02.04.4 Zuweisungsoperator

- **Die Zuweisung an Objekte einer Klasse wird mit dem Zuweisungs-operator definiert.**
  - Allgemeine Schreibweise: `T& T::operator=(const T&)`
  - Beispiel: `CPoint& CPoint::operator=(const CPoint&)`
- **Bei Definition mit Zuweisung wird implizit ein Copy-Konstruktor aufgerufen, nur beivorgängiger Definition und anschliessender Zuweisung (Initialisierung), wird der Zuweisungsoperator benutzt.**
  - Beispiele:

```
CPoint P1(5,10);    // normaler Konstruktor
CPoint P2(P1);      // expliziter Copy-Konstruktor
CPoint P3 = P1;     // impliziter Copy-Konstruktor
CPoint P4;          // normaler Default-Konstruktor
P4 = P1;            // Zuweisungsoperator
```

- **Beim Zuweisungsoperator sind grundsätzlich die gleichen Regeln zu berücksichtigen wie bei einem Copy-Konstruktor.**

Zuweisungsoperator II

- **Regeln für die Implementation eines Zuweisungsoperators:**
  - Wird ein Zuweisungsoperator nicht explizit deklariert, erzeugt der Compiler automatisch einen public deklarierten Default-Zuweisungsoperator.
  - Beim Default-Zuweisungsoperator erfolgt die Zuweisung elementweise.
  - Werden innerhalb einer Klasse Daten-Member dynamisch alloziert,muss die explizite Implementation eines Zuweisungsoperators geprüft werden.
  - Wird für eine Klasse ein spezifischer Copy-Konstruktor implementiert, muss die explizite Implementation eines Zuweisungsoperators geprüft werden.

Zuweisungsoperator III

- **Eine Referenz als Rückgabewert des Zuweisungsoperators erlaubt die Verkettung der Zuweisungsoperation in einem Ausdruck:**
  - Beispiel - einfacher Aufruf (auch mit void Rückgabetyp möglich):
```
C2 = C1;           // impliziter Aufruf
C2.operator=(C1);  // expliziter Aufruf
```
  - Beispiel - verketteter Aufruf (nur mit Referenzrückgabetyp möglich):
```
C3 = C2 = C1;                    // impliziter Aufruf
C3.operator=(C2.operator=(C1)); // expliziter Aufruf
```
  - Beispiel - Implementation:
```
CCircle& CCircle::operator=(const CCircle& pSource)
{
   Radius    = pSource.Radius;
       pPoint->X = pSource.pPoint->X;
       pPoint->Y = pSource.pPoint->Y;
       return *this;
}
```


02.04.5 Destruktor

- **Analog dem automatischen Konstruktoraufruf beim Erstellen eines Objektes, wird beim Löschen des Objektes durch den Compiler implizit der Destruktor aufgerufen.**
- **Der Destruktor kann verwendet werden um beim Löschen eines Objektes dynamisch allozierten Speicher freizugeben oder um geöffnete Files zu schliessen etc.**
- **Der Funktionsname des Destruktors ist identisch mit dem Klassen-namen mit einem vorangestellten Tilde-Zeichen "~" (auch als Komplement des Konstruktors bezeichnet).**
  - Allgemeine Schreibweise: `T::~T()`
  - Beispiel: `CPoint::~CPoint();`
- **Der Destruktor-Aufruf erfolgt unabhängig davon ob das Objekt statisch oder dynamisch instanziert wurde.**

Destruktor II

- **Regeln für die Implementation eines Destruktors:**
  - Ein Destruktor kann keinen Funktionswert zurückgegeben (nicht einmal void) und besitzt keine Argumente, weshalb ein Destruktor nicht überladen werden kann.
  - Da der Rumpf eines Destruktors ausgeführt wird bevor die Destruktoren der eingebetteten Daten-Member aufgerufen werden, können Klassen-Member innerhalb des Destruktors normal verwendet werden.
  - Wird für eine Klasse kein expliziter Destruktor deklariert, erzeugt der Compiler einen public deklarierten Default-Destruktor, welcher die Daten-Member elementweise löscht.
  - Wird im Konstruktor dynamisch Speicher alloziert, muss der Destruktor im allgemeinen explizit implementiert werden.
  - In Vererbungshierarchien sind Destruktoren als virtual zu deklarieren.
- **Für global definierte Objekte wird der Destruktor automatisch aufgerufen und zwar nach Beenden des Programms.**

02.04.6 static Daten- und Funktions-Member

- **Static Daten-Member sind nicht an ein bestimmtes Objekt einer Klasse gebunden, sondern speichern ihren Wert einmal für alle Objekte dieser Klasse (Class Member / Class Data).**
- **Die Deklaration eines static Daten-Members innerhalb einer Klassen-deklaration ist noch keine Definition (Linker – Error)! Diese muss an separater Stelle des Programms implementiert werden.**
- **Static Daten-Member können ohne die Instanzierung eines Objektes (unter Berücksichtigung der Zugriffsrechte) angesprochen werden.**

static Daten-Member II

- **Beispiel - Static Daten-Member Deklaration (Header-File *.h):**

```
class CPoint {
   float X, Y;          // X,Y Koordinaten des Punktes
public:
   static bool Visible; // Flag für Sichtbarkeit aller Punkte
public:
   void set(float, float);
   void setVisibility(bool);
};
```

speicher muss explizit alloziert werden.
ansonsten linker fehler - unreference exception

- **Beispiel - Static Daten-Member Definition (Source-File *.cpp):**

```
bool CPoint::Visible = true;  // Definition und Initialisierung
```

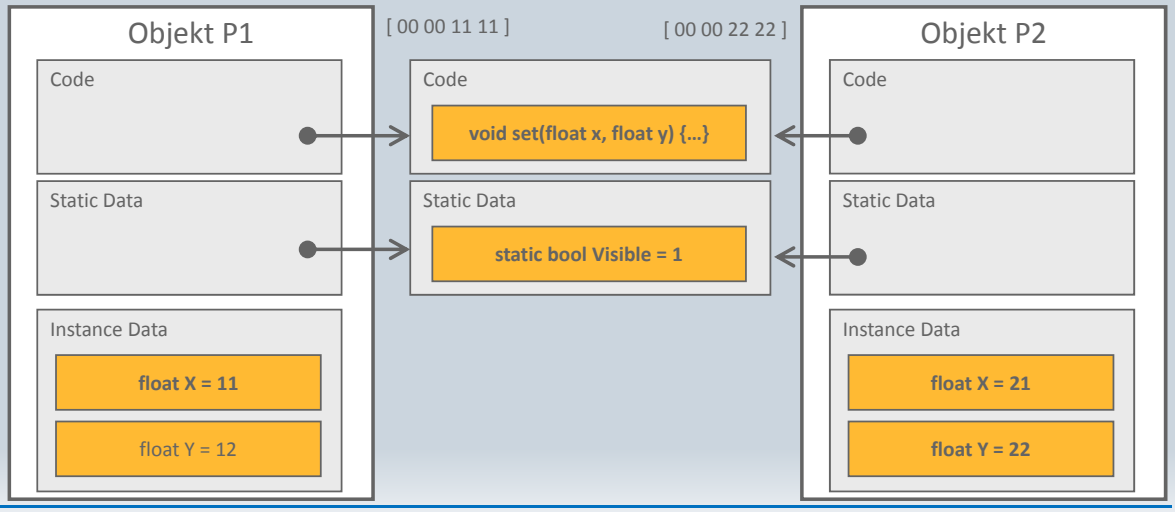
- **Beispiel - Zugriff auf einen Static Daten-Member:**

```
cout << "Visibility = " << CPoint::Visible; // Klassen Scope
```

zugriff auch via objet oder pointer auf's objekt mÖglich.

static Daten-Member III

- **Objekt Memory-Layout (schematisch)**



static Member-Funktionen

- Eine static Member-Funktion ist eine in der Klasse deklarierte Funktion die unabhängig von einem Objekt verwendet werden kann. (z.Bsp. für eine mathematische Utility Klasse).
- Eine static Member-Funktion besitzt somit keinen this-Pointer, weshalb innerhalb einer solchen Funktion nicht auf non-static Members zugegriffen werden kann, sondern nur auf mit static deklarierte Daten und Funktionen.
- Eine static Member-Funktion darf weder virtuell deklariert, noch darf der selbe Name für ein static und ein non-static Klassen-Member verwendet werden.
- Demo-Beispiel - Static Data-Member als Objektzähler

02.04.7 Überladen von Funktions-Member

- Analog wie normale Funktionen können auch Konstruktoren und andere Funktions-Member einer Klasse überladen werden (statische oder signaturgebundene Polymorphie).
- Für überladene Member-Funktionen gelten die gleichen Regeln bezüglich der Eindeutigkeit der Signatur wie für normale Funktionen.
- Jede Deklaration einer überladenen Member-Funktion wird als eigenständige Funktion mit Klassen-Scope implementiert.
- Die Vererbung von überladenen Member-Funktionen ist ebenfalls möglich.

02.04.8 Operatoren für Klassen

- **Neben Funktionen können für Klassen auch Operatoren deklariert und überladen werden (Operator Overloading).**
- **Eine Deklaration des "Funktionsnamens" für Operatoren besteht aus dem Keyword `operator` und einem entsprechenden Operator-zeichen.**
  - Beispiel: `bool CLine::operator>=(const CLine &aLine);`
- **Das verwendete Operatorzeichen kann beliebig ausgewählt werden, sollte aber möglichst dem üblichen Gebrauch entsprechen.**

Operatoren für Klassen II

- **Für die Definition von Operatoren gelten folgende Regeln:**
  - Es ist nicht möglich neue Operatorzeichen zu definieren oder bestehende Operatorzeichen zu erweitern (z.B. ist ** für die Potenzierung ist nicht zulässig).
  - Die Priorität für die Auflösung von Ausdrücken ist ebenfalls festgelegt und kann nicht verändert werden (Sprachstandard).
  - Die für vordefinierte Datentypen verwendeten Operatoren werden nicht automatisch auf abstrakte Datentypen (Klassen) übertragen.
  - Die Operatoren für vordefinierte Datentypen sind festgelegt und können weder umdefiniert noch erweitert werden.
  - Die Operatoren ***new*** und ***delete*** können ebenfalls überladen werden.
  - Die folgenden Operatoren können nicht überladen werden:
    - Operator sizeof() (Grösse)
    - Operator . (Elementzugriff)
    - Operator :: (Scope-Operator)
    - Operator ? (Konditional-Operator)

P3 = P1 + P2;
1. plus operator
2. erstellt temp objekt,
3. copy construktor
4. destruktor temp objekt
5. = operator
6. lÖschen des temp objekts

Operatoren für Klassen III

- **Beispiel - Operator+ Deklaration:**

```
CLine operator+(CLine& addLine);
```

- **Beispiel - Operator+ Implementation:**

&CLine nicht sinnvoll, da CLine Addresse nach return out of scope ist. (wird auf dem stack gelegt)

```
CLine CLine::operator+(CLine& addLine)
{
   CLine tmpLine;      // Definition temp. lokales Objekt

   // Vektor-Addition der temp. Linien Komponenten
   tmpLine.P1 = P1;
   tmpLine.P2 = P2 + addLine.P2 - addLine.P1;
   return tmpLine;          // impliziter Aufruf Copy-Konstruktor
}
```

danach noch impliziter Destruktor aufruf

- **Beispiel - Operator+ Aufruf:**

```
L3 = L1 + L2 ;             // normale Schreibweise
L3 = L1.operator+(L2);     // explizite Schreibweise
```


02.05 Default Arguments

- **Die Spezifikation von Default-Argumenten in C++ erlaubt die optionale Angabe von Parametern beim Aufruf einer Funktion.**
  - Beispiel: `void ShowMessage(char* text, int pos_x=0, int pos_y=0);`
- **Einschränkung: Hat ein Argument einen Default-Wert, müssen alle nachfolgend aufgeführten Argumente ebenfalls mit Default-Wert deklariert werden.**
- **Wirkungsvoller Mechanismus für die Implementation von nachträglichen Code-Erweiterungen oder Anpassungen.**
- **Durch die Verwendung überladener Funktionen und/oder durch implizite Typen-konversion können Mehrdeutigkeiten entstehen (Compile- oder Runtime-Fehler)!**

Default Arguments II

- **Beispiel:**

```
void ShowNum(double d, int prec=6, bool fix=true) {
   cout.precision(prec);
   if (fix) {
      cout << fixed;
   } else {
      cout << scientific;
   }
   cout << "Value = " << d << endl;
}

...

// Verwendung
double num = 12345.123456789012345;
...
ShowNum(num);                 // Value = 12345.123457
ShowNum(num, 2);              // Value = 12345.12
ShowNum(num, 10, false);      // Value = 1.2345123457e+004
ShowNum(num, false);          // Value = 12345 (!!!)
```


02.06 friend Funktionen

- Eine friend Deklaration erlaubt Non-Member Funktionen den Zugriff auf geschützte (protected, private) Daten-Member einer Klasse.
- Eine Friend-Deklaration wird durch Zugriffsspezifizierer (public, protected, private) nicht beeinflusst und kann an beliebiger Stelle innerhalb der Klassendeklaration erfolgen.
- Als Friend-Funktionen können sowohl globale Funktionen als auch Member-Funktionen einer anderen Klasse verwendet werden.
- Ist eine Friend-Funktion kein Klassen-Member, ist der implizite this-Pointer undefiniert und kann nicht benutzt werden. (Objekte müssen dann als Argumente übergeben werden.)
- Friend Beziehungen werden weder vererbt noch sind sie transitiv.


```
A friend B
B friend C
a is not fried C
```

friend Funktionen II

- Friend-Funktionen widersprechen dem Kapselungsprinzip und werden sinnvollerweise nur für globale Operatoren eingesetzt.
- Beispiel - Deklaration von Friend-Funktionen (Reihenfolge!):

```
class CPoint; // forward declaration
CLine {
        CPoint* pP1; CPoint* pP2;
public:
        void CLine::list(void)
         { cout << "p= " << pP1->X << ", " << pP1->Y << ... ;};
};

class CPoint {
        friend void CLine::list(void);
private:
        float X; float Y;
public:
        CPoint(void);
        ...
};
```

klasse legt selbst ob, sie friends hat oder nicht

friend Klassen

- Neben einzelnen Funktionen können auch ganze Klassen als `friend` deklariert werden.
- Beispiel - Deklaration von CLine als friend-Klasse:

```
class CPoint {
        friend class CLine;
private:
        float X;
        float Y;
public:
        CPoint(void);
        ...
};
```

02.07 inline Funktionen

- Die Definition einer `inline` Funktion ist ein Hinweis für den Compiler keinen Funktionsaufruf durchzuführen, sondern direkt den Code der Funktion an den entsprechenden Stellen im Programm einzusetzen.

- Beispiel – Klassendeklaration mit Impliziter inline-Definition:

```
class CPoint{
private:
   float X, Y;   // X und Y Koordinate des Punktes
public:
   // Implizite inline Implementation in der Klassendeklaration
   void set(float x, float y) {X = x; Y = y;}
};
```

- Beispiel - Explizite inline-Definition einer Funktion:

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

- Alle `inline` Funktionen müssen im Header-File implementiert sein! egal ob explizit oder implizit

CPP-02 61

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

02.08 const Funktionen

- Member-Funktionen welche die Daten-Member eines Objektes nicht verändern, können als `const` deklariert werden.

- Der `const` Spezifizierer am Schluss der Funktionsdeklaration bildet einen Teil der Signatur einer Member-Funktion und wird deshalb beim Überladen einer Funktion ebenfalls berücksichtigt.

- Innerhalb einer `const` Member-Funktion können nur Member-Funktionen verwendet werden, die ebenfalls mit `const` deklariert sind.

- Auf ein mit `const` definiertes Objekt kann nur mit `const` Member-Funktionen zugegriffen werden (selbst wenn der Zugriff nur lesend erfolgt).

CPP-02 62

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

beispiel const CLine cLine;
cLine.list();
cLine.set(a,b,c,d) <-- compile error, da mehtode nicht als const implementiert

const Funktionen II

- **Beispiel:**

```
// Deklaration einer const Member-Funktion
class CLine {
public:
        void list(void) const;
        int getLength(void);
        ...
};

// Implementation einer const Member-Funktion
void CLine::list( void ) const {
        cout << "P1 x:" << P1.x << " y:" << P1.y << endl;
        ...
}

// Verwendung einer const Member-Funktion
const CLine L1(1,1,5,5); // const Objektdefinition
L1.list();                    // Ok, const Funktionsaufruf
int i = L1.getLength();       // Compiler Error: non-const Function
```


02.09 Effective C++

- C++ wird im Vergleich zu C oft zu unrecht als "langsam" bezeichnet, weil Code ineffizient implementiert wird. Zu beachten sind deshalb die folgenden Regeln:
- Weil Konstruktoren, Copy-Konstruktoren und Zuweisungs-Operatoren sehr oft aufgerufen werden, sollten diese möglichst effizient implementiert werden.
- Für einen Funktionsaufruf mit Objekt-Parametern sollte nach Möglichkeit ein "Call By Reference" Aufruf verwendet werden.
- Die Verwendung einer Initialisiererliste für Objekt-Member ist effizienter als eine Zuweisung im Konstruktor.