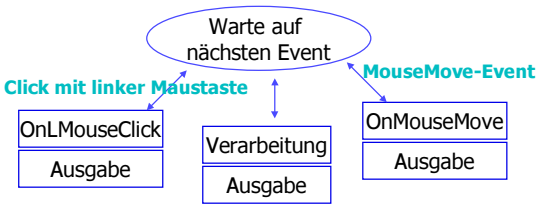
Klassische vs. ereignisorientierte Programmierung

- **Klassisch**
  - Dateneingabe (von Tastatur oder Datei oder ...)
  - Datenverarbeitung (Berechnung, Auslösen von Aktionen, ...)
  - Datenausgabe (auf Bildschirm oder in Datei)
  - **Programmierer gibt die Zeitpunkte und die Reihenfolge** von Eingaben, Verarbeitungsschritten und Ausgaben **an**!
- **Ereignisorientiert**
  - **Anwender entscheidet**, welche Funktionalität gerade zur Lösung seiner Aufgabe benötigt wird
  - Anwender entscheidet, wann Eingaben abgeschlossen sind und Verarbeitungsschritte angestoßen werden sollen
  - Aktionen müssen nicht vollständig durchlaufen werden, **Abbruch-Button**
  - **Benutzerereignisse** (Events) stoßen Verarbeitungsschritte an (Event-Handler)
  - Event-Handler führt notwendige Aktionen durch und kehrt dann zurück (Warten auf neue Events)

Klassische vs. ereignisorientierte Programmierung



- **Festlegung des Ablaufs durch Programmierer**
  - Programmfortschritt erst nach erfolgter Eingabe durch den Anwender
  - Vorziehen oder Verschieben von Eingaben kaum möglich



- **Ablauffestlegung durch Anwender**
  - Events als Auslöse- (Trigger-) Mechanismus für Verarbeitungs-schritte
  - Aktionsabbruch und Vorziehen weiterer Aktionen jederzeit möglich

Auszug aus den Window-Manager-Events (L-M)

ON_WM_LBUTTONDBLCLK( )	afx_msg void OnLButtonDblClk(UINT, CPoint);
ON_WM_LBUTTONDOWN( )	afx_msg void OnLButtonDown(UINT, CPoint);
ON_WM_LBUTTONUP( )	afx_msg void OnLButtonUp(UINT, CPoint);
ON_WM_MBUTTONDBLCLK( )	afx_msg void OnMButtonDblClk(UINT, CPoint);
ON_WM_MBUTTONDOWN( )	afx_msg void OnMButtonDown(UINT, CPoint);
ON_WM_MBUTTONUP( )	afx_msg void OnMButtonUp(UINT, CPoint);
ON_WM_MDIACTIVATE( )	afx_msg void OnMDIActivate(BOOL, CWnd*, CWnd*);
ON_WM_MEASUREITEM( )	afx_msg void OnMeasureItem(LPMEASUREITEMSTRUCT);
ON_WM_MENUCHAR( )	afx_msg LONG OnMenuChar(UINT, UINT, CMenu *);
ON_WM_MENUSELECT( )	afx_msg void OnMenuSelect(UINT, UINT, HMENU);
ON_WM_MOUSEACTIVATE( )	afx_msg int OnMouseActivate(CWnd*, UINT, UINT);
ON_WM_MOUSEMOVE( )	afx_msg void OnMouseMove(UINT, CPoint);
ON_WM_MOVE( )	afx_msg void OnMove(int, int);
ON_WM_MOVING( )	afx_msg void OnMoving(UINT, LPRECT);

Vielen Dank für Ihre Aufmerksamkeit und Ihr Durchhalten

C++ - Programmierung
Übersicht (1)

- **Einstieg**
  - Dynamische Speicheranforderung: malloc & new
  - Verwaltung dynamischer Objekte durch eine lineare Liste (sortiert / unsortiert)
  - Übungsaufgabe: Adreßverwaltung/To Do - Verwaltung
- **Übersicht über Programmierparadigmen**
  - Prozedural - Modular - Datenabstraktion - Objektorientierung - Generisch
- **Typen und Deklarationen**
  - Bekannte Typen aus C + bool + enum
  - Konstanten
    - Konstante Werte
    - Konstante Zeiger und Zeiger auf Konstante
  - Referenzen
  - Funktionen
    - Statische Variable
    - Argumentübergabe call-by-value, call-by-reference

C++ - Programmierung
Übersicht (2)

- **Benutzerdefinierte Typen (Klassen)**
  - Übungsaufgabe: Implementierung eines Stacks/Fifos
  - Erweiterung des Konzepts "Struktur" --> Klasse
    - Membervariable - Elementfunktionen - Konstruktoren - Statische Klassenelemente
  - Übungsaufgabe: Adreßverwaltung mit Klassen
  - Selbst-Referenz, this
  - Kopieren von Klassen-Objekten --> Eigener Copy-Konstruktor
  - Übungsaufgabe: Fahrstuhlsimulation
- **Ableiten von Klassen**
  - Ansätze für die Darstellung von Verwandtschaftsbeziehungen
  - Zugriffskontrolle
    - von außerhalb - aus einer weiter abgeleiteten Klasse - Exklusiv aus der abgeleiteten Klasse
  - virtuelle Funktionen, Basisklassenzeiger
  - Objektorientierung (Abstraktion) durch Basisklasse
  - Konstruktion/Destruktion bei abgeleiteten Klassen
  - Klassenhierarchien
  - Übungsaufgabe: Verwaltung von Fahrzeugdaten

C++ - Programmierung
Übersicht (3)

- **Überladen von Funktionsnamen**
  - Finden der richtigen Funktionsversion
  - Überladene Funktionsnamen: Auflösung mit Konvertierungen
  - Überladen von Operatoren
    - Operator-Funktionen
    - Ein- und zweistellige Operatoren
    - Element- und Nicht-Elementoperatoren
  - Beispiel/Übungsaufgabe: Rationale Zahlen
  - Gemischte Arithmetik
    - Initialisierung und Zuweisung mit einem Skalar
    - Konstruktoren und Konvertierungen
- **Ausnahmebehandlung**
  - Problemsituation Ausnahmefall
  - Ausnahmebehandlungskonzept (try - throw - catch)
    - Beispiel: Fifo
  - Woher kommen die Ausnahme-Typen?
  - Hierarchische Ausnahmebehandlung

C++ - Programmierung
Übersicht (4)

- **Templates**
  - Definition eines einfachen Templates (Beispiel: Fifo)
- **Namespaces**
- **C++ - Standardbibliothek**
  - Listen, Queues, Vektoren, Mengen
- **Allgemeine Programmierkenntnisse**
  - Abhängigkeiten im Programm - Verwaltung durch "MAKE"
  - Versionskontrollsysteme
  - Programmaufbau bestehend aus mehreren *.h und *.c - Dateien
  - Dokumentation von Schnittstellen und Beziehungen zwischen Klassen
    - Beispiele: Adreßverwaltung u. Aufzugsimulation