

THE

# C++

PROGRAMMING LANGUAGE


## CPP-02 – Classes and Objects

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



Berner Fachhochschule  
Haute école spécialisée bernoise  
Bern University of Applied Sciences

CPP-02		Table of Contents – Classes and Objects	
02.01	Was ist eine Klasse / Klassen-Syntax		4
02.02	Erzeugung und Verwendung von Instanzen		14
02.03	Zugriffsrechte und Geltungsbereich		20
02.04	Spezielle Funktions-Member		25
02.04.1	Funktionsüberladung (Overloading)		26
02.04.2	Konstruktor		31
02.04.3	Copy-Konstruktor		36
02.04.4	Zuweisungsoperator		41
02.04.5	Destruktor		44
CPP-02		2	



Berner Fachhochschule  
Haute école spécialisée bernoise  
Bern University of Applied Sciences

## CPP-02

## Table of Contents – Classes and Objects II

02.04.6	static Daten- und Funktions-Member	46
02.04.7	Überladen von Funktions-Member	50
02.04.8	Operatoren für Klassen	51
02.05	Default Arguments	54
02.06	friend Funktionen und Klassen	56
02.07	inline Funktionen	59
02.08	const Funktionen	60
02.09	Effective C++	62

CPP-02

3

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

## 02.01

## Was ist eine Klasse



- Klassen bilden den zentralen Datentyp für die objektorientierte Programmierung.
- Klassen ermöglichen die Abstraktion, Kapselung und Vererbung von Informationen und Operationen sowie die Instanzierung (Erzeugung) von Objekten.
- Klassen werden im Source-Code nur als Datentyp deklariert und existieren damit nur zur Kompilationszeit. Eine Klasse benötigt deshalb zur Laufzeit kein Memory (im Gegensatz zu einem instanziierten Objekt dieser Klasse).

CPP-02

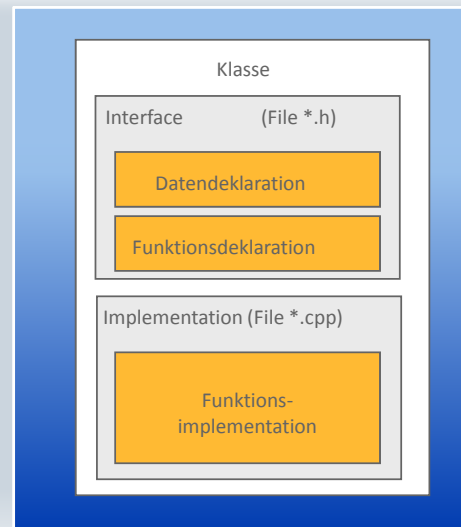
4

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

## Was ist eine Klasse II



- Eine Klasse wird zur Deklaration eines benutzerdefinierten, abstrakten Daten-typs verwendet, welcher neben Daten auch die auf diese Daten anwendbaren Funktionen umfasst.
- In C++ spricht man von Daten- bzw. Funktions-Member oder zusammen-fassend von Klassen-Member.



CPP-02 5

## Was ist eine Klasse – Modularisierung III



- Zur besseren Trennung und Modularisierung werden in C++ Interface und Implementierung einer Klasse physisch getrennt.
- Die Deklaration einer Klasse (Interface) wird in einem Header-File mit File Extension .h (.hpp oder .hxx) gespeichert.
- Die Implementation der Klasse erfolgt in einem Code-File mit File Extension .cpp (.cxx).
- Eine weitergehende Modularisierung kann durch die Verwendung von statischen und dynamische Libraries oder durch eigenständig lauffähige Software-Komponenten erreicht werden.

CPP-02 6

## Klassen - Syntax



### ■ Beispiel - Deklaration einer abgeleiteten Klasse:

```

class CMyWindow : public CWindow
{
private:
    int X;
    int Y;
public:
    CMyWindow();
    CMyWindow( int x, int y );
    void SetWindowPos( int x, int y );
};
  
```

Diagram labels and arrows:

- C++ Keyword** points to `class`.
- Klassenname** points to `CMyWindow`.
- Basisklasse** points to `CWindow`.
- Zugriffs-spezifizierer** points to `private:` and `public:`.
- Data Member** points to `int X;` and `int Y;`.
- Function Member** points to `CMyWindow();`, `CMyWindow( int x, int y );`, and `void SetWindowPos( int x, int y );`.

CPP-02 7

## Klassen Syntax – Beispiel II



### ■ Beispiel - Interface (Deklaration) einer Klasse (Header File \*.h):

```

class CPoint
{
public:
    float X;           // X Koordinate des Punktes
    float Y;           // Y Koordinate des Punktes

    void set(float x, float y); // Deklaration der Funktion set()
};                      // Deklaration abschliessen mit ;
  
```

### ■ Beispiel - Implementation einer Klasse (Source File \*.cpp):

```

#include "header.h"
void CPoint::set(float x, float y)
{
    X = x;
    Y = y;
}
  
```

Diagram label: **Scope-Operator** points to `CPoint::`.

CPP-02 8

## Spezifikation von Speicherklassen



- **C++ unterscheidet verschiedene Speicherklassen (storage class):**
  - Automatischer Speicher (**Stack**, Automatic Memory → run-time)
  - Statischer Speicher (**Static Memory**, Data Section → compile-time)
  - Dynamischer Speicher (**Heap**, Dynamic Memory, Free Store → run-time)
- **Durch die explizite Angabe der Speicherklasse kann der Geltungs-bereich bzw. die Lebensdauer einer Variablen beeinflusst werden:**
  - **static**      Verlängert Lebensdauer lokaler Variablen auf die Programmlebensdauer. Default-Speicherklasse für globale Variablen (Static Memory).
  - **extern**      Variablen werden nur deklariert (ohne Speicher-Allokation). Die Definition erfolgt in einem anderen, durch den Linker referenzierten, Modul. (extern linkage / binding)
  - **auto**        Default-Speicherklasse für lokale Variablen (Automatic Memory)
  - **register**    Hinweis zur Performance-Optimierung für den Compiler (CPU Register)

CPP-02    9

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

## Spezifikation von Speicherklassen II



	Definition ausserhalb eines Blockes (mit static)	Definition ausserhalb eines Blockes (ohne static)	Definition innerhalb eines Blockes (mit static)	Definition innerhalb eines Blockes (ohne static)
<b>Geltungsbereich</b>	File global	Programm global	lokal	lokal
<b>Lebensdauer</b>	gesamte Laufzeit des Programms	gesamte Laufzeit des Programms	gesamte Laufzeit des Programms	Laufzeit des Blockes
<b>Speicherklasse</b>	static	static	static	auto
<b>Initialisierung</b>	implizit zu 0	implizit zu 0	implizit zu 0	keine
<b>Speicherung</b>	Static Memory	Static Memory	Static Memory	Stack

- Globale, mit static definierte Variablen werden zur Kontrolle des Bindings verwendet:
  - Globale Definition mit static (File global) → Internal Linkage
  - Globale Definition ohne static (Programm global) → External Linkage
- Lokale, mit static definierte Variablen werden initialisiert, wenn der Kontrollfluss zum ersten mal die Definition erreicht.

CPP-02    10

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

möglichkeit new/delete Überladen -> eigenes speichermanagement implementieren

### Nutzung von Dynamischem Speicher



- Dynamische Variablen werden für Daten benötigt deren Grösse zur Kompilationszeit nicht bekannt ist.
- Dynamische Variablen werden in C++ mittels `new` Operator im Dynamic Memory alloziert und mittels `delete` Operator gelöscht.  
Löschen eines nullpointers ist unproblematisch
- Die Lebensdauer einer dynamische Variablen endet mit deren Freigabe (`delete`) oder bei Programmende.
- Falls `new` bei der Erzeugung einer neuen Variablen nicht genügend Speicher allozieren kann, wird eine `bad_alloc` Exception zurück-gegeben.
- In einem C++ Programm müssen für Klassen immer die Operatoren `new` und `delete` verwendet werden (Konstruktor/Destruktor Aufrufe)!

CPP-02 11

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

### Nutzung von Dynamischem Speicher



- Beispiele - Allokation von dynamischem Speicher mit `new`:

```
// Einfache Beispiele
char* str = malloc(20);      // in C   - alloziert 20 Byte
char* ch  = new char;        // in C++ - alloziert ein char
char* str = new char[20];    // in C++ - alloziert Array von 20 char

// Beispiel mit komplexerer Datenstruktur
struct Node {int item; Node* left; Node* right;};
...

// Allokation in C
struct Node* node = (struct Node*) malloc(sizeof(struct Node));

// Allokation in C++
Node* node = new Node;
Object* objects = new Object[100];
```

CPP-02 12

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

## Nutzung von Dynamischem Speicher II



### ■ Beispiele - Freigabe von dynamischem Speicher mit delete:

```
free(Object);           // in C
delete Object;          // in C++ für das Löschen einfacher Variablen
delete[] objects;       // in C++ für das Löschen von Arrays
```

- Anwendung von delete auf einen Null-Pointer hat keine Wirkung.
- Falls new ein Array alloziert, muss beim dazugehörigen delete ebenfalls die Array Notation (delete[ ]) verwendet werden (damit die dazugehörigen Destruktoren aufgerufen werden).
- Werden mit new allozierte Speicherbereiche nicht explizit wieder freigegeben, können Memory Leaks entstehen.

CPP-02 13

## 02.02

## Erzeugung von Klassen Instanzen



- Aus einer Klassendeklaration können Objekte (Instanzen) dieser Klasse statisch oder dynamisch definiert (instanziert) werden.
- Bei der Instanzierung von Objekten wird vom Compiler Memory bereitgestellt, wobei jede Instanz eine eigene Kopie der Daten-Member dieser Klasse enthält. Der Code der Funktions-Member dieser Klasse wird von allen Instanzen gemeinsam verwendet.
- Beispiel - Statische Instanzierung:

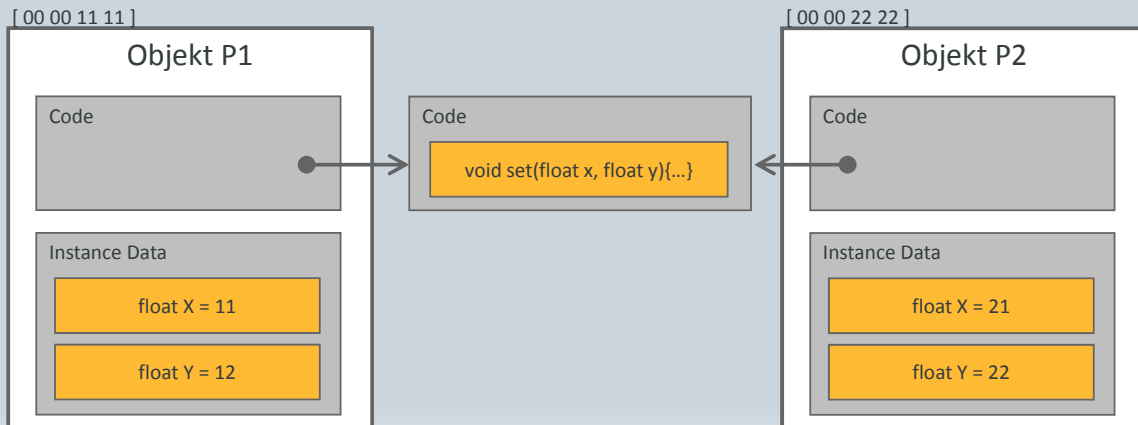
```
// Definition zweier Objekte der Klasse CPoint
CPoint P1, P2;
```

CPP-02 14

## Erzeugung von Klassen Instanzen II



- Objekt Memory-Layout (schematisch)



CPP-02 15

## Erzeugung von Klassen Instanzen III



- Beispiel - Dynamische Instanziierung:

```
// Definition zweier Pointer auf Objekte der Klasse CPoint
CPoint *pP1, *pP2;

// dynamische Erzeugung der Objekte
pP1 = new CPoint;
pP2 = new CPoint;

// Verwendung der dynamisch angelegten Objekte
...

// Löschen der Objekte
delete pP1;
delete pP2;
```

CPP-02 16



## Verwendung von Instanzen



### ■ Beispiel - Verwendung statischer Objekte:

```
CPoint P1, P2;           // Objekte der Klasse CPoint
float x;                 // Hilfswert
...

// Ansprechen der Objekt Daten-Member
P1.X = 25;
P1.Y = P1.X;
x = P1.CPoint::X;        // explizite Schreibweise
...

// Ansprechen der Objekt Funktions-Member
P2.set(100, 100);        // normale Schreibweise
P2.CPoint::set(100, 100); // explizite Schreibweise
...
```

CPP-02 17

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

## Verwendung von Instanzen II



### ■ Beispiel - Verwendung dynamischer Objekte:

```
// Definition zweier Pointer auf Objekte der Klasse CPoint
CPoint *pP1, *pP2;

// dynamische Erzeugung der Objekte
pP1 = new CPoint;
pP2 = new CPoint;

// Ansprechen der Member dynamischer Objekte
pP1->X = 10;
(*pP1).Y = 20;
pP1->set(10, 20);
pP2->CPoint::set(200, 150); // explizite Schreibweise

// Löschen der Objekte
delete pP1;
delete pP2;
```

CPP-02 18

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

## this Pointer - Implizites Funktions-Argument



- Zur eindeutigen Identifizierung der Objektdaten wird jeder Member-Funktion implizit ein zusätzliches „this“ Argument übergeben.
- Dieses Argument entspricht einem const-Pointer auf das aufrufende Objekt und kann mit dem Keyword **this** explizit referenziert werden.
  - Beispiel:
 

```
// P1.set(x,y) --> set(CPoint* const this, float x, float y);

void CPoint::set(int x, int y)
{
    this->X = x;           // identisch mit: X = x
    this->Y = y;           // identisch mit: Y = y
    func( this );         // Funktionsaufruf: func(CPoint* ptr)
}
```
- Der this-Pointer hat innerhalb der Member-Funktion einen konstanten Wert und kann nicht modifiziert werden.
- Static Member-Funktionen haben keinen this Pointer.

CPP-02

19

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

### 02.03

## Zugriffsrechte und Geltungsbereich



- Das OO-Konzept der Kapselung wird in C++ mit der Spezifikation der folgenden Klassen-Zugriffsrechte realisiert:
  - **public:** Auf public Klassen-Member kann aus jeder Funktion eines Programms zugegriffen werden. Verwendet für Member die den Zugang zu Daten und Funktionen der Klasse ermöglichen sollen.
  - **protected:** Auf protected Klassen-Member kann aus den eigenen Member-Funktionen und aus den Funktionen vererbter Klassen zugegriffen werden. Von Aussen (globaler Scope oder bei nicht vererbten Klassen) besteht jedoch keine Zugriffsmöglichkeit.
  - **private:** Klassen-Member mit private Zugriffsspezifizierer können nur von Member- und Friend-Funktionen der eigenen Klasse angesprochen werden (vollständige Kapselung gegen aussen).
- Der Zugriffsschutz wird durch den Compiler geprüft und erfolgt damit statisch während dem kompilieren.

CPP-02

20

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

## Zugriffsrechte und Geltungsbereich II



- Das standard Zugriffsrecht für Klassen die mittels `class` definiert werden ist `private`, für Klassen die mittels `struct` definiert werden gilt `public` Zugriff.
- Innerhalb einer Klassendeklaration können beliebig viele Zugriffs-spezifizierer in beliebiger Reihenfolge definiert werden. Ein defini-erter Zugriff gilt für alle folgenden Member bis zum nächsten Zugriffsspezifizierer oder bis zum Ende der Klassendeklaration.
- Die Namen von Klassen-Member haben den Scope Klasse, weshalb der selbe Name für Daten- oder Funktions-Member in mehreren Klassen definiert werden kann.

Beispiel: `CPoint P1; CLine L1;`  
`...`  
`P1.set(150, 100); // Aufruf von CPoint::set()`  
`L1.set(15, 25); // Aufruf von CLine::set()`

CPP-02

21

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

## Verschachtelte Klassendeklaration



- Die Deklaration einer lokal (innerhalb einer Klasse) verwendeten Klasse kann innerhalb der Deklaration der umgebenden Klasse erfolgen (Nested Class Declaration).

Beispiel:

```
class CLine
{
    // verschachtelte Klassendeklaration
    class CPoint {
        float X;
        float Y;
    public:
        void set(float x, float y);
    };

    private:
        CPoint P1, P2;
        ...
};
```

CPP-02

22

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

## Verschachtelte Klassendeklaration II



- Der verschachtelte (eingebettete) Klassenname besitzt nur lokalen Scope innerhalb der umgebenden Klasse.
- Die Implementation einer Member-Funktion einer verschachtelten Klasse muss deshalb mit dem Namen der umgebenden Klasse qualifiziert werden.
- Beispiel: `void CLine::CPoint::set(float, float) {...};`
- Member-Funktionen einer verschachtelten Klasse haben keine besonderen Zugriffsrechte auf Klassen-Member der umgebenden Klasse und umgekehrt.

CPP-02 23

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

## forward Deklaration



- Eine Forward-Deklaration deklariert den Namen einer Klasse, sodass Pointer und Referenzen auf die Klasse erzeugt werden können, bevor die Klasse vollständig deklariert ist.
- Eine Forward Deklaration wird meist in Zusammenhang mit einer friend Klasse verwendet oder um zu vermeiden, dass ein Header-File ein `#include` von weiteren Header-Files beinhalten muss.
- Beispiel - Header-File Deklaration:

```
class CPoint;

class CLine {
private:
    CPoint* pP1;
    CPoint* pP2;
    ...
};
```

CPP-02 24

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

## 02.04

## Spezielle Class-Member Funktionen



- Funktionsüberladung (Overloading)
- Default - Konstruktor
- Default - Copy-Konstruktor
- Default - Zuweisungsoperator
- Default - Destruktor
- static Daten- und Funktions-Member
- Überladen von Funktions-Member
- Operatoren für Klassen

CPP-02 25

### 02.04.1

### Funktionsüberladung (Overloading)



- In C++ werden Funktionen aufgrund ihres Namens sowie der Anzahl und der Typen ihrer Argumente unterschieden (Signatur).
- Die Deklaration von Funktionen mit dem selben Namen aber unterschiedlicher Signatur wird als Funktionsüberladung bezeichnet (Statische oder signaturgebundene Polymorphie).
- Die eindeutige Funktionsunterscheidung für den Linker erfolgt durch die Generierung eindeutiger Symbole, welche neben dem Funktionsnamen auch die Signatur berücksichtigen (Name Mangling)
- Dieses Name Mangling erfolgt statisch zur Kompilationszeit (Static Binding) und ist Compiler und Plattform abhängig!

CPP-02 26

## Funktionsüberladung (Overloading) II



- Für die Funktionsüberladung gelten folgende Regeln:
  - Funktionen die sich nur durch den Rückgabotyp unterscheiden, sind nicht überladbar.
  - `Type` und `const Type` sind als Typen von Argumenten nicht unterschiedlich genug.
  - `Type` und `Type&` sind als Typen von Argumenten nicht unterschiedlich genug.
  - `Type*` und `Type[ ]` sind als Typen von Argumenten nicht unterschiedlich genug.
  - Argument-Typen die sich nur durch ein `typedef` unterscheiden, sind nicht überladbar.
  - Kombinationen von obigen Regeln führen ebenfalls zu ungültigen Überladungen.

CPP-02 27

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

## Funktionsüberladung (Overloading) III



- Beispiel:
 

```
// typedef Deklaration
typedef char* PSTR;

// Gültige Funktionsüberladung
void print( char* str, int width );
void print( char* str );
void print( int value );
void print( double value );

// Ungültige Funktionsüberladung - Rückgabotyp
int print( char* str );

// Ungültige Funktionsüberladung - typedef
void print( PSTR str );

// Ungültige Funktionsüberladung - int&
void print( int& value );
```

CPP-02 28

Demo: CPP-02-D.02\_FunctionOverloading / Name Mangling

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

## Einbindung von C Code in C++



- Aufgrund des C++ Name Mangling Mechanismus kann der Linker die Namen extern definierter C Symbole nicht auflösen.
- Mit der `extern "C"` Directive kann das Name Mangling des C++ Compilers explizit ausgeschaltet werden.

nur namen wird als symbol berÜcksichtig -> wie in C

- Beispiel zur Einbindung einer einzelnen C-Funktion in C++

```
extern "C" int MyOldCFunction(char*);
```

- Beispiel zur Einbindung ganzer C-Header Files in C++

```
extern "C" {
    #include "My_Old_C_Lib.h"
}
```

CPP-02 29

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

## Einbindung von C Code in C++ II



- C-Header File zur gleichzeitigen Verwendung in C und C++

```
#ifdef __cplusplus
extern "C" {
#endif

    int MyOldCFunction1(char*);
    int MyOldCFunction2(char*);

#ifdef __cplusplus
}
#endif
```

CPP-02 30

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

## 02.04.2 Konstruktor



- C++ verwendet für die Initialisierung eines Objektes spezielle Member-Funktionen, die sogenannten Konstruktoren.
  - Beispiel: 

```
// Konstruktor Funktionsdeklaration für Klasse
CPoint
    CPoint(int x, int y);
```
- Der Konstruktor wird beim Instanzieren eines Objektes automatisch durch den Compiler aufgerufen, unabhängig davon ob die Instanzierung statisch oder dynamisch erfolgt.
- Ein Konstruktor der ohne Argumente aufgerufen werden kann, wird als Default-Konstruktor (Standard-Konstruktor) bezeichnet.
  - Allgemeine Schreibweise: `T::T()`
  - Beispiel: `CPoint::CPoint();`

CPP-02

31

 Berner Fachhochschule  
 Haute école spécialisée bernoise  
 Bern University of Applied Sciences

## Konstruktor II



- **Regeln zur Implementation von Konstruktoren:**
  - Der Funktionsname eines Konstruktors ist identisch mit dem Klassennamen.
  - Eine Klasse kann mehrere Konstruktoren besitzen. Diese müssen aber in Bezug auf den Typ bzw. die Anzahl der Argumente verschieden sein (vgl. Funktionsüberladung).
  - Ein Konstruktor darf keinen Rückgabetyt besitzen (auch nicht void).
  - Ein Konstruktor kann nicht static oder virtual deklariert werden.
  - Falls in der Klassendeklaration kein Konstruktor spezifiziert wird, erzeugt der Compiler automatisch einen public deklarierten Default-Konstruktor.
  - Wird für eine Klasse ein Konstruktor definiert, wird der Default-Konstruktor vom Compiler nicht automatisch erzeugt. Ein allenfalls notwendiger Default-Konstruktor muss in diesem Fall manuell implementiert werden.
- Für global definierte Objekte wird der Konstruktor automatisch aufgerufen und zwar vor dem Start des Programms (main).

CPP-02

32

 Berner Fachhochschule  
 Haute école spécialisée bernoise  
 Bern University of Applied Sciences



## Konstruktor III



### ■ Beispiel - Konstruktordeklaration:

```
class CPoint {
    float X, Y;                // X,Y Koordinaten des Punktes
public:
    CPoint(void);              // Default-Konstruktor
    CPoint(float x, float y);   // Konstruktor
    void set(float x, float y); // Deklaration der Funktion set()
};
```

### ■ Beispiel - Konstruktorimplementation:

```
CPoint::CPoint(void)
{
    X = 0; Y = 0;
}

CPoint::CPoint(float x, float y)
{
    X = x; Y = y;
}
```

CPP-02 33

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

## Konstruktor IV



### ■ Beispiel - "statischer" Konstruktoraufruf:

```
CPoint P1;                // impliziter Aufruf von CPoint()
CPoint P2();              // Fehler: Funktionsdeklaration !!
CPoint P3(100.0, 50.0);   // Aufruf von CPoint(float, float)
```

### ■ Beispiel - "dynamischer" Konstruktoraufruf:

```
CPoint* pP1 = new CPoint;    // impliziter Aufruf CPoint()
CPoint* pP2 = new CPoint();  // expliziter Aufruf CPoint()
CPoint* pP3 = new CPoint(100, 50); // expliziter Aufruf
                                // CPoint(float,float)
```

CPP-02 34

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

## Konstruktor Initialisiererliste

nur im konstruktor möglich



- Die Initialisierung von Daten-Member eines Objektes kann im Konstruktor auch mit Hilfe einer Initialisiererliste erfolgen.

Beispiel:

```
CPoint::CPoint(float x, float y)
: X(x), Y(y)
{
    ...
}
```

- Diese Art der Initialisierung ist effizienter als eine Zuweisung im Rumpf des Konstruktors, bei const- Daten-Membren ist es die einzig mögliche Art.
- Eine Initialisiererliste kann nur bei Konstruktoren (inkl. Copy-Konstruktoren) verwendet werden, aber nicht bei "normalen" Member-Funktionen.

CPP-02

35

Berner Fachhochschule  
Haute école spécialisée bernoise  
Bern University of Applied Sciences

### 02.04.3

## Copy-Konstruktor



- Ein Copy-Konstruktor ist ein Konstruktor um Objekte zu kopieren.
- Der Copy-Konstruktor wird mit einem Referenz-Parameter auf das zu kopierende Objekt als Argument deklariert:
  - Allgemeine Schreibweise: `T::T(const T&)`
  - Beispiel: `CPoint::CPoint(const CPoint&)`
- Implizit wird der Copy-Konstruktor aufgerufen, wenn ein Objekt einer Funktion "by value" übergeben wird, oder wenn ein Objekt bei der Instanzierung durch eine Objektzuweisung initialisiert wird.

Beispiele:

```
CPoint P1(5,10);           // normaler Konstruktor
CPoint P2(P1);             // expliziter Copy-Konstruktor
CPoint P3 = P1; ist effizienter // impliziter Copy-Konstruktor
CPoint P4 = CPoint(P1);    // impliziter Copy-Konstruktor
CPoint P5 = CPoint(5,10);  // normaler Konstruktor
CPoint* pP6 = new CPoint(P1); // expliziter Copy-Konstruktor
```

CPP-02

36

Berner Fachhochschule  
Haute école spécialisée bernoise  
Bern University of Applied Sciences

## Copy-Konstruktor II



- **Regeln zur Implementation von Copy-Konstruktoren:**
  - Wird ein Copy-Konstruktor nicht explizit deklariert, erzeugt der Compiler automatisch einen public deklarierten Default-Copy-Konstruktor.
  - Der Default-Copy-Konstruktor kopiert Daten-Member elementweise.
  - Bei der Variablen-Definition mit einer Zuweisung wird implizit ein Copy-Konstruktor aufgerufen.
  - **Werden innerhalb einer Klasse Daten-Member dynamisch alloziert, muss die explizite Implementation eines Copy-Konstruktors geprüft werden.**
  - Wird für eine Klasse ein spezifischer Zuweisungsoperator implementiert, muss die explizite Implementation eines Copy-Konstruktors geprüft werden.

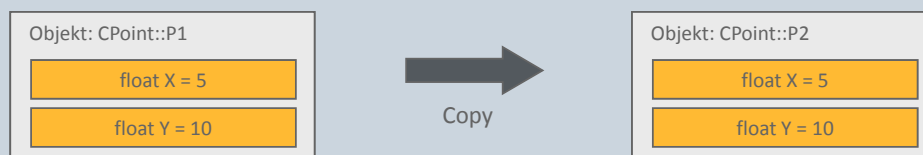
CPP-02 37

 Berner Fachhochschule  
Haute école spécialisée bernoise  
Bern University of Applied Sciences

## Copy-Konstruktor III



- **Für Objekte mit statischen Membern kopiert der Default-Copy-Konstruktor die Daten-Member elementweise (Shallow Copy):**



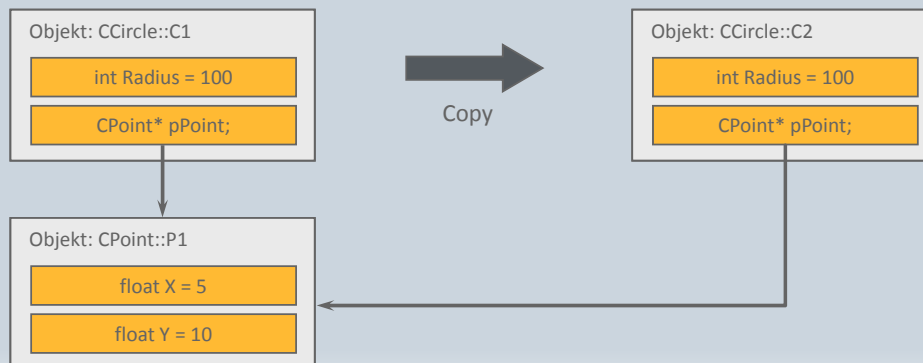
CPP-02 38

 Berner Fachhochschule  
Haute école spécialisée bernoise  
Bern University of Applied Sciences

## Copy-Konstruktor IV

C++

- Für Objekte mit dynamischen Membern kopiert der Default-Copy-Konstruktor auch Pointer Daten-Member elementweise:



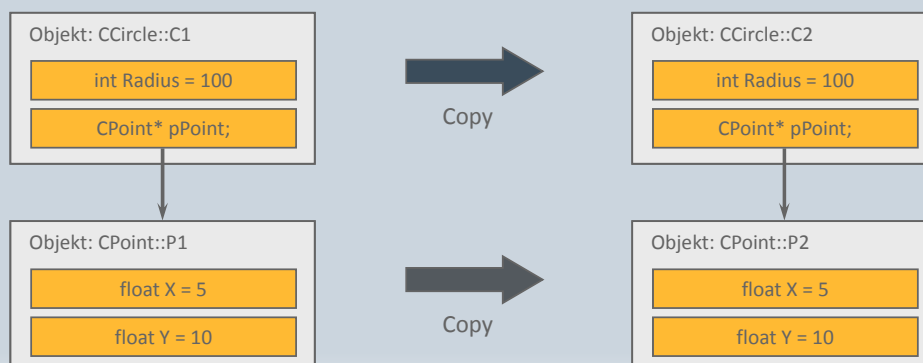
CPP-02 39

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

## Copy-Konstruktor V

C++

- Für Objekte mit dynamischen Membern muss meist ein spezieller Copy-Konstruktor manuell implementiert werden (Deep Copy):



CPP-02 40

Demo: CPP-02-D.03\_CopyConstructor

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

copy-constructor unterbinden:  
Type(const Type& that) = delete;