C++ Klassen

Grundlagen

Übersicht

- Klassen im Überblick
  - Definition und Nutzung
  - Datenelemente
  - Elementfunktionen
- this-Zeiger
- Konstruktor / Destruktor
- Beispiel:

Definition einer Klasse

Die Definition einer Klasse erfolgt in zwei Schritten:

1. Deklaration der Klassenelemente
   - der Datenelemente
   - der Elementfunktionen
   - Festlegung des Zugriffschutzes auf Datenelemente und Elementfunktionen

2. Definition der Elementfunktionen

```
class Klasse
{
Schutzangabe:// max. 3
             // Varianten
   Deklaration der Datenelemente
   Deklaration der Elementfunktionen
};
```

```
//Achtung immer ein ";" am Ende
//(d.h. class {} ist Anweisung)
```

```
Definition der Elementfunktionen
```

Kapselung und Informationsschutz

Kapselung:

- Eine Klasse besteht aus Daten (Variablen, Objekte anderer Klassen) und Funktionen (Members), die in einem einzigen Konstrukt zusammengefasst werden.

Informationsschutz (Sichtbarkeit):

- Nicht auf alle Komponenten (Daten und/oder Funktionen), die die Klasse beschreiben, kann vom Anwender zugegriffen werden. Es gibt drei Stufen des Informationsschutzes:
  - private: Zugriff haben nur die Komponenten der Klasse selbst oder Funktionen bzw. Klassen, die als Freund der zu definierenden Klasse erklärt werden. (**Standardvariante**, wenn die Angabe beim ersten Deklarationsblock fehlt).
  - public: Zugriff haben Funktionen der Klasse selbst, abgeleitete Klassen, Objekte der Klassen sowie Freund-Klassen und -Funktionen.
  - protected: Abgeleitete Klassen haben Zugriff wie auf ein public-Element. Für alle anderen verhält es sich wie ein private-Element.

Deklaration und Nutzung

```
class Datum
{
private:                              //kann weggelassen werden
    int ta, mo, ja;                   //members only (or friends)

public:
    bool set(int tag, int monat, int jahr);    //Deklaration Member-Funktion
    int tag(void);                             //Deklaration Member-Funktion
    int monat(void);                           //Deklaration Member-Funktion
    int jahr(void);                            //Deklaration Member-Funktion
};

.........

bool result;
Datum dat, *pdat = &dat;              //instanziere Objekte
result = dat.set(10, 10, 2005);       //Zugriff über Zugriffsoperator „.“
result = pdat->set(10, 10, 2005);     //Zugriff über Zugriffsoperator „->“
result = dat.set(30, 2, 2008);        //wird hoffentlich verhindert
dat.mo = 10;                          //!!!! compiler error
cout << dat.jahr();                   //Zugriff über Member-Funktion
```

Definition der Elementfunktionen

- Die Definition der Elementfunktionen erfolgt im Regelfall außerhalb der Klassendeklaration.
  - Die Zuordnung der Elementfunktion zur Klasse wird für den Compiler über den Klassennamen und den Bereichsoperator „::“ hergestellt.
  - Dabei hat die Elementfunktion direkten Zugriff auf alle Daten und Methoden der Klasse, ohne sie deklarieren oder Parameter übergeben zu müssen.
- Falls die Elementfunktion innerhalb der Klassendeklaration erfolgt, wird sie automatisch als inline-Funktion angelegt.

```
class Datum
{
  private:           // ist default

    int ta, mo, ja; // members only
                    // (or friends)
  public:
     bool set(int tag, int monat,
              int jahr);  // Deklaration
                          // Member-Funk.

     int tag(void) {return ta;} // inline
                                // Defin.

     int monat(void) {return mo;}// inline
                                 // Def.

     int jahr(void) {return ja;} // inline
                                 // Def.
};
```

Definition der Elementfunktionen

```
bool Datum::set(int tag, int monat, int jahr)
{
    if (tag < 1 || monat < 1 || monat > 12 || jahr < 0)     return false;

    switch (monat)
    {
          case 2:
            bool schaltjahr = jahr%4 == 0 && jahr%100 != 0 || jahr%400 == 0;
            if (tag > 29 || (tag == 29 &&  !schaltjahr))     return false;
            break;
          case 4: case 6: case 9: case 11:
            if (tag > 30)     return false;
            break;
          default:
            if (tag > 31)     return false;
    }
    ta = tag;
    mo = monat;
    ja = jahr;
    return true;
}
```

Die Faustregel, dass die Jahreszahl durch 4 teilbar ist, reicht alleine nicht. Volle Jahrhunderte sind keine Schaltjahre, außer die Jahreszahl lässt sich wiederum durch 400 teilen

Verwendung der Klasse Datum

- Zumindest die Klassendefinition muss vorhanden sein
  - z.B. auch durch Header Datei
- Durch Verwendung von Datum (wie ein Typ) wird ein Objekt mit Namen "heute" instanziiert.
  - Damit sind die Daten des Objekts vorhanden
- Die Elementfunktion des Objektes heute werden aufgerufen
  - heute.set()
  - heute.monat(),….

```
#include ….

class Datum {…};     // z.b. Aus Header

bool Datum::set () {…}


int main ()
{
    Datum heute;

    heute.set(26,3,2004);

    cout << "Heute ist der : " <<
    heute.tag()
         << "." << heute.monat()
         << "." << heute.jahr() << endl;

    return 0;

}
```

Zugriff über Zeiger

- Um von einer Funktion, die nicht Elementfunktion der Klasse ist, auf die Datenelemente der Klasse zugreifen zu können, muss ein
  - Objekt (Zugriffsoperator „.“) oder
  - die Referenz auf ein Objekt (Zugriffsoperator „.“)
  - oder ein Zeiger auf ein Objekt (Zugriffsoperator „->“)

  verwendet werden.

```
class Demo
{
    public:
      int cli;
      oid func(int j) {cli = j;}
};


.........
Demo dcl;
dcl.cli = 5;
dcl.func(10);

Demo *pdcl = &dcl;
pdcl->cli = 15;
pdcl->func(20);
```

Der this-Zeiger

- Bei der Instanzierung von Objekten einer Klasse besitzt jedes Objekt seine eigenen Member-Variablen (soweit keine besonderen Maßnahmen ergriffen werden: static).
- Die Member-Funktionen existieren nur einmal und bedienen alle Objekte.

```
class Demo
   {
   public:
        int cli;
        void func(int j)
        {cli = j;}
   };

.........
   Demo obj1, obj2;
        //instanziiere Objekte
   obj1.func(10);
        //mutiere obj1.cli
   obj2.func(20);
        //mutiere obj2.cli
```

Woher „weiß“ die *einzige* Member-Funktion, von welchem Objekt sie aufgerufen wurde und welche Member-Variable damit zu bearbeiten ist?

Der this-Zeiger

Antwort:

- Der Compiler ergänzt die Parameterliste einer Elementfunktion automatisch um einen Zeiger (this-Zeiger), der beim Aufruf die Adresse desjenigen Objekts enthält, das die Elementfunktion aufruft.

```
class Demo
   //Definition im C++-Quelltext
   {
   public:
     int cli;
     void func(int j){cli = j;}
   };
```

```
class Demo

   //"Ergänzung" durch den
   Compiler
   {
   public:
     int cli;
     void func(Demo *this,
                int j)
        {this->cli = j;}
   };
```

Konstruktor- und Destruktor-Funktionen

- Konstruktor- und Destruktor-Funktionen einer Klasse sind spezielle Elementfunktionen, die automatisch aufgerufen werden, wenn ein Objekt der Klasse instanziert wird bzw. wenn das Objekt den Bezugsrahmen verlässt. Sie dienen der Initialisierung bzw. Deinitialisierung des Objekts.
- Eigenschaften des Konstruktors:
  - Der Name der Elementfunktion muss mit dem Namen der Klasse übereinstimmen.
  - Er hat keinen Rückgabewert (auch nicht void).
  - Er wird automatisch aufgerufen
    - wenn ein Objekt der Klasse instanziert wird,
    - wenn ein Zeiger auf ein Objekt der Klasse definiert und mit dem new-Operator initialisiert wird,
    - wenn eine Funktion ein Objekt als Rückgabewert hat,
    - wenn einer Funktion ein Objekt als Aktualparameter übergeben wird.
  - Er kann von const-deklarierten Objekten der Klasse aufgerufen werden, ohne selbst const-deklariert zu sein.
  - Er kann nicht als virtuelle Funktion deklariert sein.
  - Er wird nicht auf abgeleitete Klassen vererbt.

Konstruktor- und Destruktor-Funktionen

- Weitere Eigenschaften des Konstruktors:
  - Er ist automatisch inline, wenn er innerhalb der Klassendefinition definiert wird.
  - Er kann mit anderen Konstruktoren überladen werden.
  - Seine Formalparameterliste darf Standardwerte aufweisen.
  - Er kann Member-Variablen initialisieren und jede andere Aktion ausführen.
  - Er kann public, private oder protected sein. Ist der Konstruktor private-deklariert, so können nur Elementfunktionen der Klasse oder friend-Funktionen der Klasse Objekte instanzieren (private class).

Beispiel:

```
class Demo
    {
    public:
      int k;
      Demo(int i) {k = i;}
      // inline-Definition des
      // Konstruktors
    };

.......

Demo objDemo1(10); //Instanz.
                   //+Init.:
                   //objDemo1.k = 10

Demo *ptrDemo = new Demo(20);
        // Instanzierung +
        // Init.:
        // ptrDemo->k = 20

const Demo objDemo2(30);
        //auch mit const-Objekt
```

Beispiel : Konstruktor

```
class Datum
{
     private:
       int ta, mo, ja;
    public:
       Datum (int t, int m, int j)
          { ta=t; mo = m; ja = j; } // Achtung jetzt ohne die Überprüfung
                                        // { this->set(t,m,j);} Aufruf der Elementfunktion


}

int main ()
{
    Datum heute(26,3,2004);

    cout << "Heute ist der : " << heute.tag()
         << "." << heute.monat()
         << "." << heute.jahr() << endl;

    return 0;

}
```

Der Destruktor

Eigenschaften des Destruktors:

- Der Name der Elementfunktion muss mit dem Namen der Klasse übereinstimmen und mit dem Symbol „~“ eingeleitet werden.
- Er hat *keinen* Rückgabewert (auch nicht void).
- Die Parameterliste ist leer und damit vom Typ void.
- Er kann *nicht* überladen werden.
- Er wird *automatisch* aufgerufen
  - wenn ein Objekt den Bezugsrahmen verlässt,
  - wenn ein Zeiger auf ein Objekt der Klasse, der mit dem new-Operator definiert wurde, mit delete deallokiert wird.
- Er kann public, private oder protected sein. Ist der Destruktor private-deklariert, so können nur Elementfunktionen der Klasse oder friend-Funktionen der Klasse auf Objekte zugreifen (private class).
- Er kann *auch* als virtuelle Funktion deklariert sein.
- Er kann beliebige Aktionen wie jede normale Funktion ausführen.

Beispiel: Konstruktor / Destruktor

Beispiel:

```
class Demo
{
private:
int elemente;
double *vektor;
public:
    Demo(int n);            //Deklaration des Konstruktors
    ~Demo(void);            //Deklaration des Destruktors
};

.......
Demo::Demo(int n)                       //Definition des Konstruktors
{
    elemente = n;
    vektor = new double[n];             //Speicherallokation für Zeigerinhalt
    for (int i = 0; i < elemente; i++)
        vektor[i] = 0.;                 //Initialisierung des Zeigerinhalts
}

Demo::~Demo(void)           //Definition des Destruktors
{
    delete vektor;          //Speicherfreigabe für Zeigerinhalt
}
```

Aufrufmöglichkeiten

Varianten in der Schreibweise für die Parameterübergabe an den Konstruktor:

Explizite Form:

```
Demo obj1 = Demo(2, 3, 4);
```

Verkürzte Schreibweise:

```
Demo obj1(2, 3, 4);
```

Alternative Schreibweise, wenn nur *ein* Argument vorhanden ist:

```
Demo obj2 = 10;              //Initialisierung - keine Zuweisung
```

Schreibweise, wenn *kein* Argument vorhanden ist:

```
Demo obj3;                  //nicht: Demo obj3( )!!!  Interpretation?
```

Sind Objekte selbst als Member-Variablen einer Klasse definiert, so können diese *nicht* wie oben initialisiert werden.

Lösung: *Initialisierungsliste* (nächstes Kapitel)

Überladen eines Konstruktors

Beispiel:

```
class Demo
{
public:
    Demo(int i, double x);        //Deklaration des Konstruktors
    Demo(int j);                  //1. überladene Deklaration
    Demo(double y);               //2. überladene Deklaration
    Demo(void);                   //3. überladene Deklaration
};

.......
Demo obj1 = Demo(17, 3.14);       //explizit
Demo obj2(22, 2.78);              //verkürzt
Demo obj3 (33);                   //verkürzt
Demo obj4 = 7.7;                  //speziell bei einem Argument
Demo obj5 = 10;                   //speziell bei einem Argument
Demo obj6;                        //kein Argument
Demo obj7( );                     //Fehler: Deklaration einer Funktion
```

Spezielle Konstruktoren: Standard-Konstruktor

Der Standard-Konstruktor enthält nur void als Formalparameter oder Formalparameter, die mit Standardwerten vorbelegt sind. Der Standard-Konstruktor wird vom Compiler, wenn vorhanden, im Gegensatz zu anderen Konstruktoren zu spezifischen Aufgaben (z.B. bei der Initialisierung von Objektfeldern) eingesetzt.

Beispiel:

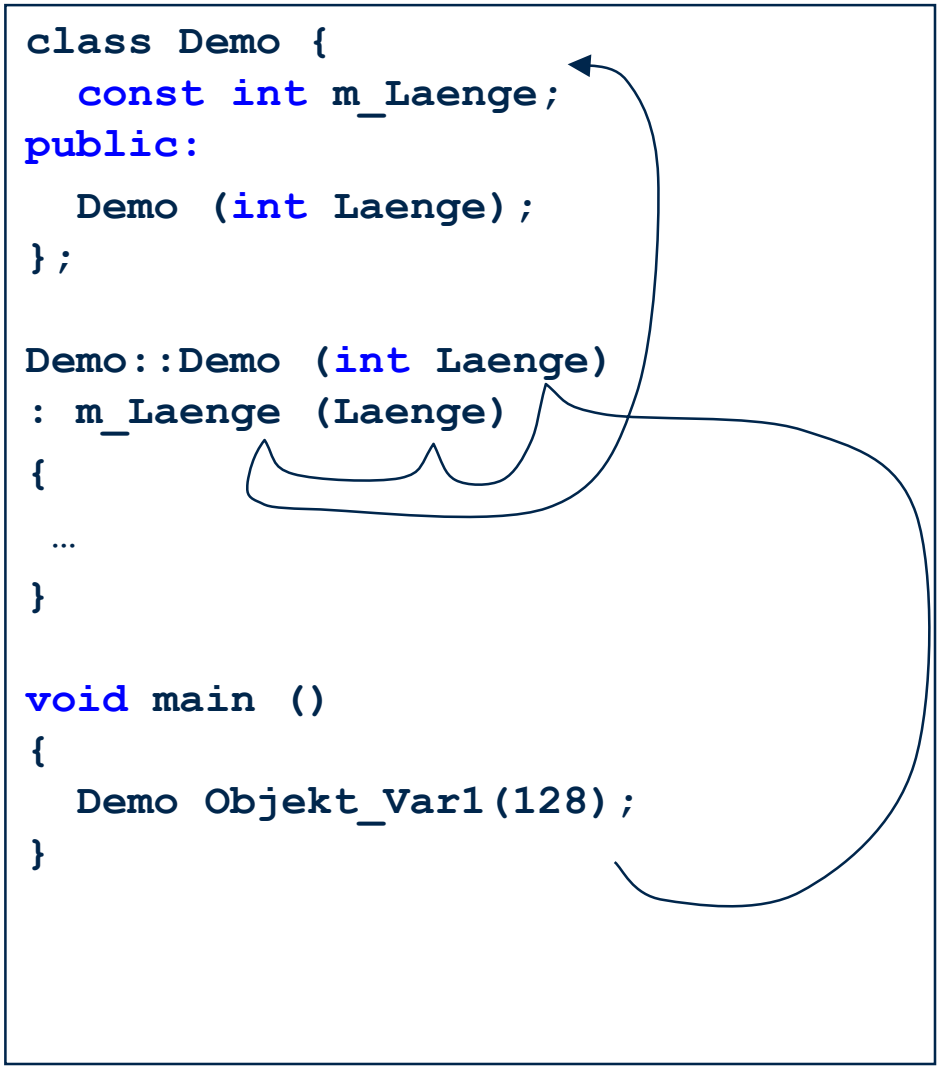
```
class Demo
{
private:
    int i;
    double x;
public:
    Demo(void) {i = 0; x = 0.;}                  //Standard-Konstruktor
    Demo(int k=0, double y=0.) {i = k; x = y;}   //auch Standard-
                                                 //Konstruktor
};
```

Empfehlung:

Fügen Sie immer einen Standard-Konstruktor in die Definition der Klasse ein. Sie vermeiden Probleme mit Feldern von Objekten.

Attributinitialisierung aus Konstruktorparameter

- typischer Fall zur Initialisierung von Attributen
- const Attribute können nur so initialisiert werden !
- bedenke: die Klassendeklaration ist nur eine Schablone, kein ausführbarer Code
  - ausgeführt wird der Konstruktor für jedes Objekt
- Konvention für Namen (sehr zu empfehlen):
  - Attribute beginnen mit m_
  - Parametername = Attributname ohne m_



Default-Parameter

- Zuordnung von Standardwerten zu formalen Parametern
  - nur in der Prototypdeklaration
  - nicht in der Definition des Funktionskörpers
- für alle Arten von Funktionen, auch für Konstruktoren
- die letzten aktuellen Parameter dürfen beim Aufruf weggelassen werden
  - Parameter ohne Default-Werte immer vor Param. mit Default

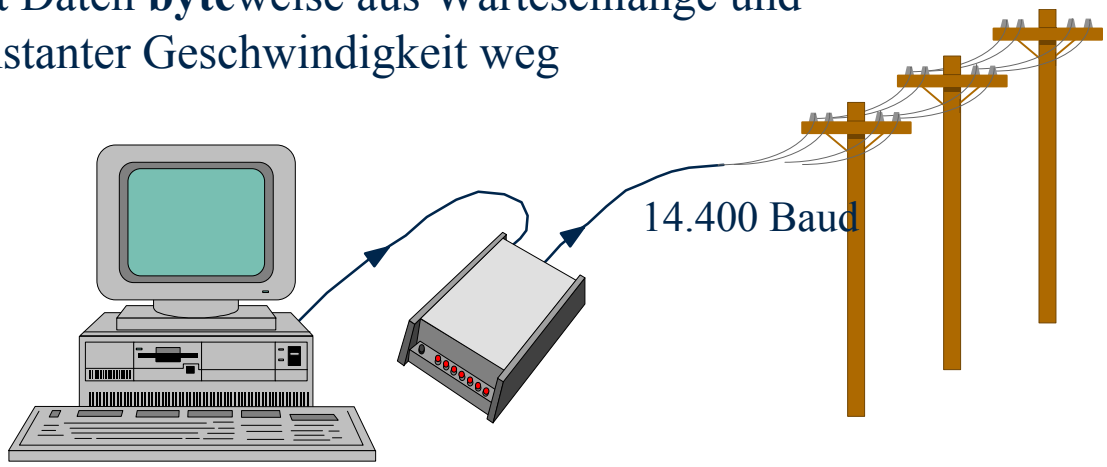
```
class Demo {
   const int m_Laenge;
public:
   Demo (int Laenge=0);
};

Demo::Demo (int Laenge)
   : m_Laenge (Laenge)
{
 ...
}

   void main ()
   {
     Demo Objekt_Var1(128);
   }
```

Am Beispiel FIFO (Warteschlange)

Abstraktion und Kapselung von Details

- Funktionsprinzip: first in, first out (FIFO)

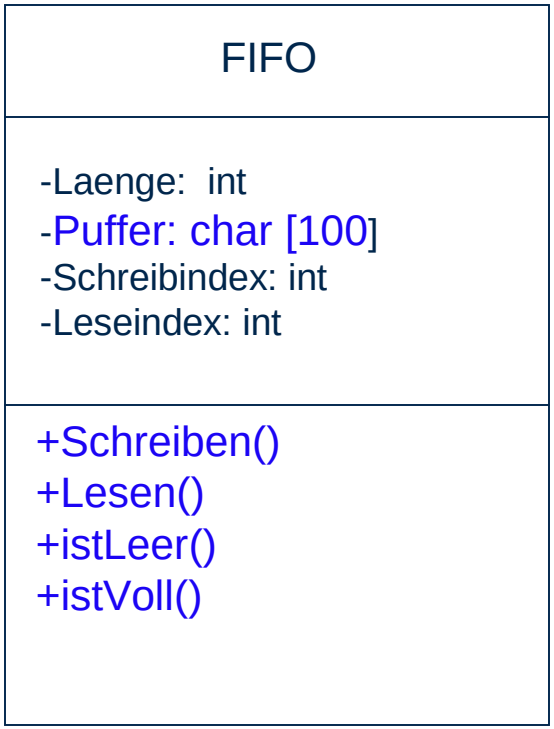
- Anwendung: Erzeuger / Verbraucher - Systeme
  - Computer überträgt Datei via Modem
    - Transferprogramm liest Daten **block**weise von Festplatte und schreibt sie in die Warteschlange
    - Modemtreiber liest Daten **byte**weise aus Warteschlange und schickt sie mit konstanter Geschwindigkeit weg



FIFO - Klassendiagramm

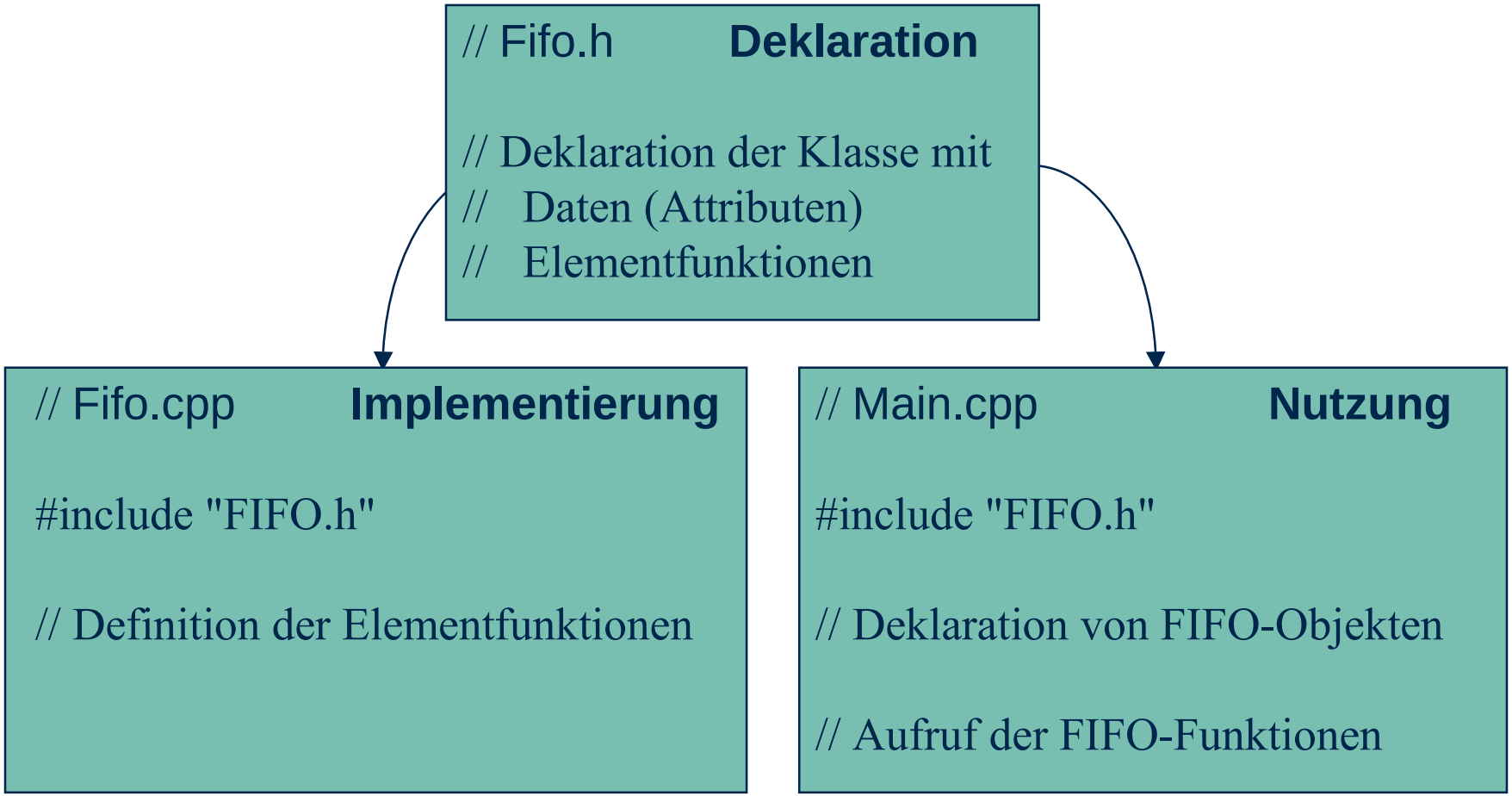
- Welche Funktionen stellt die Warteschlange zur Verfügung:
  - Schreiben()
  - Lesen(),…
- Dies bilden das sichtbare "Interface" der Klasse (+ in UML)
- Welche Daten gehören zum FIFO Puffer:
  - sicher der Puffer
  - evtl. zusätzliche Hilfsattribute

  Wichtig: Die Implementierung muss nicht sichtbar sein



FIFO: Definition und Nutzung

in größeren Systemen



FIFO: Deklaration

typischerweise in Headerdatei

- Klassendeklaration
  - Einführung des Namens
    - gilt fürderhin als Typname
  - Liste von Elementen
- Elemente (members)
  - Attribute (Daten)
    - guter Stil: private
  - Elementfunktionen
- Zugriffsrechte
  - private: Zugriff nur für eigene Elementfunktionen

    Informationen sind gekapselt, aber leider nicht verborgen
  - public: Zugriff für alle Nutzer

Fifo.h

```
class FIFO {
private: // Standard
  const int Laenge;
  char Puffer [100];
  int freierSchreibIndex;
  int geleerterLeseIndex;
  int Inkrement (int Index);
public:
  FIFO ();  // Konstruktor
  ~FIFO (); // Destruktor
  void Schreiben (char Byte);
  char Lesen ();
  bool istLeer ();
  bool istVoll ();
};
```

hier wird noch kein Speicher allokiert !

Namen gleich Klassenname; kein Ergebnis

Deklaration ist Anweisung und muß mit ; abgeschlossen werden

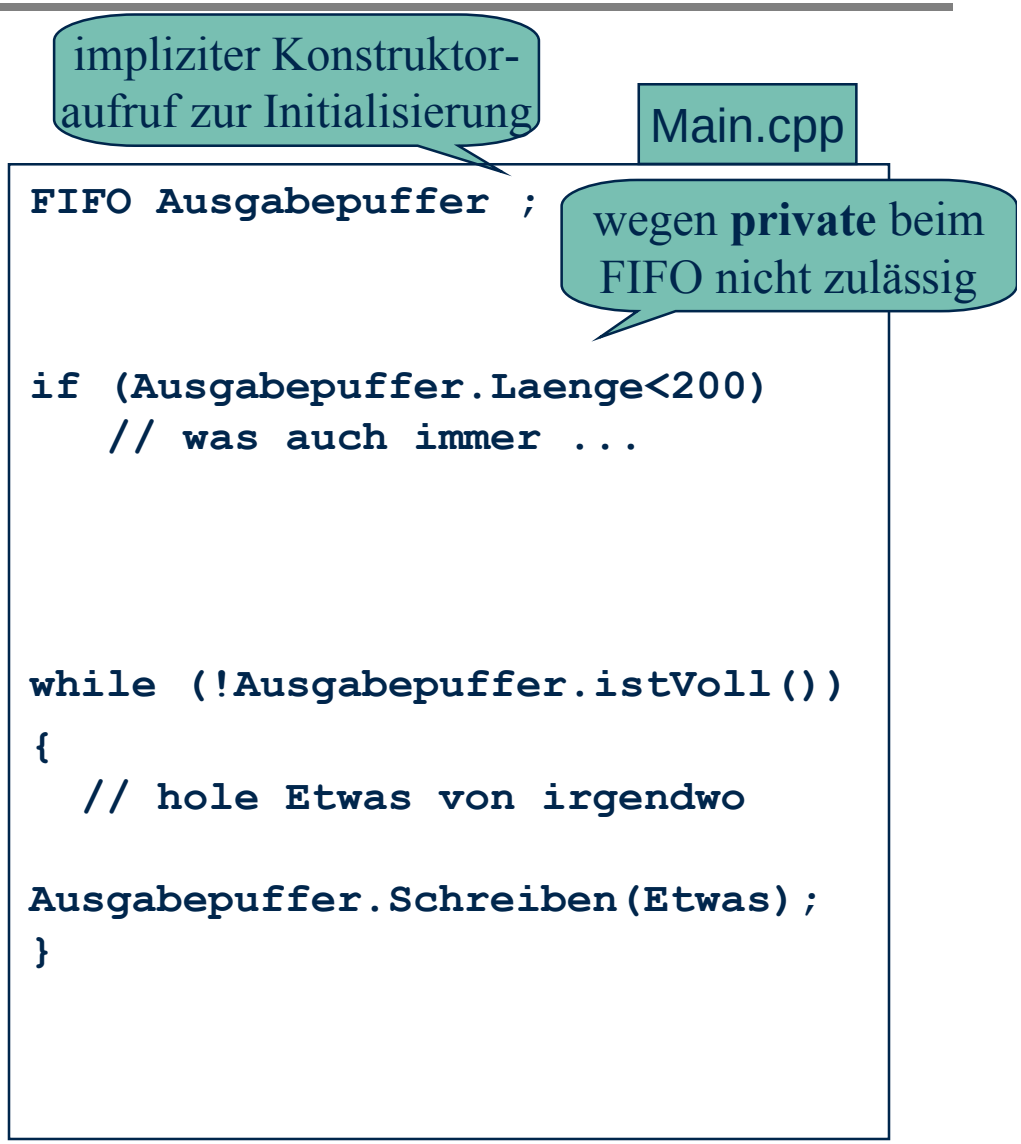
FIFO: Nutzung

- Definition eines Objekts
  - allokiert Speicher

  Klassenname Objektname
  (*aktuelleKonstruktorparameter*);
- Zugriff auf Attribute
  - vorzugsweise durch `private` unterbinden

  Objektname . Attributname
- Aufruf von Elementfunktionen

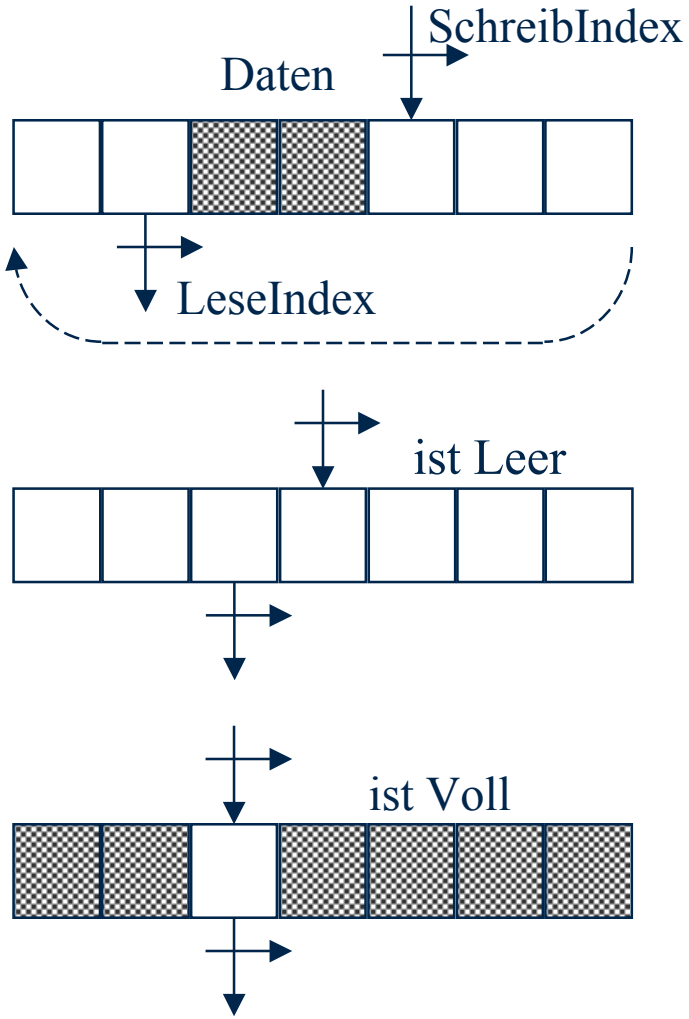
  Objektname . Funktionsname
  (*aktuelleParameter*);




```
FIFO Ausgabepuffer ;


if (Ausgabepuffer.Laenge<200)
   // was auch immer ...



while (!Ausgabepuffer.istVoll())
{
  // hole Etwas von irgendwo

Ausgabepuffer.Schreiben(Etwas);
}
```

FIFO: Implementierungsidee

- Puffer ist ein Array, in das zyklisch geschrieben wird
- ein LeseIndex läuft hinter einem SchreibIndex her
- SchreibIndex zeigt auf das nächste freie Element
- LeseIndex zeigt auf das wieder freie Element
- gültige Daten zwischen den beiden Indizes
- Inkrementierung eines Index modulo Arraylänge
- Füllstand an Indizes erkennbar



FIFO: Definition ausgelagerter Elementfunktionen

- Was passiert wenn der Puffer überläuft ?
  - Erste Näherung (assert())
- Hilfsfunktion Inkrement um der linearen Puffer zu nutzen
  - %Pufferlänge

Fifo.cpp

```
void FIFO::Schreiben (char Byte)
    {
     assert (!istVoll());
     Puffer[freierSchreibIndex]=Byte;
     freierSchreibIndex=
       Inkrement(freierSchreibIndex);
    }
```