# Inhaltsverzeichnis

# 1. Grundlagen der Programmiersprache C++

## 1.1 Streams

a) Output streams

```
cout      cout<<“Heute ist Freitag, es sind“
              <<18.5<<“Grad\n“
          cout<<“Ergebnis:“<<i<<endl;      // endl wie \n
          cout<<“jetzt hexadezimal:“<<hex<<i<<endl;
```

weitere: `dec,oct,setw(n),setfill('*'),setprecision(k)`

➔ Manipulatoren beziehen sich nur auf die direkt nachfolgende Ausgabe, jedoch bleiben hex,dec,oct auch danach gültig.

```
cerr
```

b) Input streams

```
cin       cin>>i;
```

Beispiel:

```
#include <iostream.h>

void main()
{
  char Name[20];

  cout<<“Name:“;
  cin>>Name;
  cout<<“Name:“<<Name;
}
```

## 1.2 Prototypen

- es muss für jede Funktion ein Prototyp vorhanden sein, wenn sie vor der Definition verwendet wird
- Funktionen mit variabler Argumentliste sind nicht erlaubt

## 1.3 Default-Funktionswerte

```
void myF(int i=5,double d=1.234)
{
  cout<<“i:“<<i<<“d:“<<d<<endl;
}

myF(7,9.876);       // i:7 d:9.876
myF(7);             // i:7 d:1.234
myF();              // i:5 d:1.234
```

➔ Die Funktionsparameter sind nur von rechts nach links weglassbar!

## 1.4 Includes

```
#include <iostream.h>          // Old Style
#include <iostream>

using namespace std;
```

```
namespace WBusch
{
  int Max;
    ...
}

WBusch::Max;
```

➔ Verwendung von mehreren Bezeichnern gleichen Namens möglich, wenn sie sich in unterschiedlichen Namespaces befinden.

```
namespace A=WBusch;
A::Max;

#include <cstdlib>        // entspricht <stdlib.h>
```

## 1.5 Variablendefinitionen

- an beliebiger Stelle innerhalb einer Funktion möglich
- auch in Schleifen

```
for(int i=0;i<N;i++)
{
  //irgendwas mit i
    ...
}

while(...)
{
  long l1;
  int i1;
    ...
}
```

## 1.6 Überladene Funktionen

Es können mehrere Funktionen gleichen Namens, aber mit unterschiedlicher Parameterliste angelegt werden:

```
int Abs(int i)       {return abs(i);}
long Abs(long l)     {return labs(l);}
double Abs(double d) {return fabs(d);}

x = Abs(3.145);
    ...
int i;
    ...
y = Abs(i);
```

➔ Der Compiler sucht sich die passende Funktion entsprechend der Datentypen heraus.

```
int Max(int i, int j) {return(a>b?a:b);}
int Max(int i, int j, int k)
{
  int x = i>j?i:j; return(k>x?k:x);
}
```

**oder {return Max(Max(i,j),k);}**

```
int main()
{
  cout<<Max(1,100)<<'\n';
  cout<<Max(1,2,3)<<'\n';
  return 0;
}
```

ABER:

```
int f1(int i, int j=77);
int f1(int i);                  // → Compilerfehler!!!
```

## 1.7 Inline-Funktionen

```
inline int f()
{
    ...
}
```

➔ Programmcode der Inline-Funktion wird direkt an die entsprechende Stelle innerhalb des Hauptprogramms auf Laufzeitebene eingefügt (ähnlich Makros), sonst aber wie normale Funktion.

## 1.8 Typisierte Konstanten

```
const int N=10;

char vC[N];

cout<<"size of vC:"<<sizeof vC<<endl;     // 10
```

## 1.9 const in Verbindung mit Pointern

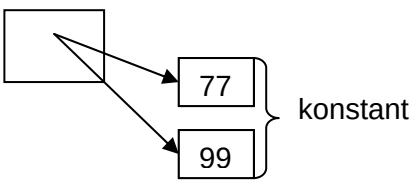
- const vor dem Typ in einer Vereinbarung

```
(const int *pI) Pointer auf Konstante (1)
```

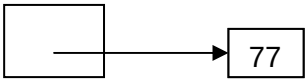
- const nach dem Typ in einer Vereinbarung

```
(int *const pI) Konstanter Pointer    (2)
```

Bsp.: `(1)  i = *pI;`




```
(2)  *pI = *pI+5;   // Wert ändern erlaubt!
     pI++;          // Pointer erhöhen verboten!
```




- konstanter Pointer auf eine Konstanten

```
(const int *const pi)
```

Initialisierung von Konstanten

```
const double *zk;
double d;
zk = &d;
d = 3.1415;
*zk = 3.1415;       // Fehler!
```

➔ Die Variable selber kann geändert werden, aber nicht über den Pointer (letzte Zeile), da dieser der Vereinbarung nach auf eine Konstante zeigt.

```
int i;

const int iz;       // falsch, Konstanten müssen bei der Vereinbarung
                    // initialisiert werden, z.B. const int iz=1;
const int *zik;
int *const kz;      // falsch, konstanter Pointer muss ebenfalls bei der
                    // Vereinbarung initialisiert werden
const int *const kzik;   // falsch, muss auch init. werden


int j = 'a';
const int jk = j;
const int *zjk = &jk;
int *const kzj = &j;
const int *const kzjk = &jk;

j = jk;
zjk = &jk;
kzj = zjk;          // falsch, da kzj ein konstanter Pointer ist!
zjk = kzjk;
kzjk = &jk;
jk = *kzjk;         // falsch, da jk eine Konstante ist!
```

## 1.10 Referenzen

Referenzen stellen einen neuen Zugang zu einem bereits existierendem Objekt dar.

Zugriff bisher über: Namen, Pointer

➔ Falls die Variable bereits einen Namen hat, ist die Referenz ein Synonym.

Bsp.:
```
int DrJekyll;
int& MrHyde = DrJekyll;
```

➔ Variable ist trotzdem nur ein einziges Mal im Speicher vorhanden
➔ alle Änderungen an den Referenzen einer Variable werden auch am Original vollzogen (auch anders herum)

Syntax: `<Typname>&`  `//& nach dem Typnamen`

➔ Referenz ist grundsätzlich bei der Vereinbarung zu initialisieren!

Ausnahmen:
- bei Verwendung als Parameter in einer Funktion
- als Returnwert

```
const int& MrHyde = DrJekyll;
```

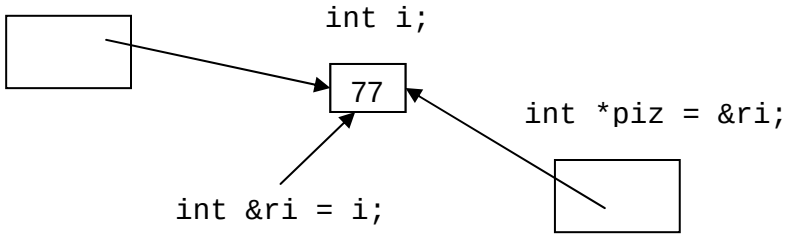
➔ DrJekyll nicht über MrHyde veränderbar, da Konstante!

```
const int DrJekyll;
int& MrHyde = DrJekyll;        // Fehler!!!
```

Beispiel:

```
int main()
{
  int actint = 123;
  int& other = actint;

  cout<<actint<<endl;     // 123
  cout<<other<<endl;      // 123

  other++;

  cout<<actint<<endl;     // 124
  cout<<other<<endl;      // 124

  actint++;

  cout<<actint<<endl;     // 125
  cout<<other<<endl;      // 125

  return 0;
}
```

```
  int *pi = &i;
```




➔ Pointer ist ein eigenes Datenobjekt, die Referenz nicht
➔ es ist nicht möglich, einen Pointer auf eine Referenz anzulegen, dieser wird dann automatisch auf das Original der Referenz gesetzt.

Referenzen als Funktionsparameter

```
struct s
{
  int num;
  char text[1000+1];
};

s so = {123,“struct muss in C++ nicht mehr angegeben werden!“};
```

```
void valFunc(s Value)
{
  cout<<Value.num<<endl;       // lokale Variablen
  cout<<Value.text<<endl;
  Value.num = 0xFFFF;
}

void ptrFunc(s *pValue)
{
  cout<<pValue->num<<endl;
  cout<<pValue->text<<endl;
  pValue->num = 0xFFFF;        // Wert wird über Pointer geändert!
}

void refFunc(s& rValue)
{
  cout<<rValue.num<<endl;
  cout<<rValue.text<<endl;
  rValue.num = 0xFFFF;         // Wert wird über Referenz geändert!
}

int main()
{
  valFunc(so);                 // (1)
  ptrFunc(&so);
  refFunc(so);                 // kein Unterschied beim Aufruf zu (1)

  return 0;
}
```

Referenzen als Returnwerte

```
int myNum = 0;

int& num()
{
  return myNum;
}

int main()
{
  int i;

  i = num();          // i erhält über den Aufruf von num() den Wert von
                      // myNum!
  num()=5;            // Referenzfunktion kann als gültiger L-Wert
                      // verwendet werden, trägt hier 5 in myNum ein!
  cout<<myNum<<endl;

  return 0;
}
```

Beispiel:

```
#include <iostream>
```

```
int& f(int *vI, int i)
{
  return vI[i];
}

int main()
{
  const int N=10;
  int v1[N]={0};
  int v2[N]={0};

  for(int i=0;i<N;i++)
    f(v1,i) = f(v2,i) = N-i;

  for(int i=0;i<N;i++)
    cout<<f(v1,i)<<"..."<<f(v2,i)<<endl;

  return 0;
}
```

## 1.11 Objektorientierte Programmierung

➔ Verbesserung der Qualität / Effektivität der Programmierung

Allgemein gibt es folgende Arten der Programmierung:
- imperative Programmierung (Assembler) ➔ befehlsorientiert, sequentiell
- prozeduale Programmierung ➔ Modularisierung, jede Routine löst Teilproblem
- objektorientierte Programmierung (s. unten)

Schaffung von sog. **Klassen** für bestimmte Sachverhalte (zentraler Bestandteil):
- benutzerdefinierter Datentyp
- beschreibt den Status (Wertebelegung aller Variablen) und das Verhalten (alle Funktionen)

➔ von „außen“ kein direkter Zugriff auf Variablen/Werte, sondern Steuerung nur über Funktionen
➔ Daten können öffentlich (**public**) oder verborgen (**private**) sein

Beispiel (Vergleich C / C++):

```
struct tDate
{
    int Year;
    int Month;
    int Day;
};

void setDate(tDate *pDate,int Y,int M, int D);
void displayDate(tDate *pDate);
```

C

```
 (1)     (2)
struct tDate
{
      int Year;
      int Month;      } (3)
      int Date;
      void setDate(int Y,int M,int D);  } (4)
      void displayDate();
};
```

C++

1) class key ➔ entweder struct oder class möglich
2) class name oder Vererbungsbeziehung
3) Memberdaten
4) Memberfunktionen (Methoden)

```
tDate today;              // Objekt der Klasse tDate erzeugt

today.setDate(2004,4,16); // Funktionsaufrufe mit Punktoperator (.)
today.Day = -47;          // darf eigentlich nicht passieren!
```

➔ daher:

```
class Date
{
 private:
    int Year;
    int Month;
    int Day;
 public:
    void setDate(int Y,int M,int D);
    void displayDate();
};
```

- eigentliche Daten sind verborgen (private), Zugriff nur über Funktionen möglich
- bei “struct“ als class key ist alles public, wenn nicht anders ausgewiesen
- bei “class“ hingegen ist alles private, wenn nicht anders ausgewiesen
- sonst kein Unterschied

```
void Date::setDate(int Y,int M,int D)
{
    Year = Y;
    Month = M;
    Day = D;
}

void Date::displayDate()
{
    cout<<Day<<”.”<<Month<<”.”<<Year;
}
```

➔ Die Memberfunktionen zum Zugriff auf die Daten sollten möglichst so programmiert werden, dass keine ungültigen Daten eingegeben werden können.

Beispiel:

```
void Date::setDate(int Y,int M,int D)
{
    static int DaysPerMonth[]=
      {0,31,28,31,30,31,30,31,31,30,31,30,31};

    Month = max(1,M);
    Month = min(12,Month);
    Day = max(1,D);
    Day = min(Day,DaysPerMonth[Month]);
    Year = max(1,Y);
}
```

➔ zur ordentlichen Initialisierung von Daten sollte ein **Konstruktor** innerhalb der Klasse vorhanden sein:

```
class Date
{
 private:
    int Year;
    int Month;
    int Day;
 public:
    Date(int i,int M,int D);        // Konstruktor, hat Namen der Klasse

    void setDate(int Y,int M,int D);
    void displayDate();
};

Date::Date(int Y,int M,int D)
{
    static int DaysPerMonth[]=
      {0,31,28,31,30,31,30,31,31,30,31,30,31};

    Month = max(1,M);
    Month = min(12,Month);
    Day = max(1,D);
    Day = min(Day,DaysPerMonth[Month]);
    Year = max(1,Y);
}
```

➔ Initialisierung von Daten dann nur noch über diesen Konstruktor:

```
Date toDay(2004,4,16);
Date DayX;                // Fehler, da es dafür keinen gültigen
                          // Konstruktor gibt
```

- bei der Initialisierung von Objekten muss sich nach dem vorhandenen Konstruktor gerichtet werden, wenn einer vorhanden ist
- für die Initalisierung eines Objektes ohne Daten (`Date DayX;`) muss ein Default-Konstruktor vorhanden sein, sonst Fehler
- „kleine“ Funktionen können auch innerhalb einer Klasse ausprogrammiert werden

➔ Destruktor zum sauberen Abbauen eines Objektes

```
~Date();

Date::~Date()
{
 ...
}
```

Anschauliches Beispiel

```
#include <iostream>
#include <string.h>

using namespace std;

class demo
{
 public:
    demo(const char *txt);
    ~demo();
 private:
    char name[30];
};

demo::demo(const char *txt)
{
    strncpy(name,txt,30);
    cout<<”Constructor called for “<<name<<endl;
}

demo::~demo()
{
    cout<<”Destructor called for “<<name<<endl;
}

void func()
{
    demo localObject(“localFuncObject”);
    static demo localStaticObject(“localFuncStaticObject”);
    cout<<”End of localFunc”<<endl;
}

demo GlobalObject(“GlobalObject”);

void main()
{
    demo localMainObject(“localMainObject”);
    cout<<”in main 1”<<endl;
    func();
    cout<<”in main 2”<<endl;
    func();
    cout<<”End of main”<<endl;
}
```

Ausgaben:

```
Constructor called for GlobalObject
Constructor called for localMainObject
in main 1
Constructor called for localFuncObject
Constructor called for localFuncStaticObject
End of localFunc
Destructor called for localFuncObject
in main 2
Constructor called for localFuncObject
End of localFunc
Destructor called for localFuncObject
```

```
End of main
Destructor called for localMainObject
Destructor called for localFuncStaticObject
Destructor called for GlobalObject
```

Zugriff auf Datenfelder

```
class Date
{
 public:
    Date(int Y,int M,int D);
    int getDay(){return Day;}
    int getMonth(){return Month;}
    int getYear(){return Year;}

    void setDay(int d);
    void setMonth(int m);
    void setYear(int y);
 private:
    int Day;
    int Month;
    int Year;
};
```

➔ Klasse enthält einen Konstruktor und getrennte Funktionen zur Eingabe und Ausgabe
➔ eigentliche Daten sind private

```
int &getDay(){return Day;}
getDay() = 77;                  // <----- Gefahr!!!
```

➔ große Gefahr bei Verwendung von Referenzen in Funktionen innerhalb des public-Bereichs, da diese dann auch als gültiger L-Value auftreten können und somit direkter Zugriff auf die eigentlichen Daten besteht.

```
class X
{
 private: int privateint;
 public:  int publicint;
}Objx;

int *pI;

pI = &Objx.publicint;
pI = &Objx.privateint;          // Fehler, Pointer auf private object

int x::*pI;
```

(Pointer `pI` darf nur auf ein Integerobjekt innerhalb der Klasse `X` zeigen)

Konstante Objekte

➔ Vereinbarung von Objekten als Konstanten

```
class Date
{
    ...
    const Date myDate(30,4,2004);
    ...
};
```

Beispiel:

```
class Date
{
    int d,m,y;
 public:
    Date(int pd,int pm,int py)
    {
       d=pd; m=pm; y=py;
    }

    void setDate(int pd,int pm,int py)
    {
       d=pd; m=pm; y=py;
    }

    void display()const                    // typische const-Funktion
    {
       cout<<d<<".”<<m<<".”<<y;
    }
};
```

➔ möglichst alle Funktionen, die Memberdaten nicht verändern, sollten als const vereinbart sein!
➔ Funktionen gleichen Namens und gleicher Parameterliste, eine normal und eine const, möglich! (Aufruf entsprechend der Vereinbarung des Objektes)

Member-Objekte

```
class Pers
{
 public:
    Pers(char *name,char *addr,int BD,int BM,int BY);
 private:
    char Name[30];
    char Addr[40];
    Date Birthday;        // Member-Objekt der Klasse Date (s.o.)
};

Pers P(“Hans Meier“,“Maxstr.7, 08150 Huckebeinhausen“,1,7,2000);
```

➔ In diesem Fall wird zuerst der Default-Konstruktor der Klasse `Date` genommen, um das Objekt P zu erzeugen und danach erst werden die Konstanten initialisiert.
➔ Damit kein undefiniertes Verhalten auftritt, ist es zweckmäßig, bei der Funktionsdeklaration den sog. **Memberinitialisierer** anzugeben, z.B.:

```
Pers(char *name,char *addr,int BD,int BM,int BY)::Birthday(BD,BM,BY);
```

## 1.12 Dynamischer Speicher

➔ **new-Operator** gibt einen Pointer auf den neuen Speicher zurück (wie malloc in C)
➔ Operand ist der Typ des zu erstellenden Objektes

Beispiel:

```
Date *pFirst, *pSecond;

pFirst = new Date;  // Speicher bereitstellen und Default-Konstruktor
                    // der Klasse Date aufrufen!
```

```
pSecond = new Date(30,4,2004);
```

➔ bei Nichterfolg wegen Speichermangel wird NULL zurückgegeben
➔ **delete-Operator** zum Freigeben des Speichers

```
delete pFirst;      // kann auch auf NULL-Pointer angewandt werden!

int length = ...;
int *pInts = new int[length];

delete []pInts;
```

➔ Die Größe von Arrays beim Erzeugen kann dynamisch über Variablen eingetragen werden!

Besonderheit beim Bereitstellen von Speicher in C++

➔ `set_new_handler` Funktion bekommt einen Funktionspointer der Form `int(*f)(size_t size)`übergeben (`#include <new>`)
➔ die als Parameter angegebene Funktion wird immer dann aufgerufen, wenn beim new-Operator ein Fehler auftritt
➔ es können auch mehrere Fehlerbehandlungsfunktionen im Programm vorhanden sein, nur muss dann `set_new_handler` an unterschiedlichen Stellen immer mit der jeweils aktuellen Fehlerbehandlungsfunktion neu gesetzt werden

Beispiel:

```
int MemError(size_t size)
{
    cerr<<size<<" Bytes not available"<<endl;
    exit(-99);
    return 0;
}
```

## 1.13 Klassen mit Pointermembern

➔ Beispiel einer Klasse "String" zur Erstellung und Bearbeitung von Strings

```
class String
{
    int len;
    char *pBuf;

  public:
    String() {len = 0; pBuf = NULL;}
    String(const char *s)
    {
          len = strlen(s);
          pBuf = new char[len+1];
          strcpy(pBuf,s);
    }
    String(char C,int n=1)     // n Zeichen werden mit 'C' aufgefuellt
    {
          len = n;
          pBuf = new char[len+1];
          memset(pBuf,C,len);
          pBuf[len]=0;
    }
```

```
    void set(int i, char C)
    {
         if (i<len) pBuf[i]=C;
         else cerr<<"IndexError"<<endl;
    }

    char get(int i)const
    {
         if (i<len) return pBuf[i];
         else {cerr<<"IndexError"<<endl; return 0;}
    }

    void display()const {cout<<pBuf<<endl;}
    ~String(){delete []pBuf;}
};

int main()
{
    String s("Max");
    s.display();              // Max wird ausgegeben
    s.set(1,'u');             // Max wird in Mux geaendert
    char c = s.get(2);        // 'x' von Max wird an c uebergeben
    int sizeS = sizeof s;     // moeglicherweise 8
    String *pS = new String ("Max");
    String x;
    x = s;                    // Memberweises Kopieren der Objekte

    return 0;
}
```

➔ Probleme beim memberweisen Kopieren (x=s) beim Abbau eines der beiden Objekte, da dann auch das Ziel des Pointers PtStr gelöscht bzw. freigegeben wird (irgendwann undefiniertes Verhalten zur Laufzeit).

➔ „neuer" Zuweisungsoperator "="

```
class Cstring
{
    ...
 public:
    void operator =(const Cstring &other)
    {
      len   = other.len;
      delete PtStr;
      PtStr = new char[len+1];
      strcpy(PtStr,other.PtStr);
    }
};
```

➔ wieder Problem, da bei Zuweisung desselben Objektes (S1=S1 bzw. S1.operator=(S1)) ein undefiniertes Verhalten auftritt, daher besser:

```
void operator =(const Cstring &other)
{
    len = other.len;
    if (&other!=this)
    {
      delete PtStr;
      PtStr = new char[len+1];
      strcpy(PtStr,other.PtStr);
    }
}
```

➔ `this` gibt die Adresse des aktuellen Objektes zurück, die Zuweisung wird hierbei also nur ausgeführt, wenn die beiden Objekte nicht die gleichen sind!
2. Variante:

```
void operator =(const Cstring &other)
{
    char *tmp;                        // temporäres Element
    len = other.len;
    tmp = new char[len+1];
    strcpy(tmp,other.PtStr);
    delete PtStr;
    PtStr = tmp;
}
```

➔ um auch z.B. S2=S3=S1 ausführen zu können, Referenzen verwenden:

```
Cstring &operator =(const Cstring &other)
{
    ...
    return *this;
}
```

Beispiel:

```
CString S1("Max");
CString S2(S1);
CString S2=S1;
```

➔ funktioniert nicht, da C++ bei Initialisierungen nicht die (operator =)-Funktion benutzt, sondern einen speziellen Konstruktor für diese Kombination erwartet, daher:

```
class Cstring
{
    ...
    Cstring(const Cstring &other)        // Kopierkonstruktor
    {
      len = other.len;
      PtStr = new char[len+1];
      strcpy(PtStr,other.PtStr);
    }
    ...
};
```

➔ Klassen, die Resourcen verwalten, benötigen immer einen **Destruktor**, **Kopierkonstruktor** und die **(operator =) Funktion** !
➔ es wird in C++ deutlich unterschieden zwischen Initialisierung und Zuweisung (siehe oben)!

## 1.14 Friends

➔ Klassen oder Funktionen können als **Friend** der eigenen Klasse bestimmt werden
➔ Friends können dann auf die privaten Funktionen/Daten der eigenen Klasse zugreifen

```
class myClass
{
  friend class yourClass;
  friend void change(my Class &x);

  private:
    int mySecret;
};

class yourClass
{
  public:
    void ChangeIt(myClass &x)
    {
      x.mySecret++;
    }
};

friend void change(my Class &x)
{
    x.mySecret+=7;
}
```

➔ eigenständige Friend-Funktion, die das zugehörige Klassenobjekt als Parameter übergeben bekommt

➔ Friend-Funktionen müssen immer in der Klasse deklariert sein, deren Friend sie sein sollen!

## 1.15 Arrays von Objekten

```
Date Birthdays[10];

Date BirthDays[10] = {Date(2,10,1950),
                      Date(9,6,1953),
                      Date(21,12,1955),
                      Date(3,2,1960)
                     };
```

➔ 4 von 10 Objekte werden mit Werten initialisiert, der Rest wird mit dem Default-Konstruktor der Klasse “Date“ initialisiert
➔ existiert kein Default-Konstruktor in der Klasse “Date“, tritt ein Fehler auf!

Beispiel:

```
Cstring *Text;
Text = new Cstring[5];
    ...
delete []Text;
```

## 1.16 Static Member / Static Methoden

```
class Konto
{
  private:
    char *Kunde;
    char *Ktonr;
    static double Zinssatz;
    ...
};
```

➔ keine Initialisierung von static-Objekten innerhalb der Klasse, sondern:

```
static double Konto::Zinssatz = 2.3;      // ausserhalb, im C++ File
```

➔ muss nur 1x zentral initialisiert werden, bei Änderung ändert sich der Wert für alle Objekte!
➔ Static Member sind unabhängig vom jeweiligen Objekt
➔ Beispiel: Objektzähler

➔ Static Methoden können nur auf Static Member zugreifen und nicht auf die Daten selber
➔ können bereits aufgerufen werden, wenn es noch garkeine Objekte der Klasse gibt

## 1.17 Nützliches Beispiel zu “this“

```
class myClass
{
    int a,b;
  public:
    myClass(int a,int b)
    {
      this->a = a;
      this->b = b;
    }
};
```

➔ “this“ bezieht sich immer auf die eigene Klasse
➔ Verwendung von gleichen Namen der externen und privaten Variablen möglich

## 1.18 Überladene new- und delete-Operatoren

```
class Name
{
    Name();
    Name(...);

    void *operator new(size_t size);
    void  operator delete(void *ptr);
};
```

➔ können dann beliebig selbst ausprogrammiert werden (Fehlerbehandlung, etc.)
➔ wenn ein Array von “Name“ erzeugt werden soll, wird nicht der new-Operator ausgeführt!

## 1.19 Operatorüberladung

- es können keine komplett neuen Operatoren erzeugt werden
- Operatorprioritäten bleiben erhalten
- Eigenschaften unär und binär können nicht verändert werden
- Assoziativität (links oder rechts) kann nicht verändert werden
- Bedeutung bezüglich eingebauter Datentypen lässt sich nicht verändern (z.B. aus + wird -)
- sollte immer in Übereinstimmung mit der ursprünglichen Bedeutung geschehen

binärer Operator als Memberfunktion, @ steht für einen beliebigen gültigen Operator (z.B. =):

```
erg_typ operator @(typ2 arg2);
```

Operator als Memberfunktion (z.b. – oder + als Vorzeichen):

```
erg_typ operator @();

friend erg_typ operator @(typ1 arg1,typ2 arg2);
friend erg_typ operator @(typ1 arg1);
```

Beispiel:

```
class Fraction
{
    ...
    Fraction operator +(const Fraction &other)const;
    {
         ...
    }
    ...
};
```

➔ Überladung des + Operators anstatt der Memberfunktion “add“

➔ + und += sind verschiedene Operatoren, sie müssen separat überladen werden!

```
Fraction FA,FB(2,3),FC(3,4);

FA = FB+FC;
FA = FB+10;
```

➔ nur dann erlaubt, wenn ein Konstruktor der Form `Fraction(long Num,long Den=1)` vorhanden ist, damit die 10 als 10/1 interpretiert wird.

```
FA = 10+FB;
```

➔ nur dann erlaubt, wenn spezielle Operator-Funktion für Integer-Werte vorhanden ist

```
friend Fraction operator +(int Op1,const Fraction &Op2);
```

➔ immer wenn der 1. Operand nicht vom Typ der eigenen Klasse ist, muss die Operatorfunktion eine friend-Funktion sein !!!

Beispiel:

```
class IntArr
{
  public:
    IntArr(int Len);
    int getLen()const;
    int &operator [](int idx);
    ~IntArr();
    IntArr(IntArr &other);
    IntArr &operator =(IntArr &other);
  private:
    int *pInt;
    int len;
};

int &IntArr::operator [](int idx)
{
    static int dummy = 0;

    if(idx >= 0 && idx < len)
         return pInt[idx];

    cerr<<”IndexError: “<<idx<<endl;
    return dummy;
}

int main()
{
    IntArr A(7);

    for (int i=0;i<7;i++)
         A[i] = i;           // Ausfuehrung des überladenen []-Operators

    for (int i=0;i<A.getLen();i++)
         cout<<A[i]<<endl;

    return 0;
}
```

Überladung des Output-Operators

```
friend &ostream operator <<(const ostream &OS,Cstring S)
{
    OS<<S.PtStr;
    return OS;
}
```

Konvertierungsoperator

- zum Konvertieren eines Objektes in einen bestimmten Datentyp

Beispiel:

```
class Fraction
{
    ...
    operator float()
    {
         return (float)(Num/Den);
    }
};

int main()
{
    float f;
    Fraction FA(2,3);

    f = FA.operator float();
    f = float(FA);
    f = (float)FA;
    f = FA;
}
```

## 1.20 Vererbung

Beispiel:

```
class Beschaeftigter                    // Basisklasse
{
    public:
         Beschaeftigter();              // Default-Konstruktor
         Beschaeftiger(const char*name); // Konstruktor Stammdaten
         char *GetName()const;          // Abfrage Stammdaten
    private:
         char Name[30];                 // Stammdaten
};

class Lohnempfaenger : public Beschaeftigter   //abgeleitete Klasse
{
    public:
         Lohnempfaenger(const char*name);
         void SetLohn(float wg);
         void SetStunden(float hr);
    private:
         float Lohn;
         float Stunden;
};

class Haendler : public Lohnempfaenger
{
    public:
         Haendler(const char*name);
         void SetKommission(float comm);
         void SetUmsatz(float Sales);
    private:
         float Kommission;
         float Umsatz;
};
```

```
int main()
{
    Beschaeftigter B(“Hans Lehmann“);
    Lohnempfaenger C(“Max Mueller“);

    // C ist sowohl vom Datentyp Beschaeftigter als auch Lohnempfaenger

    C.SetLohn(12.5);
    C.SetStunden(30);

    Beschaeftigter X;
    X = B;
    X = C;

    // aus dem Lohnempfaenger C wird ein normaler Beschaeftigter

    C = X;

    // Gefahr!!! (undefinierte Daten, da Teile fehlen)
}
```

➔ abgeleitete Klassen können nicht auf private-Member der Basisklasse(n) zugreifen, aber sehr wohl auf die public-Member(-Funktionen)

➔ zum Zugreifen auf private Daten der Basisklasse:

```
class Lohnempfaenger : public Beschaeftigter
{
    Lohnempfaenger(char*name) : Beschaeftigter(name) {}
    Lohnempfaenger(char*name,float l,float s) : Beschaeftigter(name)
    {
         Lohn = L;
         Stunden = S;
    }
};
```

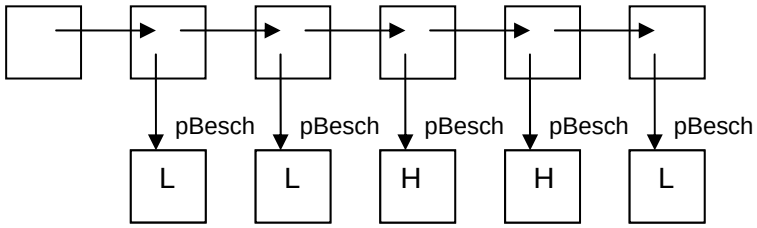
Beispiel mit Pointern:

```
Lohnempfaenger *pA;  // &Lohnempfaenger
                     // &Haendler
Haendler       *pH;  // &Haendler
Beschaeftigter *pB;  // &Beschaeftigter
                     // &Lohnempfaenger
                     // &Haendler

Lohnempfaenger A;

pB = &A;
pB->calcLohn();
```

➔ calcLohn() ist eine **überschriebene** Funktion in diesem Fall (siehe nächste Beispiele)

➔ Typ des Pointers bestimmt die auszuführende Funktion
➔ calcLohn() wird also in diesem Fall von der Klasse `Beschaeftigter` ausgeführt!!!

## 1.21 Dynamische Bindung – virtuelle Funktionen




Pointer ist stets vom Typ „Beschäftiger“, die Objekte sind entweder vom Typ „Lohnempfänger“ (L) oder „Händler“ (H)

➔ erst zur Laufzeit wird je nach Zielobjekt, auf das der Pointer aktuell zeigt, entschieden, welche Funktion (von welcher Klasse) ausgeführt wird

```
class Beschaeftigter
{
    public:
          ...
          virtual float calcLohn() {return 0.0F}
          // float calcLohn()=0; ➔ abstrakte Klasse, nur virt. Funktionen
};

class Lohnempfaenger : public Beschaeftigter
{
    public:
          ...
          virtual float calcLohn() {...};
};

class Haendler : public Lohnempfaenger
{
    public:
          ...
          virtual float calcLohn() {...};
};
int main()
{
    Lohnempfaenger L(“Max Lehmann“);

    Beschaeftigter *pB = &L;
    pB->calcLohn();
}
```

➔ es wird die `calcLohn()` Funktion der Klasse `Lohnempfaenger` aufgerufen !!!

➔ von abstrakten Klassen (siehe Klasse `Beschaeftigter`) können keine Objekte erstellt werden, sie können daher nur in einer Vererbungshierarchie verwendet werden (als Basisklasse)

➔ zum Abbau der Objekte entsprechenden virtuellen Destruktor nutzen:

```
virtual ~xyz() {...}
```

## 1.22 Protected

- Zugriff von einer abgeleiteten Klasse auf protected-Daten möglich, aber nicht von aussen
- protected ist „strenger“ als private, da es nicht erlaubt, auf die Daten eines Objekts der gleichen Klasse zuzugreifen, sondern nur auf die des aktuellen Objektes, mit dem gearbeitet wird (bei private jedoch möglich)

## 1.23 private / public – Ableitung

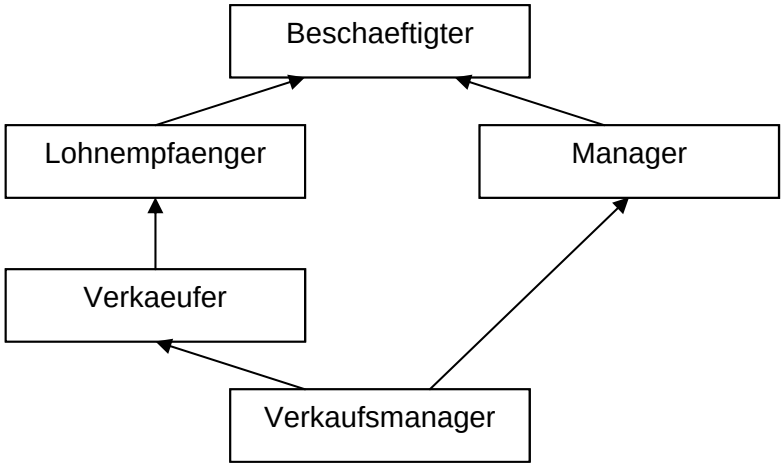
```
class xyz : public Basisklasse      // public-Ableitung
```

➔ public-Member der Basisklasse sind auch public-Member der abgeleiteten Klasse

```
class xyz : private Basisklasse     // private-Ableitung
```

➔ public-Member der Basisklasse sind jetzt private-Member der abgeleiteten Klasse und somit in weiteren Ableitungen nicht mehr bekannt!

## 1.24 Mehrfachvererbung




```
class Verkaufsmanager : public Verkaeufer, public Manager
{
    ...
};
```

➔ `Verkaufsmanager` erbt die Stammdaten von `Beschaeftigter` 2 mal (hat dann 2 Namen) !
➔ daher `virtual` bei den Klassen verwenden, die direkt von der Basisklasse erben:

```
class Lohnempfaenger : public virtual Beschaeftigter
{
    ...
};

class Manager : public virtual Beschaeftigter
{
    ...
};
```

## 1.25 Templates

- dienen der Generierung von Klassen, insbesondere von sog. „Containerklassen“, z.B. Vektoren, Hashtables, Listen
- sind intelligente Makros; Sichtbarkeits -und Typregeln werden berücksichtigt
- sind gewissermaßen eine Konstruktionsvorschrift für Klassen
- Typ wird als Klassenparameter aufgerufen
- jedem Template wird template <class T> vorangestellt
- innerhalb des Templates wird T als bekannter Datentyp verstanden
- Datentyp der generierten Klasse besteht aus TemplateName<Typ>

**Funktionen des Templates müssen in einem Header-File ausprogrammiert werden (entweder in demselben oder in einem anderen) !!!!**

## 1.26 Exceptions (Fehlerbehandlungsroutinen)

```
try      // Bereich, in dem eine Exception auftreten kann
{
    ...
    throw ExceptionA(“Fehler“);
    ...
}
catch(ExceptionA e1)
{
    ...
}
catch(int why)
{
    ...
}
catch(...)
{
    ...
}
```

# 2. Weiterführende Programmiertechniken

## 2.1 Listentechnik (Ringliste)

```
class LElem
{
    public:
         LElem(Object *pItem) {this->pItem = pItem;pNext=pPrev=NULL;}
        ~LElem() {delete pItem;}
         LElem() {pItem=pNext=pPrev=NULL;}
         LElem *getNext();
         LElem *getPrev();
         LElem *getItem();

         void setNext (LElem *Next);
         void setPrev (LElem *Prev);
         void setItem (Object *pItem);
    protected:
         LElem *pNext;
         LElem *pPrev;
         Object *pItem;
};
```

➔ damit einzelne Listenelemente ordnungsgemäß abgebaut werden können, virtuellen Destruktor in eigener Klasse verwenden:

```
class Object
{
    virtual ~Object() {}
    virtual int operator <(CObj *other);
    ...
};

class List : public LElem
{
    ...        // Listenkopf ist gleichzeitig ein leeres Listenelement
};
```

➔ intelligentes Listenelement bei einer Ringliste!
➔ Listenelement fügt sich immer hinter dem aktuellen Element ein

## 2.2 Einlesen von Zeichenketten mit Leerzeichen

```
cin>>myStr;         // es wird nur bis zum 1. Leerzeichen eingelesen
```

deshalb:

`get()` bzw. `getline()` des Inputstreams (`istream`) verwenden!