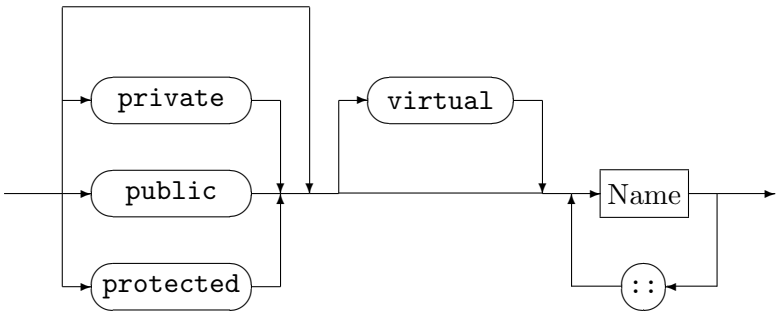
Abgeleitete Klassen

Basisklassenangabe



Durch die Basisklassenangabe in ihrer Definition wird die Klasse um die Komponenten der Basisklasse erweitert (Vererbung).

Die Komp.fkt. und die als `friend` in der abgeleiteten Klasse vereinbarten Funktionen können auf die als `protected` oder `public` gekennzeichneten Komponenten der Basisklasse zugreifen.

Durch die Kennzeichnung der Basisklasse als `public` werden die `public`-Komponenten der Basisklasse `public`-Komponenten der abgeleiteten Klasse und die `protected`-Komponenten der Basisklasse `protected`-Komponenten der abgeleiteten Klasse.
Bei einer Kennzeichnung der Basisklasse als `protected` werden die `public`- und `protected`-Komponenten der Basisklasse zu `protected`-Komponenten der abgeleiteten Klasse.
Bei einer Kennzeichnung der Basisklasse als `private` werden die `public`- und `protected`-Komponenten zu `private`-Komponenten der abgeleiteten Klasse.
Bei Nichtkennzeichnung der Basisklasse ist die Voreinstellung `private`, wenn die Basisklasse als `class` und `public`, wenn sie als `struct` definiert wurde.
Private Komponenten der Basisklasse werden allerdings *nicht* zu privaten Komponenten der abgeleiteten Klasse.

In der abgeleiteten Klasse können gleichnamige Komponenten wie in der Basisklasse definiert werden. Für Objekte der abgeleiteten Klasse haben sie Vorrang vor den Komponenten der Basisklasse.

Bsp.:

```
class Base {
  private:
    int a;
  protected:
    int b;
  public:
    int c,d;
  friend void fbase();
};

class Derived : public Base {
  private:
    int d,e;
  protected:
    int f;
  public:
    int g;
  friend void fderived();
};
```

```
void fbase()     // Befreundet mit Klasse Base
{
  Base base;
  Derived derived;
  base.a = 1; base.b = 2; base.c = 3; base.d = 4;     // ok
  derived.a = 1; derived.b = 2;                       // ok, etwas erstaunlich
  derived.c = 3; derived.Base::d = 4;                 // ok
  derived.d = 5;                                      // geht nicht
  derived.e = 6; derived.f = 7;                       // geht nicht
  derived.g = 8;                                      // ok
}

void fderived()  // Befreundet mit Klasse Derived
{
  Base base;
  Derived derived;
  base.a = 1; base.b = 2;                             // geht nicht
  base.c = 3; base.d = 4;                             // ok
  derived.a = 1;                                      // geht nicht
  derived.b = 2; derived.c = 3;                       // ok
  derived.Base::d = 4;                                // ok
  derived.d = 5;                                      // ok
  derived.e = 6; derived.f = 7; derived.g = 8;        // ok
}

int main()
{
  Base base;
  Derived derived;
  base.a = 1; base.b = 2;                             // geht nicht
  base.c = 1; base.d = 2;                             // ok
  derived.a = 1; derived.b = 2;                       // geht nicht
  derived.c = 3; derived.Base::d = 4;                 // ok
  derived.d = 5; derived.e = 6; derived.f = 7;        // geht nicht
  derived.g = 8;                                      // ok
}
```

Die Konstruktoren und Destruktoren der Basisklasse werden *nicht* vererbt. In der Konstruktorinitialisierungsliste kann neben den Komponentennamen der abgeleiteten Klasse auch der Basisklassenkonstruktor vorkommen und zur Initialisierung der Basisklasse dienen. Sofern er nicht angegeben ist, erfolgt die Initialisierung mit dem Standardkonstruktor. Die Komponenten der Basisklasse werden vor den Komponenten der abgeleiteten Klasse initialisiert.

Komponentenfunktionen der abgeleiteten Klasse haben ebenfalls Vorrang vor gleichnamigen Komponentenfunktionen der Basisklasse bei übereinstimmender Parameterliste.

Bsp.: Vektordatentyp mit Bereichsüberprüfung

```
#include <iostream>
#include <vector>
#include <stdexcept>

using namespace std;

template<class T> class chkvector : public vector<T> {
  public:
    chkvector()      : vector<T>()  {}
    chkvector(int n) : vector<T>(n) {}
    T& operator[](int i)             { return vector<T>::at(i); }
    const T& operator[](int i) const { return vector<T>::at(i); }
};

#define vector chkvector // disgusting macro hack (Stroustrup)

int main()
{
  int i,n;
  try {
    cout << "Vektordimension ? "; cin >> n;
    vector<double> a(n);
    for (i=0; i<=n; i++) cout << (a[i]=i) << " ";
  }
  catch (out_of_range ausnahme) {
    cerr << "Ausnahme out_of_range entdeckt" << endl;
    cerr << "Ausnahme: " << ausnahme.what() << endl;
  }
  catch (...) {
    cerr << "Andere Ausnahme entdeckt" << endl;
  }
  return 0;
}
```

Ausgabe:

```
Vektordimension ? 4
0 1 2 3 Ausnahme out_of_range entdeckt
Ausnahme: vector::_M_range_check: __n (which is 4) >= this->size() (which is 4)
```

Typumwandlungen zwischen Basisklasse und abgeleiteter Klasse

Die abgeleitete Klasse D kann als Spezialisierung des Basisdatentyp B betrachtet werden, wobei die Spezialisierung durch die Hinzunahme neuer Komponenten zustandekommt. Die Typumwandlung $D \rightarrow B$ erfolgt implizit. Das gilt nicht für die Typumwandlung $B \rightarrow D$, weil ein Objekt vom Typ B weniger Komponenten besitzen kann, als ein Objekt vom Typ D (und deshalb nicht alle für ein Objekt vom Typ D definierten Operationen auch auf ein Objekt vom Typ B anwendbar sind).

Virtuelle Funktionen

Komponentenfunktionen in einer Basisklasse und in einer davon abgeleiteten mit dem gleichen Namen und der gleichen Parameterliste können auch über Zeiger aufgerufen werden. Bei nichtvirtuellen Komponentenfunktionen hängt es vom Zeigertyp ab (nicht aber vom Zeigerwert), ob die Funktion der Basisklasse oder die der abgeleiteten Klasse ausgewählt wird.
Bei als `virtual` in einer Basisklasse vereinbarten Funktionen wird für einen Basisklassenfunktionszeiger dennoch die Funktion der abgeleiteten Klasse ausgewählt, wenn der Zeiger auf ein Objekt der abgeleiteten Klasse zeigt.

Bsp.:

```
#include <iostream>

using namespace std;

class Base
{
  public:
    void f()         { cout << "f - Basisklasse" << endl; }
    virtual void v() { cout << "v - Basisklasse" << endl; }
};

class Derived : public Base
{
  public:
    void f() { cout << "f - abgeleitete Klasse" << endl; }
    void v() { cout << "v - abgeleitete Klasse" << endl; }  // automatisch virtuell
};

int main()
{
  Derived d;
  Base *bp = &d;     // bzw. Base& b = d;
  bp->f();           //      b.f();
  bp->v();           //      b.v();
  return 0;
}
```

Ausgabe:

```
f - Basisklasse
v - abgeleitete Klasse
```