

Programmieren in C++ (8): Laufzeit-Typidentifikationen RTTI

Silke Seehusen, Fachhochschule Lübeck

Die statische Typüberprüfung ist ein Paradigma der Programmiersprache C++. Um dennoch in eingeschränkten Fällen Typüberprüfungen zur Laufzeit zu ermöglichen, wurden die Laufzeit-Typidentifikationen eingeführt, die in diesem Artikel an einigen Beispielen vorgestellt werden.

Unterschiedliche Typen

Ein Grundgedanke bei der Entwicklung der Programmiersprache C++ war, eine möglichst strikte Typüberprüfung zur Übersetzungszeit vorzunehmen. Damit sollte die alte "Unsitte" der C-Programmierung durchbrochen werden, Typen in jeweils benötigte andere Typen zu konvertieren. Diese Unsitte führt oft zu sehr fehlerhafter, insbesondere wartungsunfreundlicher Software. Eine Typüberprüfung, die explizit in ein Programm "reinprogrammiert" werden muß, wird schon mal vergessen oder sogar prinzipiell weggelassen mit der Begründung "Das kann doch nie schiefgehen!".

Als Konsequenz wurde in C++ die strikte Typüberprüfung zur Übersetzungszeit und für kontrollierte Typumwandlungen die Zeiger- und Referenzkompatibilität von Basisklassen und abgeleiteten Klassen und virtuelle Funktionen eingeführt (siehe z.B. [1], [2]). Desweiteren wurden Schablonen (templates) insbesondere für Container-Klassen in den Sprachumfang aufgenommen. (siehe z.B. [3]).

Da es immer noch Fälle gibt, in denen eine Typidentifikation sinnvoll ist, die erst zur Laufzeit bestimmt werden kann, wird die **Laufzeit-Typidentifikation** (*runtime type identification*, RTTI) zur Verfügung gestellt, siehe Abbildung 1.

Operatoren zum Umgang mit Typen			
Operator	Rückgabewert	Argument	Beispiel
<code>typeid</code>	<code>const type_info&</code>	Objekt, Klasse oder Referenz	<code>typeid(a)</code>
<code>dynamic_cast<T></code>	T	Zeiger oder Referenz	<code>dynamic_cast<A*>(p_b)</code>

Abbildung 1: Operatoren zur Laufzeit-Typidentifikation

Typidentifikation

Von jedem Objekt kann zur Laufzeit die Typidentifikation ermittelt werden. Dazu gibt es den Operator *typeid*, der, angewendet auf ein Objekt oder eine Klasse, die Typidentifikation zurückgibt. Diese ist selbst vom Typ *type_info*. Typidentifikationen können miteinander verglichen werden, siehe Abbildung 2, und es gibt eine dem Typ zugeordnete Zeichenkette, die jedoch implementierungsabhängig ist. Sie kann sogar bei unterschiedlichen Programmaufrufen verschieden sein.

Klasse <code>type_info</code>			
Methode	Rückgabewert	Argumente	Beispiel
<code>name</code>	<code>const char*</code>		<code>cout << typeid(a).name()</code>
<code>==</code>	<code>bool</code>	<code>const type_info&</code>	<code>if (typeid(a) == typeid(b)) ...</code>
<code>!=</code>	<code>bool</code>	<code>const type_info&</code>	<code>if (typeid(a) != typeid(b)) ...</code>
<code>before</code>	<code>bool</code>	<code>const type_info&</code>	<code>if (typeid(a).before(typeid(b)) ...</code>

Abbildung 2: Methoden der Klasse `type_info`

Die folgenden Beispiele zeigen zunächst nur, daß die Typidentifikation unabhängig davon ist, ob sie von einer Klasse, einem Objekt oder einer Referenz auf ein Objekt bestimmt wird. Desweiteren spielt es keine Rolle, ob das Objekt konstant ist.

```
#include <typeinfo>
class A    {};
class B    { public: virtual
            void vf(){} };
// ...
assert (
    typeid(A) != typeid(B) &&
    typeid(A) == typeid(A&));
A  a, a2;
assert (
    typeid(a) == typeid(A) &&
    typeid(a) == typeid(a2));
A & r_a = a;
assert (
    typeid(r_a)==typeid(A));
const A c_a = a;
assert (
    typeid(c_a)==typeid(A));
}
```

Wird `typeid` in einem Programm benutzt, so muß vor der Benutzung `type_info` durch `#include <typeinfo>` bekannt gemacht werden.

Typidentifikation und Referenzen

Betrachten wir nun noch Referenzen: Hier ergibt sich die Frage, ob die Typidentifikation einer Referenz und des Ziels der Referenz immer gleich sind. Leider ist das nicht unbedingt der Fall — das hängt davon ab, ob es sich um polymorphe Klassen handelt oder nicht.

Eine **polymorphe Klasse** ist eine Klasse, die eine virtuelle Funktion deklariert oder erbt. Im obigen Beispiel ist `B` eine polymorphe Klasse. Der Operator `typeid` liefert, angewendet auf eine Referenz einer polymorphen Klasse, die Typidentifikation des vollständigen Objekts zurück, auf das die Referenz zeigt.

Ein Objekt wird **vollständig** (*complete*) genannt, wenn es kein Teilobjekt (*sub-object*) eines anderen Objekts ist. Ein Teilobjekt kann ein Elementobjekt oder ein Basisklassenteilobjekt sein.

```
class Bc: public B {};
Bc bc;
B & r_b = bc;
assert (
    typeid(r_b) == typeid(bc) );
```

Aber bei nichtpolymorphen Klassen:

```
class AC: public A {};
AC ac;
```

```
A & r_a = ac;
assert (
    typeid(r_a) != typeid(ac) );
```

Der Unterschied in der Behandlung ist nicht gerade übersichtlich. Er resultiert aus der Implementierung von abgeleiteten Klassen.

Typnamen

Von einer Typidentifikation kann mit der Methode *name()* ein Name erhalten werden, der selbst wieder implementierungsabhängig ist:

```
cout
<< typeid(B).name() << endl
<< typeid(a).name() << endl
<< typeid(ac).name() << endl
<< typeid(r_a).name() << endl;
```

Übersetzt auf einer NeXT mit einem GNU-C++-Übersetzer liefert das Beispiel die Ausgabe:

```
TD$1B
TD$1A
TD$Q29g__Fv.4022Ac
TD$1A
```

Auf einer Sun mit der gleichen Übersetzerversion gibt es immerhin ähnliche Namen, nur das Dollarzeichen ist durch einen Punkt ersetzt. Der dritte Typname ist wohl etwas länger, weil der Typ innerhalb einer Funktion vereinbart ist. Es gibt jedoch keine Vorschriften für den Namen.

Dynamischer Wandel mit *dynamic_cast*

Wird der Operator *dynamic_cast<T>* auf einen Ausdruck *v* angewendet, so liefert er als Ergebnis den nach *T* konvertierten Ausdruck zurück, wenn eine solche Konvertierung zulässig ist. *T* muß ein Zeiger oder eine Referenz eines vollständigen Objekts sein.

Ist eine Konvertierung nicht zulässig, wird ein Nullzeiger zurückgeliefert, wenn *T* ein Zeiger ist. Es wird die Ausnahme *bad_cast* gemeldet, wenn *T* eine Referenz ist.

Ausnahmen (exceptions)		
Ausnahme	wann gemeldet	Basisklasse
<i>bad_cast</i>	bei einem ungültigen Ausdruck von <i>dynamic_cast</i>	<i>exception</i>
<i>bad_typeid</i>	wenn ein Null-Zeiger in einem Ausdruck von <i>typeid</i> auftritt	<i>exception</i>

Abbildung 3: Ausnahmen, die von *typeid* oder *dynamic_cast* gemeldet werden können

Konvertierungen sind insbesondere von abgeleiteten Klassen zu Basisklassen zulässig.

Eine Konvertierung ist auch von einem Zeiger oder einer Referenz einer Basisklasse zu einer abgeleiteten Klasse zulässig, wenn der Zeiger oder die Referenz auf ein Objekt der abgeleiteten Klasse zeigt oder die abgeleitete Klasse eine Basisklasse des Objektes ist.

Mit *dynamic_cast* ist ein sogenanntes sicheres "down cast" und ein sicheres "up cast" möglich:

```

class C
{ public:
    virtual void p()
        {cout<<"C";}
};
class D : public C
{ public:
    void pp(){cout<<"PP";}
};
D d;
C* p_c = &d;
D* p_d =
    dynamic_cast<D*>(p_c);
if ( p_d )
    p_d->pp();

```

Beim sicheren "down cast" können die speziellen Eigenschaften einer Klasse ausgenutzt werden, auch wenn nur ein Zeiger oder eine Referenz auf eine Basisklasse z.B. als Parameter übergeben wird.

Ein sicheres "up cast" ist dann sinnvoll, wenn aus der Eigenschaft, daß eine Klasse eine andere bestimmte Klasse als Basisklasse besitzt, die Eigenschaften der Basisklasse ausgenutzt werden. Das ist insbesondere in größeren Vererbungshierarchien und bei mehrfacher Vererbung sinnvoll.

Beispiele

Bisher haben wir die Sprachkonstrukte pur und an Trivialbeispielen vorgestellt. Die Frage bei neuen Konstrukten ist immer: Sind sie wirklich praktisch sinnvoll einsetzbar?

Das folgende Beispiel ist in Zusammenhang mit der Implementierung einer kleinen Personalverwaltung nach dem Entwurfsmuster Model/View/Controller entstanden. Ein Teil der Klassenhierarchie ist in Abbildung 4 wiedergegeben.

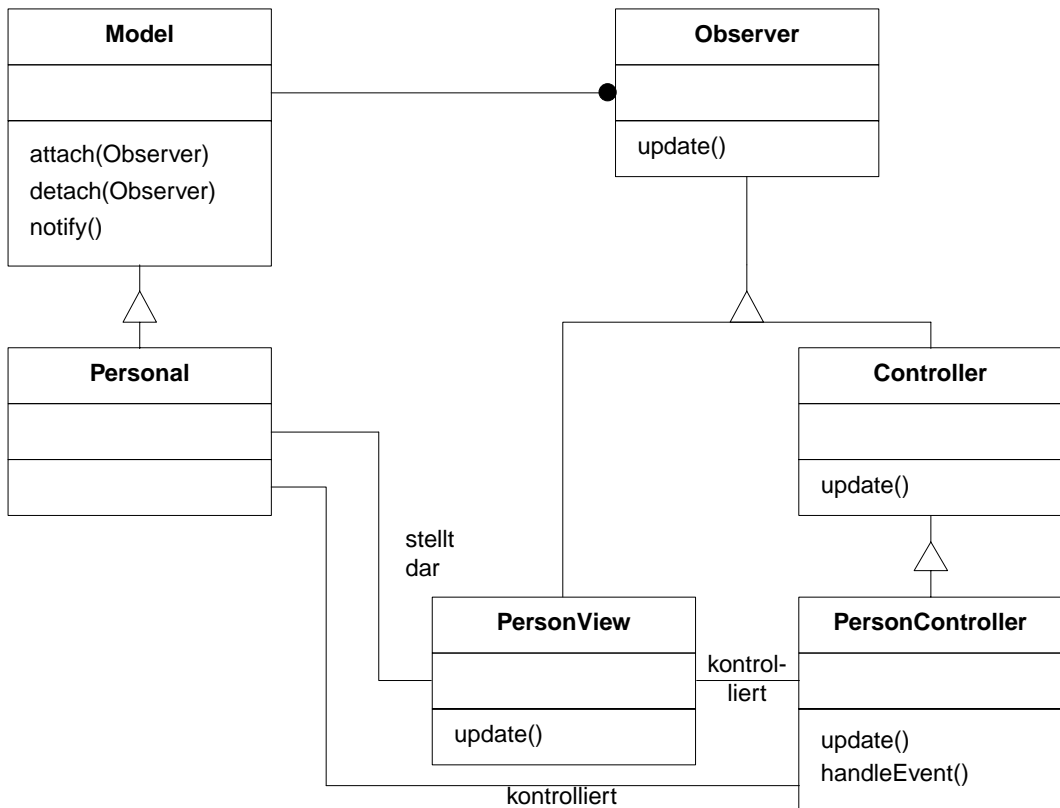


Abbildung 4: Struktur nach dem Entwurfsmuster Model/View/Controller

Der inhaltliche Kern der Anwendung steckt im Modell, dem Personal, von dem nur die oberste Klasse, *Personal*, in Abbildung 4 angegeben ist.

Die Personalverwaltung verwaltet eine Menge von Personen. Die bearbeitete Person wird in einer *PersonView* dargestellt. Eine einfache Sicht enthält Abbildung 5.



Abbildung 5: Personalverwaltung

Die Sicht wird von einem Controller, *PersonController* gesteuert. In dieser Anwendung gibt es eine 1:1-Beziehung zwischen der Sicht und dem Controller.

In unserem Beispiel untersuchen wir, wie die Beziehungen zwischen Modell, Sicht und Controller implementiert werden. Wird z.B. die Beziehung "stellt dar" zwischen der Sicht *PersonView* und dem Modell *Personal* direkt über einen Zeiger implementiert, der ein Elementobjekt von *PersonView* ist und auf *Personal* zeigt, so

brauchen wir keine dynamische Typüberprüfung.

Soll jedoch, wie beim Controller angedeutet, ein Teil des Entwurfsmusters in einer Klassenbibliothek bereitgestellt werden, ist die dynamische Typüberprüfung sehr hilfreich.

In dem Beispiel wird in der Klasse *Controller* die Beziehung zwischen einem Controller und seinem Modell und seiner Sicht implementiert:

```
class Controller : public
                    Observer
{
public:
    Controller( Subject *,
               Observer * );
    virtual ~Controller();
    virtual
        void update( Subject *);
protected:
    Subject * Model()
        { return _model;}
    Observer * View()
        { return _view;}
private:
    Subject * _model;
    // Der Controller kon-
    // trolliert das Subjekt
    Observer * _view;
    // Der Controller ist
    // einer Sicht zugeordnet.
};
```

Der spezielle Controller *PersonController* einer Personensicht erbt alle Attribute von *Controller* und implementiert die fürs Personal spezifischen Eigenschaften eines Controllers.

Dazu gehört unter anderem, auf das Drücken des Kopfes "Einfügen" mit dem Auslesen der entsprechenden Attribute und dem Einfügen einer Person, hier speziell von einer Beschäftigten der Klasse *BESCH* zu reagieren:

```
BESCH neu(personalNr, name,
           gehalt);
Personal* p_Personal =
    dynamic_cast<Personal*>
    (Model());
if ( p_Personal )
    p_Personal->insert(neu);
```

Ein Objekt der Klasse *PersonController* kennt über einen Zeiger vom Typ *Subject** sein Modell. Wenn der Personencontroller aber eine Person einfügen will, braucht er die spezifische Methode *insert*, die das Personal zur Verfügung stellt. Ein sicherer Weg stellt hier die Benutzung von *dynamic_cast* dar.

Die Alternative wäre eine direkte Typkonvertierung ohne Überprüfung. Das wäre keine Sollbruchstelle sondern eine ausgezeichnete Fehlerquelle. Der Fehler würde voraussichtlich bei kaum einem Test der neuen Software entdeckt. Er tritt höchstwahrscheinlich bei irgend einer dieser lästigen Wartungsarbeiten auf.

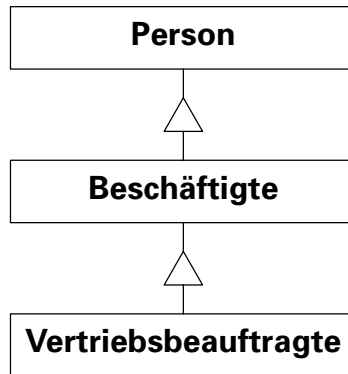
Die dynamische Konvertierung wird im obigen Beispiel auch beim Zugriff des Personencontrollers auf seine Sicht benutzt.

Wirkliche Gleichheit

Als ein Beispiel zur Benutzung von *typeid* betrachten wir den Vergleichsoperator `==`, der zwei Objekte einer Klasse *T* vergleicht. Er ist in der Regel als Methode der Klasse *T* definiert:

```
bool operator ==( const T & )
                const
```

Liegt nun eine Vererbungshierarchie zwischen Personen (PERSON), Beschäftigten (BESCH) und Vertriebsbeauftragten (VERTRIEB) vor wie:



dann ist es sinnvoll, den Vergleich als virtuellen Operator zu definieren und zwei Objekte nur als gleich anzuerkennen, wenn sie auch den gleichen Typ haben:

```
bool BESCH::operator ==
    ( const PERSON & p )
    const
{
    if(typeid(*this)==typeid(p))
    { try
      { const BESCH &b =
        dynamic_cast
          <const BESCH&>(p);
        return
          _PersNr == b._PersNr;
      }
      catch( bad_cast )
      { return false; }
    }
    else { return false; }
}
```

Zum einen wird nur wirklich verglichen, wenn mit einem Objekt mit exakt dem gleichen Typ verglichen werden soll. Das verhindert an dieser Stelle, daß Beschäftigte und Vertriebsbeauftragte gleich sein können. In einer Menge von Zeigern auf Personen kann eine Beschäftigte mit der Personalnummer 1234 gesucht werden mit:

```
typedef set
    < PERSON*, less<PERSON*> >
    Set;
Set personal;
// ... Personal einlesen
BESCH* gesucht =
    new BESCH(1234);
Set::iterator i;
for ( i=personal.begin();
      i != personal.end() &&
```

```

    **i != *gesucht; // <—
    i++ );
if ( i != personal.end() )
    cout << "gefunden: "
        << **i << endl;

```

Durch die Definition von `==` und einem Template wird automatisch der entsprechende Operator `!=` definiert, der oben benutzt wird.

Wenn der Vergleichsoperator für Vertriebsbeauftragte nicht überdefiniert wird, wird auch für diese der oben angegebene Operator aufgerufen. Sollen Vertriebsbeauftragte anders verglichen werden, so sollte der Operator überdefiniert werden.

Polymorphismus über Zeiger- und Referenzkompatibilität und virtuelle Funktionen gehen Hand in Hand mit dynamischer Typkontrolle.

Die obige Definition der Gleichheit hat ihre Tücken. Die Objekte, die beim Aufruf des Vergleichs links und rechts vom Gleichheitszeichen stehen, bestimmen den Operator, der aufgerufen wird, und der Übersetzer ist im Hinblick auf automatische Konvertierung sehr großzügig. Die Übersichtlichkeit und Verständlichkeit kann bei gleichzeitiger Benutzung aller Konzepte zur Implementierung von Polymorphismus verloren gehen.

Zu beachten ist bei diesem Beispiel zur Zeit auch, daß es z.B. falsche Ergebnisse liefert, wenn es mit dem GNU-Übersetzer Version 2.7.2 übersetzt wird. Der "Work-around" ist hier, beim `dynamic_cast` weder als Typparameter noch als Argument ein `const` zu verwenden. Aber vielleicht ist das schon behoben, wenn Sie diesen Artikel lesen.

Die zur Zeit verfügbaren Übersetzer haben jeweils verschiedene Abweichungen vom vorgeschlagenen Standard. Wenn Sie nicht gerade einen nagelneuen Übersetzer besitzen, können Sie mit den Beispielen eventuell Probleme haben.

Schnittstellen zu C

Bei der Benutzung von C-Bibliotheken tritt oft der Fall auf, daß ein Zeiger auf ein Objekt als Parameter übergeben wird. Wenn insbesondere Call-Back-Funktionen mit Zeigern auf Objekte aufgerufen werden, erhöht eine dynamische Typkontrolle die Programmsicherheit.

Im Beispiel der Personalverwaltung sind die Funktionen, die beim Drücken eines Knopfs aufgerufen werden, solche Funktionen. Dort werden die Funktionen dem Tcl/Tk-Interpreter mit einem Zeiger auf sogenannte Klientendaten bekanntgemacht. Die Klientendaten enthalten in unserem Fall einen Zeiger auf den "Arbeiter", der die eigentliche Funktion ausführen soll, und einen Zeiger auf die Methode des Arbeiters, die er ausführen soll. Die Funktion, die zunächst beim Drücken eines Knopfs aufgerufen wird, sieht dann z.B. folgendermaßen aus:

```

TclCommand* p_c =
    dynamic_cast<TclCommand*>
        (clientdata);
if ( p_c && p_c->_arbeiter
    && p_c->_methode )
    return
        ((p_c->_arbeiter)->*
         (p_c->_methode))
        (interp, argc, argv);
else return TCL_ERROR;

```

`clientdata` sind hier die übergebenen Daten. In dem Ausdruck `p->*m(a,b,c)` ist `p` ein Zeiger auf ein Objekt, `m` ein Zeiger auf eine Methode des Objekts, und `(a,b,c)` sind die aktuellen Parameter des Methodenaufrufs.

Zum Schluß weise ich auf einen beliebten Fehler hin: Ein "down cast" ist nach dem vorgeschlagenen Standard auf polymorphen Typen definiert. Wenn der Übersetzer dieses nicht überprüfen kann, ist der Effekt eines "down cast" von einem nicht polymorphen Typ eben undefiniert ...

Zusammenfassung

Die Laufzeit-Typidentifikation und die dynamische Konvertierung dienen der Handhabung von Objekten, deren genauer Typ erst zur Laufzeit bekannt ist.

- Die Laufzeit-Typabfragen erlauben eine flexible und dennoch sehr sichere Implementierung von Klassenbibliotheken.
- Wenn eine Typkonvertierung sein muß, kann sie relativ sicher gestaltet werden.
- Die dynamische Konvertierung sollte nach wie vor nicht verwendet werden, wenn die statische Typüberprüfung hinreichend ist.
- Die dynamische Konvertierung ist nur ein halber Weg zu Implementierungsmöglichkeiten wie Metaklassen.
- Eine dynamische Typüberprüfung macht ein Programm in der Regel nicht übersichtlicher, sondern nur in wenigen Ausnahmefällen.
- Einige zur Zeit verfügbare Übersetzer weisen jeweils verschiedene Abweichungen vom vorgeschlagenen Standard auf.
- Die Laufzeit-Typidentifikationen sind für die Implementierung von Bibliotheken, Frameworks und Entwurfsmustern geeignet. In der Entwicklung von normaler Anwendungs-Software sollten sie, wenn überhaupt, sparsam eingesetzt werden.

Quellen

Die vollständigen Programme befinden sich auf dem FTP-Server *ftp.informatik.uni-osnabrueck.de* unter *pub/hanser/um*.

Literatur

- [1] S. Seehusen "Programmieren in C++, Teil II: Einfache Vererbung, Teil III, Templates, Teil VII, Die Standard Template Library STL" **unix/mail** 3/95, 4/95, 3/96.
- [2] S. Seehusen, "Programmieren in C++, Teil III, Templates: Wir fertigen Schablonen und gehen in Serie" **unix/mail** 4/95.
- [3] S. Seehusen, "Programmieren in C++: Die Standard Template Library STL" **unix/mail** 3/96.