Technisches Programmieren in C++

Objektorientierte Programmierung

Karl Riedling
Institut für Sensor- und Aktuatorsysteme

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
- Abgeleitete Klassen
- Zugriff auf Klassen-Elemente
- Schablonen (*Templates*)
- Spezielle Funktionen in Klassen
- Überladen (*Overloading*)
- Konventionelle Programmierung und OOP

Objektorientierte Programmierung (OOP)

- **Eigenschaften von Klassen**
- Abgeleitete Klassen
- Zugriff auf Klassen-Elemente
- Schablonen (*Templates*)
- Spezielle Funktionen in Klassen
- Überladen (*Overloading*)
- Konventionelle Programmierung und OOP

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - **Klassen-Typen**
  - Klassen-Objekte
  - Definition von Klassen
  - Klassenelement-Funktionen
  - Datenelemente von Klassen
  - Verschachtelte Klassen

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Typen
    - **Allgemeines**
    - Anonyme Klassen
    - Definition einer Klasse

Klassen-Typen — Allgemeines

- *Klassen-Typen* sind jene, die mit den folgenden Schlüsselworten definiert werden:
  - `class`,
  - `struct` und
  - `union`
- Klassen-Typen können ineinander geschachtelt definiert werden; in diesem Fall gilt die Definition einer "inneren" Klasse nur innerhalb der sie umschließenden Klasse.

Klassen-Typen — Allgemeines

- Generell können alle Klassen-Typen beliebig viele der folgenden Elemente enthalten:
  - *Datenelemente*, die den Zustand und die Eigenschaften eines *Objekts* der Klasse beschreiben;
  - *Konstruktor-Funktionen*, die *neue Objekte* der Klasse initialisieren;
  - *Destruktor-Funktionen*, die "aufräumen", wenn ein Objekt nicht mehr benötigt wird;
  - *Klassenelement*-Funktionen, die das Verhalten eines Klassenobjekts bestimmen und für die betreffende Klasse spezifische Operationen vornehmen.

Klassen-Typen — Allgemeines

- Zugriff auf Klassenelemente

Eigenschaft	Strukturen	Klassen	*Unions*
Schlüssel-wort	`struct`	`class`	`union`
Standard-Zugriffsrecht	`public`	`private`	`public`
Zugriffsbe-schränkungen	keine	keine	immer nur ein Element

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Typen
    - Allgemeines
    - **Anonyme Klassen**
    - Definition einer Klasse

Anonyme Klassen

- "*Anonyme Klassen*" = Klassen ohne Angabe eines Klassennamens.
- Anwendungen:
  - `typedef`-Namensdefinitionen
  - Eingeschlossene Klassen

Anonyme Klassen

- `typedef`-Namensdefinitionen:

```
typedef struct
{
  short x;
  short y;
} POINT;
```


Anonyme Klassen

- Eingeschlossene Klassen:

```
struct PTWert
{
  POINT pt;
  union
  {
    int iWert;
    long lWert;
  };
};
```

Anonyme Klassen

- Eingeschlossene Klassen (Fortsetzung):

```
PTWert ptw;
int i = ptw.iWert;
POINT p = ptw.pt;
short j = ptw.pt.x;
```

- In C++ (nicht in ANSI-C!) ist ein Zugriff auf Elemente der *eingeschlossenen* Klasse ebenso möglich, wie wenn diese Elemente der *einschließenden* Klasse wären.

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Typen
    - Allgemeines
    - Anonyme Klassen
    - **Definition einer Klasse**

Definition einer Klasse

- Eine Klasse gilt mit dem Ende ihrer Definition als definiert.
- Es ist dabei gleichgültig, ob *Klassenelement-Funktionen* bereits *definiert* wurden, oder ob sie nur *deklariert* wurden.

Definition einer Klasse

```
class Point
{
public:
  Point ()          // Konstruktor definiert
    {nx = ny = 0;}
  short& x ();      // Zugriffsfunktionen
  short& y ();      // deklariert
private:
  short nx;
  short ny;
};
// ab hier kann Point verwendet werden
```


Definition einer Klasse

- Innerhalb der Definition einer Klasse können jedoch *Zeiger* oder *Referenzen* auf diese Klasse vorkommen:

```
struct LinkedList
{
  LinkedList *vor;    // Link-Zeiger nach
  LinkedList *rueck;  // vorne und hinten
  char inhalt [80];
};
```


Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Typen
  - **Klassen-Objekte**
  - Definition von Klassen
  - Klassenelement-Funktionen
  - Datenelemente von Klassen
  - Verschachtelte Klassen

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Objekte
    - **Operationen mit Klassen-Objekten**
    - Leere Klassen

Operationen mit Klassen-Objekten

- C++ erlaubt *von sich aus* die folgenden Operationen mit und an Objekten einer Klasse:
  - Zuweisungen: Standardmäßig durch bitweise Kopie; alternativ: benutzerdefinierter Zuweisungsoperator (Funktion `operator=`).
  - Initialisierungen unter Verwendung eines Kopier-Konstruktors.
- Beliebige weitere Operationen können durch benutzerdefinierte Funktionen realisiert werden.

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Objekte
    - Operationen mit Klassen-Objekten
    - **Leere Klassen**

Leere Klassen

- Leere Klassen (ohne Datenelemente) werden als *Container* für bestimmte Funktionen verwendet.
- Objekte einer leeren Klasse dürfen definiert werden.
- Sie haben eine von Null verschiedene Größe!

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Typen
  - Klassen-Objekte
  - **Definition von Klassen**
  - Klassenelement-Funktionen
  - Datenelemente von Klassen
  - Verschachtelte Klassen

Definition von Klassen

- Die *Deklaration* einer *Klasse*, also die Auflistung der *Elemente* der Klasse, muss *genau einmal* erfolgen.
- Änderungen der Zahl oder Type der Klassenelemente sind *nach* der Deklaration nicht mehr zulässig.
- *Objekte einer Klasse* können in beliebiger Zahl *definiert* werden.

Definition von Klassen

- Ihre *Initialisierung* kann im Zuge ihrer Definition erfolgen durch:
  - eine Liste der Anfangswerte des Objekts in "`{ }`", für Klassen
    - ohne Konstruktor;
    - ohne Basisklassen und virtuellen Funktionen;
    - ohne Elemente, die *nicht* `public` sind;
  - einen geeigneten Konstruktor der Klasse.

Definition von Klassen

- Eine Initialisierung nicht-statischer Datenelemente innerhalb der *Deklaration* der Klasse ist unzulässig.
- Statische Datenobjekte müssen außerhalb der Deklaration der Klasse explizit initialisiert werden.

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Typen
  - Klassen-Objekte
  - Definition von Klassen
  - **Klassenelement-Funktionen**
  - Datenelemente von Klassen
  - Verschachtelte Klassen

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassenelement-Funktionen
    - **Nicht-statische Funktionen**
    - Der `this`-Zeiger
    - Statische Funktionen

Nicht-statische Funktionen

- Nicht-statische Funktionen, die innerhalb einer Klasse deklariert oder definiert wurden, müssen grundsätzlich mit dem Elementauswahl-Operator ("`.`" bzw. "`->`") aufgerufen werden.
- Sie beziehen sich immer auf das spezifische Objekt der Klasse, für das sie aufgerufen wurden.
- Eine Ausnahme besteht beim Aufruf dieser Funktionen innerhalb *anderer* Klassenelement-Funktionen derselben Klasse.

Nicht-statische Funktionen

```
#include <iostream.h>

class Point
{
public:
  short& x() { return _x; }
  short& y() { return _y; }
  void Show ()
  { cout << "x = " << x() << ", y = " << y() << "\n";}
private:
  short _x, _y;
};
```

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassenelement-Funktionen
    - Nicht-statische Funktionen
    - **Der `this`-Zeiger**
    - Statische Funktionen

Der `this`-Zeiger

- Der Zeiger `this` ist für alle nicht-statischen Klassenelement-Funktionen definiert.
- Er zeigt auf jenes Objekt, für das die Klassenelement-Funktion aufgerufen wurde.
- Die Type von `this` ist *`type`* `* const`, also die eines *konstanten Zeigers*.
- `this` kann daher nicht auf ein anderes Objekt "umgebogen" werden.
- Alle Zugriffe einer Klassenelement-Funktion auf Elemente der Klasse erfolgen unter impliziter Verwendung von `this`.

Der `this`-Zeiger

- Eine explizite Verwendung von `this` ist im Allgemeinen nicht erforderlich:

```
#include <iostream.h>
class Point
{
public:
  short& x() { return this->_x; }
  short& y() { return this->_y; }
  void Show ()
  { cout << "x = " << this->x() <<
              ", y = " << this->y() << "\n" }
                 // this is overkill!
private:
  short _x, _y;
};
```


Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassenelement-Funktionen
    - Nicht-statische Funktionen
    - Der `this`-Zeiger
    - **Statische Funktionen**

Statische Funktionen

- Statische Klassenelement-Funktionen haben *keinen* `this`-Zeiger.
- Sie können daher *nicht* auf *Objekte*, sondern nur auf die folgenden Elemente der Klasse zugreifen:
  - Statische Datenelemente;
  - Enumeratoren;
  - Eingeschlossene Typen.
- Sie können ohne Verwendung eines Objekts ihrer Klasse unter Angabe des Klassennamens, gefolgt von "`::`", aufgerufen werden.

Statische Funktionen

- Ein Aufruf mit dem Elementauswahl-Operator ("`.`" oder "`->`") unter Verwendung eines Objekts der Klasse ist jedoch zulässig, wenn ein solches existiert:

Statische Funktionen

```
#include <iostream.h>
class Punkt
{
public:
  Punkt()  { PunkteZahl++; }  // Konstruktor
  ~Punkt() { PunkteZahl--; }  // Destruktor
  // Zugriffsfunktionen:
  unsigned& x() { return xKoord; }
  unsigned& y() { return yKoord; }
  // statische Zugriffsfunktion:
  static int Anzahl() { return PunkteZahl; }
  static int PunkteZahl;
```


Statische Funktionen

```
private:
  unsigned xKoord;
  unsigned yKoord;
};

int Punkt::PunkteZahl = 0;

int main ()
{
  cout << "Anzahl der Punkte: " <<
    Punkt::Anzahl() << "\n";
  Punkt p1;                          // neues Objekt
  cout << "Anzahl der Punkte: " <<
    p1.Anzahl() << "\n";
}
```


Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Typen
  - Klassen-Objekte
  - Definition von Klassen
  - Klassenelement-Funktionen
  - **Datenelemente von Klassen**
  - Verschachtelte Klassen

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Datenelemente von Klassen
    - **Statische Daten**
    - `unions`
    - Bitfelder

Statische Daten

- Statische Datenelemente einer Klasse sind mit dem Schlüsselwort `static` versehen.
- Sie werden in der Definition der Klasse wohl *deklariert*, müssen aber gesondert außerhalb der Klassendefinition *definiert* und eventuell initialisiert werden.
- Sie sind eigenständige Objekte, deren Type im Allgemeinen *nicht* die Type der Klasse ist.
- Auf sie kann entweder über ein Objekt der Klasse mit dem Elementauswahl-Operator ("`.`" oder "`->`") zugegriffen werden, oder unter Angabe des Klassennamens, gefolgt von "`::`".

Statische Daten

- Statische Datenelemente, die nicht in der Definition der Klasse als `public` deklariert wurden, sind nur für Klassenelement-Funktionen und solche, die mit `friend` deklariert wurden, zugänglich.

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Datenelemente von Klassen
    - Statische Daten
    - **unions**
    - Bitfelder

unions

- C++-unions können auch Klassenelement-Funktionen enthalten.
- Sie dürfen auch Konstruktoren und Destruktoren, jedoch *keine virtuellen Funktionen* enthalten.
- Sie dürfen weder *Basisklassen* sein noch haben.
- Sie dürfen *nicht* die folgenden Datentypen als Elemente beinhalten:
  - Klassen-Typen mit Konstruktoren und/oder Destruktoren;
  - Klassen-Typen mit einem benutzerdefinierten Zuweisungs-Operator;
  - Statische Datenelemente.

unions

- unions dürfen auch *anonym* sein (d.h., keinen Namen haben); in diesem Fall deklarieren sie ein *Objekt* und keine *Type*.
- Für anonyme unions gelten die folgenden Einschränkungen:
  - Sie sind immer public;
  - Sie dürfen keine Funktions-Elemente haben.
- Das folgende Beispiel verwendet eine union, um wahlweise ein Zeichen, einen Zeiger auf einen String und eine ganzzahlige Variable zu speichern und auszugeben:

unions

```
#include <iostream.h>
class Data
{
  enum DataType { Char, Int, String };
  DataType type;                        // ist private!
  union
  {
    char CharData;
    int IntData;
    char *StringData;
  };
  _data (int i)                         // Hilfsfunktion
  { IntData = i; type = Int; }
```


unions

```
public:
  // Konstruktoren:
  Data () { _data (0); }            // Default
  Data (int i) { _data (i); }
  Data (char ch) { CharData = ch; type = Char; }
  Data (char *psz){ StringData = psz; type = String; }
  // Ausgabe-Funktion:
  void Print ();                // hier nur deklariert
};
```


unions

```
void Data::Print (void)
{
  switch (type)
  {
  case Char:
    cout << "char:   " << CharData << "\n";
    break;
  case Int:
    cout << "int:    " << IntData << "\n";
    break;
  case String:
    cout << "string: " << StringData << "\n";
  }
}
```

unions

```
int main ()
{
  Data d1;                      // Default: int = 0
  Data d2 = 'K';                // type = Char
  Data d3 = 17;                 // type = Int
  Data d4 = "Hello, world!";    // String

  d1.Print();
  d2.Print();
  d3.Print();
  d4.Print();
  d2.CharData = 'L';
  d2.Print();
}
```


Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Datenelemente von Klassen
    - Statische Daten
    - unions
    - **Bitfelder**

Bitfelder

- Klassen, die mit `struct` oder `class` deklariert wurden, können Elemente (Bitfelder, *Bit Fields)* enthalten, die kleiner als eine ganzzahlige Datentype sind.
- Das folgende Demo-Programm zeigt die *Equipment List* des PC-BIOS an:

Bitfelder

```
typedef unsigned short WORD;

struct EquipList
{
  WORD FloppyPresent   : 1;
  WORD NPUPresent      : 1;
  WORD                 : 2; // XT-RAM
  WORD ActiveVideoMode : 2;
  WORD FloppyCount     : 2;
  WORD DMAPresent      : 1;
  WORD COMPortCount    : 3;
  WORD GamePortPresent : 1;
  WORD                 : 1; // PCJr
  WORD LPTPortCount    : 2;
};
```


Bitfelder

```
#include <iostream.h>
extern "C" int biosequip(void); //
  Bibliotheksfunktion
int main ()
{
  int i = biosequip ();
  EquipList Equip = *(EquipList *) &i;
  if (Equip.FloppyPresent)
    cout << (Equip.FloppyCount + 1);
  else
    cout << "Keine";
  cout << " Diskettenlaufwerk(e)\n";
// usw. siehe Skriptum
```


Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
  - Klassen-Typen
  - Klassen-Objekte
  - Definition von Klassen
  - Klassenelement-Funktionen
  - Datenelemente von Klassen
  - **Verschachtelte Klassen**

Verschachtelte Klassen

- Eine Klasse kann innerhalb einer anderen Klasse deklariert werden — *verschachtelte Klasse* (*Nested Class*).
- Eine verschachtelte Klasse ist unmittelbar nur von der sie umgebenden Klasse aus zugänglich.
- Für andere Zugriffe muss ein voll spezifizierter Name (*Fully Qualified Name* — `umschließende Klasse :: eingeschlossene Klasse`) angegeben werden.

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
- **Abgeleitete Klassen**
- Zugriff auf Klassen-Elemente
- Schablonen (*Templates*)
- Spezielle Funktionen in Klassen
- Überladen (*Overloading*)
- Konventionelle Programmierung und OOP

Objektorientierte Programmierung (OOP)

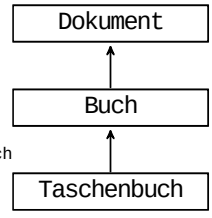
- Abgeleitete Klassen
  - **Einfache Vererbung (*Inheritance*)**
  - Virtuelle Funktionen
  - Abstrakte Klassen
  - Mehrfache Vererbung
  - Mehrfache Basisklassen
  - Virtuelle Basisklassen
  - Mehrdeutigkeiten

Einfache Vererbung

- Eine Klasse, die eine Spezialisierung darstellt, kann durch *Vererbung* (*Inheritance*) aus einer anderen Klasse abgeleitet werden.
- Bei der *einfachen Vererbung* hat jede abgeleitete Klasse genau eine (direkte) *Basisklasse*:

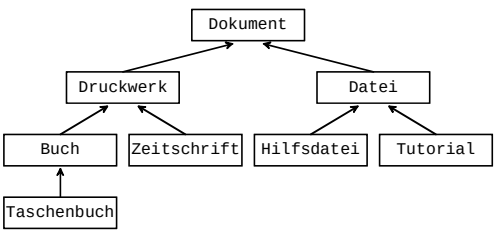
Einfache Vererbung

```
class Dokument
{ /* Klassenelemente */ };

// Buch abgeleitet von Dokument
class Buch : public Dokument
{ /* Klassenelemente */ };

// Taschenbuch abgeleitet von Buch
class Taschenbuch : public Buch
{ /* Klassenelemente */ };
```



Einfache Vererbung

- Eine Klasse kann als Basisklasse beliebig vieler abgeleiteter Klassen dienen:



Einfache Vererbung

- Eine abgeleitete Klasse enthält alle Elemente der Basisklasse plus zusätzliche Elemente, die für die abgeleitete Klasse deklariert wurden.
- Eine abgeleitete Klasse kann daher auf alle Elemente ihrer Basisklasse (sowie allfälliger *indirekter Basisklassen*) zugreifen.
- Elemente der Basisklasse und der abgeleiteten Klasse können daher auf gleiche Weise verwendet werden.
- In der abgeleiteten Klasse können auch Elemente der Basisklasse neu definiert werden.

Einfache Vererbung

- Der Operator "`::`" erlaubt die Unterscheidung zwischen gleichnamigen Objekten der Basisklasse und der abgeleiteten Klasse:

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

class Dokument
{
public:
  char *Name;             // Name des Dokuments
  void PrintName() { cout << Name << "\n"; }
};
```


Einfache Vererbung

```
class Buch : public Dokument
{
public:
  Buch (char *name, int seiten);
  void PrintName ();
private:
  int Seitenzahl;
};
Buch::Buch (char *name, int seiten)     // Konstruktor
{
  Name = new char [strlen (name) + 1];
  strcpy (Name, name);                  // kopieren
  Seitenzahl = seiten;
}
```


Einfache Vererbung

```
void Buch::PrintName ()
{
  cout << "Titel:  " << Name << "\nSeiten: "
    << Seitenzahl << "\n";
}
int main ()
{
  Buch Stroustrup ("The C++ Programming Language, "
    "2nd Ed", 669);
  Stroustrup.PrintName ();
  Stroustrup.Dokument::PrintName ();
}
```


Einfache Vererbung

- Hier wird einmal die mit der Klasse `Buch` und dann die mit der Klasse `Dokument` definierte Funktion `PrintName()` aufgerufen. Die Ausgabe ist daher:

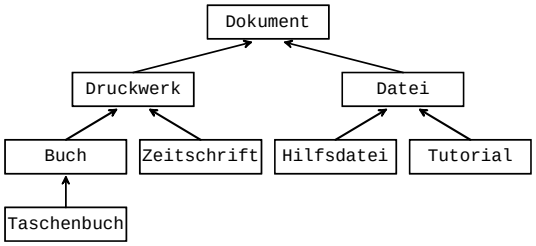
```
Titel:  The C++ Programming Language, 2nd Ed
Seiten: 669
The C++ Programming Language, 2nd Ed
```


Einfache Vererbung

- Das folgende Beispiel zeigt den Aufbau einer "Bibliothek" aus einer *heterogenen Liste* (Feld von Zeigern auf Objekte mit unterschiedlichen Typen):

```
// erforderliche Header-Dateien

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

Einfache Vererbung



Einfache Vererbung

```
class Dokument // Basisklasse für alle anderen Klassen
{
public:
  char *Name;                  // Name des Dokuments
  Dokument () { }              // Konstruktoren
  Dokument (char *name) { Speichern (name); }
  void PrintName ()            // Default-Ausgabe
    { cout << Name << "\n"; }
  void Speichern (char *name)  // Hilfsfunktion
  {
    Name = new char [strlen (name) + 1];
    strcpy (Name, name);
  }
};
```


Einfache Vererbung

```
// Abgeleitete Klasse Kategorie A (ohne PrintName())
class Druckwerk : public Dokument
{
public:
  Druckwerk () { }
  Druckwerk (char *name)
    { Speichern (name); }
};
```


Einfache Vererbung

```
// Abgeleitete Klasse Kategorie B (mit PrintName())
class Datei : public Dokument
{
public:
  Datei () { }
  Datei (char *name)
    { Speichern (name); }
  void PrintName ()
    { cout << "Datei: " << Name << "\n"; }
};
```


Einfache Vererbung

```
class Buch : public Druckwerk
{
// analog als Kategorie B (mit PrintName()) definiert
};
class Zeitschrift : public Druckwerk
{
// analog als Kategorie B (mit PrintName()) definiert
};
class Taschenbuch : public Buch
{
// analog als Kategorie A (ohne PrintName()) definiert
};
```


Einfache Vererbung

```
class Hilfsdatei : public Datei
{
// analog als Kategorie A (ohne PrintName()) definiert
};
class Tutorial : public Datei
{
// analog als Kategorie A (ohne PrintName()) definiert
};
const int BibGroesse = 10;    // Anzahl der Einträge
```

Einfache Vererbung

```
int main ()
{
  Dokument *Bibliothek [BibGroesse]; // "Datenbank"
  cout << "Typ der Eintragung:\n(D)okument, "
    "D(r)uckwerk, D(a)tei, (B)uch, (Z)eitschrift,\n"
    "(T)aschenbuch, (H)ilfsdatei, T(u)torial:\n\n";
  for (int i = 0; i < BibGroesse; i++)
  {
    char cDtyp;                        // Schalt-Zeichen
    char buffer [128];
    cout << "Typ:   ";
    cin.getline (buffer, 128, '\n');
    cDtyp = *buffer;
```


Einfache Vererbung

```
    cout << "Titel: ";
    cin.getline (buffer, 128, '\n');
    // konvertiere cDtyp in Kleinbuchstaben
    switch (tolower (cDtyp))
    {
    case 'd':
      Bibliothek[i] = new Dokument (buffer);
      break;
    case 'r':
      Bibliothek[i] = new Druckwerk (buffer);
      break;
    // und so weiter analog für die anderen Klassen
```


Einfache Vererbung

```
    default:
      i--;                // Kompensation von "i++"
    }
  }
  // gib alle Eintragungen aus
  for (i = 0; i < BibGroesse; i++)
    Bibliothek[i]->PrintName();
}
```


Einfache Vererbung

- Bei der Ausgabe der "Bibliotheksliste" wird nur die `PrintName()`-Funktion der Klasse `Dokument` verwendet, weil `Bibliothek` vom Typ `Dokument*` ist.
- Die Charakteristiken der abgeleiteten Klassen gehen verloren.
- Dieses Problem kann durch Verwendung *virtueller Funktionen* behoben werden.

Objektorientierte Programmierung (OOP)

- Abgeleitete Klassen
  - Einfache Vererbung (*Inheritance*)
  - **Virtuelle Funktionen**
  - Abstrakte Klassen
  - Mehrfache Vererbung
  - Mehrfache Basisklassen
  - Virtuelle Basisklassen
  - Mehrdeutigkeiten

Virtuelle Funktionen

- Die gemeinsamen Eigenschaften abgeleiteter Klassen werden im Allgemeinen durch ihre Basisklasse repräsentiert; spezifische Eigenschaften einer Klasse werden in dieser selbst festgelegt.
- Klassenelement-Funktionen werden mit dem Schlüsselwort `virtual` als *virtuell* deklariert.
- Bei der Erstellung eines Objekts einer (abgeleiteten) Klasse wird vom Compiler automatisch die Information abgespeichert, die es erlaubt, auch dann die korrekten Klassenelement-Funktionen aufzurufen, wenn nur ein *Zeiger* auf die *Basisklasse* übergeben wird.

Virtuelle Funktionen

- Bei Verwendung virtueller Funktionen im letzten Beispiel braucht nur die Basisklasse geändert zu werden.
- In den abgeleiteten Klassen *kann*, *muss* aber nicht die Klasse `PrintName` als `virtual` deklariert werden.
- Die Deklaration der Basisklasse unter Verwendung virtueller Funktionen sieht folgendermaßen aus:

Virtuelle Funktionen

```
// Basisklasse für alle anderen Klassen
class Dokument
{
public:
  char *Name;
  Dokument () { }
  Dokument (char *name) { Speichern (name); }
  virtual void PrintName () { cout << Name << "\n"; }
  void Speichern (char *name)
  {
    Name = new char [strlen (name) + 1];
    strcpy (Name, name);
  }
};
```


Virtuelle Funktionen

- Die wesentlichen Unterschiede zwischen den Versionen mit und ohne virtuelle Funktionen sind:
  - Der Compiler legt bei der Ausführung des Konstruktors für jedes Objekt jeder Klasse und für jede Familie virtueller Funktionen einen Zeiger auf eine Tabelle ab, die (unter anderem) einen Funktionszeiger auf die korrekte virtuelle Funktion enthält.
  - Bei der Ausführung einer virtuellen Funktion wird aufgrund der im Objekt abgespeicherten Informationen der korrekte Funktionszeiger ausgewählt und die passende virtuelle Funktion ausgeführt.

Virtuelle Funktionen

- Alle Implementierungen einer virtuellen Funktion müssen die gleiche Ergebnistype haben.
- In einer Basisklasse kann eine *reine virtuelle Funktion* (*Pure Virtual Function*) in der Form

  `virtual void PrintName () = 0;`

  deklariert werden.
- In diesem Fall *muss* in jeder von der Basisklasse abgeleiteten Klasse eine geeignete Implementierung dieser Funktion vorgesehen werden.
- Für den Aufruf einer Klassenelement-Funktion gilt grundsätzlich:

Virtuelle Funktionen

- Aufruf über ein Objekt (`Obj.Funkt()`):
  - Es wird *immer* die für die Klasse des Objekts gültige Implementierung der Funktion ausgeführt, gleichgültig, ob die Funktion virtuell ist oder nicht.
- Aufruf über Zeiger oder Referenz:
  - Virtuelle Funktion: Aufruf der Klassenelement-Funktion, die dem zugrunde liegenden Objekt entspricht.
  - Nicht-virtuelle Funktion: Aufruf der Klassenelement-Funktion, die der Klasse des Zeigers oder der Referenz entspricht.

Virtuelle Funktionen

- Die Verwendung des Schlüsselwortes `virtual` ist nur bei Klassenelement-Funktionen zulässig.
- Seine Verwendung bei der Deklaration einer virtuellen Funktion in einer *abgeleiteten* Klasse ist optional.
- Alle Funktionen in einer abgeleiteten Klasse, die eine virtuelle Funktion einer Basisklasse neu definieren, sind *a priori* virtuell.
- Virtuelle Funktionen in einer Basisklasse *müssen definiert* werden, außer, sie wurden als *reine virtuelle Funktionen* deklariert.

Virtuelle Funktionen

- Der Aufrufmechanismus virtueller Funktionen kann mit dem Operator "`::`" und unter Angabe der Ziel-Klasse ausgeschaltet werden.

Objektorientierte Programmierung (OOP)

- Abgeleitete Klassen
  - Einfache Vererbung (*Inheritance*)
  - Virtuelle Funktionen
  - **Abstrakte Klassen**
  - Mehrfache Vererbung
  - Mehrfache Basisklassen
  - Virtuelle Basisklassen
  - Mehrdeutigkeiten

Abstrakte Klassen

- Abstrakte Klassen sind solche, die *mindestens eine reine virtuelle Funktion* enthalten oder von einer abstrakten Basisklasse abgeleitet wurden, ohne eine Implementierung für deren reine virtuelle Funktionen zu beinhalten.
- Sie können verwendet werden, um ein Protokoll zu erzwingen (z.B. spezifische Funktionen für jede von ihnen abgeleitete Klasse).

Abstrakte Klassen

- Abstrakte Klassen existieren ausschließlich als Basisklassen für abgeleitete Klassen.
- Es ist nicht zulässig, ein Objekt einer abstrakten Klasse zu definieren.

Abstrakte Klassen

- Wenn in unserem Bibliotheksprogramm-Beispiel die Funktion `PrintName` in der Klasse `Dokument` als *reine virtuelle Funktion*

```
virtual void PrintName () = 0;
```

deklariert worden wäre, dann hätte dies die folgenden Konsequenzen:

  - Die Klassen `Dokument` und `Druckwerk` wären *abstrakte* Klassen; es dürften keine Objekte dieser Klassen (beispielsweise im `switch`-Befehl in `main()`) generiert werden.

Abstrakte Klassen

  - Für alle anderen abgeleiteten Klassen unseres Beispiels würde sich nichts ändern.
  - Es wäre aber nicht möglich, eine Klasse zu definieren, die direkt von `Dokument` oder `Druckwerk` abgeleitet wäre, aber keine `PrintName()`-Funktion definiert hätte.

Abstrakte Klassen

- Obwohl es unzulässig ist, *Objekte* einer abstrakten Klasse zu definieren, können *Zeiger* und *Referenzen* auf eine abstrakte Klasse verwendet werden.
- Es ist daher nach wie vor möglich, in unserem Beispielprogramm die *heterogene Liste* der Bibliotheks-eintragungen als

  `Dokument *Bibliothek [BibGroesse];`

  zu definieren und auf ihre Elemente zuzugreifen.

Abstrakte Klassen

- Die Verwendung abstrakter Klassen ist unzulässig für:
  - Funktionsargumente und -ergebnisse;
  - Explizite Typkonversion.
- Bei direktem oder indirektem Aufruf einer reinen virtuellen Funktion durch den Konstruktor einer Klasse ist das Ergebnis undefiniert.

Abstrakte Klassen

- Reine virtuelle Funktionen können in einer abstrakten Klasse auch *definiert* werden; sie können jedoch nur unter Verwendung der Syntax

  `<Abstrakte Klasse>::<Funktion>()`

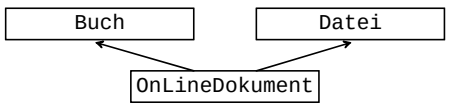
  aufgerufen werden.
- Damit ist es beispielsweise möglich, reine virtuelle Destruktoren für eine abstrakte Klasse zu definieren.

Objektorientierte Programmierung (OOP)

- Abgeleitete Klassen
  - Einfache Vererbung (*Inheritance*)
  - Virtuelle Funktionen
  - Abstrakte Klassen
  - **Mehrfache Vererbung**
  - Mehrfache Basisklassen
  - Virtuelle Basisklassen
  - Mehrdeutigkeiten

Mehrfache Vererbung

- Eine abgeleitete Klasse in C++ kann auch *mehrere* direkte Basisklassen haben:



Mehrfache Vererbung

```
class Buch
{
  // Klassenelemente
};
class Datei
{
  // Klassenelemente
}
class OnLineDokument : public Buch, public Datei
{
  // neue Klassenelemente
}
```

Mehrfache Vererbung

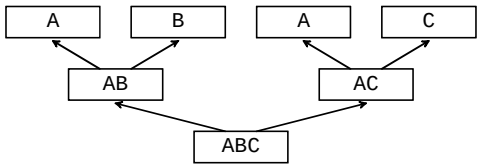
- Die abgeleitete Klasse vereinigt die Eigenschaften aller ihrer Basisklassen.
- Die Reihenfolge, in der die Basisklassen deklariert werden, bestimmt:
  - die Reihenfolge, in der die Konstruktoren der Basisklassen bei der Definition eines Objekts der abgeleiteten Klasse aufgerufen werden;
  - die *umgekehrte* Reihenfolge des Aufrufs der Destruktoren.

Objektorientierte Programmierung (OOP)

- Abgeleitete Klassen
  - Einfache Vererbung (*Inheritance*)
  - Virtuelle Funktionen
  - Abstrakte Klassen
  - Mehrfache Vererbung
  - **Mehrfache Basisklassen**
  - Virtuelle Basisklassen
  - Mehrdeutigkeiten

Mehrfache Basisklassen

- Bei Verwendung mehrfacher Vererbung zur Erstellung abgeleiteter Klassen kann die selbe Klasse mehrfach *indirekte* (aber nicht *direkte*) Basisklasse sein:

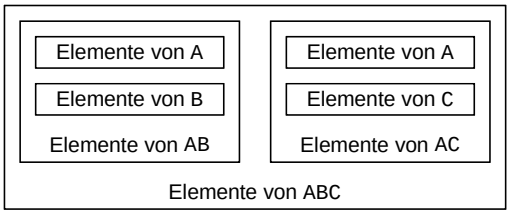


Mehrfache Basisklassen

```
class A { /* Klassenelemente von A */ };
class B { /* Klassenelemente von B */ };
class C { /* Klassenelemente von C */ };
class AB : public A, public B
{
  // Klassenelemente von AB
}
class AC : public A, public C
{
  // Klassenelemente von AC
}
class ABC : public AB, public AC
{
  // Klassenelemente von ABC
}
```


Mehrfache Basisklassen

- Die logische Struktur der abgeleiteten Klasse ABC ist dann:



Objektorientierte Programmierung (OOP)

- Abgeleitete Klassen
  - Einfache Vererbung (*Inheritance*)
  - Virtuelle Funktionen
  - Abstrakte Klassen
  - Mehrfache Vererbung
  - Mehrfache Basisklassen
  - **Virtuelle Basisklassen**
  - Mehrdeutigkeiten

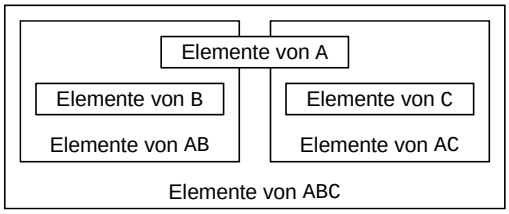
Virtuelle Basisklassen

- Im obigen Beispiel enthält ein Objekt der Klasse ABC *zweimal* die Elemente der Klasse A.
- Dies führt zu Mehrdeutigkeiten bei Zugriffen auf ein Element der Klasse A.
- Deklaration der mehrfach aufscheinenden Basisklasse als *virtuelle* Basisklasse
  - Verbesserung der Effizienz der Datenspeicherung;
  - Verbesserung der Eindeutigkeit der Zuordnung.
- Im folgenden Fall wird für ein Objekt der abgeleiteten Klassen nur *ein* Satz der Elemente der virtuellen Basisklasse angelegt:

Virtuelle Basisklassen

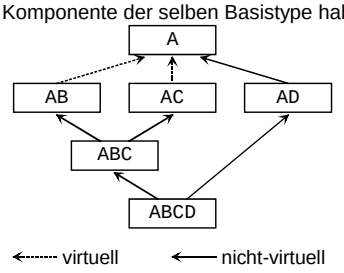
```
class AB : virtual public A, public B
{
  // Klassenelemente von AB
}
class AC : virtual public A, public C
{
  // Klassenelemente von AC
}
class ABC : public AB, public AC
{
  // Klassenelemente von ABC
}
```


Virtuelle Basisklassen



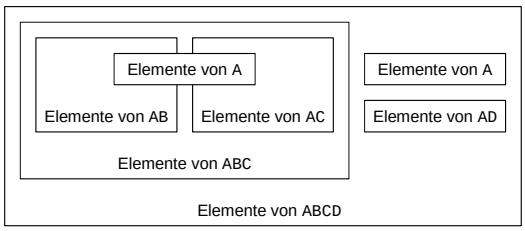
Virtuelle Basisklassen

- Eine Klasse kann auch eine *virtuelle* und eine *nicht-virtuelle* Komponente der selben Basistype haben:



Virtuelle Basisklassen

- Dieser Vererbungsstruktur entspricht die folgende Struktur eines Objektes der Klasse ABCD:



Virtuelle Basisklassen

- Unter bestimmten Voraussetzungen kann die Verwendung virtueller Basisklassen zusätzlichen Overhead an Programmcode und Datengröße bedeuten.

Objektorientierte Programmierung (OOP)

- Abgeleitete Klassen
  - Einfache Vererbung (*Inheritance*)
  - Virtuelle Funktionen
  - Abstrakte Klassen
  - Mehrfache Vererbung
  - Mehrfache Basisklassen
  - Virtuelle Basisklassen
  - **Mehrdeutigkeiten**

Mehrdeutigkeiten

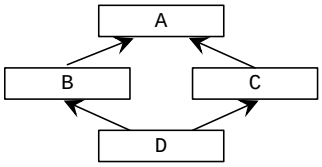
- Im Zuge einer mehrfachen Vererbung kann eine Klasse den selben Namen auf mehr als einem Pfad erben — *Mehrdeutigkeit* (*Ambiguity*).
- Diese Mehrdeutigkeit kann durch vollständige Spezifikation des Klassennamens beseitigt werden.
- Bei Verwendung virtueller Basisklassen existiert nur *ein* Satz der Elemente der virtuellen Basisklasse in der abgeleiteten Klasse. Jeder Zugriff auf ein Element der *virtuellen* Basisklasse ist daher *immer eindeutig*.

Mehrdeutigkeiten

- Im Gegensatz dazu ist bei *nicht-virtueller* Deklaration der Basisklasse *jeder* Zugriff auf ein Element der Basisklasse *mehrdeutig*.
- Wenn ein Name in einer Basisklasse und in einer abgeleiteten Klasse definiert wurde, *dominiert* die Implementierung des Namens in der abgeleiteten Klasse.
- Im Falle einer potenziellen Mehrdeutigkeit wird immer die Implementierung der abgeleiteten Klasse verwendet.

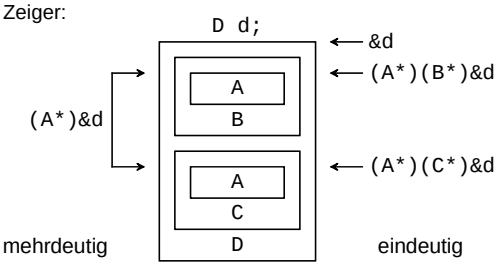
Mehrdeutigkeiten

- Die *explizite Konversion von Zeigern* oder *Referenzen* auf Klassen kann Mehrdeutigkeiten verursachen, wenn nicht-virtuelle Basisklassen verwendet werden:



Mehrdeutigkeiten

- Für ein Objekt `d` der Klasse `D` gelten dann die folgenden Zeiger:



Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
- Abgeleitete Klassen
- **Zugriff auf Klassen-Elemente**
- Schablonen (*Templates*)
- Spezielle Funktionen in Klassen
- Überladen (*Overloading*)
- Konventionelle Programmierung und OOP

Objektorientierte Programmierung (OOP)

- Zugriff auf Klassen-Elemente
  - **Zugriffskontrolle**
  - Das Schlüsselwort `friend`

Zugriffskontrolle

- Schlüsselworte für die Steuerung des Zugriffs auf Klassenelemente:
  - `private`: Klassenelemente zugänglich für:
    - Klassenelement-Funktionen der Klasse;
    - Klassen und Funktionen, die als `friend` der Klasse deklariert wurden.
  - `protected`: Klassenelemente zugänglich für:
    - Klassenelement-Funktionen der Klasse sowie von Klassen, die von der Klasse abgeleitet wurden;
    - Klassen und Funktionen, die als `friend` der Klasse deklariert wurden.

Zugriffskontrolle

  - `public`: Klassenelemente zugänglich für:
    - beliebige Funktionen.
- Diese Zugriffskontrolle kann durch explizite Typenumwandlung umgangen werden.
- Ohne Verwendung eines der drei Zugriffs-Schlüsselwörter ist der Zugriff auf Klassentypen, die mit `struct` oder `union` deklariert wurden, `public`, und auf solche, die mit `class` deklariert wurden, `private`.
- Alle Klassenelemente, die in der Deklaration der Klasse einem der Zugriffs-Schlüsselwörter folgen, unterliegen dem entsprechenden Zugriffstyp.

Zugriffskontrolle

- Diese Spezifikation endet mit dem nächsten Zugriffs-Schlüsselwort oder dem Ende der Deklaration.
- Es können beliebig viele Zugriffs-Schlüsselwörter in der Deklaration einer Klasse verwendet werden.
- Für abgeleitete Klassen hängt der Zugriff auf die Elemente einer Basisklasse von den Zugriffsrechten innerhalb der Basisklasse und von dem bei der Ableitung spezifizierten Zugriffsrecht ab:

Zugriffskontrolle

in Basis-klasse:	**Zugriffsrecht bei der Ableitung:**		
	`private`	**`protected`**	**`public`**
`private`	immer unzugänglich		
`protected`	`private`	`protected`	`protected`
`public`	`private`	`protected`	`public`

Zugriffskontrolle

- Bei der Definition einer abgeleiteten Klasse darf das Schlüsselwort für den Zugriffs-Typ auf die Basisklasse weggelassen werden.

Zugriffskontrolle

- In diesem Fall wird der gleiche Zugriffs-Typ verwendet, der standardmäßig für den definierten Klassentyp gilt:

```
class Abgeleitet : Basis
```

ist äquivalent zu:

```
class Abgeleitet : private Basis
```

und

```
struct Abgeleitet : Basis
```

ist äquivalent zu:

```
struct Abgeleitet : public Basis
```


Objektorientierte Programmierung (OOP)

- Zugriff auf Klassen-Elemente
  - Zugriffskontrolle
  - **Das Schlüsselwort `friend`**

Objektorientierte Programmierung (OOP)

- Zugriff auf Klassen-Elemente
  - Das Schlüsselwort `friend`
    - **Allgemeines**
    - `friend`-Funktionen
    - Klassen und Klassenelemente als `friends`

Das Schlüsselwort `friend` — Allgemeines

- Das Schlüsselwort `friend` erlaubt einen Zugriff externer Funktionen oder Klassen auf Elemente einer Klasse, die als `protected` oder `private` deklariert wurden.

Objektorientierte Programmierung (OOP)

- Zugriff auf Klassen-Elemente
  - Das Schlüsselwort `friend`
    - Allgemeines
    - **`friend`-Funktionen**
    - Klassen und Klassenelemente als `friends`

`friend`-Funktionen

- Funktionen, die mit `friend` deklariert wurden, gelten nicht als Elemente der Klasse.
- Sie sind vielmehr "gewöhnliche" Funktionen, die nicht mit dem Namen eines Klassen-Objekts und dem Operator "`.`" oder "`->`" aufgerufen zu werden brauchen.
- Es ist gleichgültig, *wo* innerhalb der Definition einer Klasse die Deklaration von `friends` erfolgt.
- Es müssen aber immer *alle* `friend`-Funktionen, unabhängig davon, ob sie gleiche oder unterschiedliche Namen haben, *explizit* als `friend` deklariert werden.
- Das folgende Beispiel illustriert eine Klasse `Punkt` und einen überladenen (*overloaded) Operator* `operator+`:

friend-Funktionen

```
#include <iostream.h>
class Punkt
{
public:
  Punkt (short x, short y) { _x = x; _y = y; }
  void Print()
  { cout << "Punkt (" << _x << ", " << _y << ")\n"; }
private:
  short _x, _y;
  // friend-Deklarationen
  friend Punkt operator+ (Punkt&, int);
  friend Punkt operator+ (int, Punkt&);
};
```


friend-Funktionen

```
// friend-Funktion für Operation "Punkt + Offset"
Punkt operator+ (Punkt& pt, int offset)
{
  Punkt temp = pt;
  temp._x += offset;    // _x und _y sind private!
  temp._y += offset;
  return temp;
}
// friend-Funktion für Operation "Offset + Punkt"
Punkt operator+ (int offset, Punkt& pt)
{
  Punkt temp = pt;
  temp._x += offset;    // _x und _y sind private!
  temp._y += offset;
  return temp;
}
```


friend-Funktionen

```
int main ()
{
  Punkt p(10, 20);
  p.Print();
  p = p + 5;      // ruft 1. operator+ auf
  p.Print();
  p = 3 + p;      // ruft 2. operator+ auf
  p.Print();
}
```


Objektorientierte Programmierung (OOP)

- Zugriff auf Klassen-Elemente
  - Das Schlüsselwort `friend`
    - Allgemeines
    - `friend`-Funktionen
    - **Klassen und Klassenelemente als `friend`s**

Klassen und Klassenelemente als `friend`s

- Klassenelement-Funktionen oder ganze Klassen können als `friend` einer Klasse (mit gleichen Zugriffsrechten auf Elemente dieser Klasse wie deren eigene Funktionen) deklariert werden:

Klassen und Klassenelemente als `friend`s

```
class A;        // Vorwärts-Deklaration
// Deklarationen ohne Definition der Funktionen
class B
{
public:
  int Func1(A);
  int Func2(A);
};
class A
{
private:
  int _a;
  friend int B::Func1(A);
  friend class C;
};
```

Klassen und Klassenelemente als `friends`

```
// Definition der Funktionen in Klasse B
int B::Func1(A a)
{
  return a._a; // Ok: Func1 ist friend von A
}
int B::Func2(A a)
{
  return a._a; // unzulässig: Func2 ist nicht friend
}
```


Klassen und Klassenelemente als `friends`

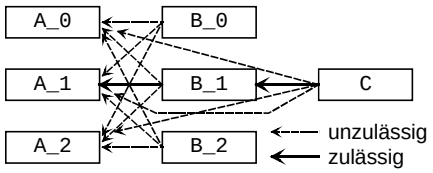
```
// Definition von Klassen C und D
class C
{
public:
  int Func3(A a) { return a._a; }
                // Ok: Klasse C ist friend von A
};
class D : public C
{
public:
  int Func4(A a) { return a._a; }
                // unzulässig: Freundschaften
                // können nicht vererbt werden
};
```


Klassen und Klassenelemente als `friends`

- Das folgende Beispiel illustriert die Wirkung von `friend` bei abgeleiteten Klassen und "Freunden von Freunden":

Klassen und Klassenelemente als `friends`

- Klasse `B_1` ist als `friend` von `A_1` deklariert, und Klasse `C` als `friend` von `B_1`. Klasse *n* ist von der Klasse *n*–1 abgeleitet:



Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
- Abgeleitete Klassen
- Zugriff auf Klassen-Elemente
- **Schablonen (*Templates*)**
- Spezielle Funktionen in Klassen
- Überladen (*Overloading*)
- Konventionelle Programmierung und OOP

Objektorientierte Programmierung (OOP)

- Schablonen (*Templates*)
  - **Allgemeines**
  - Schablonen für Klassen (*Class Templates*)
  - Funktions-Schablonen (*Function Templates*)

Schablonen (*Templates*) — Allgemeines

- Schablonen (*Templates*) stellen eine Verallgemeinerung einer Klasse oder Funktion dar, wobei die Type einzelner Klassenelemente oder Funktionsargumente offen bleibt.
- Unter Verwendung einer Schablone können Familien von Klassen oder überladenen Funktionen definiert werden.
- Die spezielle typmäßige Implementierung dieser Klassen oder Funktionen wird erst bei der Definition von Objekten oder beim Aufruf der Funktion mit Argumenten eines bestimmten Typs festgelegt.

Schablonen (*Templates*) — Allgemeines

- Schablonen verhalten sich ähnlich wie Makros, erlauben jedoch im Gegensatz zu diesen die volle Typprüfung von C++.

Objektorientierte Programmierung (OOP)

- Schablonen (*Templates*)
  - Allgemeines
  - **Schablonen für Klassen (*Class Templates*)**
  - Funktions-Schablonen (*Function Templates*)

Schablonen für Klassen (*Class Templates*)

- *Schablonen für Klassen* werden ähnlich definiert wie eine Klasse; der Klassen-Definition wird jedoch voran-gestellt:

  `template <class Type> …`
- In der eigentlichen Definition der Klasse kann beliebig oft der bei der Definition der Schablone verwendete Typen-name (*Type*) als Platzhalter für die Typenbezeichnung vorkommen.
- Bei der Definition eines *Objekts* der Klasse muss dann, ebenfalls in "< >", die tatsächliche (fundamentale oder abgeleitete) Type angegeben werden, für die das Objekt angelegt werden soll.

Schablonen für Klassen (*Class Templates*)

- Beispiel: Verwendung einer Schablone für eine allgemeine Stack-Klasse:

Schablonen für Klassen (*Class Templates*)

```
#include <iostream.h>
template <class T> class stack // allgemeine Klasse "stack"
{
  T *basis;
  T *zeiger;
  int gr;
public:
  stack (int groesse)              // Konstruktor
  { basis = zeiger = new T [gr = groesse]; }
  ~stack ()                        // Destruktor
  { delete [] basis; }
  void push (T t)           // Objekt auf den Stack schieben
  {
    if (zeiger - basis < gr)      // Überlaufschutz
      *zeiger++ = t;
  }
```

Schablonen für Klassen (*Class Templates*)

```
  T pop (void)              // Objekt vom Stack holen
  {
    if (zeiger > basis)          // Überlaufschutz
      return *--zeiger;
    else
      return 0;
  }
  int frei (void)           // freie Stack-Eintragungen
  { return gr - (zeiger - basis); }

}; // Ende template <class T> class stack
```


Schablonen für Klassen (*Class Templates*)

```
int main ()
{
  int stackgr;
  cout << "Stack-Größe:  ";
  cin >> stackgr;
  stack <int> si (stackgr);   // Implementierung für ints
  stack <char> sc (stackgr);  // Implementierung für chars
  do                          // int-Stack auffüllen
  {
    int i;
    cout << "Zahl eingeben: ";
    cin >> i;
    si.push (i);      // Elementfunktion mit int-Argument
    cout << "Noch frei: " << si.frei() << " int-Plätze\n";
  }
  while (si.frei () > 0);
```


Schablonen für Klassen (*Class Templates*)

```
  cout << "\n";
  do                          // char-Stack auffüllen
  {
    char ch;
    cout << "Zeichen eingeben: ";
    cin >> ch;
    sc.push (ch);     // Elementfunktion mit char-Argument
    cout << "Noch frei: " << sc.frei() << " char-Plätze\n";
  }
  while (sc.frei () > 0);
  cout << "Stacks ausleeren:\n";       // Stacks ausräumen
  for (int i = 0; i < stackgr; i++)
    cout << "int = " << si.pop() << "; char = " << sc.pop()
      << "\n";
}
```


Schablonen für Klassen (*Class Templates*)

- Die gleiche Funktionalität hätte auch *ohne* Verwendung von Schablonen erzielt werden können.
- In diesem Fall hätte jedoch für jede auf einem Stack aufzubewahrende Datentype eine *eigene* Klasse definiert werden müssen, also beispielsweise:

Schablonen für Klassen (*Class Templates*)

```
class int_stack
{
  int *basis;
  int *zeiger;
  int gr;
public:
  stack (int groesse)            // Konstruktor
  { basis = zeiger = new int [gr = groesse]; }
    // ... und so weiter
};
class char_stack
{
  char *basis;
  char *zeiger;
  int gr;
    // ... und so weiter
};
```


Schablonen für Klassen (*Class Templates*)

- Die Definition der Klassenelement-Funktionen einer Schablone kann auch außerhalb der Definition der Schablone erfolgen:

Schablonen für Klassen (*Class Templates*)

```
template <class T> class stack
{
  T *basis;
  T *zeiger;
  int gr;
public:
  stack (int);                  // Konstruktor
  ~stack ();                    // Destruktor
  void push (T);
  T pop (void);
  int frei (void);
};
template <class T> stack <T>::stack (int groesse)
{ basis = zeiger = new T [gr = groesse]; }
template <class T> stack <T>::~stack ()
{ delete [] basis; }
```


Schablonen für Klassen (*Class Templates*)

```
template <class T> void stack <T>::push (T t)
{
  if (zeiger - basis < gr)
    *zeiger++ = t;
}
template <class T> T stack <T>::pop (void)
{
  if (zeiger > basis)
    return *--zeiger;
  else
    return 0;
}
template <class T> int stack <T>::int frei (void)
{ return gr - (zeiger - basis); }
```


Objektorientierte Programmierung (OOP)

- Schablonen (*Templates*)
  - Allgemeines
  - Schablonen für Klassen (*Class Templates*)
  - **Funktions-Schablonen (*Function Templates*)**

Funktions-Schablonen (*Function Templates*)

- Beispiel: Definition einer Schablone für ein eindimensionales Feld beliebiger Größe und Type und eine Funktion, mit der die Elemente dieses Feldes sortiert werden können.
- Dieser Sortieralgorithmus ist für (fast) alle Datentypen einsetzbar:

Funktions-Schablonen (*Function Templates*)

```
#include <iostream.h>
template <class Type> class Feld
{
public:
  Feld (int groesse = 1)   // (Default-) Konstruktor
  { f = new Type [gr = groesse]; }
  ~Feld ()                 // Destruktor
  { delete[] f; }
  int gr;                  // Datenelement: Größe
  Type *f;                 // Zeiger auf Feldelemente
};
```


Funktions-Schablonen (*Function Templates*)

```
// generalisierte Sortierfunktion
template <class T> void sort (Feld<T>& F)
{
  unsigned n = F.gr;       // Feldgröße
  for (int i = 0; i < n - 1; i++)
  {
    for (int j = i + 1; j < n; j++)
    {
      if (F.f[i] > F.f[j])
      {
        T temp = F.f[i];
        F.f[i] = F.f[j];
        F.f[j] = temp;
      }
    }
  }
}
```

Funktions-Schablonen (*Function Templates*)

```
// spezielle Ausgabefunktion
void show (int zahl, Feld <char>& Fc, Feld <int>& Fi,
  Feld <double>& Fd)
{
  for (int i = 0; i < zahl; i++)
    cout << Fc.f[i] << "\t" << Fi.f[i] << "\t" <<
      Fd.f[i] << "\n";
}
int main ()
{
  int zahl;
  cout << "Feldgröße:       ";
  cin >> zahl;
  // "Felder" definieren:
  Feld <char> Fc (zahl);              // chars
  Feld <int> Fi (zahl);               // ints
  Feld <double> Fd (zahl);            // doubles
```


Funktions-Schablonen (*Function Templates*)

```
  for (int i = 0; i < zahl; i++)
  {
    cout << "Zeichen:         ";
    cin >> Fc.f[i];
    cout << "Ganze Zahl:      ";
    cin >> Fi.f[i];
    cout << "Gleitkommazahl:  ";
    cin >> Fd.f[i];
  }
  cout << "\nUnsortiert:\n";
  show (zahl, Fc, Fi, Fd);
  sort (Fc);                              // sortieren
  sort (Fi);
  sort (Fd);
  cout << "\nSortiert:\n";
  show (zahl, Fc, Fi, Fd);
}
```


Funktions-Schablonen (*Function Templates*)

- Diese Sortierfunktion ist grundsätzlich für alle Datentypen brauchbar, für die ein sinnvoller Vergleich "`if (F.f[i] > F.f[j])`" existiert.
- Für manche Datentypen (z.B. `char*` – *Strings*) gilt dies nicht; für diese ist der Vergleich mit der Bibliotheksfunktion `strcmp()` vorzunehmen.
- Für solche Fälle kann eine Vergleichsfunktion *als Schablone* definiert werden, die für fast alle Datentypen verwendet wird; für spezielle Typen (z.B. `char*`) wird nach den Regeln des *Function Overloading* eine spezielle Funktion mit gleichem Basis-Namen definiert:

Funktions-Schablonen (*Function Templates*)

```
#include <iostream.h>
#include <string.h>            // Definition von strcmp()
// generalisierte Klasse "Vergleich"
template <class T> class Vergleich
{
public:
  static int groesser (T& a, T& b)
  { return a > b; }
};
// spezielle Klasse für char*
class Vergleich <char*>
{
public:
  static int groesser (char *a, char *b)
  { return strcmp (a, b) > 0; }
};
```


Funktions-Schablonen (*Function Templates*)

- In der Sortierfunktion `sort()` wird die Zeile

  `if (F.f[i] > F.f[j])`

  ersetzt durch:

  `if (Vergleich<T>::groesser(F.f[i], F.f[j]))`

Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
- Abgeleitete Klassen
- Zugriff auf Klassen-Elemente
- Schablonen (*Templates*)
- **Spezielle Funktionen in Klassen**
- Überladen (*Overloading*)
- Konventionelle Programmierung und OOP

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - **Allgemeines**
  - Konstruktoren
  - Destruktoren
  - Temporäre Objekte
  - Konversionen
  - Kopierfunktionen
  - Spezielle Initialisierungen

Spezielle Funktionen in Klassen

- Spezielle Funktionen in Klassen können ausschließlich als Teil der Definition einer Klasse definiert werden.
- Sie bestimmen, wie Objekte der Klasse erzeugt, vernichtet, kopiert und in Elemente einer anderen Klasse konvertiert werden.
- Sie können zum Teil auch implizit vom Compiler aufgerufen werden.
- Sie unterliegen ebenso den Zugriffsregeln wie gewöhnliche Klassenelement-Funktionen.

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Allgemeines
  - **Konstruktoren**
  - Destruktoren
  - Temporäre Objekte
  - Konversionen
  - Kopierfunktionen
  - Spezielle Initialisierungen

Konstruktoren

- Klassenelement-Funktionen mit dem Namen der Klasse sind Konstruktoren.
- Konstruktoren haben keine Ergebnis-Type und dürfen auch kein Resultat zurückgeben.
- Es ist unzulässig, einen Funktionszeiger auf einen Konstruktor zu definieren.
- Entsprechend den Regeln für *Function Overloading* können beliebig viele Konstruktoren einer Klasse definiert werden.
- Wenn eine Klasse einen Konstruktor besitzt, wird jedes ihrer Objekte unter Verwendung des Konstruktors initialisiert.

Konstruktoren

- Solche Objekte können sein:
  - Globale Objekte;
  - Lokale Objekte (Gültigkeit innerhalb einer Funktion oder eines Blocks);
  - Dynamische Objekte (vom *Free Store*);
  - Temporäre Objekte (Compiler-generiert);
  - Objekte, die Elemente einer anderen Klasse sind;
  - Jener Teil des Objektes einer *abgeleiteten* Klasse, der zur *Basisklasse* gehört und bei der Erstellung eines Objektes der abgeleiteten Klasse mit erstellt wird.

Konstruktoren

- In C++ existieren zwei spezielle Typen von Konstruktoren:
  - Default-Konstruktor: Hat kein Argument oder kann ohne Argument aufgerufen werden. Wird implizit vom Compiler verwendet, um ein Standard-Objekt der Klasse zu erzeugen.
  - Kopier-Konstruktor: Ein einziges Argument, dessen Type eine Referenz auf die Klasse ist. Wird vom Compiler implizit verwendet, um bei der Erstellung eines neuen Objektes den Inhalt eines bestehenden Objektes zu duplizieren.

Konstruktoren

- Der Compiler kann einen Default- und einen Kopier-Konstruktor generieren, wenn keiner spezifiziert wurde.
- Konstruktoren initialisieren die zur Behandlung virtueller Basisklassen und Funktionen benötigten Tabellen und rufen die Konstruktoren von Basisklassen und/oder Element-Klassen auf.
- Der compilergenerierte Kopierkonstruktor kopiert zusätzlich die Elemente des Quell-Objekts.
- Wenn keine Konstruktoren von Basis- oder Element-klassen existieren, werden deren Elemente bitweise kopiert.

Konstruktoren

- Konstruktoren können beliebige Klassenelement-Funktionen aufrufen.
- Felder werden elementweise in aufsteigender Reihen-folge unter Verwendung des Default-Konstruktors erstellt.
- Konstruktoren können Objekte initialisieren, die als `const` oder `volatile` deklariert wurden, können *selbst* jedoch nicht als `const`, `volatile`, `virtual` oder `static` deklariert werden.

Konstruktoren

- Konstruktoren können explizit aufgerufen werden, beispielsweise in Funktionsaufrufen:

```
DrawLine (Point(17, 23), Point(35, 46));
```

  oder bei Initialisierungen:

```
Point pt = Point(39, 46);
```

- In einem solchen Fall existiert das Objekt, das durch den Konstruktor erstellt wird, nur für die Dauer des jeweiligen Ausdrucks.

Konstruktoren

- Objekte abgeleiteter Klassen werden durch Aufruf der Konstruktoren, beginnend mit der untersten Basisklasse, erstellt. Damit ist sichergestellt, dass beim Aufruf jedes Konstruktors *alle* seine Basisklassen vollständig aufgebaut sind.
- Analog werden Konstruktoren der Klassen der *Elemente* einer Klasse aufgerufen, bevor der Konstruktor jener Klasse ausgeführt wird, in der sie enthalten sind.

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Allgemeines
  - Konstruktoren
  - **Destruktoren**
  - Temporäre Objekte
  - Konversionen
  - Kopierfunktionen
  - Spezielle Initialisierungen

Destruktoren

- Destruktoren zerstören Klassenobjekte, die nicht mehr benötigt werden.
- Der Name des Destruktors ist immer der der Klasse mit einer vorangestellten Tilde ("~").
- Destruktoren haben keine Funktionsargumente.
- Destruktoren haben kein Ergebnis und keine Ergebnis-type.
- Destruktoren dürfen nicht als `const`, `volatile` oder `static` deklariert werden (aber auf Objekte ange-wendet werden, die als `const`, `volatile` oder `static` deklariert wurden).

Destruktoren

- Sie können als virtuelle (und sogar als reine virtuelle Funktionen in einer abstrakten Klasse) deklariert werden.
- In diesem Fall können Objekte zerstört werden, deren Type nicht bekannt ist; der korrekte Destruktor wird aufgrund des virtuellen Aufrufmechanismus automatisch gewählt.
- Es ist unzulässig, einen Funktionszeiger auf einen Destruktor zu definieren.
- Abgeleitete Klassen erben nie den Destruktor ihrer Basisklasse. Wenn für eine Klasse kein Destruktor existiert, generiert der Compiler einen solchen.

Destruktoren

- Destruktoren werden in den folgenden Fällen ausgeführt:
  - Ein mit `new` allokiertes Objekt wird mit `delete` zerstört. In diesem Fall wird immer das "am meisten abgeleitete" Objekt zerstört, sofern virtuelle Destruktoren verwendet werden.
  - Ein lokales Objekt mit Blockgültigkeit verliert seine Gültigkeit am Ende des Blocks.
  - Die Lebensdauer eines temporären Objekts endet.
  - Im Falle globaler oder statischer Objekte, wenn das Programm beendet wird.
  - Wenn der Destruktor explizit aufgerufen wird.

Destruktoren

- Wenn eine Basisklasse oder eine Element-Klasse einen zugänglichen Destruktor hat, generiert der Compiler für alle abgeleiteten Klassen einen Default-Destruktor, sofern für diese Klassen kein Destruktor definiert wurde. Dieser compilergenerierte Destruktor ist `public`; er ruft die Destruktoren der Basisklassen und der Element-Klassen auf.
- Destruktoren können beliebige Klassenelement-Funktionen aufrufen.
- Bei der Ausführung eines Destruktors werden — soweit zutreffend — die folgenden Schritte durchlaufen:

Destruktoren

- Der Destruktor der Klasse wird aufgerufen und komplett ausgeführt.
- Wenn die Klasse Elemente enthält, die Objekte anderer Klassen sind, werden für diese Objekte die Destruktoren ihrer Klassen in der umgekehrten Reihenfolge ihrer Deklaration aufgerufen.
- Die Destruktoren nicht-virtueller Basisklassen werden in der umgekehrten Reihenfolge ihrer Deklaration aufgerufen.
- Die Destruktoren virtueller Basisklassen werden in der umgekehrten Reihenfolge ihrer Deklaration aufgerufen.

Destruktoren

- Ein expliziter Aufruf eines Destruktors kann wie folgt erfolgen:

```
class Point;
Point pt;
Point *ppt = &pt;

// nicht-virtueller Aufruf:
pt.Point::~Point();          // oder
ppt->Point::~Point();

// virtueller Aufruf:
pt.~Point();                 // oder
ppt->~Point();
```


Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Allgemeines
  - Konstruktoren
  - Destruktoren
  - **Temporäre Objekte**
  - Konversionen
  - Kopierfunktionen
  - Spezielle Initialisierungen

Temporäre Objekte

- Temporäre Objekte werden automatisch vom Compiler in den folgenden Fällen erstellt:
  - Initialisierung einer (`const`) Referenz mit einer Type, die sich von der des zugrunde liegenden Objekts unterscheidet.
  - Zur Aufnahme des mit `return` zurückgegebenen Ergebnisses einer Funktion, wenn:
    - die Funktion als Resultat ein `class`- oder `struct`-*Objekt* hat, und
    - das Ergebnis der Funktion *nicht* in ein anderes Objekt kopiert wird.

Temporäre Objekte

  - Für das Ergebnis einer Typenumwandlung in eine benutzerdefinierte Type.
- Diese temporären Objekte werden automatisch wieder zerstört, sobald sie nicht mehr benötigt werden.

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Allgemeines
  - Konstruktoren
  - Destruktoren
  - Temporäre Objekte
  - **Konversionen**
  - Kopierfunktionen
  - Spezielle Initialisierungen

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Konversionen
    - **Allgemeines**
    - Konversions-Konstruktoren
    - Konversionsfunktionen

Konversionen — Allgemeines

- Die Konversion eines Klassen-Objekts in ein Objekt einer anderen Klasse erfolgt durch:
  - Konversion mittels Konstruktor.
  - Umsetzung mittels einer benutzerdefinierten Konversionsfunktion.
- Benutzerdefinierte Konversionen werden nur dann verwendet, wenn sie eindeutig sind; ansonsten erfolgt eine Fehlermeldung.

Konversionen — Allgemeines

- Konversionen erfolgen in den folgenden Fällen:
  - Explizite Konversion;
  - Initialisierung eines Objekts mit einem Ausdruck einer anderen Type;
  - Funktionsaufruf mit einem aktuellen Argument, dessen Type von der des formalen Arguments abweicht;
  - Zuweisung des Ergebnisses einer Funktion auf ein Objekt einer anderen Type;
  - Ausdruck mit Objekten verschiedener Typen;
  - Die Type eines Ausdrucks ist von der erwarteten verschieden.

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Konversionen
    - Allgemeines
    - **Konversions-Konstruktoren**
    - Konversionsfunktionen

Konversions-Konstruktoren

- Konversions-Konstruktor: Konstruktor der Ziel-Klasse der Konversion mit einem einzigen Argument mit der Type der Quell-Klasse.

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Konversionen
    - Allgemeines
    - Konversions-Konstruktoren
    - **Konversionsfunktionen**

Konversionsfunktionen

- Konversionsfunktionen erlauben die explizite Typenumwandlung durch *Type Casts* oder funktionsartigen Aufruf.
- Sie werden als Klassenelement-Funktionen mit dem Namen "`operator` *`type`*`()`" definiert.
- Sie haben keine Argumente und ein Resultat mit der durch *`type`* festgelegten Type.

Konversionsfunktionen

- Für Konversionsfunktionen gelten die folgenden Regeln:
  - Klassen, Aufzählungen und `typedefs` dürfen nicht innerhalb der Definition einer Konversionsfunktion, sondern separat zuvor definiert werden.
  - Konversionsfunktionen haben keine Argumente.
  - Die Type des Ergebnisses einer Konversionsfunktion ist durch ihren Namen festgelegt. Die Angabe einer Ergebnistype ist unzulässig.
  - Konversionsfunktionen dürfen als virtuell deklariert werden.
  - Konversionsfunktionen (und *Operator Overloading*) können zu Mehrdeutigkeiten führen.

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Allgemeines
  - Konstruktoren
  - Destruktoren
  - Temporäre Objekte
  - Konversionen
  - **Kopierfunktionen**
  - Spezielle Initialisierungen

Kopierfunktionen

- Kopiervorgänge erfolgen in zwei Fällen:
  - Initialisierung
  - Zuweisung.
- Für eine C++-Klasse können beide Kopierfunktionen unabhängig festgelegt werden:
  - *Initialisierungen* durch benutzerdefinierten Kopier-Konstruktor;
  - *Zuweisungen* durch benutzerdefinierte `operator=()`-Funktion.

Kopierfunktionen

- Kopier-Konstruktoren und Zuweisungs-Funktionen haben ein einziges Argument der Type "Referenz auf die Klasse".
- Es sollte zweckmäßigerweise als "**`const`** *`Klassenname`*`&`" definiert werden, um auch Initialisierungen durch konstante Objekte zu erlauben.
- Zuweisungsfunktionen haben die Deklaration *`type`*`& `*`type`*`::operator=(const `*`type`*`&)`. Die Deklaration der Resultattype kann unterbleiben.

Kopierfunktionen

- Wenn kein Kopier-Konstruktor bzw. keine Zuweisungs-Funktion definiert wurde, generiert der Compiler eine solche.
- Die compilergenerierten Funktionen kopieren das Quellobjekt elementweise auf das Zielobjekt.
- Das folgende Beispiel illustriert die Eigenschaften von Kopier-Konstruktor und benutzerdefinierter Zuweisungs-funktion:

Kopierfunktionen

```
#include <iostream.h>
class Point
{
public:
  // Standard- und Default-Konstruktor:
  Point (int x = 0, int y = 0)
  { _x = x; _y = y; }
  // Kopier-Konstruktor:
  Point (const Point& pt)
  { _x = 3*pt._x; _y = 3*pt._y;  }
  // Zuweisungs-Operator:
  Point& operator=(const Point& pt)
  {
    _x = 5*pt._x;
    _y = 5*pt._y;
    return *this;
  }
```


Kopierfunktionen

```
  // Ausgabe-Funktion:
  show ()
  { cout << "x = " << _x << ", y = " << _y << "\n"; }
private:
  short _x, _y;
};                      // Ende der Definition von Point
int main ()
{
  Point pt1 (10, 20);            // Standard-Konstruktor
  Point pt2 = pt1;               // Kopier-Konstruktor
  Point pt3;                     // Default-Konstruktor
  pt1.show();
  pt2.show();
  pt3.show();
  pt3 = pt1;              // benutzerdefinierte Zuweisung
  pt3.show();
}
```


Kopierfunktionen

- Zur Verdeutlichung der Mechanismen von Kopier-konstruktor und benutzerdefinierter Zuweisungsfunktion wurden die Punkt-Koordinaten mit 3 bzw. 5 multipliziert. Das Programm hat die Ausgabe:

```
x = 10, y = 20
x = 30, y = 60
x = 0, y = 0
x = 50, y = 100
```

Kopierfunktionen

- Das vorige Programm wäre auch ohne die Definitionen von `Point (const Point& pt)` und `Point& operator=(const Point& pt)` fehlerfrei übersetzbar und lauffähig; seine Ausgabe wäre in diesem Fall:

```
x = 10, y = 20
x = 10, y = 20
x = 0, y = 0
x = 10, y = 20
```


Kopierfunktionen

- Für Initialisierungen und Zuweisungen zwischen Objekten einer Basisklasse und einer abgeleiteten Klasse gilt:
  - Objekte einer *Basisklasse* können mit einem Objekt einer von dieser *abgeleiteten Klasse* initialisiert werden bzw. ihnen kann ein Objekt einer von ihnen abgeleiteten Klasse zugewiesen werden.
  - Initialisierungen und Zuweisungen in umgekehrter Richtung sind nicht zulässig.

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Allgemeines
  - Konstruktoren
  - Destruktoren
  - Temporäre Objekte
  - Konversionen
  - Kopierfunktionen
  - **Spezielle Initialisierungen**

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Spezielle Initialisierungen
    - **Felder von Klassenobjekten**
    - Klassen-Objekte als Klassenelemente
    - Initialisierung von Basisklassen

Felder von Klassenobjekten

- Felder von Klassen-Objekten werden — soweit vorhanden — unter Verwendung des Konstruktors der Klasse initialisiert.
- Wenn eine Liste von Initialisierungswerten angegeben wird, die *weniger* Elemente enthält, als zur Initialisierung des gesamten Feldes notwendig wären, werden die ersten Feldelemente mit den Werten aus der Liste initialisiert, alle weiteren unter Verwendung des Default-Konstruktors:

Felder von Klassenobjekten

```
#include <iostream.h>

class Punkt
{
public:
  Punkt (short x = 3, short y = 5)
  { _x = x; _y = y; }

  Zeige ()
  { cout << "x = " << _x << ", y = " << _y << "\n"; }

private:
  short _x, _y;
};
```

Felder von Klassenobjekten

```
int main ()
{
  Punkt pt[5] = {
    Punkt (11, 13),            // pt[0]
    Punkt (17, 19),            // pt[1]
    Punkt (23, 29) };          // pt[2]
  for (int i = 0; i < 5; i++)
    pt[i].Zeige();
}
```

- Die Ausgabe des Programms ist:

```
x = 11, y = 13
x = 17, y = 19
x = 23, y = 29
x = 3, y = 5
x = 3, y = 5
```


Felder von Klassenobjekten

- Felder von *statischen* Klassenelementen können wie "gewöhnliche" Felder bei ihrer Definition außerhalb der Klassen-Definition initialisiert werden.

Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Spezielle Initialisierungen
    - Felder von Klassenobjekten
    - **Klassen-Objekte als Klassenelemente**
    - Initialisierung von Basisklassen

Klassen-Objekte als Klassenelemente

- Klassen können Elemente enthalten, die ihrerseits Objekte einer *anderen* Klasse sind.
- Diese Objekte können aber nur unter einer der folgenden Voraussetzungen initialisiert werden:
  - Die Klasse des eingeschlossenen Objekts benötigt keinen Konstruktor.
  - Die Klasse des eingeschlossenen Objekts hat einen zugänglichen Default-Konstruktor.
  - Alle Konstruktoren der umschließenden Klasse initialisieren das Objekt explizit.

Klassen-Objekte als Klassenelemente

- Im folgenden Beispiel enthält die Klasse `Rect` zwei Objekte der Klasse `Point`:

```
#include <iostream.h>
class Point
{
public:
  Point (short x = 0, short y = 0)
  { _x = x; _y = y; } // Standard- und Default-Konstruktor
  Show ()
  { cout << "x = " << _x << ", y = " << _y << "\n"; }
private:
   short _x, _y;
};
```


Klassen-Objekte als Klassenelemente

```
class Rect
{
public:
  Rect (short x1, short y1, short x2, short y2);
                                          // Deklaration
  Show ()
  {
    cout << "Punkt 1: ";
    _pt1.Show();
    cout << "Punkt 2: ";
    _pt2.Show();
  }
private:
  Point _pt1, _pt2;
};
```

Klassen-Objekte als Klassenelemente

```
// Definition des Konstruktors von Rect
Rect::Rect (short x1, short y1, short x2, short y2) :
   _pt1 (x1, y1), _pt2 (x2, y2)
  { }                    // LEERE Funktionsdefinition!
int main ()
{
  Rect r (1, 2, 3, 4);
  r.Show();
}
```

- Der Konstruktor von `Rect` ruft zweimal den Konstruktor von `Point` auf, um die Objekte `_pt1` bzw. `_pt2` zu initialisieren. Das Programm hat die Ausgabe:

```
Punkt 1: x = 1, y = 2
Punkt 2: x = 3, y = 4
```


Objektorientierte Programmierung (OOP)

- Spezielle Funktionen in Klassen
  - Spezielle Initialisierungen
    - Felder von Klassenobjekten
    - Klassen-Objekte als Klassenelemente
    - **Initialisierung von Basisklassen**

Initialisierung von Basisklassen

- Basisklassen werden im wesentlichen ebenso initialisiert wie Objekte von Klassen als Klassenelemente:

```
class Window                        // Basisklasse
{
public:
  Window (Rect rSize);              // Konstruktor
};
class DialogBox : public Window     // abgeleitete Klasse
{
public:
   DialogBox (Rect rSize);         // Konstruktor
};
// Definition des Konstruktors von DialogBox
DialogBox::DialogBox (Rect rSize) : Window (rSize)
{ }
```


Initialisierung von Basisklassen

- Bei der Erstellung eines Objekts der Type `DialogBox` wurde hier der Konstruktor der Klasse `Window` mit dem Argument `rSize` aufgerufen. Im obigen Beispiel erfolgte keine Initialisierung der Klassenelemente von `DialogBox` (wohl aber eine der Elemente von `Window`).

Initialisierung von Basisklassen

- Die Elemente einer virtuellen Basisklasse müssen entweder explizit vom Konstruktor der "am meisten abgeleiteten" Klasse initialisiert werden, oder die virtuelle Basisklasse muss einen Default-Konstruktor haben.
- Die Initialisierung der der virtuellen Basisklasse zugehörigen Objekte innerhalb einer anderen abgeleiteten Klasse als der "am meisten abgeleiteten" wird ignoriert.
- Das folgende Demo-Programm basiert in wesentlichen Teilen auf dem "Bibliotheksverwaltungsprogramm" von Abschnitt "Einfache Vererbung":

Initialisierung von Basisklassen

```
#include <iostream.h>
#include <string.h>          // für strcpy() und strncpy()
class Dokument
{
public:
  Dokument() { anzahl++; }
  Dokument (const char *s1, const char *s2, int i)
  {
    strncpy (_s1, s1, sizeof (_s1) - 1);
    _s1 [sizeof (_s1) - 1] = 0;
    strncpy (_s2, s2, sizeof (_s2) - 1);
    _s2 [sizeof (_s2) - 1] = 0;
    _i = i;
    anzahl++;
  }
```

Initialisierung von Basisklassen

```
  virtual ~Dokument() { anzahl--; }
  virtual void zeige() { } = 0;
  static int anzahl;    // Anzahl der definierten Objekte
protected:
  char _s1[50], _s2[30];
  int _i;
};
int Dokument::anzahl = 0;        // Definiere anzahl
```


Initialisierung von Basisklassen

```
class Buch : public Dokument
{
public:
  Buch (const char *Titel, const char *Autor,
    int Seiten) : Dokument (Titel, Autor, Seiten) { }
  void zeige ()
  {
    cout << "Autor:  " << _s2 << ", Titel: "
      << _s1 << "\n\t" << _i << " Seiten\n";
  }
};
```


Initialisierung von Basisklassen

```
class Manual : public Dokument
{
public:
  Manual (const char *Titel, int Seiten) :
    Dokument (Titel, "", Seiten) { }
  void zeige ()
  {
    cout << "Manual: " << _s1 << " - " << _i
      << " Seiten\n";
  }
};
```


Initialisierung von Basisklassen

```
class Datei : public Dokument
{
public:
  Datei (const char *Inhalt, const char *Pfad,
    int Bytes) : Dokument (Inhalt, Pfad, Bytes) { }
  void zeige ()
  {
    cout << "Datei:  " << _s1 << ", Pfad: " << _s2
      << "\n\t" << _i << " Bytes\n";
  }
};

const int max_Eintr = 6;

Dokument *doc[max_Eintr];        // Feld von Zeigern
```


Initialisierung von Basisklassen

```
int main () {
  doc[0] = new Buch ("The C++ Programming Language",
    "Bjarne Stroustrup", 669);
  doc[1] = new Buch ("C - The Complete Reference",
    "Herbert Schildt", 773);
  doc[2] = new Manual ("Microsoft Visual C++ - C++ "
    "Language Reference", 468);
  doc[3] = new Manual ("Microsoft Quick C Run Time "
    "Library Reference", 687);
  doc[4] = new Datei ("GNU C++ Compiler-Dokumentation",
    "c:\\gnu\\docs\\gcc\\gcc.tex", 173664);
  doc[5] = new Datei ("GNU C Run Time Library Reference",
    "c:\\gnu\\docs\\djgpp\\libcref.i", 226882);
  cout << Dokument::anzahl << " Eintragungen:\n";
  for (int i = 0; i < Dokument::anzahl; i++)
    doc[i]->zeige();              // Liste ausschreiben
}
```


Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
- Abgeleitete Klassen
- Zugriff auf Klassen-Elemente
- Schablonen (*Templates*)
- Spezielle Funktionen in Klassen
- **Überladen (*Overloading*)**
- Konventionelle Programmierung und OOP

Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - **Überladen von Funktionen (*Function Overloading*)**
  - Überladen von Operatoren (*Operator Overloading*)

Function Overloading

- "Überladen" von Funktionen (*Function Overloading*) erlaubt die Definition von Funktionen mit identischen Namen, aber mit unterschiedlicher Zahl und Type der formalen Argumente.
- Der Compiler wählt zum Zeitpunkt der Übersetzung die passende Funktion aufgrund der aktuellen Argumente nach folgenden Regeln (mit abnehmender Präzedenz).
- Existieren mehrere gleichnamige Funktionen mit gleicher Präzedenz, so erfolgt eine Fehlermeldung.
  - Eine exakte Übereinstimmung besteht.

Function Overloading

- Eine triviale Konversion ist erforderlich:
  - Objekt in Referenz oder umgekehrt;
  - eindimensionales Feld in Zeiger;
  - Funktionsaufruf in Funktionszeiger-Aufruf ;
  - Objekt oder Zeiger in `volatile` oder `const` Objekt oder Zeiger.
- *Integral Promotion* ist erforderlich.
- Eine Standard-Konversion existiert.
- Eine benutzerdefinierte Konversion existiert.
- Eine variable Anzahl von Funktionsargumenten ("`...`") war deklariert.

Function Overloading

- Funktionen mit *n* voreingestellten Argumenten werden als Satz von *n*+1 individuell verschiedenen Funktionen betrachtet.
- Nicht-statische Klassenelement-Funktionen haben einen impliziten `this`-Zeiger; die Type dieses `this`-Zeigers muss mit der des entsprechenden Arguments übereinstimmen.
- Die Auswahlregeln für *overloaded* Funktionen implizieren die Regeln, wodurch sich solche Funktionen voneinander unterscheiden *müssen*, um als unterschiedliche Funktionen zu gelten:

Function Overloading

- Anzahl und/oder Typen der Argumente müssen unterschiedlich sein.
- Unterschiede im Typ des Resultats allein sind *nicht* ausreichend.
- In der Argumentliste gelten Objekte und Referenzen auf ein Objekt gleichen Typs als identisch:

```
int func (int&, double&);
```

ist identisch zu

```
int func (int, double);
```


Function Overloading

- Eindimensionale Felder und Zeiger der gleichen Type sind identisch.
- Mehrdimensionale Felder sind dann unterschiedlich, wenn sich ihre zweiten und folgenden Felddimensionen voneinander unterscheiden.
- Klassenelement-Funktionen dürfen sich nicht nur darin unterscheiden, dass eine `static` ist und die andere nicht.
- `typedef`-Definitionen werden in die ihnen zugrunde liegende Type umgesetzt. Eine Type und ein ihr mit `typedef` zugewiesenes Synonym sind identisch.

Function Overloading

- Aufzählungstypen können zur Unterscheidung herangezogen werden.
- Wenn Referenzen oder Zeiger mit `const` oder `volatile` als Attribut versehen sind, werden sie als unterschiedlich von der Basistype betrachtet.

- Funktionen werden nur dann als überladen behandelt, wenn sie sich innerhalb desselben Gültigkeitsbereichs befinden:
  - Funktionen innerhalb eines übergeordneten Gültigkeitsbereichs werden verborgen;
  - Funktionen mit gleichem Namen in einer Basisklasse werden verborgen.

Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Funktionen (*Function Overloading*)
  - **Überladen von Operatoren (*Operator Overloading*)**

Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Operatoren (*Operator Overloading*)
    - **Allgemeines**
    - Unäre Operatoren
    - Binäre Operatoren
    - Der Zuweisungsoperator (`operator=`)
    - Der Funktionsaufruf (`operator()`)
    - Der Feldindex-Operator (`operator[]`)
    - Die Elementauswahl-Operatoren
    - Die Operatoren `new` und `delete`

Operator Overloading — Allgemeines

- Die Funktion der meisten Operatoren von C++ kann bei Bedarf für bestimmte Datentypen geändert werden.
- Diese Neudefinition ist entweder global oder im Rahmen einer Klasse möglich.
- Die Implementierung erfolgt durch globale oder Klassenelement-Funktionen.
- Der Name einer Funktion zur Definition eines überladenen Operators ist `operator` *op*, wobei *op* einer der C++-Operatoren mit Ausnahme der folgenden ist:

Operator Overloading — Allgemeines

- Operatoren, die *nicht* durch Überladen von Operatoren (*Operator Overloading*) umdefiniert werden können:

Operator	Bezeichnung
`.`	Elementauswahl
`.*`	Elementauswahl mit Zeiger auf Element
`::`	Gültigkeitsbereich
`? :`	Bedingte Ausführung
`sizeof`	Objektgröße
`#`	Präprozessor-Befehl
`##`	Präprozessor-Befehl

Operator Overloading — Allgemeines

- Die folgenden Regeln für *Operator Overloading* gelten generell (mit Ausnahme der Operatoren `new` und `delete`):
- Operator-Funktionen müssen Klassenelement-Funktionen sein oder ein Argument haben, das einer Klassen- oder Aufzählungstype angehört oder eine Referenz auf eine solche ist.
- Es dürfen beliebig viele Implementierungen für einen Operator vorgesehen werden.

Operator Overloading — Allgemeines

- Für benutzerdefinierte Implementierungen eines Operators gelten die gleichen Regeln bezüglich Präzedenz, Anzahl und Anordnung der Operanden wie für die Standard-Definition.
- Ein als *Klassenelement*-Funktion definierter *unärer* Operator hat *kein* Argument; als *globale* Funktion benötigt er *ein* Argument.
- Ein als *Klassenelement*-Funktion deklarierter *binärer* Operator benötigt *ein* Argument; als *globale* Funktion benötigt er *zwei*.

Operator Overloading — Allgemeines

- Die Verwendung von voreingestellten Argumenten für überladene Operatoren ist unzulässig.
- Abgeleitete Klassen erben alle Operatoren ihrer Basisklasse mit Ausnahme des `operator=`.
- Das erste Argument von als Klassenelement-Funktionen definierten überladenen Operatoren hat immer die Type der Klasse, für/mit der der Operator aufgerufen wird (also die Type der Klasse, in der er definiert wurde, oder einer Basisklasse davon). Für dieses Argument werden keine wie immer gearteten Typ-Konversionen vorgenommen.

Operator Overloading — Allgemeines

- *Operator Overloading* erlaubt eine völlige Umdefinition des Verhaltens von Operatoren. Die erwartete Semantik sollte jedoch erhalten bleiben; beispielsweise sollten also die folgenden Ausdrücke äquivalent bleiben:

```
var = var + 1;
var += 1;
var++;
++var;
```


Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Operatoren (*Operator Overloading*)
    - Allgemeines
    - **Unäre Operatoren**
    - Binäre Operatoren
    - Der Zuweisungsoperator (`operator=`)
    - Der Funktionsaufruf (`operator()`)
    - Der Feldindex-Operator (`operator[]`)
    - Die Elementauswahl-Operatoren
    - Die Operatoren `new` und `delete`

Unäre Operatoren

- Als *Klassenelement-Funktionen* werden überladene *unäre* Operatoren folgendermaßen deklariert:

```
Ergebnistype operator op ();
```

- Als *globale Funktionen* müssen sie so deklariert werden:

```
Ergebnistype operator op (arg);
```

- *`Ergebnistype`* ist die beliebige und durch nichts eingeschränkte Type des Ergebnisses der Operator-Funktion, *`op`* einer der unären Operatoren, und *`arg`* ein Argument mit Klassen- oder Aufzählungstyp.

Unäre Operatoren

- Bei den Inkrement- und Dekrement-Operatoren ("++" bzw. "--") wird folgendermaßen zwischen der Präfix- ("++i") und der Postfix-Version ("i++") unterschieden:
  - Die Präfix-Version des Operators wird wie jede andere unäre Operator-Funktion deklariert;
  - die Postfix-Version hat ein zusätzliches Argument vom Typ `int`.
- Das folgende Beispiel illustriert die Definition und Anwendung dieser beiden Operatoren am Beispiel einer Klasse Punkt:

Unäre Operatoren

```
#include <iostream.h>
class Punkt
{
public:
  // Deklaration der Inkrement-Operatoren
  Punkt& operator++();           // Präfix
  Punkt operator++(int);         // Postfix
  // Definition der Dekrement-Operatoren
  Punkt& operator--()            // Präfix
  {
    _x--;                        // Standard-Dekrement
    _y--;
    return *this;
  }
```


Unäre Operatoren

```
  Punkt operator--(int)          // Postfix
  {
    Punkt temp = *this;          // temporäres Objekt
    --*this;                     // ruft operator--() auf
    return temp;
  }
  Punkt (const short& x = 0, const short& y = 0)
  { _x = x; _y = y; }            // (Default-) Konstruktor
  Zeige(char *s)                 // Anzeigefunktion
  { cout << s << ": x = " << _x << ", y = " << _y << "\n"; }
private:
  short _x, _y;
};
```


Unäre Operatoren

```
// Definition der Inkrement-Operatoren
Punkt& Punkt::operator++()       // Präfix
{
  _x++;                          // Standard-Inkrement
  _y++;
  return *this;
}
Punkt Punkt::operator++(int)     // Postfix
{
  Punkt temp = *this;            // temporäre Kopie
  ++*this;                       // ruft operator++() auf
  return temp;
}
```


Unäre Operatoren

```
int main ()
{
  Punkt p1 (3, 5);
  p1.Zeige("p1");
  ++p1;                          // Präfix-Inkrement
  p1.Zeige("p1");
  Punkt p2 = p1++;               // Postfix-Inkrement
  p1.Zeige("p1");
  p2.Zeige("p2");
  --p1;                          // Präfix-Dekrement
  p1.Zeige("p1");
  p2 = p1--;                     // Postfix-Dekrement
  p1.Zeige("p1");
  p2.Zeige("p2");
}
```


Unäre Operatoren

- Alternativ hätten die Funktionen `operator++()` und `operator--()` global definiert und als `friend` deklariert werden können:

Unäre Operatoren

```
#include <iostream.h>
class Punkt
{
public:
  friend Punkt& operator++(Punkt&);// Präfix-Operatoren
  friend Punkt& operator--(Punkt&);
  friend Punkt operator++(Punkt&, int);// Postfix-Operatoren
  friend Punkt operator--(Punkt&, int);
  Punkt (const short& x = 0, const short& y = 0)
  { _x = x; _y = y; }              // (Default-) Konstruktor
  Zeige(char *s) // Zugriff
  { cout << s << ": x = " << _x << ", y = " << _y << "\n"; }
private:
  short _x, _y;
};
```

Unäre Operatoren

```
// Definition der Inkrement-Operatoren
Punkt& operator++(Punkt& p)          // Präfix
{
  p._x++;                            // Standard-Inkrement
  p._y++;
  return p;
}
Punkt operator++(Punkt& p, int)      // Postfix
{
  Punkt temp = p;                    // temporäre Kopie
  ++p;                               // ruft operator++() auf
  return temp;
}

// Die Definition der Dekrement-Operatoren erfolgt analog
```


Unäre Operatoren

- Das `int`-Argument der Postfix-Version der Inkrement- und Dekrement-Operatoren wird normalerweise nicht benutzt, obwohl dies prinzipiell möglich wäre.
- Es enthält bei gewöhnlichem Aufruf des Operators den Wert `0`.
- Eine Übergabe des `int`-Arguments an die Funktion `operator++()` ist nur durch expliziten Aufruf wie im folgenden Beispiel möglich:

Unäre Operatoren

```
#include <iostream.h>
class Int                            // Neudefinition von int
{
public:                 // Default- und Kopier-Konstruktor
  Int (const int& i = 0) { _i = i; }
  Int& operator++(int n) {
    if (n)                           // Spezial-Aufruf
      _i += n;
    else                             // "gewöhnlicher" Aufruf
      _i++;
    return *this;
  }
  Show() { cout << "i = " << _i << "\n"; }
private:
  int _i;
};
```


Unäre Operatoren

```
int main()
{
  Int i = 3;                         // "Int", NICHT "int"
  i++;                               // "gewöhnlicher" Aufruf
  i.Show();
  i.operator++ (7);                  // Spezial-Aufruf
  i.Show();
}
```


Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Operatoren (*Operator Overloading*)
    - Allgemeines
    - Unäre Operatoren
    - **Binäre Operatoren**
    - Der Zuweisungsoperator (`operator=`)
    - Der Funktionsaufruf (`operator()`)
    - Der Feldindex-Operator (`operator[]`)
    - Die Elementauswahl-Operatoren
    - Die Operatoren `new` und `delete`

Binäre Operatoren

- Als Klassenelement-Funktionen werden überladene binäre Operatoren folgendermaßen deklariert:
  `Ergebnistype operator op (arg);`
- Als globale Funktionen müssen sie so deklariert werden:
  `Ergebnistype operator op (arg1, arg2);`
- *`Ergebnistype`* ist die beliebige und durch nichts eingeschränkte Type des Ergebnisses der Operator-Funktion, *`op`* einer der binären Operatoren, und *`arg`* (bzw. *`arg1`* und *`arg2`*) ein Argument.

Binäre Operatoren

- *arg* kann von beliebiger Type sein, während von *arg1* und *arg2* mindestens eines eine Klassen- oder Aufzählungstype haben muss.
- Spezielle binäre Operatoren sind der Zuweisungs-Operator ("="), der Funktionsaufruf ("( )"), der Feldindex-Operator ("[ ]") und die Elementauswahl-Operatoren ("->" und "->*").

Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Operatoren (*Operator Overloading*)
    - Allgemeines
    - Unäre Operatoren
    - Binäre Operatoren
    - **Der Zuweisungsoperator (`operator=`)**
    - Der Funktionsaufruf (`operator()`)
    - Der Feldindex-Operator (`operator[]`)
    - Die Elementauswahl-Operatoren
    - Die Operatoren `new` und `delete`

Der Zuweisungsoperator (`operator=`)

- Die Deklaration des Zuweisungsoperators ist identisch zu der aller anderen binären Operatoren, mit den folgenden *Ausnahmen*:
- Die Funktion `operator=` *muss* eine *nichtstatische Klassenelement*-Funktion sein. Eine Definition als globale Funktion ist unzulässig.
- Die Funktion `operator=` wird nicht von abgeleiteten Klassen geerbt.
- Sofern keine explizite Definition von `operator=` erfolgte, generiert der Compiler eine solche Funktion.
- Das folgende Beispiel zeigt eine Implementierung von `operator=`:

Der Zuweisungsoperator (`operator=`)

```
#include <iostream.h>
class Point
{
public:
  Point (int x = 0, int y = 0)
  { _x = x; _y = y; }  // Standard- und Default-Konstruktor
  Point& operator=(const Point& pt)  // Zuweisungs-Operator
  {
    _x = 3*pt._x;
    _y = 5*pt._y;
    return *this;
  }
  show ()                          // Ausgabe-Funktion
  { cout << "x = " << _x << ", y = " << _y << "\n"; }
private:
  short _x, _y;
};
```


Der Zuweisungsoperator (`operator=`)

```
int main ()
{
  Point pt1 (10, 20);
  Point pt2, pt3;
  pt1.show();
  pt2.show();
  pt3.show();
  pt3 = pt2 = pt1;
            // mehrfache benutzerdefinierte Zuweisung
  pt1.show();
  pt2.show();
  pt3.show();
}
```


Der Zuweisungsoperator (`operator=`)

- Die `operator=`-Funktion gibt den Wert des Objekts, für das sie aufgerufen wurde (`*this`), als Resultat zurück. Das ist notwendig, um die auch sonst in C++ übliche mehrfache Zuweisung realisieren zu können.

Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Operatoren (*Operator Overloading*)
    - Allgemeines
    - Unäre Operatoren
    - Binäre Operatoren
    - Der Zuweisungsoperator (`operator=`)
    - **Der Funktionsaufruf (`operator()`)**
    - Der Feldindex-Operator (`operator[]`)
    - Die Elementauswahl-Operatoren
    - Die Operatoren `new` und `delete`

Der Funktionsaufruf (`operator()`)

- Der Funktionsaufruf-Operator muss eine *nichtstatische Klassenelement*-Funktion sein.
- Er erlaubt die Ausführung von Operationen, die eine beliebige Zahl von Argumenten (Operanden) benötigen.
- Das folgende Beispiel zeigt eine Anwendung, die eine Änderung der Koordinaten eines Punktes in *einer* Operation erlaubt:

Der Funktionsaufruf (`operator()`)

```
#include <iostream.h>
class Punkt
{
public:
  Punkt() { _x = _y = 0; }
  Punkt& operator() (short dx, short dy)
  {
    _x += dx;
    _y += dy;
    return *this;
  }
  Zeige()
  { cout << "x = " << _x << ", y = " << _y << "\n"; }
private:
  short _x, _y;
};
```


Der Funktionsaufruf (`operator()`)

```
int main()
{
  Punkt pt;
  pt.Zeige();
  pt (3, 5);
  pt.Zeige();
}
```

- Die Funktionsklammern folgen hier dem Namen eines *Objekts* und nicht dem einer *Funktion*!
- *Operator Overloading* des Funktionsaufruf-Operators definiert nicht den existierenden Funktionsaufruf-Operator neu, sondern *erweitert* seine Anwendbarkeit im Zusammenhang mit Objekten, die *nicht* Funktionen sind.

Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Operatoren (*Operator Overloading*)
    - Allgemeines
    - Unäre Operatoren
    - Binäre Operatoren
    - Der Zuweisungsoperator (`operator=`)
    - Der Funktionsaufruf (`operator()`)
    - **Der Feldindex-Operator (`operator[]`)**
    - Die Elementauswahl-Operatoren
    - Die Operatoren `new` und `delete`

Der Feldindex-Operator (`operator[]`)

- Der Feldindex-Operator ist ein *binärer* Operator.
- Er muss eine *nichtstatische Klassenelement*-Funktion sein mit genau einem Argument von beliebiger Type, das den gewünschten Feldindex angibt.
- Beispiel: Prüfung der Feldgrenzen eines eindimensionalen Feldes (Vektors):

Der Feldindex-Operator (`operator[]`)

```
#include <iostream.h>
class IntVektor
{
public:
  int& operator[](int iIndex)
  {
    static int iFehler = 0;           // "Schutz"-Variable
    if (iIndex >= 0 && iIndex < _iG)
      return _iE [iIndex];
    else                              // Überlauf
    {
      cout << "Ueberlauf Element [" << iIndex << "] !\n";
      return iFehler;
    }
  }
```


Der Feldindex-Operator (`operator[]`)

```
  IntVektor (int nElem = 1)   // Konstruktor
  {
    _iE = new int [nElem];    // Feld allokieren
    _iG = nElem;              // Feldgröße
  }
  ~IntVektor()                // Destruktor
  { delete[] _iE; }
private:
  int *_iE;                   // Zeiger auf Feldelemente
  int _iG;                    // Feldgröße
};
```


Der Feldindex-Operator (`operator[]`)

```
int main()
{
  IntVektor v (10);           // Vektor mit 10 Elementen
  for (int i = -1; i <= 10; i++)
    v[i] = i;                 // initialisiere Feld
  v[2] = v[7];                // teste Zuweisung
  cout << "Lesen ...\n";
  for (i = -1; i <= 10; i++)  // Ausgabe
    cout << "Element v[" << i << "] = " << v[i] << "\n";
}
```


Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Operatoren (*Operator Overloading*)
    - Allgemeines
    - Unäre Operatoren
    - Binäre Operatoren
    - Der Zuweisungsoperator (`operator=`)
    - Der Funktionsaufruf (`operator()`)
    - Der Feldindex-Operator (`operator[]`)
    - **Die Elementauswahl-Operatoren**
    - Die Operatoren `new` und `delete`

Die Elementauswahl-Operatoren

- Die Elementauswahl-Operatoren "`->`" und "`->*`" gelten als *unäre* Operatoren.
- Sie müssen im Rahmen einer Klasse definiert werden und einen Zeiger auf diese Klasse als Ergebnis zurückgeben.
- Eine Umdefinition von "`.`" und "`.*`" ist jedoch *nicht* möglich.

Objektorientierte Programmierung (OOP)

- Überladen (*Overloading*)
  - Überladen von Operatoren (*Operator Overloading*)
    - Allgemeines
    - Unäre Operatoren
    - Binäre Operatoren
    - Der Zuweisungsoperator (`operator=`)
    - Der Funktionsaufruf (`operator()`)
    - Der Feldindex-Operator (`operator[]`)
    - Die Elementauswahl-Operatoren
    - **Die Operatoren `new` und `delete`**

Die Operatoren new und delete

- Die Funktion `operator new()` muss mindestens ein Argument der Type `size_t` (definiert in `stddef.h`) haben; weitere Argumente mit beliebiger Zahl und Type sind möglich.
- Das Ergebnis von `operator new()` hat immer die Type `void*`.
- Die Funktion `operator delete()` muss mindestens ein Argument der Type `void*` haben; optional ein weiteres Argument der Type `size_t` (für Größe des Speicherblocks).
- `operator delete()` hat die Ergebnistype `void`.

Die Operatoren new und delete

- Bei Definition als Klassenelement-Funktionen sind beide Funktionen statische; sie können daher nicht virtuell sein.
- Für die Allokierung von Objekten der fundamentalen Typen und von Klassen, die keine Definition für `operator new()` vorgesehen haben, wird *immer* der globale (Standard-) `operator new()` verwendet.
- Beispiel: Buchführung über Speicherallokationen:

Die Operatoren new und delete

```
#include <iostream.h>
#include <stdlib.h>        // für size_t
#include <malloc.h>
int _fLog = 0;      // Flag: "Buchführung" ein/aus
int BAll = 0;       // Zahl der allokierten Speicherblöcke
void *operator new (size_t n)          // global definiert
{
  static int in_new = 0;               // Schutz-Flag
  if (_fLog && ! in_new)  // verhindere rekursive Zugriffe
  {
    in_new = 1;
    cout << "Speicherblock " << ++BAll << " mit " << n <<
      " Bytes allokiert\n";
    in_new = 0;
  }
  return malloc (n);
}
```


Die Operatoren new und delete

```
void operator delete (void *pMem)      // global definiert
{
  static int in_del = 0;               // Schutz-Flag
  if (_fLog && ! in_del)  // verhindere rekursive Zugriffe
  {
    in_del = 1;
    cout << "Speicherblock deallokiert; noch "
      << --BAll << " Blöcke\n";
    in_del = 0;
  }
  free (pMem);
}
```


Die Operatoren new und delete

```
int main ()
{
  char *Blocks[5];
  _fLog = 1;                    // Buchführung einschalten
  for (int i = 0; i < 5; i++)
    Blocks[i] = new char[100];
  for (i = 0; i < 5; i++)
    delete Blocks[i];
}
```


Objektorientierte Programmierung (OOP)

- Eigenschaften von Klassen
- Abgeleitete Klassen
- Zugriff auf Klassen-Elemente
- Schablonen (*Templates*)
- Spezielle Funktionen in Klassen
- Überladen (*Overloading*)
- **Konventionelle Programmierung und OOP**

Objektorientierte Programmierung (OOP)

- Konventionelle Programmierung und OOP
  - **Vorteilhafte Anwendungsfälle für OOP**
  - OOP-Mechanismen und ihre Konsequenzen
  - Vorteile von OOP
  - Strategien zum Einsatz von OOP

Vorteilhafte Anwendungsfälle für OOP

- Grundsätzlich sind alle Problemstellungen auch mit *konventioneller* Programmierung bearbeitbar.
- *Vorteile* der *objektorientierten* Programmierung in den folgenden Situationen:
  - Größere Zahl von Objekten in einem System (Programm oder Gruppe von Programmen), die *ähnliche*, aber doch *unterschiedliche* Behandlung erfordern, z.B.:
    - Datenbanken für heterogene Objekte;
    - graphische Systeme.

Vorteilhafte Anwendungsfälle für OOP

- Arithmetische und logischen Operationen mit Objekten:
  - Komplexe Zahlen;
  - Vektoroperationen.
- Details der Implementierung von Funktionen sind uninteressant.
- *Abstrakte Konzepte* für Operationen an oder mit möglichst vielen verschiedenartigen Objekten:
  - Warteschlangen, Stacks u.ä.;
  - Daten-Kommunikation.

Objektorientierte Programmierung (OOP)

- Konventionelle Programmierung und OOP
  - Vorteilhafte Anwendungsfälle für OOP
  - **OOP-Mechanismen und ihre Konsequenzen**
  - Vorteile von OOP
  - Strategien zum Einsatz von OOP

OOP-Mechanismen und ihre Konsequenzen

- Klassenspezifische Funktionen und Operatoren als Teil einer Klasse definiert— *Function* und *Operator Over-loading*; *virtuelle Funktionen*:
  - Es ist unmöglich, ein Objekt mit einer ungeeigneten Funktion zu bearbeiten.
  - *Ein* konsistenter Name für *überladene* Funktionen erlaubt Konzentration auf die *Operationen* und nicht auf den Funktionsaufruf.

OOP-Mechanismen und ihre Konsequenzen

- Beziehung verwandter Objekt-Typen auf gemeinsame Basis-Typen — *Vererbung*:
  - *Gemeinsamkeiten* stehen in den Basisklassen, *Unterschiede* in den abgeleiteten Klassen.
  - Nebenwirkungen von Änderungen und Ergänzungen an Struktur oder Definition der Objekt-Typen sind weniger wahrscheinlich.

OOP-Mechanismen und ihre Konsequenzen

- Die Daten und Funktionen sind eng verbunden — *Einkapselung*:
  - Unerwünschte Auswirkungen von Änderungen oder Erweiterungen des Programms werden unwahrscheinlicher.
- Grundlegende Funktionen in allgemeiner Form definiert — *Schablonen*:
  - Die Programmentwicklung wird einfacher, die Programme übersichtlicher und die Gefahr von Fehlern geringer.

Objektorientierte Programmierung (OOP)

- Konventionelle Programmierung und OOP
  - Vorteilhafte Anwendungsfälle für OOP
  - OOP-Mechanismen und ihre Konsequenzen
  - **Vorteile von OOP**
  - Strategien zum Einsatz von OOP

Vorteile von OOP

- Programme sind besser strukturiert und daher übersichtlicher.
- Fehlerquellen fallen weg, oder Fehler können vom Compiler erkannt werden:
  - Strikte Typenprüfung;
  - weniger globale Variable;
  - Datenobjekte innerhalb der Definition einer Klasse eingekapselt.

Vorteile von OOP

- Die *Wartbarkeit* eines Programms wird verbessert:
  - Erweiterung des Funktionsumfangs erfordert weniger Aufwand;
  - Nebenwirkungen leichter vermeidbar.
- Einsatz objektorientierter Methoden erfordert im allgemeinen etwas größeren *Planungsaufwand*:
  - Konzepte *vor* dem Beginn der eigentlichen Programmierarbeit erstellen und potentielle Erweiterungen überlegen.

Vorteile von OOP

- Objektorientierte Methoden sind — je nach Aufgabenstellung — in unterschiedlichem Umfang einsetzbar; sinnvoll dann und nur dann, wenn mit einem Vorteil im Hinblick auf Klarheit, Wartbarkeit und/oder Einfachheit des Programms verbunden.

Objektorientierte Programmierung (OOP)

- Konventionelle Programmierung und OOP
  - Vorteilhafte Anwendungsfälle für OOP
  - OOP-Mechanismen und ihre Konsequenzen
  - Vorteile von OOP
  - **Strategien zum Einsatz von OOP**

Strategien zum Einsatz von OOP

- Enthält das Programm Daten, die irgendwie zusammengehören (logisch oder physikalisch, weil z.B. gemeinsam abzuspeichern)?
  - Wenn ja: Zusammengehörige Daten in Klassen-Objekten (typisch `structs`) zusammenfassen.
  - Beispiele: Konfigurations-Parameter eines Programms; lokale Daten einer Funktion, die diese an weitere untergeordnete Funktionen weitergeben soll.

Strategien zum Einsatz von OOP

- Ist die Anzahl der zusammenzufassenden Datenelemente groß, und lassen sie sich sinnvoll in *Untergruppen* einteilen?
  - Wenn ja: Klassen (wieder meist `structs`) für die Untergruppen definieren; diese Untergruppen können zweckmäßigerweise in einer "Container-Klasse" zusammengefasst werden.
  - Beispiele: Konfigurations-Daten; generell: stark strukturierte Daten in wenigen Objekten mit vielen Elementen.

Strategien zum Einsatz von OOP

- Gibt es Datentypen, die teils gemeinsame und teils individuell verschiedene Elemente oder Eigenschaften haben?
  - Wenn ja: Die gemeinsamen Eigenschaften (Elemente) in einer *Basisklasse* zusammenfassen; die individuellen Unterschiede in von der Basisklasse *abgeleiteten* Klassen definieren.
  - Beispiele: Datenbanken mit heterogenen Objekten; Behandlung der Ein- oder Ausgabe in graphisch orientierten Systemen.

Strategien zum Einsatz von OOP

- Gibt es Funktionen, die ausschließlich oder vorwiegend auf Datenelemente *einer* Klasse zugreifen, und die sich je nach dem Typ des betroffenen Klassenobjekts (potentiell) unterschiedlich verhalten sollen?
  - Wenn ja: Diese Funktionen als Elemente dieser Klassen unter Verwendung von *Function Overloading* und virtuellen Funktionen definieren.
  - Beispiele: Funktionen für die Ein- und Ausgabe von Daten in einem Datenbankprogramm; Ausgabe graphischer Objekte.

Strategien zum Einsatz von OOP

- Wird die Erstellung und/oder Wartung des Programms durch Verwendung speziell für die Bearbeitung von bestimmten Objekten definierter Operatoren erleichtert?
  - Wenn ja: Operatoren als Klassenelement-Funktionen (mit *Operator Overloading*) definieren.
  - Beispiele: Komplexe Arithmetik; Vektorrechnung.
- Soll ein *Konzept* auf mehrere verschiedene Datentypen angewandt werden?
  - Wenn ja: Dieses Konzept als *Template* definieren.
  - Beispiele: Warteschlangen, Stacks, Sortieren …