Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 4

4. Ergänzungen zur Laufzeitpolymorphie

4.1. Abstrakte Klassen

4.2. Laufzeittypinformation

4.3. Typkonvertierungsoperator `dynamic_cast`

Abstrakte Klassen in C++

- **Eigenschaften**

  ◇ Eine Klasse, für die wenigstens eine **rein-virtuelle Funktion** deklariert ist, wird als **abstrakte Klasse** bezeichnet.

  ◇ Eine **rein-virtuelle Funktion** besitzt **keine Definition** in der Basisklasse. Für sie ist **lediglich** das **Interface** (Parameter und Funktionstyp) jedoch keine konkrete Implementierung festgelegt.
  Eine **abstrakte Klasse** ist damit **unvollständig** definiert. Von ihr lassen sich **keine konkreten Objekte anlegen**.
  Sie kann **nur als Basisklasse** zur Ableitung anderer Klassen verwendet werden.

  ◇ Jede von einer abstrakten Klasse abgeleitete Klasse, von der konkrete Objekte erzeugt werden sollen, muß für sämtliche geerbten rein-virtuellen Funktionen Definitionen enthalten (→ **konkrete Klasse**).
  Eine rein-virtuelle Funktion, die in einer abgeleiteten Klasse nicht definiert wird, bleibt rein-virtuell für diese Klasse.
  Die abgeleitete Klasse ist damit ebenfalls abstrakt.

  ◇ Eine **abstrakte Klasse** darf **nicht als Parameter-Typ** und **nicht als Funktions-Rückgabe-Typ** verwendet werden und darf **nicht in** einer **expliziten Typ-Konvertierung** auftreten.

  ◇ **Pointer** und **Referenzen** auf **abstrakte Klassen** sind dagegen **zulässig**.

- **Anwendung :**

  Abstrakte Klassen dienen in einer Klassenhierarchie zur **Zusammenfassung gemeinsamer Eigenschaften** unterschiedlicher konkreter Objekte unter einem **Oberbegriff** und zur Bereitstellung eines **gemeinsamen Methoden-Interfaces** ohne Implementierungs-Details festzulegen.
  Die Implementierung der Methoden kann **geändert** oder **ergänzt** werden (neue abgeleitete Klassen), **ohne** daß dies **Auswirkungen auf die Schnittstelle** und damit auf die Anwendung der Methoden hat.

- **Beispiel :**

```
class Punkt { private: int x, y; /* ... */ };

class GeoObj                                          // abstrakte Klasse
{ public:
    virtual void move(const Punkt&) = 0;
    virtual void draw(void) = 0;
    // ...
  protected:
    GeoObj(const Punkt& p) : refpunkt(p) {}
    Punkt refpunkt;
};

class Linie : public GeoObj                           // konkrete Klasse
{ public:
    void move(const Punkt& p) { /* Verschiebe Linie ... */ }
    void draw(void)           { /* Zeichne Linie    ... */ }
    // ...
};

class Kreis : public GeoObj                           // konkrete Klasse
{ public:
    void move(const Punkt& p) { /* Verschiebe Kreis ... */ }
    void draw(void)           { /* Zeichne Kreis    ... */ }
    // ...
};
```

Demonstrationsprogramm zu abstrakten Klassen in C++

```
// ------------------------------------------------------------------------
// Programm ABCLBSP
// ------------------------------------------------------------------------
// Demonstrationsbeispiel zu abstrakten Klassen
// ------------------------------------------------------------------------

#include <iostream>
using namespace std;

class Num                                           // abstrakte Klasse
{ public:
    Num(int i=0) { wert=i;}
    virtual void showNum(void) = 0;
  protected:
    int wert;
};

class DecNum : public Num                           // konkrete Klasse
{ public:
    DecNum(int i=0) : Num(i) { }
    void showNum(void)
    { cout <<"wert dezimal   : " << dec << wert << '\n'; }
};

class HexNum : public Num                           // konkrete Klasse
{ public:
    HexNum(int i=0) : Num(i) { }
    void showNum(void)
    { cout <<"wert sedezimal : " << hex << wert << '\n'; }
};

class OctNum : public Num                           // konkrete Klasse
{ public:
    OctNum(int i=0) : Num(i) { }
    void showNum(void)
    { cout <<"wert oktal     : " << oct << wert << '\n'; }
};

void main(void)
{ Num *baseptr[3];
  DecNum wd(10);
  HexNum wh(511);
  OctNum wo(63);
  baseptr[0]=&wd;            // implizite Typwandlung DecNum* --> Num*
  baseptr[1]=&wh;            // implizite Typwandlung HexNum* --> Num*
  baseptr[2]=&wo;            // implizite Typwandlung OctNum* --> Num*
  cout << '\n';
  for (int i=0; i<3; i++)
    baseptr[i]->showNum();
}
```

- **Ausgabe des Programms :**

```
wert dezimal   : 10
wert sedezimal : 1ff
wert oktal     : 77
```

Laufzeit-Typinformation in C++ (1)

• Einführung

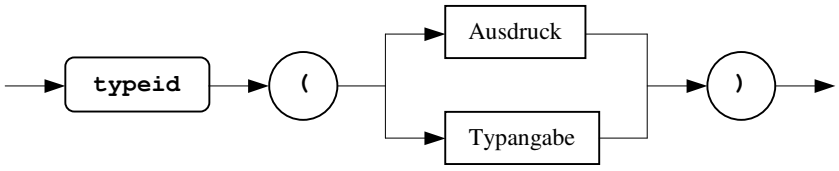
Ein **Zeiger** bzw eine **Referenz** auf ein Objekt einer **abgeleiteten Klasse** kann implizit oder explizit in einen Zeiger bzw eine Referenz auf ein Objekt einer - eindeutigen - **Basisklasse umgewandelt** werden.
Dadurch wird es möglich, mittels Basisklassen-Pointern bzw -Referenzen Objekte unterschiedlichen Typs - allerdings aus derselben Vererbungshierarchie - zu verwalten. Das bedeutet, daß der gleiche Basisklassen-Pointer (bzw -Referenz) dynamisch änderbar auf Objekte unterschiedlichen Typs zeigen kann.
Manchmal ist es wünschenswert, während der Laufzeit den **tatsächlichen Typ** des Objekts, auf das ein Basisklassen-Pointer bzw -Referenz zeigt, zu ermitteln.
→ **Laufzeit-Typinformation** (*Runtime Type Information* = **RTTI**)

• `typeid`-Operator

◇ **unärer** Operator

◇ ermöglicht die **Ermittlung von Typinformationen während der Laufzeit**

◇ **Syntax :**



◇ **Beispiele :**
```
typeid(*baseptr[1])
typeid(int)
```

◇ Der **Wert** eines **`typeid`-Ausdrucks** ist vom **Typ `const type_info&`**

→ Ein `typeid`-Ausdruck liefert als Ergebnis eine Referenz auf ein Objekt der Klasse `type_info`, durch das der Typ des Operanden-Ausdrucks bzw der Operanden-Typangabe repräsentiert wird.

Die Klasse **`type_info`** ist Bestandteil der ANSI/ISO-C++-Standardbibliothek
. Sie ist in der Header-Datei **`<typeinfo>`** (bzw `<typeinfo.h>`) definiert.

Der **Compiler** legt **für jeden Datentyp ein Objekt dieser Klasse** an.

Wenn der **Operanden-Ausdruck** eine **Referenz** oder ein **dereferenzierter Pointer != NULL** auf Objekte einer **polymorphen Klasse** ist, ist das **Ergebnis** eine **Referenz** auf das **`type_info`-Objekt**, das den **tatsächlichen Typ** des aktuell referierten vollständigen Objekts referiert.
→ Ermittlung des **dynamischen** (zur Laufzeit änderbaren) **Typs**

Ist der **Operanden-Ausdruck** ein **dereferenzierter NULL-Pointer** auf Objekte einer polymorphen Klasse wird die **Exception `bad_typeid`** geworfen.

Für **jeden anderen Typ des Operanden-Ausdrucks** (sowie für eine **Typangabe als Operand**) ist das Ergebnis eine **Referenz** auf das **`type_info`-Objekt**, das diesen **(statischen) Typ** repräsentiert.

⇒ eine (dynamische) **Laufzeit-Typinformation läßt sich nur für Objekte polymorpher Klassen** ermitteln.

• Hinweis :

In **Visual-C++** muß die Unterstützung der Laufzeit-Typinformation durch Setzen eines **Compiler-Schalters** explizit **aktiviert** werden.
Menu : **Projekt → Einstellungen** → Reiter : **C++** → Kategorie : **Programmiersprache C++**→ Checkbox : **RTTI aktiv**.

Laufzeit-Typinformation in C++ (2)

- **Beispiele für das Ergebnis eines `typeid`-Ausdrucks :**

```
#include <typeinfo>
using namespace std;

class Fahrzeug { /* ... */ };                    // nicht-polymorphe Klasse

class LandFahrz : public Fahrzeug                // polymorphe Klasse
{ public:
    virtual void fahren(float) { /* ... */ }
    // ...
};

class Auto : public LandFahrz { /* ... */ };

class Fahrrad : public LandFahrz { /* ... */ };

class Boot : public Fahrzeug  { /* ... */ };


void main(void)
{
  Fahrzeug *pclFahr;
  LandFahrz *pclLandF;
  LandFahrz clLaFz;
  Auto a;
  LandFahrz& rLaFz=a;

  pclLandF=new Fahrrad;
  typeid(*pclLandF);     // --> type_info-Objekt von Fahrrad   (dynamisch)
  pclLandF=&a;
  typeid(*pclLandF);     // --> type_info-Objekt von Auto      (dynamisch)
  typeid(rLaFz);         // --> type_info-Objekt von Auto      (dynamisch)

  pclFahr=&clLaFz;
  typeid(*pclFahr);      // --> type_info-Objekt von Fahrzeug  (statisch)
  pclFahr=new Boot;
  typeid(*pclFahr);      // --> type_info-Objekt von Fahrzeug  (statisch)
  typeid(clLaFz);        // --> type_info-Objekt von LandFahrz (statisch)
  typeid(LandFahrz);     // --> type_info-Objekt von LandFahrz (statisch)
}
```

- **Realisierung der Ermittlung der (dynamischen) Laufzeit-Typinformation**

Für **jede polymorphe Klasse** wird ein **Pointer** auf das die Klasse repräsentierende **`type_info`-Objekt** als **zusätzlicher Eintrag** mit in die **virtuelle Methoden-Tabelle** (VMT) aufgenommen.
Damit ist dieses Objekt über den **VMT-Pointer** erreichbar.
Da eine VMT nur für polymorphe Klassen existiert, ist die Ermittlung der (dynamischen) Laufzeit-Typinformation auf Objekte derartiger Klassen beschränkt.

Laufzeit-Typinformation in C++ (3)

- **Die Klasse `type_info`**

  ◇ Objekte dieser Klasse **repräsentieren Typen**

  ◇ Die Klasse ist in der C++-Header-Datei **`<typeinfo>`** (bzw `<typeinfo.h>`) definiert.

  ◇ Für **jeden Datentyp** legt der Compiler **ein Objekt** dieser Klasse an.
  Dieses enthält **implementierungsabhängige Datenkomponenten** zur Speicherung des **Typnamens** sowie eines **codierten Wertes**, der es gestattet, Typen in eine **Sortierreihenfolge** anzuordnen sowie zwei Typen auf **Gleichheit** zu überprüfen.
  Die Klasse überlädt die **Operatoren `==`** und **`!=`** und definiert eine **Memberfunktion zur Ermittlung des Typnamens** sowie eine **Memberfunktion zum Vergleich des "Reihenfolgekriteriums"** zweier Typen.

  ◇ In **ANSI/ISO-C++** ist folgende **prinzipielle Definition** der Klasse vorgesehen :

```
class type_info
{ public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
  private:
    // implementierungsabhängige Datenkomponenten zur
    // Speicherung des Typnamens und eines "Reihenfolgekriteriums"
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
};
```

  ◇ Da der **Copy-Konstruktor** und die **Zuweisungsoperator-Funktion** dieser Klasse **`private`** sind, lassen sich `type_info`-Objekte **nicht kopieren**.

- **Beispiele zur Anwendung der Memberfunktionen der Klasse `type_info`** :

```
#include <typeinfo>
using namespace std;

{ //  ...
  LandFahrz *apclFuhrpark[ANZ];
  int iAnzVelo=0;
  //  ...
  for (int i=0; i<ANZ; i++)
    if (typeid(*apclFuhrpark[i]) == typeid(Fahrrad))
      iAnzVelo++;
  //  ...
}

{ //  ...
  LandFahrz *pclFahr;
  //  ...
  cout << typeid(*pclFahr).name();
  //  ...
}
```

Der Typkonvertierungsoperator `dynamic_cast` in C++ (1)

- **`dynamic_cast`**

  ◇ Dieser Typkonvertierungsoperator ermöglicht **sichere Typkonvertierungen innerhalb von Klassenhierarchien**; insbesondere eine **sichere Rückwandlung** eines **Pointers** (Referenz) auf **Basisklasse** in **Pointer** (Referenz) auf **abgeleitete Klasse**.
  Eine eventuelle **`const`**-Eigenschaft läßt sich mit ihm **nicht entfernen**.

  ◇ Der Ausdruck **`dynamic_cast<T>(e)`**

  bewirkt die Konvertierung des Ausdrucks **`e`** in den Typ **`T`**.

  Der **Zieltyp `T`** muß ein **Pointer** oder eine **Referenz** auf eine vollständig definierte Klasse bzw der Typ **`void*`** sein. Entsprechend muß der **Quellausdruck `e`** ein **Pointer** auf ein Klassen-Objekt oder ein **Lvalue** eines Klassentyps sein.

  ◇ **Folgende Fälle** sind **zulässig :**

  a) Der **Quellausdruck `e`** ist ein Pointer auf ein Objekt bzw ein Lvalue einer - von einer Basisklasse `B` **abgeleiteten** - **Klasse `D`.** Dabei muß `B` eine zugreifbare (`public`) und eindeutige Basisklasse von `D` sein.
  `T` ist ein Pointer bzw eine Referenz auf diese **Klasse `B`.**
  In diesem Fall ist das **Ergebnis** ein Pointer bzw eine Referenz auf das im **`D`-Objekt enthaltene Teil-Objekt der Klasse `B`.**
  → dieser Fall **entspricht** der **impliziten Standardkonvertierung**

  Beispiel :

```
class B { /* ... */ };

class D : public B { /* ... */ };

// ...
D clD;
B* pclB1 = dynamic_cast<B*>(&clD);
B* pclB2 = &clD;                    // pclB1 == pclB2 !
```

  b) Der **Quellausdruck `e`** ist ein Pointer auf ein Objekt einer **polymorphen Klasse** und **`T`** ist der Typ **`void*`.**
  In diesem Fall ist das **Ergebnis** ein Pointer auf das **vollständige** durch **`e`** tatsächlich referierte Objekt.

  c) Der **Quellausdruck `e`** ist ein Pointer auf ein Objekt bzw ein Lvalue einer **polymorphen Klasse `B`** und **`T`** ist **nicht** der Typ **`void*`.**
  In diesem Fall wird mittels einer **Laufzeit-Typprüfung** des tatsächlich referierten Objekts **geprüft**, ob der Quellausdruck **`e`** in den Zieltyp **`T`** **umgewandelt** werden kann :

  ▻ Ist **`T`** ein Pointer bzw eine Referenz auf eine **von `B public` abgeleitete Klasse `D`** und wird **durch `e`** tatsächlich ein **Objekt dieser Klasse `D`** oder einer **von `D` abgeleiteten Klasse** referiert, so ist das **Ergebnis** ein Pointer bzw eine Referenz auf das **`D`-(Teil-)Objekt**.

  ▻ Ist **`T`** ein Pointer bzw eine Referenz auf eine **Klasse `A`,** die zugreifbare und eindeutige **Basisklasse** des **durch `e` tatsächlich referierten Objekts** ist, so ist das **Ergebnis** ein Pointer bzw eine Referenz auf das **`A`-Teil-Objekt** des von **`e`** referierten Objekts.

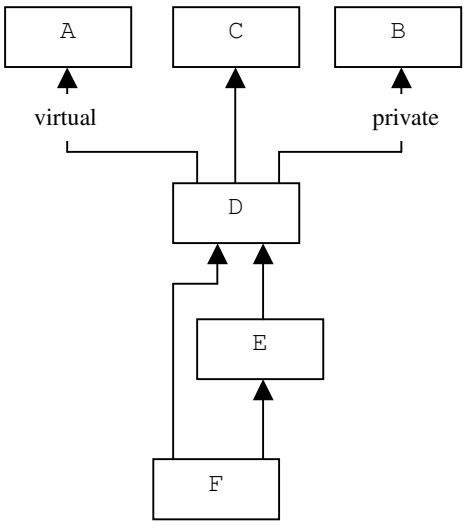
  ▻ In allen **übrigen Fällen** ist die **gewünschte Typumwandlung nicht möglich**.
  → Wenn **`T`** ein **Pointer-Typ** ist, wird als **Ergebnis** der **`NULL`-Pointer** erzeugt;
  wenn **`T`** eine **Referenz** ist, wird die **Exception `bad_cast`** geworfen.

Der Typkonvertierungsoperator `dynamic_cast` in C++ (2)

- **Beispiel für Typwandlungen bei polymorphen Klassen** :

```
class A { virtual void f(); /* ...*/ };

class B { virtual void g(); /* ...*/ };

class C { virtual void h(); /* ... */ };

class D : public virtual A, public C , private B { /* ... */ };

class E : public D { /* ... */ };

class F : public E, public D { /* ...*/ };    // in Visual-C++ 6.0 nicht zulaessig

void func(void)
{ A  a;
  D  d;
  E  e;
  F  f;
  A* ap = &a;
  B* bp = dynamic_cast<B*>(&d);    // D* → B*  Fehlschlag (private)
  D* dp = dynamic_cast<D*>(ap);    // A* → D*  Fehlschlag (ap zeigt auf A)
  C* cp = dynamic_cast<C*>(ap);    // A* → C*  Fehlschlag (ap zeigt auf A)
  ap = &d;                         // D* → A*  implizit
  dp = dynamic_cast<D*>(ap);       // A* → D*  hier o.k.(ap zeigt auf D)
  D& dr2 = dynamic_cast<D&>(*ap); // A  → D&  o.k. (ap zeigt auf D)
  cp = dynamic_cast<C*>(ap);       // A* → C*  o.k. (ap zeigt auf D, C ist auch
                                   //           Basisklasse von D)
  bp = dynamic_cast<B*>(ap);       // A* → B*  Fehlschlag (ap zeigt auf D,
  ap = &e;                         //           B ist private Basisklasse von D)
  dp = dynamic_cast<D*>(ap);       // A* → D*  o.k.(ap zeigt auf E,
                                   //           D ist public-Basisklasse von E)
  ap = &f;                         // o.k.,     nur ein A in F enthalten
                                   //           (virtuelle Basisklasse)
  dp = dynamic_cast<D*>(ap);       // A* → D*  Fehlschlag (mehrdeutig)
  E* ep1 = (E*)ap;                 // Fehler,   C-compatibler cast von
                                   //           virtueller Basis
  E* ep2 = dynamic_cast<E*>(ap);   // A* → E*  o.k.(ap zeigt auf F,
                                   //           E ist public-Basisklasse von F)
 }
```

Klassendiagramm



Objekt der Klasse `F`