


THE

C++

PROGRAMMING LANGUAGE

CPP-04 – Inheritance and Polymorphism

CPVR Vertiefungsmodul BTI-7281
Urs Künzler (urs.kuenzler@bfh.ch)




Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CPP-04 Table of Contents – Inheritance and Polymorphism		
04.01	Introduction	4
04.02	Inheritance Relation	5
04.03	Inheritance of Access Rights	9
04.04	Inheritance of Member-Functions	13
04.05	Polymorphism	16
04.06	Virtual Functions	20
04.07	Abstract Classes	24
04.08	Explicit Type Conversion (Casting)	25
04.09	Runtime Type Information (RTTI)	28

CPP-04

2



Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CPP-04

Table of Contents – Inheritance and Polymorphism II

04.10

Multiple Inheritance

29

CPP-04

3

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

04.01

Einführung

C++

■ Klassenbeziehungen

- Eine Klasse wird in einer realen Applikation nicht für sich isoliert verwendet, sondern meist bestehen vielfältige Beziehungen zu anderen Klassen.
- Die Modellierung eines Systems durch miteinander interagierende Objekte, erfordert eine entsprechende Modellierung der Beziehungen zwischen Klassen.
- C++ kennt verschiedene Möglichkeiten Klassenbeziehungen abzubilden, wobei die Vererbungsbeziehung die mächtigste darstellt und für die objektorientierte Programmierung unbedingt erforderlich ist.

CPP-04

4

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

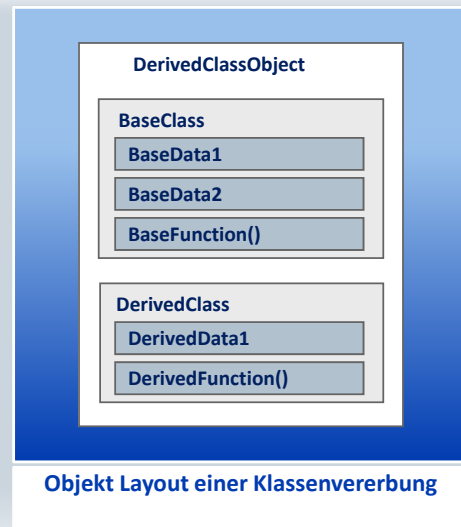
04.02

Vererbungsbeziehung



- **Vererbungsbeziehung**

- Eine Vererbungsbeziehung (Inheritance-Relation) beschreibt die hierarchische Abhängigkeit einer Unterklasse von einer Basisklasse.
- Eine Vererbungsbeziehung wird zur Abbildung eines Generalisierungs- bzw. Spezialisierungs-Verhältnisses zwischen zwei Klassen verwendet.



CPP-04

5

Vererbungsbeziehung II



- Bei der Vererbung werden die Daten- und Funktions-Member der Basisklasse (unter Berücksichtigung der Zugriffsrechte) an die abgeleitete Klasse übertragen.
- Eine Ausnahme bilden Konstruktoren, der Destruktor und der Zuweisungsoperator, die nicht an die abgeleitete Klasse vererbt werden. Die Default-Implementation dieser Member-Funktionen wird für die abgeleitete Klasse implizit vom Compiler erzeugt.
- Für die Default-Implementation dieser Member-Funktionen gelten die gleichen Regeln (vgl. Copy-Konstruktor, Zuweisungsoperator) wie bei einer nicht vererbten Klasse. *eigener copy-constructor wenn deep copy gewünscht ist*

CPP-04

6

Vererbungsbeziehung III



- Die Vererbung bewirkt eine Verschachtelung des Geltungsbereichs von Klassen, wobei der Scope einer abgeleiteten Klasse vom Scope der Basisklasse(n) umgeben wird.
- Innerhalb der Implementation einer abgeleiteten Member-Funktion kann deshalb auf alle Klassen-Member der Basisklasse(n) genauso zugegriffen werden als seien diese in der abgeleiteten Klasse definiert worden (Zugriffsrechte berücksichtigen).
- Durch mehrfaches Ableiten über mehrere Stufen können Klassenhierarchien aufgebaut werden (Bsp. GUI-Klassenbibliotheken).
- Man unterscheidet zwischen direkter bzw. indirekter Basisklasse, je nachdem ob eine vererbte Klasse unmittelbar bzw. von einer weiter oben in der Vererbungshierarchie deklarierten Klasse abgeleitet wird.

CPP-04

7

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

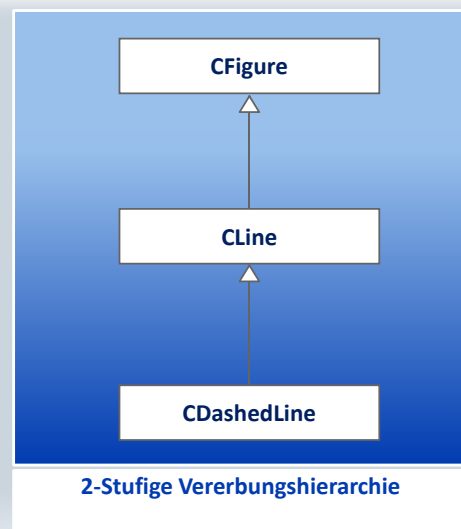
Vererbungsbeziehung - Beispiel



```
class CFigure {
protected:
    int iColor;
public:
    CFigure(void);
};

class CLine : public CFigure {
protected:
    CPoint P1, P2;
public:
    CLine(void);
    ...
};

class CDashedLine : public CLine {
public:
    CDashedLine(void);
    ...
};
```



CPP-04

8

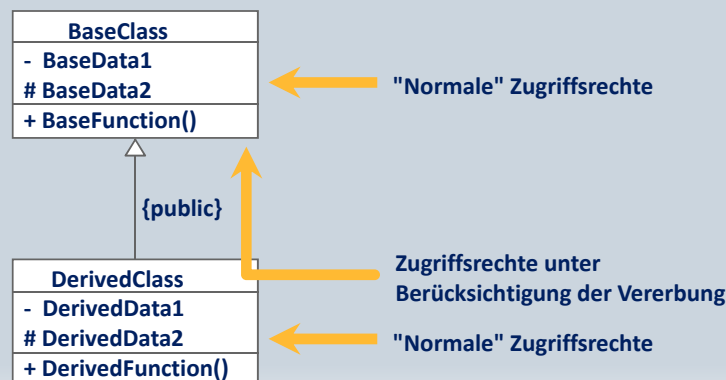
B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

04.03

Vererbung von Zugriffsrechten



- Für die Zugriffsrechte auf Member einer abgeleiteten Klasse muss berücksichtigt werden mit welchen Zugriffsrechten die Vererbung erfolgt:



CPP-04

9

 Berner Fachhochschule
 Haute école spécialisée bernoise
 Bern University of Applied Sciences

Vererbung von Zugriffsrechten II



- Regeln für die Vererbung von Zugriffsrechten:**
 - public:** Bei einer **public-Vererbung** einer Klasse werden public Member der Basis-klasse zu public Membern der abgeleiteten Klasse und protected Member der Basisklasse zu protected Membern der abgeleiteten Klasse.
 - protected:** Bei einer **protected-Vererbung** einer Klasse werden public und protected Member der Basisklasse zu protected Membern der abgeleiteten Klasse.
 - private:** Bei einer **private-Vererbung** einer Klasse werden public und protected Member der Basisklasse zu private Membern der abgeleiteten Klasse.
- Unabhängig von der Vererbung der Zugriffsrechte sind private Klassen-Member der Basis-klasse in abgeleiteten Klassen nicht zugänglich (Ausnahme: friend deklarierte Member der Basisklasse).
- Ohne explizite Angabe des Zugriffsspezifizierers wird für eine mit class deklarierte abgeleitete Klasse die private-Vererbung als Default verwendet. Für eine mit struct deklarierte abgeleitete Klasse wird als Default public-Vererbung benutzt.

CPP-04

10

 Berner Fachhochschule
 Haute école spécialisée bernoise
 Bern University of Applied Sciences

Vererbung von Zugriffsrechten III



■ Zusammenfassung der Vererbung von Zugriffsrechten:

Vererbung	Basisklasse		Abgeleitete Klasse
public	public	→	public
	protected	→	protected
	private	- - - ✗ - - -	no access
protected	public	→	protected
	protected	→	protected
	private	- - - ✗ - - -	no access
private	public	→	private
	protected	→	private
	private	- - - ✗ - - -	no access

typkompatibilität
nicht mehr gewährleistet
-> interface ist nicht
mehr kompatibel

- Bei einer public-Ableitung bleiben die Zugriffsrechte erhalten → “is-a” – Beziehung.
- Bei einer protected- oder private-Ableitung werden die Zugriffsrechte eingeschränkt → “is implemented in terms of” Beziehung
- Da bei einer private/protected-Ableitung die Schnittstelle der Basisklasse nicht vererbt wird, ist die abgeleitete Klasse nicht typkompatibel zu ihrer Basisklasse. **Konversionen von Zeigern und Referenzen sind deshalb nicht mehr möglich.**

CPP-04 11

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Typenkonversion abgeleiteter Klassen



- Zwischen einer public abgeleiteten Klasse und ihrer Basisklasse sind die folgenden impliziten Typenkonversionen definiert:
 - Ein **Objekt einer abgeleiteten Klasse** wird implizit in ein Objekt der Basisklasse konvertiert.
 - Eine **Referenz auf ein Objekt einer abgeleiteten Klasse** wird implizit in eine Referenz auf ein Objekt der Basisklasse konvertiert.
 - Ein **Pointer auf ein Objekt einer abgeleiteten Klasse** wird implizit in einen Pointer auf ein Objekt der Basisklasse konvertiert.
- Durch diese implizite Typenkonversion können bei Initialisierungen, Parameterübergaben und Zuweisungen anstelle von Objekten der Basisklasse auch Objekte der abgeleiteten Klasse stehen.
- Dadurch ist eine public abgeleitete Klasse vollständig typen-kompatibel zu ihrer Basisklasse, dies gilt aber nicht für private oder protected abgeleitete Klassen!

CPP-04 12

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

04.04

Vererbung und Member-Funktionen



- Konstruktoren und Destruktoren
- Member-Verdeckung (Hidding)
- Funktionsüberladung (Function Overloading)
- Funktionsüberschreibung (Function Overriding)
- Virtuelle Funktionen (Virtual Functions)
- Abstrakte oder rein-virtuelle Funktionen (Pure Virtual Functions)

CPP-04

13

 Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Konstruktor- und Destruktor-Verkettung



- Bei der Instanzierung von Objekten einer abgeleiteten Klasse werden zuerst die Daten-Member der Basisklasse(n) initialisiert. Dies geschieht in der logischen Reihenfolge der Klassenhierarchie durch impliziten Aufruf des Default-Konstruktors der Basisklasse(n).
- Soll bzw. kann für eine Basisklasse nicht der Default-Konstruktor aufgerufen werden, kann bzw. muss der zu verwendende Konstruktor explizit in der Initialisiererliste des Konstruktors der abgeleiteten Klasse angegeben werden. Beispiel:
`DerivedClass::DerivedClass(int x) : BaseClass(x) {...};`
- Beim Löschen eines Objektes einer abgeleiteten Klasse wird in umgekehrter Reihenfolge zuerst der Destruktor der abgeleiteten Klasse ausgeführt und danach implizit der Destruktor der Basis-klasse(n) aufgerufen.

CPP-04

14

Demo: CPP-04-D.01 - Inheritance

 Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Klassen-Member Verdeckung (Hidding)



- Wird in einer abgeleiteten Klasse ein Klassen-Member mit dem gleichen Namen wie in der Basisklasse deklariert, so verdeckt der Member der abgeleiteten Klasse denjenigen der Basisklasse. (Dies gilt auch wenn die Typen der Daten-Member bzw. die Parameter der Funktions-Member nicht übereinstimmen!)
- Auf verdeckte Member der Basisklasse kann aber durch Qualifizierung mit dem Namen der Basisklasse trotzdem zugegriffen werden (Scope-Operator).
- Beispiel - Daten-Member Verdeckung:

```
void CDashedLine::SetDashColor(int iColor) {
    int point_color = CFigure::iColor;
    int dash_color = iColor;
    ...
}
```

CPP-04

15

Demo: CPP-04-D.01 - Inheritance



04.05

Polymorphismus



- **Polymorphismus**
 - Unter Polymorphismus ("Vielgestaltigkeit") versteht man die Eigenschaft, dass sich eine Funktion mit gleichem Namen (in voneinander abgeleiteten Klassen) unterschiedlich verhalten kann.
- **Statischen Polymorphie**
 - Bei der statischen Polymorphie ermittelt der Compiler zur Compile-Zeit welche Funktion zur Laufzeit aufgerufen werden muss (static binding). In C++ realisiert durch Funktions-überladung und statische Funktionsüberschreibung (nicht virtuelle Funktionen).
- **Dynamischen Polymorphie**
 - Bei der dynamischen Polymorphie wird erst zur Laufzeit ermittelt welcher Funktions-Code ausgeführt werden muss (dynamic oder late binding). In C++ realisiert durch dynamische Funktionsüberschreibung (virtuelle Funktionen).

CPP-04

16



Funktionsüberladung (Overloading)



■ Funktionsüberladung (Overloading)

- Für das Überladen von Member-Funktionen in Vererbungsstrukturen gelten die gleichen Regeln wie für das statische Überladen von Funktionen innerhalb einer Klasse, d.h. überladene Funktionen (gleicher Name) müssen durch ihre unterschiedliche Signatur unterscheidbar sein.
- Vererbte Member-Funktionen können innerhalb der Klassen-hierarchie auch in einer abgeleiteten Klasse überladen werden.

CPP-04 17

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Funktionsüberschreibung (Overriding)



■ Funktionsüberschreibung (Overriding)



- Von Funktionsüberschreibung (Function Overriding) spricht man, wenn eine Member-Funktion einer Basisklasse in einer abgeleiteten Klasse durch eine Funktion mit gleicher Signatur überschrieben wird.
- Bei der statischen Überschreibung wird die aufzurufende Funktion zur Compile-Zeit ermittelt (static binding). Für "normal", d.h. nicht virtuell deklarierte Funktionen ist dies der Standard.
- Bei der dynamischen Überschreibung wird die konkret aufzurufende Funktion dynamisch zur Laufzeit-Zeit ermittelt (dynamic binding). Dies wird für Funktionen verwendet, die mit **virtual** deklariert sind.

zusätzliche tabelle muss unterhalten werden
virtual function pointer table

1x virtual gilt dann für alle abgeleiteten klassen auch

CPP-04 18

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Polymorphe Variablen



- In C++ können nur Pointer- und Referenz- Variablen zu verschiedenen Zeiten auf Objekte unterschiedlicher Klassen zeigen. Sie heissen deshalb auch polymorphe Variablen.
- Beispiel: `CFigure* pF1 = new CLine(15, 15, 25, 25);`
- Bei polymorphen Variablen unterscheidet man zwischen dem statischen und dem dynamischen Typ:
 - Der statische Typ ist der Typ, mit der die Variable deklariert wurde.
 - Der dynamische Typ ist der Typ des Objekts, auf das die Variable zur Laufzeit zeigt.
- Standardmässig werden die Member-Funktionen des statischen Typs einer polymorphen Variablen aufgerufen, auch wenn der dynamische Typ Member-Funktionen mit gleicher Signatur besitzt (static binding). **Verwendung des Keywords `virtual` ändert dieses Verhalten.** insbesondere ist `virtual` keyword beim destruktur wichtig

CPP-04 19

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

04.06

Virtuelle Funktionen



- **Virtuelle Funktionen**
 - Wird eine Member-Funktion einer Klasse als `virtual` spezifiziert, so kann diese in abgeleiteten Klassen dynamisch überschrieben werden.
 - Beispiel:


```
class CFigure {
public:
    virtual void List(void);
    ...
};
```
- Damit wird erreicht, dass bei einer polymorphen Variablen der dynamische Typ entscheidet welche Member-Funktion ausgeführt wird, wenn in der abgeleiteten Klasse eine Funktion mit gleicher Signatur existiert (dynamic binding).
- In den meisten anderen OO-Programmiersprachen ist der „Virtual Mechanismus“ der Default. Deshalb wird in UML implizit von virtual Operationen ausgegangen.

CPP-04 20

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Virtuelle Funktionen II



- Die überschreibende Member-Funktion in der abgeleiteten Klasse ist implizit wieder virtual, sie kann aber auch explizit als virtual spezifiziert werden.
- In der Implementation einer überschreibenden Member-Funktion kann die virtuelle Funktion der Basisklasse durch die Verwendung des Scope-Operators aufgerufen werden.

- Beispiel: `CLine::List()`

```
{
    CFigure::List(); // call base class List()
    cout << "P1 (" << x << ", " << y << ")" << endl;
    cout << "P2 (" << x << ", " << y << ")" << endl;
}
```

CPP-04 21

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Virtuelle Funktionen III



- Eine Klassen mit virtuellen Funktionen wird auch als polymorphe Klassen bezeichnet.
- Die Implementation einer polymorphe Klassen erfolgt durch eine vom Compiler automatisch erzeugte "virtual function table" die intern mit einem "virtual function pointer" (`__vfptr`) referenziert wird. [pro klassenhierarchie eine separate tabelle](#)
- Konstruktoren und static Member-Funktionen können nicht virtuell deklariert werden, da sie nicht für existierende Objekte aufgerufen werden.
- Der Destruktor einer Klasse sollte immer dann virtuell deklariert werden, wenn von einer Basisklasse weitere Klassen abgeleitet werden, für die ein eigener Destruktor implementiert ist.

CPP-04 22

Demo: CPP-04-D.01 - Inheritance

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Abstrakte oder rein virtuelle Funktionen



- Wird eine virtuelle Funktion mit "0 initialisiert", spricht man von einer abstrakten oder rein virtuellen Funktion (Pure Virtual Function).
- Beispiel:

```
class CFigure {  
public:  
    virtual void List(void) = 0;  
    ...  
};
```
- Weil rein virtuelle Funktionen in der Klasse in der sie deklariert werden nicht implementiert werden müssen, können sie für die Interface-Spezifikation verwendet werden. Eine pure virtual Funktion kann aber trotzdem eine Implementation haben, muss aber nicht.
- Ausnahme: Ein rein virtueller Destruktor muss eine Implementation besitzen weil dieser infolge der Destruktor-Verkettung automatisch aufgerufen wird (Linker – Error).

CPP-04

23

 Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

04.07

Abstrakte Klassen



- **Abstrakte Klassen**
 - Eine Klasse, die mindestens eine rein virtuelle Funktion enthält, wird als abstrakte Klasse bezeichnet.
 - Abstrakte Klassen sind nicht direkt instanzierbar und können deshalb nur als Interface-Klassen verwendet werden.
 - Eine abstrakte Klasse wird als Basisklasse einer anderen Klasse verwendet, in welcher die rein virtuellen Funktionen implementiert werden. Die abgeleitete Klasse ist damit nicht mehr abstrakt und kann instanziiert werden.
 - Die Deklaration von Pointern und Referenzen auf abstrakte Klassen ist möglich.

CPP-04

24

Demo: CPP-04.01D - Inheritance

 Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

04.08

Explizite Typenkonversion (Casting)



- Die Konversion vordefinierter Datentypen erfolgt implizit durch den Compiler gemäss einer arithmetischen Standardkonversion.
 - Beispiel: float --> double --> long double
unsigned char --> unsigned short int --> unsigned int --> unsigned long int
- Mit einer expliziten Typenkonversion kann eine Typenumwandlung gezielt durchgeführt oder eine implizite Standardkonversion den spezifischen Bedürfnissen angepasst werden.
- Konversion mittels C Cast-Notation:
 - Beispiel: `double d = (int) (12 * 1.2345); // d = 14`
- Die Konversion mittels Funktionalem-Casting ermöglicht die Typenkonversion mit einem oder mehreren Argumenten (für Klassen entspricht dies dem Aufruf eines Konstruktors).
 - Beispiel: `double d = int(12 * 1.2345); // d = 14`

CPP-04

25

 Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

C++ Cast-Operatoren



- Der Operator `static_cast<T> ()` wird verwendet, wenn eine implizite Konversion vom Typ des Ausdrucks in Typ (T) möglich ist.
 - Beispiel: `int n = 3, m = 2;
double x = static_cast<double>(n)/m; // x = 1.5`
- Mittels `const_cast<T> ()` Operator kann die const-Deklaration eines Pointers aufgehoben werden (nur für Pointer).
 - Beispiel: `const int* i = new int(0); int n = 10;
(const_cast<int>(i)) = n; // i = 10`
- Bei der Umwandlung mittels `reinterpret_cast<T> ()` wird jegliche Typenprüfung ausgeschaltet. (Verwendung zum Beispiel für void* Pointer Konvertierung oder Treiber-Entwicklung)
 - Beispiel: `char* s = "Hello World!";
double x = *(reinterpret_cast<double*>(s)); // d=2.19...`

CPP-04

26

 Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

C++ Cast-Operatoren II



- Mit dem Operator `dynamic_cast<T>()` wird dynamisch geprüft, ob der Typ eines polymorphen Klassen-Pointers in eine abgeleitete Klasse konvertiert werden kann. Ist die Umwandlung nicht möglich, wird ein Nullzeiger zurückgegeben.

▪ Beispiel:

```
class A {...}; class B : public A {...};

void f(A* a)
{
    B* b = dynamic_cast<B*>(a); // "Downcast"
    ...
}
```

- Dieser Operator funktioniert nur mit polymorphen Typen, d.h. die Basisklasse muss mind. eine virtuelle Funktion deklariert haben.

- Eine explizite Typenkonversion in C++ ist oft ein Hinweis auf ein schlechtes Applikations-Design!

CPP-04 27

 Berner Fachhochschule
 Haute école spécialisée bernoise
 Bern University of Applied Sciences

04.09

Runtime Type Information (RTTI)



- Runtime Type Information (RTTI) erlaubt zur Laufzeit Informationen über (dynamische) Datentypen (v.a. Klassen) zu erhalten.
- Der `typeid()` Operator liefert zu einem Ausdruck oder einem Typ-namen ein `type_info` - Objekt zurück, das die (dynamischen) Typ Information des Arguments repräsentiert.
- Zwei `type_info` - Objekte können miteinander auf Gleichheit bzw. Ungleichheit geprüft werden. Mit der Funktion `name()` kann der Name des Typs abgefragt werden.

▪ Beispiel:

```
class A {...}; class B : public A {...};

void ShowDiagnostics(A* a)
{
    if (typeid(*a) != typeid(A)) // Typvergleich
        cout << typeid(*a).name(); // "class B"
}
```

CPP-04 28

 Berner Fachhochschule
 Haute école spécialisée bernoise
 Bern University of Applied Sciences

04.10

Mehrfachvererbung



■ Mehrfachvererbung (Multiple Inheritance)

- In C++ können Klassen auch von mehreren direkten Basisklassen abgeleitet werden. Die abgeleitete Klasse erbt dann die Klassen-Member von jeder dieser Basisklassen.
- Bei Mehrfachvererbung können leicht Mehrdeutigkeiten entstehen. So ist der Name eines Klassen-Members bereits mehrdeutig, wenn es in den Basisklassen mehrere Deklarationen dieses Namens gibt (auch bei unterschiedlichen Typen).
- Solche Mehrdeutigkeiten können aufgelöst werden, indem der entsprechende Name vollständig qualifiziert wird.

CPP-04

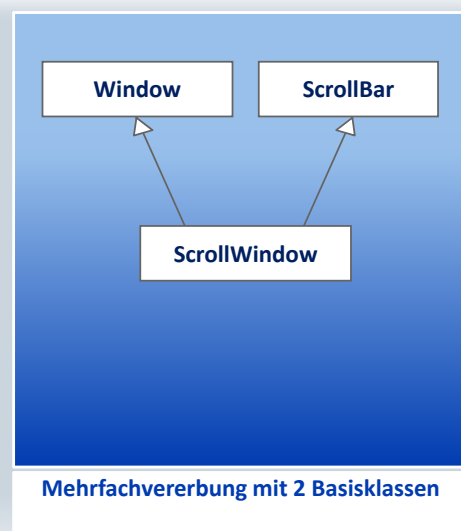
29

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Mehrfachvererbung - Beispiel



```
// Deklaration der Basisklassen
class Window {
public:
    Window(void);
    Move(int x, int y);
    ...
};
class ScrollBar {
public:
    ScrollBar(void);
    Move(int x, int y);
    ...
};
// Deklaration der mehrfach abgeleiteten Klasse
class ScrollWindow : public Window, public ScrollBar
{
public:
    ScrollWindow(void);
    ...
};
```



CPP-04

30

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Mehrfachvererbung – Beispiel II



```
// ScrollWindow Definition
ScrollWindow ScrollWnd;

// Verschieben des Fensters
ScrollWnd.Move(10, 10); // Compiler-Error: Mehrdeutigkeit
```

■ Auflösung der Mehrdeutigkeit:

```
// expliziter Funktionsaufruf mit Scope-Operator
ScrollWnd.ScrollWnd::Move(10, 10);
ScrollWnd.ScrollBar::Move(10, 10);

// Definition von Move() in der abgeleiteten Klasse
void ScrollWindow::Move(int x, int y) {
    Window::Move(x, y);
    ScrollBar::Move(x, y);
}
```

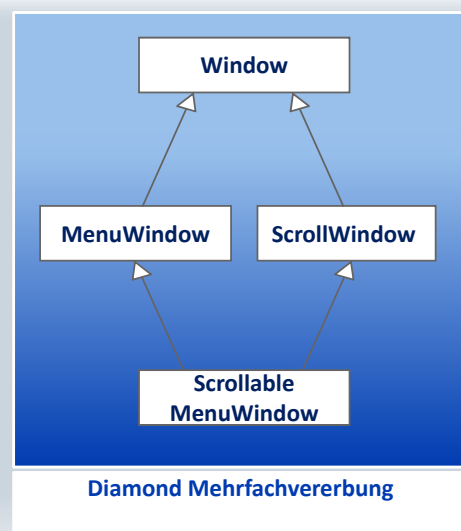
CPP-04 31

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Virtuelle Basisklassen



- Bei der Mehrfachvererbung kann es vorkommen, dass dieselbe Klasse mehrfach als indirekte Basisklasse auftritt.
- In diesem Fall ist die Basisklasse entsprechend mehrfach als Teil-objekt in der indirekt abgeleiteten Klasse enthalten.



CPP-04 32

B Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

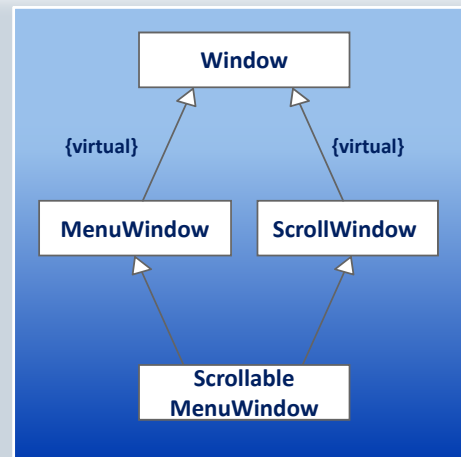
Virtuelle Basisklassen II



- Mit der **virtual** Deklaration in allen direkten Ableitungen der Basis-Klasse kann erreicht werden, dass die Basisklasse nur einmal als Teilobjekt in **indirekt** abgeleiteten Klassen enthalten ist.

- **Beispiel:**

```
class ScrollWindow:  
    virtual public Window {  
        ...  
};  
  
class MenuWindow:  
    virtual public Window {  
        ...  
};
```



Diamond Mehrfachvererbung
mit virtual Spezifikation