Programmieren / Algorithmen & Datenstrukturen 2

Abgeleitete Klassen

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Abgeleitete Klassen

Inhalt.

- Templates
- Abgeleitete Klassen
- Testgetriebene Programmierung
- Container, Iteratoren und Algorithmen der StdLib
- Fortgeschrittenes Suchen
- Fortgeschrittenes Sortieren
- Grafische Benutzeroberflächen

Generalisierung / Vererbung

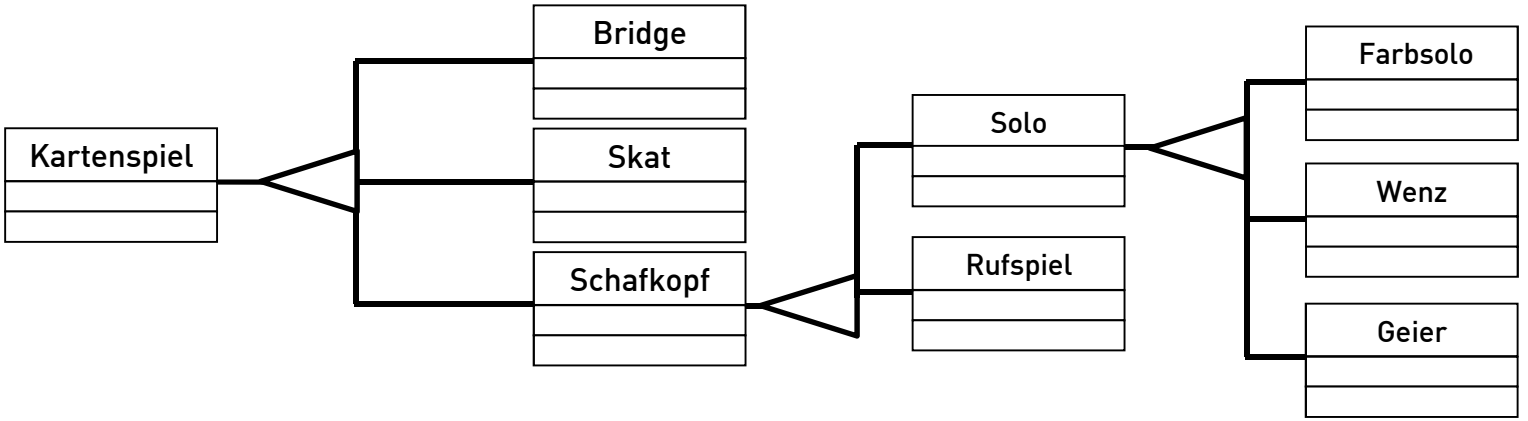
Generalisierung / Vererbung ist ein Fundamentalkonzept der sog. "objektorientierten" Programmentwicklung.

- Generalisierung / Vererbung ist ein mächtiges Abstraktionskonzept, das dazu verwendet wird,
  - einerseits Ähnlichkeiten zwischen verschiedenen Klassen zu teilen
  - und andererseits gleichzeitig ihre Unterschiede zu erhalten.
- *Generalisierung* ist die Relation zwischen einer Klasse und einer oder mehrerer verfeinerter Versionen davon.
  - Die Klasse, die verfeinert wird, nennt man Basisklasse, jede verfeinerte Version nennt man abgeleitete Klasse.
  - *Attribute* (Datenmember) und *Methoden* (Memberfunktionen), die für eine Gruppe von abgeleiteten Klassen gelten, werden einer Basisklasse zugewiesen.
- *Vererbung* ist der Mechanismus, Attribute und Methoden über eine Generalisierungsstruktur zu gewinnen.
  - Datenmember und Memberfunktionen der Basisklasse werden von den einzelnen abgeleiteten Klassen gemeinsam genutzt.
  - Man sagt, jede abgeleitete Klasse "erbt" von ihrer Basisklasse.

Generalisierung / Vererbung

Generalisierung / Vererbung ist ein Fundamentalkonzept der sog. "objektorientierten" Programmentwicklung.

- Die Generalisierungsstruktur ist zunächst hierarchisch.
  - Notation in Klassendiagrammen: ein Dreieck, das eine Basisklasse mit ihren abgeleiteten Klassen verbindet.
    - Die Basisklasse wird durch eine Linie mit der Dreieckspitze verbunden.
    - Die abgeleiteten Klassen werden durch Linien mit einer Linie verbunden, die über die Grundlinie des Dreiecks verläuft.

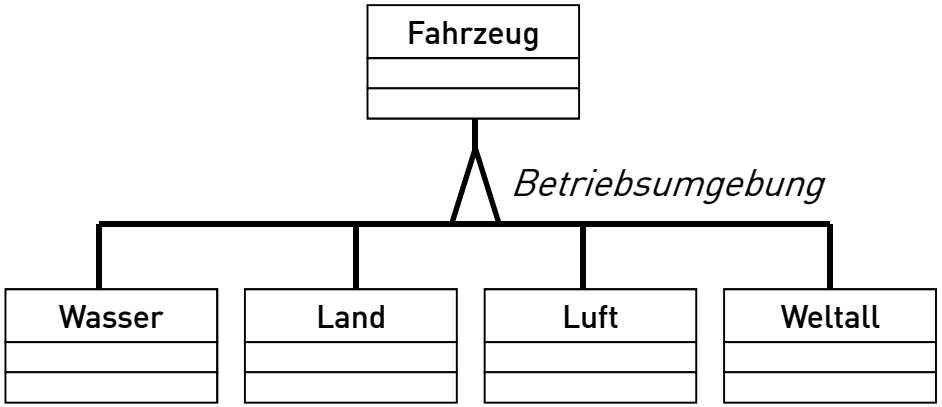


- Der Vererbungsmechanismus ist über beliebig viele Ebenen transitiv.
  - Man spricht von Vorfahrens- und Nachkommensklassen.
  - Ein Objekt vom Typ einer abgeleiteten Klasse ist gleichzeitig ein Objekt vom Typ aller Vorfahrensklassen.

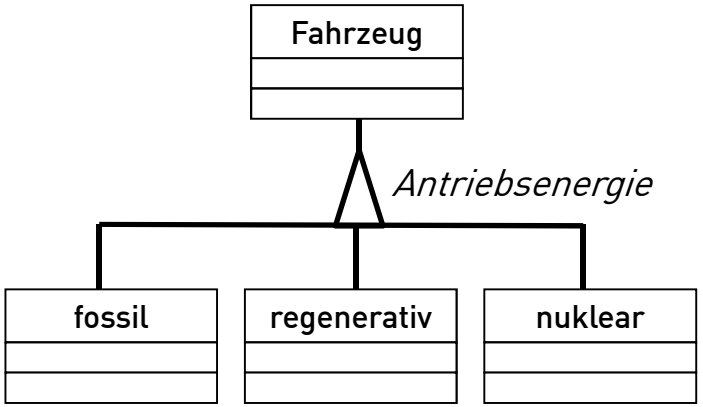
Diskriminatoren

Ein (optionaler) Diskriminator bezeichnet die Grundlage einer Generalisierung.

- Ein Diskriminator gibt an, welches Merkmal eines Objekts durch eine bestimmte Generalisierungsrelation abstrahiert wird.
  - Der Diskriminator einer Generalisierung wird im Klassendiagramm zu dem Dreieck geschrieben.
- Wenn möglich sollte jeweils nur nach einem Merkmal unterschieden werden.
- Beispiel:



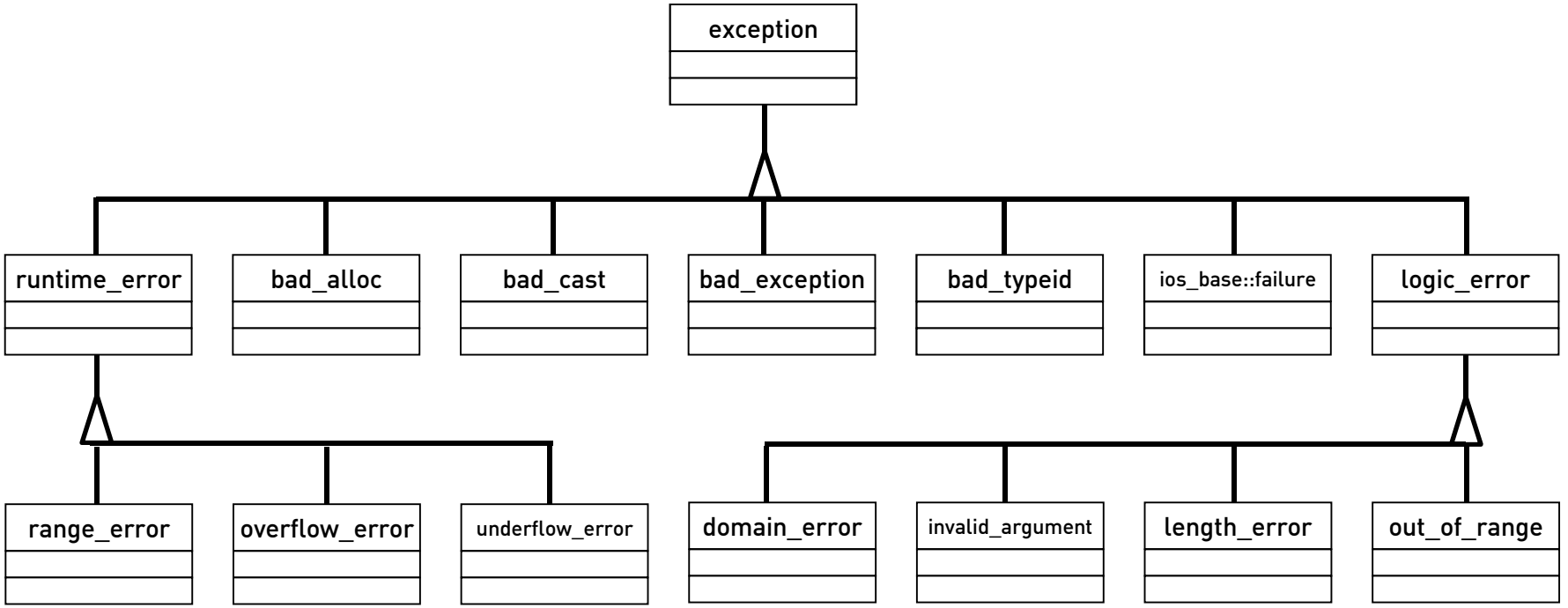
Der Betriebsumgebungs-Diskriminator für *U-Boot* ist *Wasser.*



Der Antriebsenergien-Diskriminator für *Fahrrad* ist *regenerativ.*

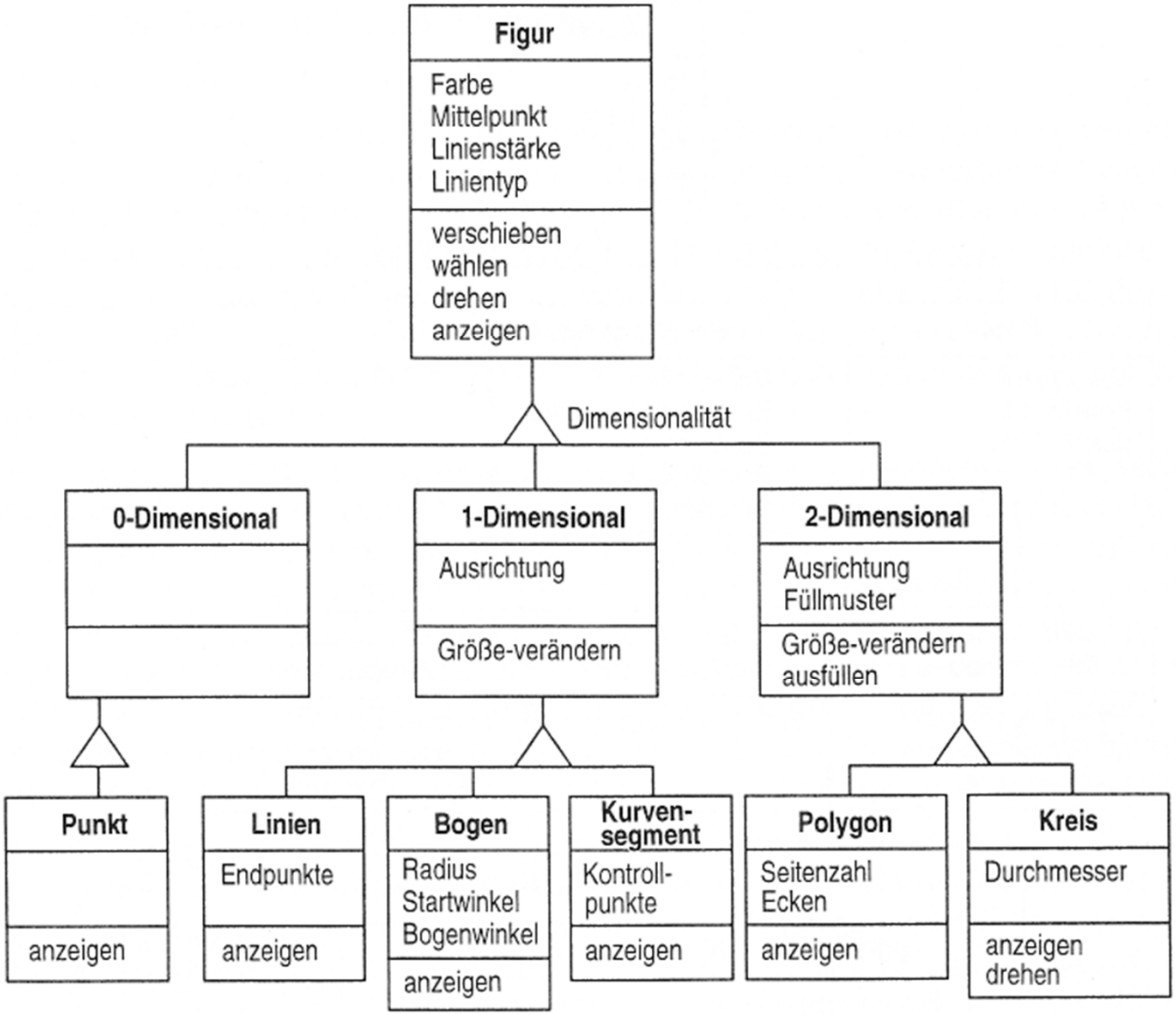
Vererbung der Ausnahme-Typen in C++

Sie kennen schon die Generalisierungsstruktur der C++ Ausnahmetypen aus der StdLib.



Klassisches Beispiel für Vererbung: Figuren

Die Abbildung zeigt eine Klassenhierarchie von grafisch darzustellenden geometrischen Figuren.



Bildquelle: Rumbaugh et al.

Abgeleitete Klassen

Ohne abgeleitete Klassen...

- Ein Programm, das mit den Beschäftigten eines Unternehmens zu tun hat, könnte eine Datenstruktur ähnlich zu dieser enthalten:

```
struct Employee {
    string name;
    int department;
    double salary;
    Employee* next; // als Link in einer Liste mit allen Mitarbeitern
    //…
};
```

- Als nächstes kann man versuchen, die Manager des Unternehmens zu definieren:

```
struct Manager {
    Employee emp;          // die Mitarbeiter-Daten des Managers
    int level;             // die Hierarchie-Stufe
    Employee* deputy;      // die Vertretung
    //…
};
```

Abgeleitete Klassen

...tauchen schnell Schwierigkeiten auf.

- Manager sind ebenfalls Beschäftigte, ihre `Employee` Daten werden im `emp` Member eines `Manager` Objekts abgelegt.

```
struct Employee {
    string name;
    int department;
    double salary;
    Employee* next;
    //…
};
```

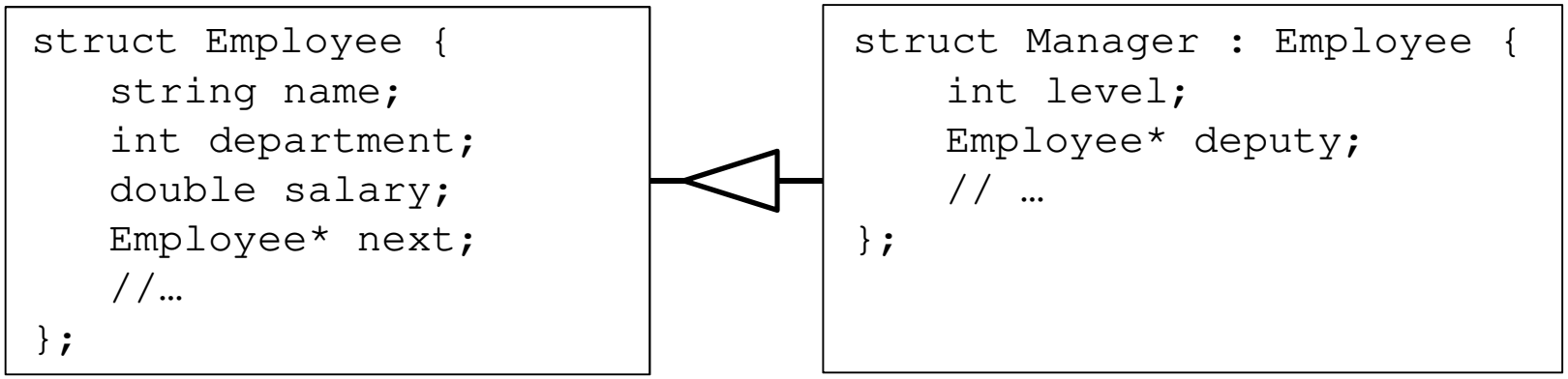
```
struct Manager {
    Employee emp;
    int level;
    Employee* deputy;
    // …
};
```

- *Schwierigkeiten:*
  - Ein `Manager*` ist nicht gleichzeitig ein `Employee*`.
  - Um ein `Manager` Objekt als `Employee` zu verwenden, um es beispielsweise in die vorgesehene `Employee` Liste einzutragen, muss man speziellen Quellcode schreiben:
    - Explizite Zeigerumwandlung,
    - oder nur die `emp` Komponente von `Manager` Objekten nutzen (z.B. in die Liste eintragen),
    - oder ...?
    - **Die Alternativen sind wenig intuitiv und fehleranfällig.**

Abgeleitete Klassen

Zur Implementierung abgeleiteter Klassen gibt es in C++ spezielle Sprachmittel.

- Richtige Herangehensweise: ein `Manager` ist ein `Employee`, der um zusätzliche Eigenschaften ergänzt wurde.



- `Employee` ist der allgemeinere Typ (*Basisklasse*), von dem `Manager` als speziellerer Typ (*abgeleitete Klasse*) verfeinert wird.
  - Der speziellere Typ `Manager` besitzt alle Member des allgemeineren Typs `Employee`.
  - Der speziellere Typ `Manager` besitzt zusätzlich einen Member namens `level` und einen Member namens `deputy`.

Abgeleitete Klassen

Zur Implementierung abgeleiteter Klassen gibt es in C++ spezielle Sprachmittel.

- Jetzt kann man eine `Employee` Objektliste erstellen, die auch Objekte vom Typ `Manager` enthält:

```
void f() {
   Manager m1{};
   Manager m2{};
   Employee e1{};
   Employee e2{};
   Employee* elist{};

   elist = &m1;       // m1 in elist eintragen
   m1.next = &e1;     // e1 in elist eintragen
   e1.next = &m2;     // m2 in elist eintragen
   m2.next = &e2;     // e2 in elist eintragen
   e2.next = nullptr; // Ende von elist
}
```

Abgeleitete Klassen

Zur Implementierung abgeleiteter Klassen gibt es in C++ spezielle Sprachmittel.

- Da `Manager` von `Employee` abgeleitet ist, kann ein `Manager*` direkt als `Employee*` benutzt werden, aber *nicht* umgekehrt – Beispiel:

```
01 void g() {
02   Manager m{};  Manager* pm{};
03   Employee e{}; Employee* pe{};
04   pe = &m;                    // OK
05   //pm = &e;                  // Fehler, nicht jeder Employee ist Manager
06   //pm->level = 2;            // das waere jetzt ein grober Fehler
07   pm = static_cast<Manager*>(pe); // OK (wegen Zeile 4):
08                                   // pm zeigt auf m vom Typ Manager
09   pm->level = 2;                  // jetzt OK
10 }
```

- Das gilt allgemein: *`Derived`* Adressen können ohne explizite Typumwandlung an *`Base`* Zeiger zugewiesen werden (wie Zeile 4), die entgegengesetzte Umwandlung muss explizit sein (wie Zeile 7).
- Vorsicht: die typische C++ Implementierung prüft zur Laufzeit *nicht*, ob das Ergebnis einer solchen Zuweisung den erwarteten Typ trägt.

Abgeleitete Klassen

Als `class` mit Kapselung.

- Benutzerdefinierte Typen sind mehr als reine Datenstrukturen

```
class Employee {
  public:
    //…


  private:
    string name;
    //…
};
```

```
class Manager : public Employee {
  public:
    void print( ) const;
    //…

  private:
    int level;
    //…
};
```

- Zugriffsversuch aus `Manager` auf `Employee`-Elemente:

```
void Manager::print( ) const {
  cout << "Name: " << name << ", Level: " << level;
}
```

Diese Methode lässt sich *nicht* kompilieren, da Member der abgeleiteten Klasse *keinen* Zugriff auf `private` Member der Basisklasse (hier: `name`) haben.

Abgeleitete Klassen

Als `class` mit Kapselung.

- Benutzerdefinierte Typen sind mehr als reine Datenstrukturen...

```
class Employee {
  public:
    void print( ) const;
    //…

  private:
    string name;
    //…
};
```

```
class Manager : public Employee {
  public:
    void print( ) const;
    //…

  private:
    int level;
    //…
};
```

- Erfolgreicher Zugriff aus `Manager` auf Member aus `Employee`:

```
void Employee::print( ) const { cout << Name: " << name; }

void Manager::print( ) const {
  Employee::print( );              // die Employee Daten ausgeben
  cout << ", Level: " << level;    // die Manager Daten ausgeben
}
```

Vererbung und Zugriffsrechte

Eine abgeleitete Klasse `A` kann sich als `public`, `protected` oder `private` aus einer Basisklasse `B` ableiten.

- **`class A : public B { /*…*/ };`**
  Die Zugriffsrechte werden unverändert aus der Basisklasse übernommen.
  - Alle Member der Basisklasse `B`, die `public` oder `protected` deklariert sind, können in der abgeleiteten Klasse `A` angesprochen werden.
  - Über ein Objekt vom Typ `A` kann auf die `public` und `protected` Member von `B` zugegriffen werden.
- `class A : protected B { /*…*/ };`
  Dieser Modus kommt selten vor.
  - Die `public` Member der Basisklasse `B` haben in der abgeleiteten Klasse `A` `protected` Status.
  - `private` und `protected` Member von `B` sind in `A` alle `private`, also nicht zugreifbar.
- `class A : private B { /*…*/ };`
  Die abgeleitete Klasse verdeckt die Basisklasse vollständig.
  - Alle Member der Basisklasse `B`, die `public` oder `protected` deklariert sind, gelten in der abgeleiteten Klasse `A` als `private`.
  - Über ein Objekt vom Typ `A` kann auf überhaupt kein Member von `B` zugegriffen werden.

Abgeleitete Klassen

Konstruktoren und Destruktoren.

- Jedes Objekt einer abgeleiteten Klasse enthält ein anonymes Objekt der Basisklasse.
  - *Konstruktion: von unten nach oben*. Zuerst wird die Basis konstruiert, dann die Member-Objekte und zum Schluss die abgeleitete Klasse selbst – Member und Basen wie immer in der Reihenfolge ihrer *Deklaration* (und *nicht* in der Reihenfolge der Initialisierer-Liste im Konstruktor – manche Compiler warnen).
  - *Destruktion: von oben nach unten.* Genau umgekehrt.
- Wenn die Basisklasse einen Konstruktor hat, wird dieser aufgerufen:

```
class Employee {
  public:
    Employee( string, int );
    //…
  private: //…
};
```

```
class Manager : public Employee {
  public:
    Manager( string, int,
             int, Employee* );
    //…
  private: //…
};
```

```
Employee::Employee( string nam, int dep )
  : name{nam}, department{dep}, salary{0.0}, next{nullptr} { /* … */ }

Manager::Manager( string nn, int dd, int lev, Employee* dpy )
  : Employee{nn,dd}, level{lev}, deputy{dpy} { /* … */ }
```

Klassen-Hierarchien

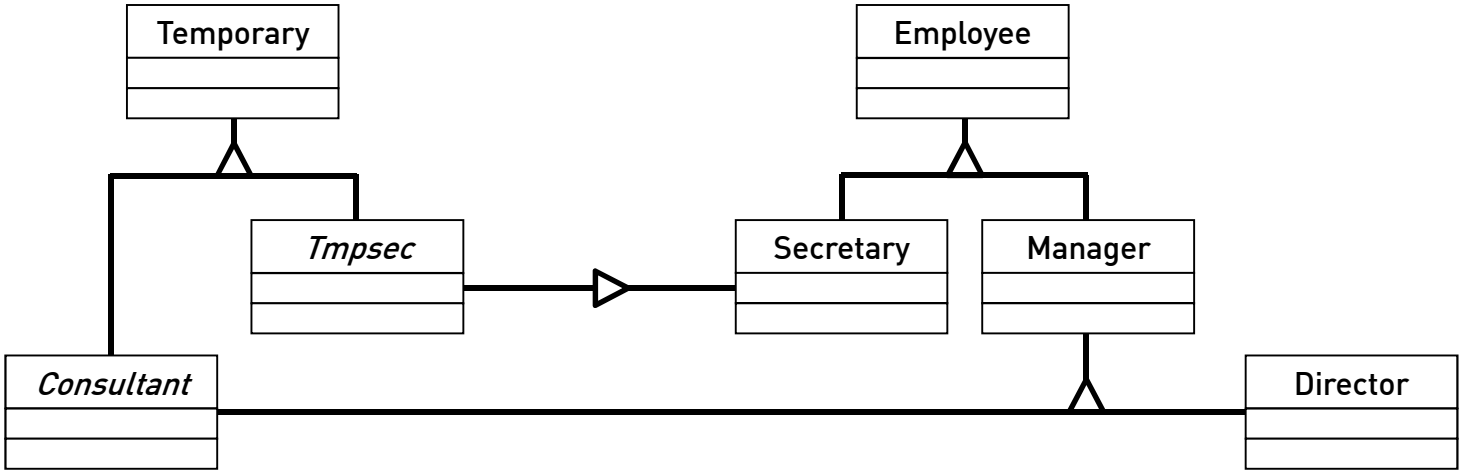
Eine abgeleitete Klasse kann wiederum als Basisklasse dienen.

- Eine Menge verwandter Klassen nennt man Klassen-Hierarchie:

```
class Employee { /*…*/ };
class Manager : public Employee { /*…*/ };
class Director : public Manager { /*…*/ };
```

- C++ kann auch Klassenstrukturen darstellen, die allgemeine, gerichtete azyklische Graphen sind:

```
class Temporary { /*…*/ };
class Secretary : public Employee { /*…*/ };
class Tmpsec : public Temporary, public Secretary { /*…*/ };
class Consultant : public Temporary, public Manager { /*…*/ };
```



Virtuelle Methoden

Gegeben sei ein `Base*`, von welchem genauen Typ ist das Objekt, auf welches gerade gezeigt wird?

- Virtuelle Methoden werden in der Basisklasse deklariert und definiert, und können in den abgeleiteten Klassen umdefiniert ("*überschrieben*") werden.

```
class Employee {
  public:
    Employee( string na, int de );
    virtual void print( ) const;
    //…
  private:
    string name;
    int department;
    Employee* next;
    static Employee* list;
    //…
};
```

```
void Employee::print() const { cout << Name << " (" << department << ")"; }
```

- Das Schlüsselwort `virtual` zeigt an, dass es für unterschiedliche abgeleitete Klassen unterschiedliche Versionen von `print()` geben kann.
  - Mit gleichen Parameterlisten (im Unterschied zum Über*laden* von Methoden).
  - Der Typ der Methode darf nicht umdefiniert werden.

Virtuelle Methoden

Gegeben sei ein `Base*`, von welchem genauen Typ ist das Objekt, auf welches gerade gezeigt wird?

- Wenn man eine neue Klasse ableitet, definiert man nun ggf. einfach die Methode passend neu.

```
class Manager : public Employee {
  public:
    Manager( string nam, int dep, int lev );
    void print( ) const;
    //…
  private:
    int level;
    Employee* deputy;
    //…
};

void Manager::print() const {
    Employee::print();
    cout << ", Level: " << level;
}
```

- Klassen, die mindestens eine virtuelle Methode besitzen, heißen auch *polymorphe Klassen*.

Virtuelle Methoden

Gegeben sei ein `Base*`, von welchem genauen Typ ist das Objekt, auf welches gerade gezeigt wird?

- Jetzt kann eine `Employee` Liste, die auch `Manager` enthält, so ausgegeben werden:

```
void Employee::print_list( ) const {
    for( Employee* p{list}; p != nullptr; p = p->next ) {
        p->print( );
        cout << std::endl; // zeilenweise
    }
}
```

- Jedes Objekt wird von `Employee::print_list()` korrekt entsprechend seines Typs ausgegeben.
- Hinweise zu `next` und `list` aus der `Employee`-Basisklasse (S. 17):

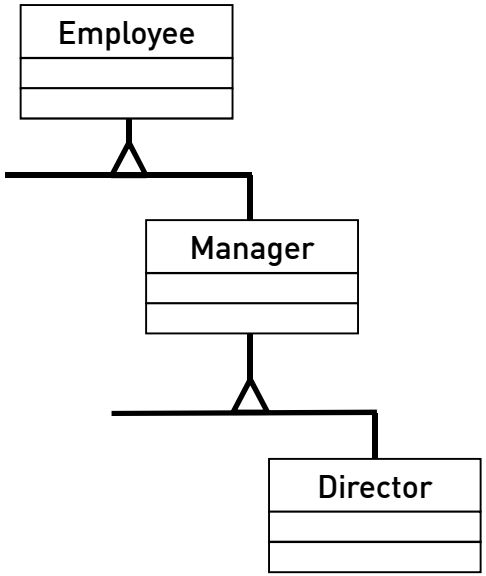
```
// Definition Konstruktor:
Employee::Employee( string na, int de )
  : name{ na }, department{ de }, next{ list } { list = this; }
// Initialisierung eines static Members (hier namens list) geht nicht in Konstruktoren,
// sondern wird im globalen Scope gemacht:
Employee* Employee::list{ nullptr };
```

Virtuelle Methoden

Indirekte Basisklassen ("Vorfahrensklassen").

- Beispiel:

```
class Employee {
   //…
   virtual void print();
};
class Manager : public Employee {
   //…
   virtual void print();
};
class Director : public Manager {
   //…
   virtual void print();
};
```



Vertiefung: virtuelle Destruktoren

Oft wird gefordert, dass polymorphe Klassen auch einen virtuellen Destruktor haben müssen (manche Compiler warnen), Begründung:

- Wenn eine Klasse virtuelle Methoden besitzt (d.h. polymorph ist) , dann wird man wohl auch von ihr abgeleitete Klassen bilden,
  - sonst wären die virtuellen Methoden / die Polymorphie wenig sinnvoll.
- Da man die abgeleiteten Klassen nicht von vornherein alle kennt, kann man auch nicht ausschließen, dass in diesen Klassen spezifische Destruktoren erforderlich sind.
- Objekte solcher *abgeleiteten* Klassen kann man natürlich auch mit einem expliziten `new` "nackt" erzeugen,
- und später im Code über einen Zeiger auf die *Basis*klasse ansprechen (Sie erinnern sich: Basisklassenzeiger können Adressen von Objekten eines von ihnen abgeleiteten Typs enthalten).
- Wird nun das explizit erforderliche `delete` auf den Basisklassenzeiger angewandt, der die Adresse eines solchen abgeleiteten Objekts hält, muss dann natürlich der richtige Destruktor verwendet werden.
  - Bei abgeleiteten Klassen stellt der C++ Mechanismus für virtuellen Methoden sicher, dass immer die korrekte Methode aufgerufen wird.
  - Damit dieser Mechanismus auch für Destruktoren greifen kann, muss die entsprechende Basisklasse dann auch einen virtuellen Destruktor besitzen.

Explizite Typumwandlungen

Diese Sprachmittel sind nur nach gründlichen Überlegungen einzusetzen, da sie die Typsicherheit drastisch verletzen können, d.h. sehr fehleranfällig sind.

- **`dynamic_cast<T>(a)`** konvertiert `a` in den Typ `T`,
  - wird zur Run-Time ausgewertet, funktioniert für alle vorgesehenen Typumwandlungen,
  - `a` muss Zeiger oder Referenz auf einen polymorphen Typ sein (d.h. auf einen Typ, der mindestens eine virtuelle Methode hat),
    - liefert den Nullzeiger auf `T`, wenn der Zeiger `a` nicht auf ein Objekt vom Typ `T` oder vom von `T` abgeleiteten Typ zeigt,
    - löst eine `std::bad_cast` Ausnahme aus, wenn die Referenz `a` nicht ein Objekt des Typs `T` oder eines von `T` abgeleiteten Typs benennt.
- Beschaffung von Typinformation (etwa zum Instrumentieren Ihres Quellcodes beim Debugging): `typeid( a )`:
  - Header `typeinfo` erforderlich,
  - wenn Sie das Gefühl haben, dass Sie in Ihrem Programm mittels `typeid` unterschiedliche Objekte nach ihrem Typ unterscheiden müssten, sollten Sie nochmals gründlich über Ihr Programmdesign und über den C++ Mechanismus für virtuelle Methoden nachdenken...

Virtuelle Methoden

Einige weitere Merkmale bei virtuellen Methoden.

- (Erst) Durch virtuelle Methoden sind abgeleitete Klassen "polymorph" und damit mehr als nur eine Kurzschreibweise in Deklarationen.
  - Compiler und Linker garantieren hier die Übereinstimmung zwischen Objekten und den auf sie angewandten Methoden.
- Beim Aufruf einer Methode über den Operator `::` (scope resolution)
  - ist sichergestellt, dass der Mechanismus für virtuelle Methodenaufrufe *nicht* verwendet wird,
    - sonst würde es eine Endlos-Rekursion geben.
  - Es kann auch `inline` Quellcode ersetzt werden.
- Eine virtuelle Methode muss für die Klasse, in der sie das erste Mal deklariert wurde, auch definiert werden.
  - Ausnahme: sog. abstrakte Klassen, die "rein virtuelle" Methoden haben.

Abstrakte Klassen

Man nennt Klassen abstrakt, die *nur* als Basisklassen für abgeleitete Klassen Sinn machen (und nicht als Typen für Objekte).

- Eine Klasse mit mindestens einer *rein virtuellen* Methode ("pure virtual") heißt abstrakte Klasse.
  - Eine virtuelle Methode wird "rein", indem man sie mit `=0` initialisiert.
  - Von abstrakten Klassen können *keine* Objekte erzeugt werden.
  - Abstrakte Klassen sind vielmehr nur als Basisklassen zur Ableitung anderer Klassen gedacht.
- Klassisches Beispiel: `Shape` als Basisklasse für Figuren (Objekte direkt vom Typ `Shape` sind sinnlos):

```
class Shape {
  public:
    virtual void move( int, int ) = 0; // rein virtuelle Methode
    virtual void draw( ) = 0;          // rein virtuelle Methode
    //…
  private:
    //…
};
```

- Man kann abstrakte Klassen meist daran erkennen, dass sich ihre virtuellen Methoden schon gar nicht sinnvoll definieren lassen.

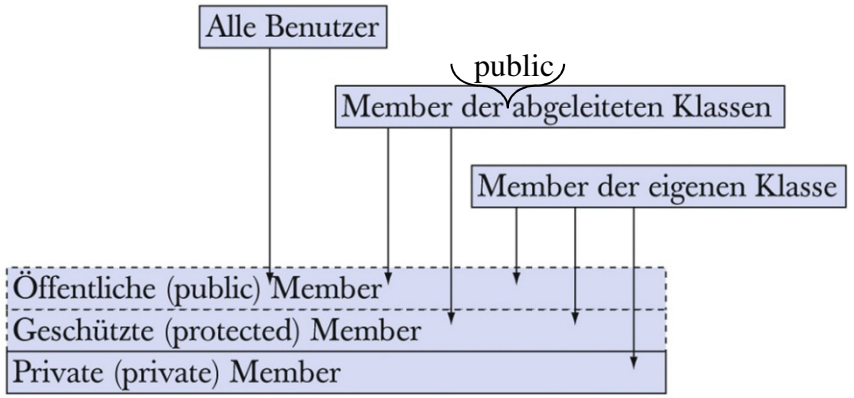
Abstrakte Klassen

Bleiben rein virtuelle Methoden ohne Definition in der abgeleiteten Klasse, ist die abgeleitete Klasse ebenfalls abstrakt.

- Mittels rein virtueller Methoden / abstrakter Klassen können Programme schichtweise implementiert werden.
- Beispiel:

```
class X {
  public:
    virtual void a( ) = 0; // rein virtuelle Methode
    virtual void b( ) = 0; // rein virtuelle Methode
};

class Y : public X { void a( ); }; // ueberschreibt X::a
class Z : public Y { void b( ); }; // ueberschreibt X::b

X obj1{}; // Fehler: Objekt namens obj1 der abstrakten Klasse X
Y obj2{}; // Fehler: Objekt namens obj2 der abstrakten Klasse Y
Z obj3{}; // ok
```

- *Hinweis:* ein seltener, aber ebenso geeigneter Weg, eine Klasse abstrakt zu machen, besteht darin, alle Konstruktoren als `protected` zu deklarieren.



Bildquelle: Stroustrup

Übung

Implementieren Sie eine Klassen-Generalisierungsstruktur.

- Implementieren Sie, auf Basis der vorgestellten Quellcodefragmente, die einfache Generalisierung mit `Employee` als Basisklasse und `Manager` als Klasse, die direkt und `public` aus `Employee` abgeleitet ist.
  - Der `Employee` Typ kapselt mindestens `string name`, `int department`, `double salary` und bietet als Operationen mindestens den Standardkonstruktor, einen weiteren Konstruktor und eine virtuelle `print()` Methode zur Ausgabe.
  - Der `Manager` Typ kapselt mindestens `int level`, `Employee* deputy` und bietet als Operationen mindestens den Standardkonstruktor, einen weiteren Konstruktor und eine `print()` Methode zur Ausgabe.
- Stellen Sie sicher, dass eine einfach verkettete Liste aufgebaut wird, die alle erzeugten Objekte (vom Typ `Employee` und vom Typ `Manager`) enthält.
- Probieren Sie Ihre Klassen über einen `main()` Treiber aus.
  - Legen Sie einige `Employee` Objekte und `Manager` Objekte an.
  - Geben Sie in einer Schleife über einen `Employee*` die Objektdetails aus.

Komplettes Beispiel: Figuren

Als komplettes Beispiel werden wir ein Programm zur Darstellung geometrischer Figuren als Punkteraster auf der Konsole entwickeln.

- Konsolenausgabe (*keine* Grafik im eigentlichen Sinn):
  - Konsolenraster als zweidimensionales `char`-Array `std::array< std::array<char,xMax>, yMax >`.
- Figuren-Bibliothek:
  - Abstrakte Basisklasse `Shape`,
  - Typ `Line` mit gerasterter Ausgabe (Bresenham-Algorithmus),
  - Typ `Rectangle`,
  - Typ `Square` (von `Rectangle` abgeleitet?)
    - Ist ein `Square` ein `Rectangle`?
    - Oder ist umgekehrt ein `Rectangle` ein `Square`?
    - Eine Antwort bietet das Liskovsche Substitutionsprinzip...
- Anwendung zum Ausprobieren:
  - Zeichenplotter für `Line`, `Rectangle`, `Square` und weitere Objekte.

Darstellung der Figuren auf einem Raster

Vorstellung des Raster-Prinzips.

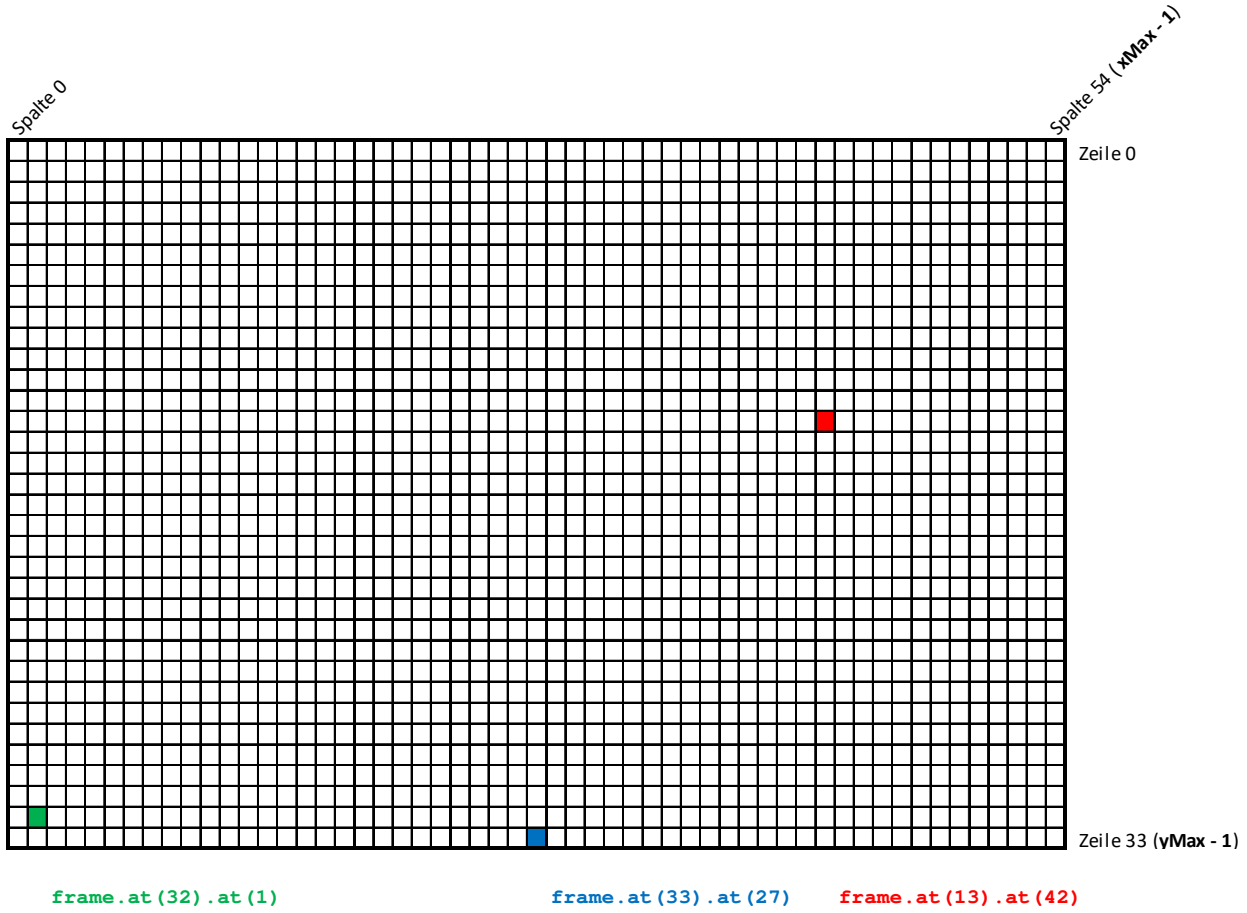
- Quellcode z.B.:

```
std::array< std::array<int,10>, 5 > f{};
f.at(0).at(8) = 8;
f.at(2).at(7) = 4;
for( int i{}; i<5; ++i) {
   for( int j{}; j<10; ++j ) {
      std::cout << f.at(i).at(j);
   }
   std::cout << std::endl;
}

/* 5 Zeilen, 10 Spalten:

   0000000080
   0000000000
   0000000400
   0000000000
   0000000000

*/
```

Darstellung der Figuren auf einem Raster

Vorstellung des Raster-Prinzips.

- Darstellung von z.B. 55 x 34 Bildschirmpunkten als
  `std::array< std::array<char,55>, 34 > frame{};`



Darstellung der Figuren auf einem Raster

Zeichenorientierte Rasterausgabe auf der Konsole ("char-Plotter").

- Eine `Frame` Klasse

```
const int xMax {55}; // Zeichen pro Frame-Zeile
const int yMax {34}; // Zeilen pro Frame

class Frame {
  public:
    enum color { black = ' ', white = '*' };
    Frame( );
    void put_point( int, int );        // Punkt im Frame speichern
    void reset( );                     // Frame leeren
    void show( ) const;                // Frame auf die Konsole ausgeben
    bool on_frame( int, int ) const;   // "Clipping"
  private:
    std::array< std::array<char,xMax>, yMax > frame;
};
```

- Vollständige Definition dieser Klasse...

Figurentypen

Abstrakte Basisklasse `Shape` für unterschiedliche Figuren.

- Eine abstrakte `Shape` Basisklasse:
  - Die Klasse soll zusätzlich eine verkettete Liste aller Objekte aufbauen, die aus den abgeleiteten Klassen erzeugt werden (Zugriff über einen `Shape*`).

```
class Shape {
  public:
    Shape() : prev{ last } { last = this; }    // Skizze Standardkonstruktor
    virtual void draw( Frame* ptarget ) const = 0;   // rein virtuelle Methode
    virtual void move( int hor, int ver ) = 0;       // rein virtuelle Methode
    // etc.
  private:
    static Shape* last; // Zeiger auf die zuletzt erzeugte Figur (d.h. Listenanfang)
    Shape* prev;         // Zeiger auf die zuvor erzeugte Figur
                         // (d.h. die naechste in der Liste)
};
Shape* Shape::last{ nullptr }; // initialisiert den static Member mit dem Nullzeiger
void show_all_shapes( Frame* ); // gibt alle Shape Objekte auf Frame* aus
```

  - Beachten Sie, dass die Klasse auch einen Mechanismus benötigt, der erzeugte Objekte aus der Liste entfernt, sobald diese ungültig geworden sind (d.h. ein passender Destruktor muss definiert werden)...

Figurentyp: gerade Linie

Klasse `Line` für gerade Linien.

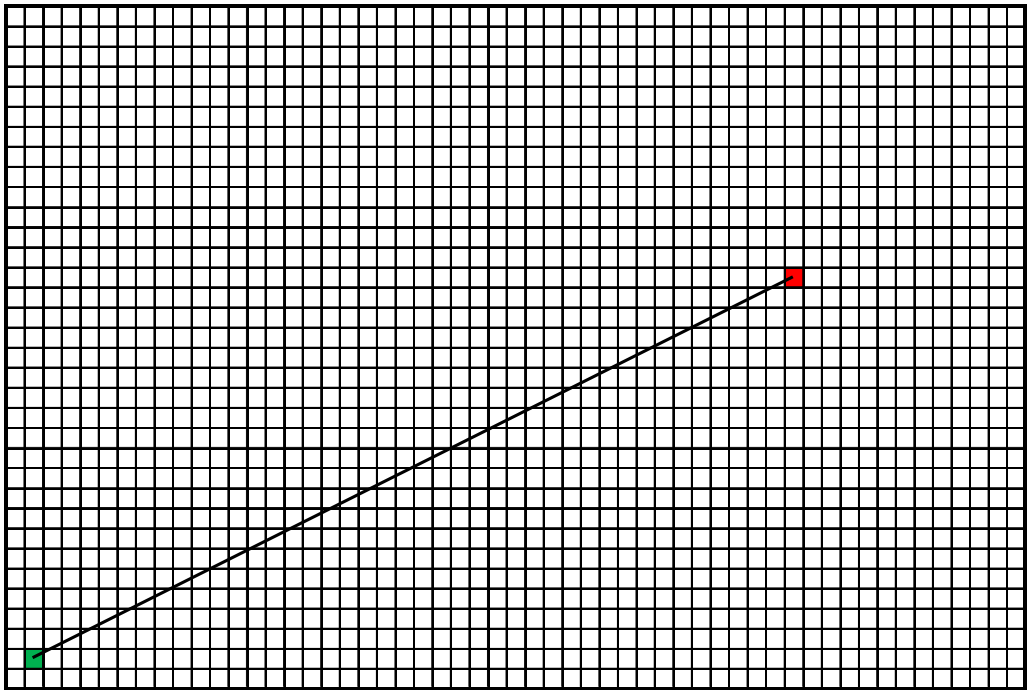
- Eine `Line` Klasse:

```
class Line : public Shape {
  public:
    Line() = delete; // keine Standardlinie
    Line( int, int, int, int ); // ein Konstruktor
    void set( int, int, int, int );
    void draw( Frame* ) const; // rein virtuelles shape::draw() ueberschreiben
    void move( int h, int v ); // rein virtuelles shape::move() ueberschreiben
  private:
    int start_x, start_y, end_x, end_y;
};

Line::Line( int sx, int sy, int ex, int ey ) // Skizze Konstuktordefinition
     : start_x{sx}, start_y{sy}, end_x{ex}, end_y{ey} { }
void Line::move( int h, int v ) { // Def. der line::move() Methode
    start_x += h; start_y += v; end_x += h; end_y += v;
}

// …
```

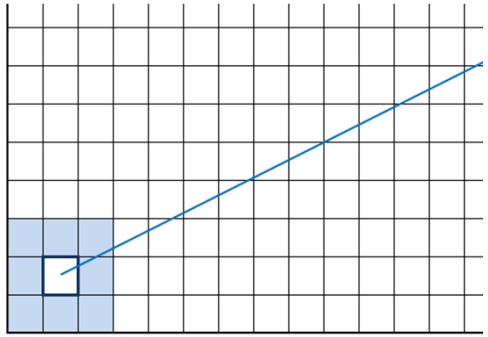
- Definition der `Line::draw()` Methode…

Eine gerade Linie zwischen zwei Rasterpunkten

Ermittlung der Rasterpunkte für eine gerade Linie zwischen zwei gegebenen Rasterpunkten (dem Anfang und dem Ende der Linie).



- Von einem gegebenen Startpunkt sind zunächst maximal acht unterschiedliche direkt nächste Punkte zu einem gegebenen Endpunkt (der z.B. in Richtung der blauen Linie liegt) möglich.

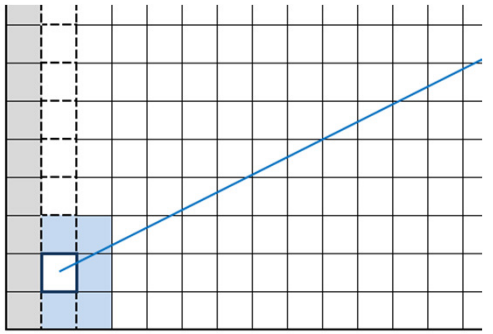


Eine gerade Linie zwischen zwei Rasterpunkten

Bresenham-Algorithmus: drei Vereinfachungsschritte.

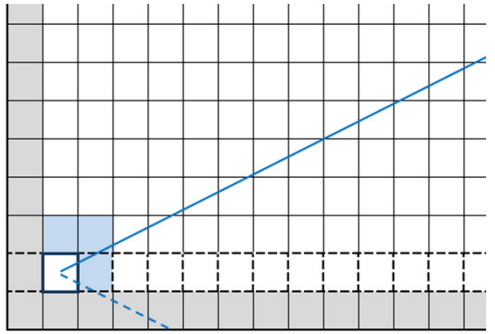
- Falls der Endpunkt links vom Startpunkt liegt, vertauschen wir Start- und Endpunkt: damit entfallen drei Möglichkeiten.

```
if( end_Xcoord < start_Xcoord ) {
   // Start- und Endpunkt vertauschen
}
```



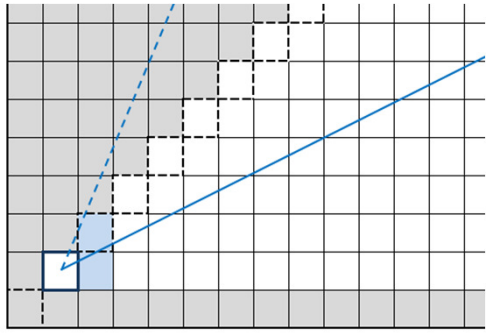
- Falls die Linie nun nach rechts unten verläuft, spiegeln wir alles an der Horizontalen und merken uns dies: es entfallen zwei weitere Möglichkeiten.

```
if( end_Ycoord > start_Ycoord ) {
   // Vorzeichen beider Y-Koordinaten umdrehen
   // dafuer einen Merker setzen
}
```



- Falls die Linie nun steiler als 45 Grad verläuft, spiegeln wir alles an 45 Grad und merken uns dies auch: es entfällt eine weitere Möglichkeit.

```
if(   (end_Xcoord - start_Xcoord)
    < (start_Ycoord - end_Ycoord) ) {
   // start_Xcoord mit start_Ycoord vertauschen
   // end_Xcoord mit end_Ycoord vertauschen
   // dafuer einen weiteren Merker setzen
}
```



Eine gerade Linie zwischen zwei Rasterpunkten

Bresenham-Algorithmus.

- Sinn der drei Vereinfachungen:
  - Sie lassen sich in einem einmaligen Durchlauf anwenden, *bevor* die einzelnen Rasterpunkte in einer Schleife möglichst effizient bestimmt werden.
  - Sie bewirken, dass in der Schleife an jedem Rasterpunkt nur zwischen zwei möglichen nächsten Punkten entschieden werden muss,
    - entweder *rechts*
    - oder *rechts-oben*.
- Durch die zweite und dritte Vereinfachung müssen wir mit Hilfe unserer beiden Merker bei der Berechnung diese vier Fälle unterscheiden:

Koordinaten	**flach**	**steil**
steigend	*(x, y)*	*(y, x)*
fallend	*(x, -y)*	*(y, -x)*

Eine gerade Linie zwischen zwei Rasterpunkten

Zur Schleife im Bresenham-Algorithmus.

- Entscheidungskriterium:

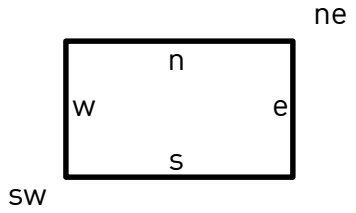
```
( end_Ycoord - start_Ycoord ) * 2 < ( end_Xcoord - start_Xcoord )
```

$2\,\mathrm{d}Y$ $\mathrm{d}X$

- Umformuliertes Entscheidungskriterium: `2dY-dX < 0`
  - `true`
    - Gehe nach *rechts*, d.h. erhöhe nur x,
    - inkrementiere um $2\,\mathrm{d}Y$ (ein positiver Wert), prüfe `<0`.
  - `false`
    - Gehe nach *rechts-oben*, d.h. erhöhe x und y,
    - inkrementiere um $2\,(\mathrm{d}Y\text{-}\mathrm{d}X)$ (ein negativer Wert), prüfe `<0`.

Figurentyp: Rechteck

Klasse `Rectangle` für Rechtecke.



- Eine `Rectangle` Klasse:

```
class Rectangle : public Shape {
  public:
    Rectangle() = delete; // kein Standardrechteck
    Rectangle( int swx, int swy, int nex, int ney ); // Skizze Konstruktor
    void draw( Frame* ); // rein virtuelles shape::draw() ueberschreiben
    void move( int h, int v ); // rein virtuelles shape::move() ueberschreiben
  private:
    int sw_x, sw_y, ne_x, ne_y;
};
```

- Vollständige Definition dieser Klasse...

Figurentyp: Quadrat

Übung: Klasse `Square` für Quadrate.

- Erinnern Sie sich an die Generalisierungs- / Vererbungssyntax:

```
class Square : public Rectangle {
    // …
};
```

- Übung:

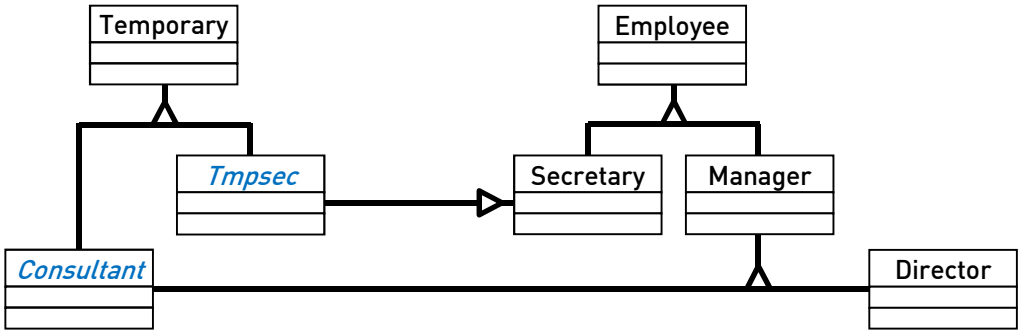
  Definieren Sie die Klasse `Square` für Quadrate.

LEERE SEITE

Mehrfache Vererbung

Eine abgeleitete Klasse kann mehr als eine direkte Basisklasse haben.

- Generalisierungsstrukturen können in C++ allgemein gerichtete azyklische Graphen sein:



- Im obigen Beispiel:
  - `Tmpsec` hat die beiden Basisklassen `Temporary` und `Secretary`

```
class Tmpsec : public Temporary, public Secretary { /*…*/ };
```

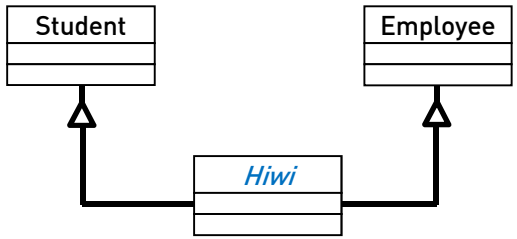
  - `Consultant` hat die beiden Basisklassen `Temporary` und `Manager`

```
class Consultant : public Temporary, public Manager { /*…*/ };
```

Mehrfache Vererbung

Eine abgeleitete Klasse kann mehr als eine direkte Basisklasse haben.

- Im folgenden Beispiel erbt `Hiwi` von `Student` und von `Employee`:



- Dies wird üblicherweise als mehrfache Vererbung bezeichnet (im Unterschied zur Vererbung mit nur einer direkten Basisklasse).
- Zusätzlich zu den jeweiligen speziell für `Hiwi` definierten Methoden ist die Vereinigungsmenge der `Student`- und `Employee`-Methoden auf `Hiwi`-Objekte anwendbar.

```
void f( Hiwi& h ) {
   h.studentnumber();          // Student::studentnumber()
   h.salary();                 // Employee::salary()
   h.teachingassignments();    // Hiwi::teachingassignments()
}
```

Mehrfache Vererbung

Eine abgeleitete Klasse kann mehr als eine direkte Basisklasse haben.

- Entsprechend kann auch ein `Hiwi`-Objekt übergeben werden, wenn ein `Student`- oder `Employee`-Objekt erwartet wird.

```
void set_vacation( Employee* );

void set_final_grade( Student* );

void g( Hiwi* ph ) {
  set_vacation( ph );      // set_vacation( static_cast<Employee*>(ph) )
  set_final_grade( ph );   // set_final_grade( static_cast<Student*>(ph) )
}
```

- Wäre man auf einfache Vererbung beschränkt, müsste man sich bei der Implementierung für eine von zwei sich gegenseitig ausschließenden Alternativen entscheiden.
  - Entweder die Klasse `Hiwi` würde *nur* von `Student` abgeleitet
  - oder die Klasse `Hiwi` würde *nur* von `Employee` abgeleitet.
  - Beides bedeutet weniger Flexibilität.

Mehrfache Vererbung

Eine abgeleitete Klasse kann mehr als eine direkte Basisklasse haben.

- Auch virtuelle Methoden arbeiten in gewohnter Weise:

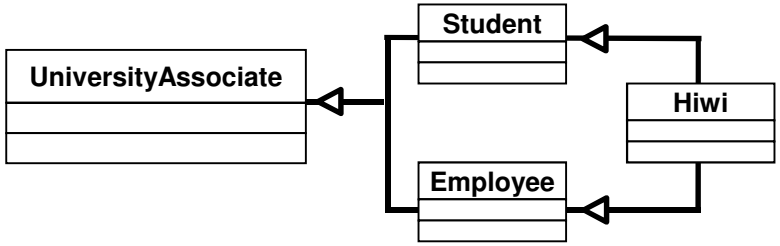
```
class Student {
   //…
   virtual void abc() = 0;
};

class Employee {
   //…
   virtual void xyz() = 0;
};

class Hiwi : public Student, public Employee {
   //…
   void abc();
   void xyz();
};
```

- Es werden nun `Hiwi::abc()` bzw. `Hiwi::xyz()` aufgerufen, wenn ein `Hiwi` über `Student*` bzw. `Employee*` verwendet wird.

Mehrfache Vererbung

Mehrfaches Vorkommen einer Vorfahrensklasse.

- Es kann vorkommen, dass eine abgeleitete Klasse eine Vorfahrensklasse mehrfach besitzt.
- Beispielsweise könnten `Student` und `Employee` von einer gemeinsamen Basisklasse `UniversityAssociate` abgeleitet sein.




```
class Student  : public UniversityAssociate       { /*…*/ };
class Employee : public UniversityAssociate       { /*…*/ };
class Hiwi     : public Student, public Employee  { /*…*/ };
```

- Das funktioniert zwar ohne Probleme,
  - für jedes `Hiwi`-Objekt werden nun einfach zwei separate `UniversityAssociate`-Objekte erzeugt,

  viele der Konsequenzen sind aber normalerweise so nicht erwünscht und können schnell verwirren.

Mehrfache Vererbung

Virtuelle Basisklassen.

- Möchte man die mehrfache Erzeugung von Basisklassen-Subobjekten verhindern, kann man die Ableitungen von der entsprechenden Basisklasse als `virtual` qualifizieren.

```
class Student  : virtual public UniversityAssociate { /*…*/ };
class Employee : virtual public UniversityAssociate { /*…*/ };
class Hiwi     : public Student, public Employee    { /*…*/ };
```

- Um mehrfache Initialisierung zu vermeiden wird der Compiler hier immer den Konstruktor der am weitesten abgeleiteten Klasse verwenden.
- Damit abgeleitete Klassen nie mit nicht initialisierten Basisklassen konfrontiert sind, werden virtuelle Basisklassen vor allen anderen initialisiert.

Mehrfache Vererbung

Mehrfache Vererbung: virtuell und nicht-virtuell.

- Normalerweise besteht ein Objekt aus so vielen Teilen, wie die Klasse Basisklassen hat, z.B. ergibt sich aus

```
class Student  : public UniversityAssociate { /*…*/ };
class Employee : public UniversityAssociate { /*…*/ };
class Hiwi     : public Student, public Employee    { /*…*/ };
```

  der auf der nächsten Seite *links* abgebildete `Hiwi` Aufbau.

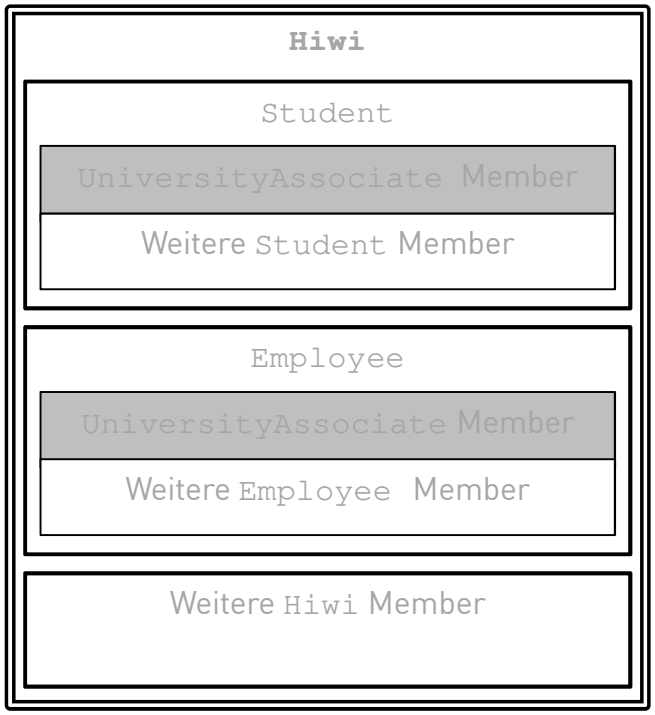
- Dagegen führt

```
class Student  : virtual public UniversityAssociate { /*…*/ };
class Employee : virtual public UniversityAssociate { /*…*/ };
class Hiwi     : public Student, public Employee    { /*…*/ };
```

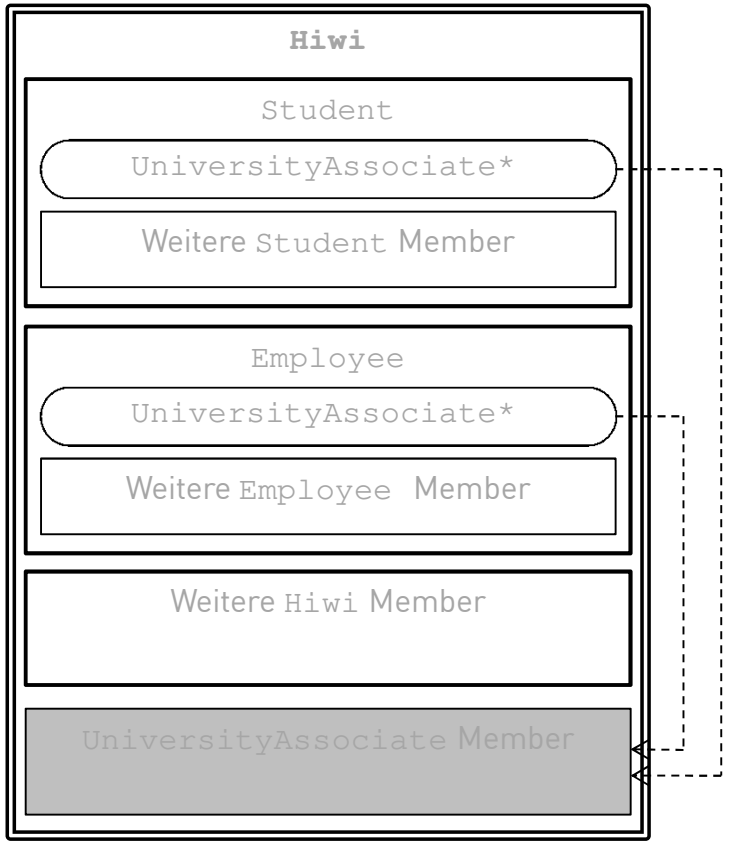
  zu dem auf der nächsten Seite *rechts* abgebildeten `Hiwi` Aufbau.

Mehrfache Vererbung

Mehrfache Vererbung: virtuell und nicht-virtuell.



nicht virtuell



virtuell

```
Hiwi monica{};
UniversityAssociate* pua = &monica; // ok
Hiwi* ph { Hiwi*(pua) }; // Fehler,
     // das geht nur mit dynamic_cast
```

Mehrfache Vererbung und Schnittstellen-Klassen

Auflösung von Mehrdeutigkeit durch vollqualifizierten Aufruf ("explizit").

- Problem: unterschiedliche Basisklassen können Methoden mit demselben Namen besitzen.

```
class Student {
    //…
    virtual void print();
};
class Employee {
    //…
    virtual void print();
};
```

- Beim Aufruf mit einem `Hiwi*` müssen die Mehrdeutigkeiten aufgelöst werden, Lösung z.B. explizit durch vollqualifizierten Aufruf:

```
void printall( Hiwi* ph ) {
    //ph->print(); // Fehler, mehrdeutig
    ph->Student::print();   // ok
    ph->Employee::print();  // ok
}
```

Schnittstellen-Klassen

Auflösung von Mehrdeutigkeit durch Definition von Schnittstellenklassen.

- Weiteres Problem: unterschiedliche Basisklassen mit gleichlautenden Methoden, die Unterschiedliches tun und beide benötigt werden, z.B.:

```
class Student {

  //…
  virtual void assign();
};
```

```
class Employee {

  //…
  virtual void assign();
};
```

```
class Hiwi : public Student, public Employee {
    //…
};
```

  - Ein `Hiwi`-Objekt könnte einen ordnungsmäßig immatrikulierten Studierenden, der am Fachbereich angestellt ist, repräsentieren.
  - Sowohl für `Student` als auch für `Employee` sind spezifische Methoden wie z.B. `assign()` definiert, die man in `Hiwi` jeweils vollständig übernehmen will / muss.
    - `Student::assign()` ordnet z.B. Lehrveranstaltungen zu.
    - `Employee::assign()` weist z.B. den direkten Vorgesetzten zu.

Schnittstellen-Klassen

Auflösung von Mehrdeutigkeit durch Definition von Schnittstellenklassen.

- Schwierigkeit:
  - `Hiwi` kann nur *eine* `assign()`-Methode enthalten.
  - *Beide* `assign()`-Versionen, die aus `Student` *und* die aus `Employee`, müssen überschrieben werden.
  - Die beiden `assign()`-Versionen können *nicht* mit derselben Methode überschrieben werden, da sie völlig Unterschiedliches tun.
- Üblicher Lösungsweg:
  - Definition von je einer *Schnittstellen-Klasse* für `Student` und `Employee`.
  - Die Schnittstellen-Klassen dienen nur dazu, neue Namen für die `assign()` Methoden einzuführen, damit diese neuen virtuellen Methoden ausgeführt werden, wenn `Student::assign()` bzw. `Employee::assign()` aufgerufen wird.

Hinweis: dieses Problem tritt (nur) auf, wenn die Methoden völlig gleich lauten; sind die Parameterlisten unterschiedlich, wird der übliche Mechanismus für überladene Funktionsnamen die Mehrdeutigkeit auflösen.

Schnittstellen-Klassen

Auflösung von Mehrdeutigkeit durch Definition von Schnittstellenklassen.

- Die Schnittstellen-Klassen:

```
class SStudent
  : public Student
{
  virtual void S_assign() = 0;
  void assign() { S_assign(); }
};
```

```
class EEmployee
  : public Employee
{
  virtual void E_assign() = 0;
  void assign() { E_assign(); }
};
```

- Nun lässt sich aus den Schnittstellen-Klassen `SStudent` und `EEmployee` eine `Hiwi` Klasse zusammensetzen, in der die Methoden `S_assign()` und `E_assign()` beliebig umdefiniert werden können:

```
class Hiwi : public SStudent, public EEmployee {
    //…
    void S_assign() { /*…*/ };
    void E_assign() { /*…*/ };
};
```

Schnittstellen-Klassen

Schnittstellen-Klassen sind ein sehr nützliches, allgemein anwendbares Lösungsmuster.

- Der eben vorgestellte Lösungsansatz über eine Schnittstellen-Klasse, die Methoden weiterreicht, ist eine grundsätzliche Technik mit sehr vielen praktischen Einsatzmöglichkeiten.
- Ein weiteres Beispiel, das an unseren Typ `myVector` und an den für `myVector` Objekte überladenen Indexoperator `[]` anknüpft.

```
class myVector {
      int sz; double* elem;
   public:
      explicit myVector( int s ) : sz{ s }, elem{ new double[s]{} } {}
      myVector( ) : myVector{ 0 } {}
      ~myVector( ) { delete[] elem; }
      //…
      int size() const { return sz; }
      virtual double& operator[]( int );
};
double& myVector::operator[]( int index ) {
    if( index<0 || sz<=index ) error( "myVector::operator[](), bad index" );
    return elem[index];
}
```

Schnittstellen-Klassen

Mit Schnittstellen-Klassen kann man Typen recht einfach spezifisch anpassen, im Beispiel den `[]` Zugriff auf `myVector` Objekte.

- `myRangeVec` als Schnittstellen-Klasse, die dem Indexoperator für den Typ `myVector` frei vom Benutzer definierbare Indexgrenzen gibt:

```
class myRangeVec : public myVector {
  public:
     myRangeVec() = delete;
     myRangeVec( int lo, int hi )
     : myVector( hi-lo+1 ), lb{lo} {}
     int low() const { return lb; }
     int high() const { return lb + size() - 1; }
     double& operator[]( int i ) {
       return myVector::operator[]( i-lb );
     }

  private:
     int lb; // lower bound, also der untere Index
};
```

Schnittstellen-Klassen

Die `myRangeVec` Schnittstelle im Einsatz.

- Nun können Programmierer Objekte vom Typ `myRangeVec` verwenden und dabei eigene Indexgrenzen angeben:

```
try {
  myRangeVec vv{1,10};   // Index von 1 bis 10
                         // vv{-3,6} geht z.B. auch

  vv[3] = 24.68;
  for( int i{■}; i<=■; ++i )   // probieren Sie einige Werte aus
       cout << vv[i] << '\n';

  myVector* p{&vv};
  std::cout << (*p)[3];        // virtual in myVector

  return 0;
}
catch( ... ) {
  cerr << "Programmabbruch: ein Ausnahmefehler";
  return -1;
}
```

Templates und abgeleitete Klassen

Zur Terminologie.

- Templates und Parameter.
  - Zugrunde liegendes Fundamentalkonzept: Generische Programmierung.
  - Auflösung beim *Übersetzen* des Programms
    - zur Build-time.
  - Sog. "parametrische Polymorphie".

- Abgeleitete Klassen und virtuelle Methoden.
  - Zugrunde liegendes Fundamentalkonzept: Generalisierung / Vererbung.
  - Auflösung beim *Ausführen* des Programms
    - zur Run-time.
  - Sog. "ad-hoc Polymorphie".

Templates und abgeleitete Klassen

Was *nicht* funktioniert.

- Sei `Base` eine Basisklasse und `Derived` eine aus `Base` abgeleitete Klasse,
  - d.h. es gilt: ein `Derived` ist ein `Base`.
- Sei `L` ein beliebiges Template,
  - d.h. es existieren z.B. `L<Derived>` und `L<Base>`.
- Es gilt *nicht* : ein `L<Derived>` ist ein `L<Base>`.
- Beispiel:

```
vector<Shape> vs{}; vector<Rectangle> vr{};
vs = vr; // Fehler
vector<Shape*> vps{}; vector<Rectangle*> vpr{};
vps = vpr; // Fehler
vps.push_back( new Rectangle{} );
// funktioniert zwar, fuehrt aber zu Fehlern, sobald vps an
// anderer Stelle z.B. als vector<Triangle*> angesehen wird
```

- Es gibt fortgeschrittene Möglichkeiten, um mit Templates Vererbungsbeziehungen auszudrücken,
  - dies wird im Rahmen dieser Lehrveranstaltung jedoch nicht vertieft.

Templates und abgeleitete Klassen

myRVec<T>

```
template<class T>
class myRVec : public std::vector<T> {
  public:
    myRVec() = delete;
    myRVec( int lo, int hi )
    : std::vector<T>( hi-lo+1 ), lb{lo} {}
    int lo() const { return lb; }
    int hi() const { return lb + std::vector<T>::size() - 1; }
    T& operator[]( int i ) { return std::vector<T>::operator[]( i-lb ); }
    T& at( int i ) { return std::vector<T>::at( i-lb ); }
    // etc.
  private:
    int lb;
};
```

```
myRVec<std::string> v1{ -1,4 };
std::cout << v1.size() << std::endl;
v1[3] = "Walk the line.";
std::cout << v1[3] << std::endl;
myRVec<std::string>* p{&v1};
std::cout << (*p).size() << std::endl;
std::cout << (*p)[3] << std::endl;
std::cout << p->at(-2) << std::endl; // Fehler
// etc.
```

Einige Beispielfragen

Abgeleitete Klassen.

- Was ist der Unterschied zwischen Generalisierung und Vererbung? Wie hängen die beiden Konzepte zusammen?
- Wie nennt man eine allgemeine Klasse, von der es verfeinerte Versionen gibt? Wie nennt man die verfeinerten Klassen?
- Wie symbolisiert man Vererbung in Klassendiagrammen?
- Erläutern Sie Diskriminatoren, geben Sie mindestens drei eigene Beispiele.
- Erläutern Sie an einem *eigenen* Beispiel den Vorteil, den die speziellen C++ Sprachmittel zur Programmierung abgeleiteter Klassen bieten.
- Erläutern Sie, wie Konstruktoren und Destruktoren bei abgeleiteten Klassen arbeiten.
- Wie kann man mit einem Objekt einer abgeleiteten Klasse auf `private` Member der Basisklasse zugreifen?

Einige Beispielfragen

Abgeleitete Klassen.

- Warum kann ein `Derived*` (der auf ein Objekt einer abgeleiteten Klasse zeigt) an einen `Base*` (der auf ein Objekt der Basisklasse zeigt) zugewiesen werden, aber nicht umgekehrt? Erklären Sie, welches Problem entstehen würde, wenn der umgekehrte Fall erlaubt wäre, wenn also ein *Derived* Zeiger die Adresse eines *Base* Objekts enthalten könnte.
- Was sind "virtuelle" Methoden? Wofür genau werden sie eingesetzt?
- Welche wesentlichen Unterschiede sehen Sie zwischen dem Über*schreiben* virtueller Methoden und dem Über*laden* von Methoden?
- Wie verhindert man beim Aufruf einer virtuellen Methoden, dass der Mechanismus für virtuelle Methoden verwendet wird?
- Was sind "rein virtuelle" Methoden? Wofür benötigt man sie? Geben Sie ein *eigenes* Beispiel.
- Warum kann man von abstrakten Klassen keine Objekte erzeugen?
- Welchen Sinn haben abstrakte Klassen, wenn man keine Objekte von Ihnen erzeugen kann?

Einige Beispielfragen

Abgeleitete Klassen.

- Was passiert, wenn eine rein virtuelle Methode einer Basisklasse in einer abgeleiteten Klasse nicht definiert wird?
- Erklären Sie mit eigenen Worten, wie man in einer abstrakte Basisklasse eine Liste der Objekte aus den von ihr abgeleiteten Klassen mitführen kann.
- Warum sind geometrische Figuren als Beispiel für den Einsatz von Vererbung so gut geeignet?
- In welcher Beziehung zueinander stehen die Typen `Shape` und `Line`, in welcher die Typen `Shape` und `Rectangle`?
- In welcher Beziehung zueinander stehen die Typen `Line` und `Rectangle`?
- In welcher Beziehung zueinander stehen die Typen `Rectangle` und `Square`?
- Warum war es so einfach, den Typ `Square` zu definieren?

Einige Beispielfragen

Abgeleitete Klassen.

- Fragen zu Rasterpunkten
  - Was macht `std::array<T,N>`?
  - Warum ist `std::array<T,N>` für die Darstellung von Rasterpunkten so gut geeignet?
  - Wofür wird der Bresenham-Algorithmus verwendet?
  - Welche drei Vereinfachungen werden im Bresenham-Algorithmus angewandt? Welche wesentlichen Vorteile ergeben sich daraus?
  - Erläutern Sie, mit eigenen Worten und ohne Quellcode-Zitate, die vier Fälle für die Ermittlung der Rasterpunkt-Koordinaten im Bresenham-Algorithmus.
  - Erklären Sie genau, was bei den Schleifendurchläufen im Bresenham-Algorithmus geschieht.
  - Wie würden Sie die Praxisrelevanz des Bresenham-Algorithmus einschätzen? Recherchieren Sie Ihre Antwort...

Einige Beispielfragen

Abgeleitete Klassen.

- Aus wie vielen direkten Basisklassen kann eine Klasse abgeleitet werden?
- Erklären Sie an einem eigenen Beispiel: welche/n Vorteil/e hat es, wenn in einer Generalisierungsstruktur mehr als eine direkte Basisklasse zulässig ist, welche/n Nachteil/e hat es?
- Geben Sie ein *eigenes* Beispiel einer Generalisierungsstruktur, in der eine abgeleitete Klasse auf zwei unterschiedlichen Wegen von einer Vorfahrensklasse erbt. Was geschieht hinsichtlich dieser Vorfahrensklasse, wenn Sie ein Objekt vom Typ der abgeleiteten Klasse erzeugen?
- Welche Einsatzmöglichkeit sehen Sie für vollqualifizierte Namen bei mehrfacher Vererbung?
- Erklären Sie das Konzept der virtuellen Basisklassen: wofür dient es, welche Besonderheiten sind zu beachten?
- Was sind Schnittstellen-Klassen, und was haben sie mit mehrfacher Vererbung zu tun?
- Sie haben das grundlegende Implementierungskonzept der *Schnittstellen-Klassen* kennen gelernt. Erläutern Sie die Technik in eigenen Worten. Welche Vorteile ergeben sich?

Nächste Einheit:
Testgetriebene Programmierung