Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 1

1. Ergänzungen zu Klassen und Objekten

1.1. Pointer auf Klassenkomponenten

1.2. Unions als Klassen

1.3. Innere und lokale Klassen

Pointer auf Klassenkomponenten in C++ (1)

- **Komponentenzeiger**
  - ◇ C++ kennt auch den abgeleiteten Typ "**Pointer auf Klassenkomponenten eines bestimmten Typs**" (→ **Komponentenzeiger,** *pointer to class ,member*).
  - ◇ Komponentenzeiger sind an eine **Klasse** und einen **Komponententyp** gebunden. Sie sind sowohl für **Datenkomponenten** als auch für **Funktionskomponenten** (Member-Funktionen) möglich.
  - ◇ **Werte** eines Komponentenzeiger-Typs identifizieren Komponenten eines bestimmten Typs innerhalb eines Objekts einer bestimmten Klasse. Es handelt sich bei diesen "Adressen" **nicht** um normale **Speicher-Adressen**, sondern um eine **Art Index** in eine – gedachte – Tabelle der Komponenten. **Zeiger auf Datenkomponenten** können als **Offset** innerhalb eines Objekts der entsprechenden Klasse aufgefasst werden.
  - ◇ Komponentenzeiger können **nicht** auf **statische Komponenten** zeigen.

- **Vereinbarung von Komponentenzeiger-Typen und -Variablen**
  - ◇ Zusätzliche Angabe des **Scope Resolution Operators** zusammen mit dem **Klassennamen** :
    - ▻ Typangabe für einen **Pointer auf Datenkomponenten** :

      `komponententyp klassenname::*`
    - ▻ Typangabe für einen **Pointer auf Funktionskomponenten** :

      `funktionstyp (klassenname::* )(parameterliste)`
  - ◇ **Beispiel :**

```
class Complex
{ public:
    Complex(float, float);
    Complex mal(const Complex& x);
    Complex plus(const Complex& x);
    float betrag(void);
    float re, im;
};
// ...

float Complex::*pfcomp;         // pfcomp ist Zeiger auf eine Komponente
                                // vom Typ float der Klasse Complex

Complex (Complex::*pcfunc)(const Complex&);
                              // pcfunc ist Zeiger auf eine Member-Funktion der
                              // Klasse Complex mit einer Complex-Referenz als
                              // Parameter und dem Funktionstyp Complex
```

  - ◇ Mittels **`typedef`** kann auch ein **Typname** für einen Komponentenzeiger-Typ vereinbart werden, der dann für Variablenvereinbarungen nutzbar ist.

    Beispiel :
```
typedef float (Complex::*PMFF)(void);
// PMFF ist ein Komponentenzeiger-Typ
```

- **Initialisierung von und Wertzuweisung an Komponentenzeigervariablen**
  - ◇ Es dürfen nur die "**Adressen**" **von Komponenten**, die vom **entsprechenden Typ** sind und auf die auch **Zugriff besteht**, zugewiesen werden (`public` bzw innerhalb Member- oder Freund-Funktionen)
    Auch hierbei ist dem **Komponentennamen** der **Scope Resolution Operator** zusammen mit dem **Klassennamen** voranzustellen.
  - ◇ **Beispiel :**
```
int main(void)
{ PMFF pffunc = &Complex::betrag;
  pfcomp = &Complex::re;
  pcfunc = &Complex::plus;
  //...
}
```

Pointer auf Klassenkomponenten in C++ (2)

- **Die Komponentenauswahloperatoren `.*` und `->*`**

  ◇ Nach Zuweisung einer Komponenten-"Adresse" an eine Komponentenzeigervariable kann über diese zu der **Komponente zugegriffen werden**.
  Dies ist aber nur in der **Bindung an ein konkretes Objekt** der Klasse möglich.
  Hierzu dienen die speziellen **Komponentenauswahl-Operatoren** `.*` und `->*`

  ◇ Beide Operatoren liegen in der **Priorität** zwischen den Multiplikations-Operatoren und den unären Operatoren.

  ◇ Der **Operator** `.*` wird auf **Objekte** angewendet, während der **Operator** `->*` auf **Objektzeiger** angewendet wird :

  ▻ Zugriff zu **Datenkomponenten** : **`objekt.*datenkomp_zeiger`**
  `objektzeiger->*datenkomp_zeiger`

  ▻ Zugriff zu **Funktionskomponenten**: **`(objekt.*funktionskomp_zeiger)(parameterliste)`**
  `(objektzeiger->*funktionskomp_zeiger)(parameterliste`)

  ◇ **Beispiel :**

```
class Complex;              // wie vorher definiert

typedef float Complex::*PDF;
typedef Complex (Complex::*PMFC)(const Complex&);

int main(void)
{
  Complex c1(1,1), c2(0,0);
  Complex *cptr=&c2;
  PDF pfcomp;               // Zeiger auf float-Komp der Klasse Complex
  PMFC pcfunc;              // Zeiger auf Funktions-Komp. von Complex

  pfcomp=&Complex::re;
  pcfunc=&Complex::plus;

  c1.*pfcomp=3.5;                 // c1.re=3.5;
  cptr->*pfcomp=7.3;              // cptr->re (=c2.re) =7.3;
  c1=(c1.*pcfunc)(c2);            // Klammerung notwendig, da () höhere
  c2=(cptr->*pcfunc)(c1);         // Priorität als .* bzw ->* hat
  // ...
}
```

- **Hinweis :**

  ◇ Der Zeigertyp "**Pointer auf Klassenkomponenten eines bestimmten Typs**" muß sorgfältig vom Zeigertyp "**Pointer auf bestimmten Typ**" unterschieden werden :

  ◇ So sind (**`float Complex::*`**) und (**`float *`**) **verschiedene** – weder implizit noch explizit ineinander konvertierbare – **Typen.**

  ◇ **Beispiel :**

```
void func(float Complex::*pfcomp);

void falsch(void)
{
  float a=5.8;
  func(&a);           // Fehler , falscher Parametertyp !
}
```

Pointer auf Klassenkomponenten in C++ (3)

- **Anmerkungen zu "Adressen" von Member-Funktionen**

  ◇ "**Adressen" von Funktionskomponenten** (Member-Funktionen) können nur über die **Bindung an die Klasse** (Scope Resolution Operator und Klassenangabe) ermittelt werden.
  Die Ermittlung der Adresse über die **Bindung** an ein **konkretes Objekt** der Klasse ist **nicht zulässig**.

  Beispiel :

```
int main(void)
{
  Complex c1(1.0,1.0);  // Klasse Complex wie vorher definiert
  float (Complex::*pffunc)(void);
  pffunc=&Complex::betrag;          // zulässig
  pffunc=&c1.betrag;                // unzulässig
  // ...
}
```

  ◇ Die **Zuweisung** der "**Adresse"** einer **nicht-statischen** Member-Funktion an eine "**einfache**" – nicht mit einer Klassenangabe versehenen – **Funktionspointervariable** ist **nicht zulässig**. Es ist **keine Speicheradresse** !

  Beispiel :

```
float (*pf)(void);
pf=&Complex::betrag;                // unzulässig
```

  ◇ **Statische** Member-Funktionen werden dagegen ähnlich wie freie – nicht an eine Klasse gebundene – Funktionen behandelt.
  Die **Adresse** einer derartigen Funktion darf einer "**einfachen**" **Funktionspointervariablen** zugewiesen werden.
  Es handelt sich um eine echte Speicheradresse.
  Die **Zuweisung** an eine entsprechende **Komponentenzeigervariable** ist dagegen **unzulässig**.
  Natürlich kann die Adresse **nur** über die **Bindung an die Klasse** ermittelt werden.
  (Gilt **analog** auch für **statische Datenkomponenten**.)

  Beispiel :

```
class Complex
{ public:
    // ...
    static float betrag(void);
    // ...
};

int main(void)
{
  float (*pf)(void);
  pf=&Complex::betrag;              // zulässig
  // ...
}
```

- **Anmerkungen zu Adressen von Datenkomponenten**

  ◇ Neben der Ermittlung der "Adresse" einer Datenkomponente über die Bindung an die Klasse, läßt sich auch die echte **Speicheradresse einer Datenkomponente** eines **konkreten Objekts** ermitteln. Eine derartige Adresse kann dann **nur** "**einfachen**" **Datenpointer-Variablen** zugewiesen werden.

  Beispiel :

```
int main(void)
{
  Complex c1(1.5,2.0);   // Klasse Complex wie vorher definiert
  float *pflt = &c1.re;  // zulässig
  // ...
}
```

Demonstrationsprogramm zu Pointer auf Klassenkomponenten in C++

```
/* ------------------------------------------------------------------ */
/* Programm kompptrdemo.   (C++-Quelldatei kpdemo_m.cpp)              */
/* ------------------------------------------------------------------ */
/* Demonstrationsprogramm zu Pointern auf Klassenkomponenten          */
/* ------------------------------------------------------------------ */

#include <iostream>

using namespace std;

class Demo
{
  public:
    void func1(void);
    void func2(void);
};

void Demo::func1(void)
{
  cout << "\nAufruf von func1()\n";
}

void Demo::func2(void)
{
  cout << "\nAufruf von func2()\n";
}

typedef void (Demo::*PDFUNC)(void);   // PDFUNC ist Komponentenzeigertyp

int main(void)
{
  Demo d;               // Objekt der Klasse Demo
  Demo *pd=&d;          // Pointer auf Objekt der Klasse Demo
  PDFUNC fptr;          // fptr ist Komponentenzeigervariable

  fptr=&Demo::func1;    // Zuweisung der Adresse einer Member-Funktion
  (d.*fptr)();          // Aufruf Member-Funktion für konkretes Objekt
  fptr=&Demo::func2;    // Zuweisung der Adresse einer anderen Member-Fkt.
  (pd->*fptr)();        // Aufruf Member-Funktion für konkretes Objekt
  return 0;
}
```

- **Ausgabe des Programms :**

```
Aufruf von func1()

Aufruf von func2()
```

Unions als Klassen in C++

- **Eigenschaften**
  - ◇ **Unions in C++** sind **abwärtskompatibel zu Unions in C**, d.h. eine gültige C-Union ist auch eine gültige C++-Union (aber nicht notwendigerweise umgekehrt).
  - ◇ **Unions in C++** sind **auch Klassen**.
    Ihre **Datenkomponenten** belegen alle den **gleichen Speicherplatz** (genauer: sie beginnen alle an der gleichen Speicheradresse).
    Neben Datenkomponenten können sie **auch Funktionskomponenten** (Member-Funktionen) - einschließlich Konstruktoren und Destruktor - besitzen.
  - ◇ **Defaultmäßig** sind alle Komponenten **`public`**.
    Einzelne oder alle Komponenten können **auch `private`** oder **`protected`** deklariert werden.
    Dies bedeutet, daß es möglich ist, den Speicherplatz für die Datenkomponenten über eine öffentliche Komponente von außen anzusprechen, während der gleiche Speicherplatz über eine private Komponente nicht von außen zugänglich ist.

- **Initialisierung von Union-Objekten :**
  - ◇ **alle** Datenkomponenten sind **`public:`** wie in C (nur erste Komponente) oder über **geeignete Konstruktoren**, die auch eine andere als die erste Komponente initialisieren können
  - ◇ **wenigstens eine `private`** Datenkomponente : **nur** über **geeignete Konstruktoren**

- **Beispiel :**

```
#include <iostream>
#include <iomanip>
using namespace std;

union Bdouble
{   Bdouble(double);                    // Konstruktor           (public !)
    void showBytes(void);               // Ausgabe als Bytefolge (public !)
  private:
    double d;
    unsigned char c[sizeof(double)];
};

Bdouble::Bdouble(double w) { d=w; }     // Konstruktor

void Bdouble::showBytes(void)           // Ausgabe double-Wert als Byte-Folge
{ cout << endl << hex << setfill('0');
  for (int i=sizeof(double)-1; i>=0; i--)
    cout << setw(2) << (c[i]&0xff) << ' ';
  cout << endl << dec;
}

int main(void)
{ Bdouble dobj(1991.829);
  dobj.showBytes();
  return 0;
}
```

```
40 9f 1f 50 e5 60 41 89
```

- **Einschränkungen für Unions :**
  - ◇ Unions können **nicht** von einer anderen Klasse a**bgeleitet** sein.
  - ◇ Unions können **keine Basisklasse** sein.
  - ◇ Eine Union darf **keine Komponente** besitzen, die einen **Konstruktor** oder **Destruktor** oder **selbstdefinierten Zuweisungs-Operator** hat.
  - ◇ Eine Union darf **keine statischen Datenkomponenten** besitzen.
  - ◇ Eine Union darf **keine virtuellen Member-Funktionen** haben.
  - ◇ **Anonyme** Unions dürfen **keine Funktionskomponenten** und **nur `public`**–Datenkomponenten aufweisen.

Innere Klassen in C++ (1)

- **Definition geschachtelter Klassen**
  - ◇ **Klassendefinitionen** können **geschachtelt** werden → Definition einer Klasse innerhalb einer anderen Klasse.
  - ◇ Eine innerhalb einer anderen Klasse definierte Klasse heißt **innere Klasse** oder eingebettete Klasse (***nested class***). Eine Klasse, innerhalb der eine andere Klasse definiert ist, wird **äußere** oder **umschließende Klasse** (***enclosing class***) genannt.
  - ◇ **Beispiel :** Rückwärts verkettete Liste, Definition des Typs der Listenelemente als innere Klasse

```
class Set                                          // äußere Klasse
{
  public :
    Set() { last=NULL; }
    void insert(int val) { last = new SetMember(val, last); }

    // ...

  private :

    class SetMember                                // innere Klasse
    {
      public :
        SetMember(int val, SetMember* n)
        { memVal=val;
          next=n;
        }
      private :
        int memVal;
        SetMember* next;
    };

    SetMember* last;
};
```

  - ◇ Die Definition der inneren Klasse kann im **`public`-Teil oder** im **`private`-Teil** der äußeren Klasse erfolgen. Im **`private`-Teil** definierte innere Klassen können **nur innerhalb** der **äußeren Klasse** (sowie von Freunden) einschließlich weiterer innerer Klassen verwendet werden, im `public`-Teil definierte Klassen können dagegen auch außerhalb benutzt werden.
  - ◇ Der Name der inneren Klasse ist **lokal** zur äußeren Klasse.
    Die **innere** Klasse befindet sich damit **im Sichtbarkeitsbereich** (Verfügbarkeitsbereich, ***scope***) der **äußeren** Klasse.
    → eine **Verwendung** des Klassennamens **außerhalb der umschließenden** Klasse erfordert die **Qualifizierung** mit deren Namen.

```
class Outer
{
  public :
    class Inner
    {
       // ...
    };
  // ...
};

void func(void)                // Verwendung der inneren Klasse von außerhalb
{
  Inner yourObj;               // fehlerhaft : Name Inner ist nicht sichtbar
  Outer::Inner myObj;          // ok
  // ...
}
```

Innere Klassen in C++ (2)

- **Beziehungen zwischen innerer und äußerer Klasse**

  ◇ Die Beziehungen zwischen innerer und äußerer Klasse **entsprechen** denen zwischen **separaten Klassen** mit der **Besonderheit**, dass sich die **innere** Klasse im **Sichtbarkeitsbereich** (*scope*) der **äußeren** Klasse befindet.
  Das bedeutet, dass die **innere Klasse** die in der **äußeren Klasse** definierten **Namen direkt verwenden** kann, der **gegenseitige Komponentenzugriff** sich aber nach den **üblichen Zugriffsrechteregeln** richtet.

  ⇨ ▻ Es existiert **keine** gegenseitige **`friend`-Eigenschaft** zwischen äußerer und innerer Klasse.
  ▻ **Memberfunktionen** der **äußeren** Klasse haben **keine besonderen Zugriffsrechte** zu den Komponenten der **inneren** Klasse **und umgekehrt**
  ▻ **Freunde** der einen Klasse sind **nicht** automatisch Freunde der anderen Klasse

  ◇ In Memberfunktionen und sonstigen **Vereinbarungen** der **inneren Klasse** dürfen unter Beachtung der Zugriffsrechte ohne konkreten Objektbezug nur **Typnamen**, **statische Komponenten** (Daten und Funktionen) sowie **Aufzählungskonstante** der **äußeren** Klasse verwendet werden.
  Eine Verwendung **nichtstatischer** Daten- oder Funktionskomponenten erfordert einen **konkreten Objektbezug** (Objekt, Referenz oder Pointer auf Objekt)

  ◇ Entsprechendes gilt für **Vereinbarungen** in der **äußeren** Klasse. Dabei müssen Typnamen, sowie die Namen von statischen Komponenten und Aufzählungskonstanten aus der **inneren Klasse** mit deren Namen **qualifiziert** werden.

  ◇ **Beispiel :**

```
class Outer
{
    enum State                                  // private !
    { READY, RUNNING, WAITING, STOPPED };

  public :
    int ioPub;
    static int ioStatPub;

    class Inner
    { public :
        int iiPub;
        static int iiStatPub;
        State st;              // ok : privater Typname kann benutzt werden,
                               //      wenn Name vorher vereinbart ist
        void doSomeIn(int i, Outer* op)
        { st=READY;     // Fehler : private Aufzaehlungskonstante von Outer
          ioPub=i;      // Fehler : nichtstatische Komponente von Outer
          ioStatPub=i;  // ok     : statische public-Komponente von Outer
          ioPriv=i;     // Fehler : private-Komponente von Outer u. nicht statisch
          ioStatPriv=i; // Fehler : private-Komponente von Outer
          op->ioPub=i;  // ok     : Zugriff zu public-Komp über Objekt-Pointer
          op->ioPriv=i; // Fehler : Zugriff zu private-Komponente
        }

      private :
        int iiPriv;
        static int iiStatPriv;
    };

    void doSomeOut(int i, Inner* ip)
    { iiStatPub=i;         // Fehler : Qualifikation mit Klassenname fehlt
      Inner::iiStatPub=i;  // ok     : statische Komponente von Inner
      Inner::iiPub=i;      // Fehler : nichtstatische Komponente von Inner
      Inner::iiStatPriv=i; // Fehler : private Komponente von Inner
      ip->iiPub=i;         // ok     : Zugriff zu public-Komp über Objekt-Pointer
      ip->iiPriv=i;        // Fehler : Zugriff zu private-Komponente
    }

  private :
    int ioPriv;
    static int ioStatPriv;
};
```

Innere Klassen in C++ (3)

- **Komponentendefinition außerhalb der Klassendefinition**

  ◇ **Memberfunktionen** innerer Klassen können auch **außerhalb der Klassendefinition** definiert werden.
  Statische Datenkomponenten müssen sogar außerhalb der Klassendefinition definiert werden.
  In beiden Fällen muß der **vollqualifizierte** Komponentenname (doppelte Qualifikation !) angegeben werden.

```
class Outer
{
  public :
    class Inner
    {
      public :
        void doSomeIn(int i, Outer* op);
        // ...
      private :
        static int iiStatPriv;
        // ...
    };
  //...
};

void Outer::Inner::doSomeIn(int i, Outer* op)
{ // ... }

int Outer::Inner::iiStatPriv=1;
```

  ◇ Dabei ist zu berücksichtigen, dass der **Sichtbarkeitsbereich** (*scope*) einer **Klasse** mit ihrem Scope-**Qualifier beginnt** (für die innere Klasse oben : mit `Outer::Inner::`) und sich **bis zum Ende** der jeweiligen Komponenten-**Definition erstreckt**.
  Das hat z.B. Konsequenzen für die Angabe des Rückgabetyps von Memberfunktionen, wenn dies ein innerhalb der äußeren Klasse definierter Typ ist.

```
class Outer
{
  public :
    class In1
    {
      // ...
    };

    class In2
    {
      public :
        In1 func(In2);
        // ...
    };
  // ...
};


In1 Outer::In2::func(In2 x)     // fehlerhaft : Qualifikation für In1 fehlt
{ // ...
}

Outer::In1 Outer::In2::func(In2 x)   // ok !
{ // ...
  return *(new In1);                 // ok : innerhalb der Funktionsdefini-
                                     //      tion ist Qualifikation für In1
}                                    //      nicht notwendig
```

Innere Klassen in C++ (4)

- **Vorwärtsdeklaration von inneren Klassen**
  - ◇ Eine **innere Klasse** kann innerhalb der Definition einer äußeren Klasse auch zunächst nur **deklariert** werden.
    → **Vorwärtsdeklaration** !
  - ◇ Die **Definition** der **inneren** Klasse muß dann entweder **innerhalb** der **äußeren** Klasse **später** oder **außerhalb** deren Klassendefinition erfolgen.
  - ◇ Bei der zweiten Möglichkeit kann die **Definition** der **inneren Klasse** nach **außen** (gegenüber dem Anwender) **verborgen** werden, da diese in einer eigenen Headerdatei, die dem Anwender nicht zugänglich gemacht wird, enthalten sein kann.
  - ◇ **Beispiel :**

```
class OuterFwd
{
  class InnerFwd1;            // Vorwärtsdeklaration
  class InnerFwd2;            // Vorwärtsdeklaration
  // ...
  class InnerFwd1             // spätere Definition
  {
   // ...
  };
  // ...
};


class OuterFwd::InnerFwd2     // Definition in eigener Headerdatei
{
  // ...
};
```

- **Gründe für die Schachtelung von Klassen**
  - ◇ Wenn eine Klasse **nur innerhalb einer anderen Klasse benötigt** wird, kann sie als **innere Klasse** der anderen Klasse definiert werden. Sie tritt **nach außen** überhaupt **nicht in Erscheinung**.
    Dadurch wird die Anzahl der globalen Namen vermindert.
    Sinnvollerweise sollte die Definition der inneren Klasse im **`private`**-Teil der äußeren Klasse erfolgen
    Beispiel : Referenzzählung
  - ◇ Wenn eine Klasse **funktionell zu einer anderen Klasse** gehört, nur im Zusammenhang mit dieser Klasse sinnvoll eingesetzt werden kann, aber durchaus **außerhalb** dieser Klasse **verwendet** wird, sollte sie als **innere Klasse** im **`public`**-Teil der anderen Klasse definiert werden.
    Beispiel : Iteratoren

Lokale Klassen in C++

- **Definition lokaler Klassen**
  - ◇ Eine Klasse kann auch **innerhalb** einer **Funktion** definiert werden → **lokale Klasse**.
  - ◇ Die Funktion, die die Definition einer lokalen Klasse enthält, wird als **umschließende Funktion** (*enclosing function*) bezeichnet.
  - ◇ **Memberfunktionen** einer lokalen Klasse müssen **innerhalb** der **Klassendefinition definiert** werden.
  - ◇ Eine lokale Klasse darf **weder statische Datenkomponenten noch statische Memberfunktionen** enthalten.
  - ◇ **Beispiel :**

```
#include <iostream>

using namespace std;


int iext=2;


void func(int i)
{
  static int is=i;
  int ia=i;

  class LocClass
  { public :
      LocClass(int i=0)  { setVal(i);}
      void setVal(int i) { iVal=(i==0?iext:i); }
      int getVal()       { return iVal; }
    private :
      int iVal;
  };

  LocClass myLoc1(ia);
  cout << "\nvalue : " << myLoc1.getVal() << endl;
  LocClass myLoc2(is);
  cout << "\nvalue : " << myLoc2.getVal() << endl;
  LocClass myLoc3(5);
  cout << "\nvalue : " << myLoc3.getVal() << endl;
  LocClass myLoc4;
  cout << "\nvalue : " << myLoc4.getVal() << endl;
}
```

- **Verwendungsbeschränkungen**
  - ◇ Der Name einer lokalen Klasse ist nur innerhalb der Funktion, in der die Klasse definiert ist, bekannt → die **Klasse** kann **nur innerhalb** dieser **Funktion verwendet** werden.
  - ◇ Die Funktion, in der die lokale Klasse definiert ist, hat **keine besondere Zugriffsberechtigung zu den Komponenten** der lokalen Klasse. Es gelten die üblichen Zugriffsrechte.
  - ◇ Innerhalb einer lokalen Klasse kann **nur zu globalen** und zu **statisch-lokalen Variablen** der umschließenden Funktion zugegriffen werden. Ein Zugriff zu lokalen `auto`-Variablen der umschließenden Funktion ist nicht möglich.
    Weiterhin können alle übrigen globalen und lokalen Namen (Funktionen, Typen, Aufzählungskonstante) der umschließenden Funktion (sofern vorher vereinbart) innerhalb der lokalen Klasse verwendet werden.
  - ◇ **Anmerkung zu Visual-C++** (6.0) : Abweichend von ANSI-C++ ist ein Zugriff zu statisch-lokalen Variablen der umschließenden Funktion nicht zulässig.