§3 Vererbung – Allgemeines

Leitideen: *Die Hinzufügung von Daten- oder Funktionskomponenten zu einer vorhandenen Klasse (Basisklasse) wird als Vererbung und die auf diese Weise erzeugte Klasse als abgeleitete Klasse bezeichnet.*

Die privaten Komponenten der Basisklasse sind vor dem Direktzugriff aus der abgeleiteten Klasse geschützt, um ein Aushebeln des Klassenkonzepts zu verhindern. Die Vorrangregel für gleichnamige Komponenten der abgeleiteten Klasse ermöglicht das "Überschreiben" der nichtprivaten Komponentenfunktionen der Basisklasse.

Durch den Mechanismus der Vererbung ist die Weiterbenutzung der hinzugefügten Funktionalität möglich, auch wenn sich die Implementierung der Basisklasse ändert.

§3 Vererbung – Allgemeines II

Leitideen: *Gelegentlich sollen Objekte abgeleiteter Klassen einheitlich als Basisklassenobjekte behandelt werden (Polymorphismus). Virtuelle Funktionen in der Basisklasse ermöglichen den korrekten Aufruf von Komponentenfunktionen abgeleiteter Klassen, weil im Objekt ein Verweis auf eine Tabelle von Funktionszeigern gespeichert wird. Das funktioniert allerdings nur für den Zugriff über Zeiger oder Referenzen.*

Die Implementierung virtueller Funktionen in der Basisklasse kann vermieden werden (rein virtuelle Funktionen). Dann können allerdings keine Basisklassenobjekte, sondern lediglich Basisklassentypzeiger und -referenzen mit Verweisen auf konkrete Objekte abgeleiteter Klassen gebildet werden (abstrakte Klassen).

§3 Vererbung – Themenübersicht

- Vererbung - Begriff, Zweck, Prinzip
- Vererbung - Attribute
- Vererbung - Gleichnamige Komponenten
- Vererbung - Konstruktoren, Destruktoren
- Virtuelle Funktionen - Zweck
- Virtuelle Funktionen - Realisierung
- Virtuelle Funktionen - Zusammenfassung
- Rein virtuelle Funktionen, abstrakte Klassen
- Virtuelle Destruktoren

Vererbung - Begriff, Zweck, Prinzip

Begriff: Erweiterung einer Klasse („abgeleitete Klasse“) um die Komp. einer bereits definierten Klasse („Basisklasse“) [Objekte der abgeleiteten Klasse haben mindestens so viele Datenkomp. wie die Objekte der Basisklasse]

Zweck: Vermeidung von Mehrfachdef., automat. Übernahme von Änderungen der Basisklasse in die abgeleitete Klasse

Prinzip: Gleichnamige Komp. der abgeleit. Klasse haben Vorrang vor denen der Basisklasse (gilt für Daten- und Fkt.komp.)

Private Komponenten der Basisklasse nicht zugreifbar, andere Basisklassenkomp. je nach Art der Ableitung.

Vererbung - Attribute

- Art der Ableitung (public, protected, private) bestimmt Zugriffsrechte auf Basisklassenkomponenten für die abgeleitete Klasse
- Am wichtigsten public-Ableitung: Schränkt nur den Zugriff auf private Basisklassenkomponenten ein
- Private Komponenten der Basisklasse werden *nicht* zu privaten Komponenten der abgeleiteten Klasse.
  Sonst: Direkter Zugriff auf Komponenten einer Klasse durch *nicht* von der Klasse autorisierte Funkt. möglich (Zugriffsschutz von privaten Komponenten ginge verloren, Datenkapselung wäre weitgehend aufgehoben.)
- Basisklasse kann den Zugriff auf ihre privaten Komp. per `friend`-Deklaration für einzelne abgeleitete Klassen erlauben (selten verwendet)
- Alternativ: Komponenten als `protected` statt `private` kennzeichnen. [Abgeleitete Klassen dürfen auf als `protected` gekennzeich. Komp. der Basisklasse zugreifen]

Vererbung - gleichnamige Komponenten

- In der abgeleiteten Klasse können gleichnamige Komp. wie in der Basisklasse definiert werden:

  Bsp.:
  ```
  struct Base              {int a,b,c,d};
  struct Derived : Base {int d,e,f,g};
  // Base: 4 Komp.   Derived: 8 Komp.
  ```

- In Objekten der abgeleiteten Klasse: Vorrang der Komp. der abgeleiteten Klasse *vor* den Komp. der Basisklasse

  Bsp.:
  ```
  Base base;
  Derived derived;
  derived.d;          // Komp. d von Derive
  Base::derived.d;    // Komp. d von Base
  ```

- „Überdecken“ von Komponentenfunktionen häufiger als „Überdecken“ von Datenkomponenten

Bsp.: Base/Derived mit Zugriffsattributen → Inf.bl.13, S.1
Vektordatentyp mit Bereichsüberprüfung → Inf.bl.13, S.3

Vererbung - Konstruktoren/Destruktoren

- *Keine* Vererbung der Konstruktoren und Destruktoren der Basisklasse.
  Sonst u. U. nur partielle Initialisierung von Objekten der abgeleiteten Klasse (d.h. lediglich Komponenten der Basisklasse werden initialisiert).
  Deshalb auch keine Vererbung der Funktion `operator=`
- Verwendung von Basisklassenkonstruktoren in Konstruktorinitialisierungsliste erlaubt und sinnvoll
- Initialisierung der Basisklassenkomponenten erfolgt vor der der übrigen Komponenten, ggf. mit Standardkonstruktoren
- Aufruf der Destruktoren in umgekehrter Reihenfolge

Virtuelle Funktionen - Zweck

- Abgeleitete Klasse als Spezialisierung der Basisklasse: Spezialisierung durch Hinzufügung weiterer (auch gleichnamiger) Komponenten
- Gelegentlich wünschenswert: Eine Reihe von Objekten aus einer Basisklasse und davon abgeleiteten Klassen einheitlich behandeln. („Polymorphismus“)

  Bsp.:
  ```
  struct Base                {void f(){}};
  struct Derived : Base {void f(){}};
  list<Base> bl;
  Derived d;
  bl.push_back(d); // Typumw (Base)d okay
  ```
  jedoch: `bl.front().f()` ruft falsches f() auf!
- Typumw. `Derived→Base` zulässig (weniger Komp.!)
  `Base→Derived` unzulässig (mehr Komp.!)
- Lösung dieses Problems durch *virtuelle* Funktionen

Virtuelle Funktionen - Realisierung

- Typumwandlungen sind am flexibelsten für Zeigertypen.

  Bsp.:
  ```
  struct Base                {void f(){}};
  struct Derived : Base {void f(){}};
  list<Base*> bpl;
  Derived d;
  bpl.push_back(&d); // Typumw (Base*)&d ok
  ```
  immer noch: `bpl.front()->f()` ruft falsches f() auf!
- *Abhilfe:* `virtual void f()` statt `void f()` in Base
- Erschließen der richtigen Funktion f() dann *allein* aus der Adresse des Datenobjekts möglich, weil für virtuelle Funktionen im Objekt zusätzlich ein Verweis auf eine Funktionszeigertabelle gespeichert wird.
- *Hintergrund:* Objekte von Klassen *ohne* virtuelle Funktionen enthalten *nur* Datenkomponenten. Deshalb Objekttyp wichtig für Auswahl gleichnamiger nicht virtueller Komponentenfunktionen.

Virtuelle Funktionen - Zusammenfassung

- *Regel:* Der Datentyp eines Klassenobjekts entscheidet, welche der gleichnamigen Komponentenfunktionen mit der gleichen Parametertypliste ausgewählt wird.
- *Ausnahme:* Komponentenfunktion der abgeleiteten Klasse wird ausgewählt, wenn:
  1. Komponentenfunktion in der Basisklasse als `virtual` definiert *und*
  2. Zugriff auf Objekt der abgeleiteten Klasse über Basisklassenzeiger bzw. -referenz erfolgt
- *Hintergrund:* Typumwandlung `Derived`→`Base` entfernt Komponenten von `Derived`, Änderung des Zeigers auf virtuelle Funktionstabelle aus Konsistenzgründen erforderl. Typumwandlung `Derived*`→`Base*` bzw. `Derived&`→ `Base&` lässt Bitmuster (bei übl. Klassenlayout) ungeändert.
- *Ergänzung:* Gleichnamige Funktionen mit der gleichen Parameterliste in abgeleiteter Klasse automat. virtuell (auch ohne `virtual`-Dekl.), falls in Basisklasse virtuell

Rein virtuelle Funktionen, Abstrakte Klassen

- *Rein virtuelle Funktion:* Virtuelle Funktion, bei der der Funktionsrumpf durch `=0` ersetzt ist
- *Abstrakte Klasse:* Klasse mit mindestens einer rein virtuellen Funktion
- Abstrakte Klassen können *keine* Objekte enthalten!
- Auch als Parameter- oder Ergebnistyp von Funktionen unzulässig
- Zeiger oder Referenzen auf abstrakte Klassen erlaubt
- Bildung abgeleiteter Klassen (mit Objekten!) möglich
- Abstrakte Klasse oft Ausgangspunkt einer Klassen-hierarchie
- Polymorphismus also auch in C++ möglich, aber etwas umständlich wegen Verwendung von Zeigern oder Referenzen
- C++ kennt auch Mehrfachvererbung – hier nicht behandelt

Virtueller Destruktor

Zweck: Vermeidung von Speicherlecks

- Beim Löschen von dynamisch allozierten Objekten über Basisklassenzeiger wird die statische Typinformation benutzt, wenn *kein* virtueller Destruktor in der Basisklasse vorhanden ist.

  Bsp.:
  ```
  struct Base                {int a;};
  struct Derived : Base {int b;};
  Base *p  = new Derived
  delete p; // Wuerde nur 4B statt
            // 8B freigeben
  ```

- Ähnlich wie bei virtuellen Funktionen ist in dieser Situation ein virtueller Destruktor erforderlich. (Im C++-Standard vorgeschrieben!)

  Bsp.:
  ```
  struct Base { int a;
        virtual ~Base() {} };
  // Restliche Zeilen unveraendert
  ```

- *Regel:* Bei Klassen mit virtuellen Funktionen auch virtuellen Destruktor definieren.