
Namensräume und Typumwandlung

Hier werden weitere Themen behandelt

Inhaltsverzeichnis

1.Namensräume.....	2
2.Typumwandlung.....	12

1. Namensräume

Bisher haben wir den Namespace `std` verwendet ohne die Bedeutung von Namensräumen exakt zu kennen.

Ein Namensraum definiert den **Sichtbarkeitsbereich** von Namen in einem C++ Programm.

In C++ wurden Namespaces eingeführt, damit man ein Programm aus unterschiedlichen Teilen zusammensetzen kann, wobei in den unterschiedlichen Teilen für unterschiedliche Objekte die selben Namen erlaubt sind.

Beispiel:

Das Softwarehaus X hat eine Bibliothek mit C++ Routinen:

```
#include <iostream>
using namespace std;
void print(string s) {
    cout << s;
}
double invers(double d) {
    return 1/d;
}
```

Das Softwarehaus Y hat eine Bibliothek mit C++ Routinen:

```
#include <iostream>
#include <string>
using namespace std;
void print(string s) {
    cout << s << endl;
}
```

```
string invers(string s) {  
    string r="";  
    for (int i=s.length()-1; i>=0; i--)  
        r = r + s.at(i);  
    return r;  
}
```

In der Anwendung sollen beide Bibliotheken verwendet werden:

```
#include "X.cpp"  
#include "Y.cpp"  
int main(){  
    double x=2;  
    x = invers(x);           // ok, da überladen  
    print("Ergebnis: ");   // welches print?  
    cout << x << endl;  
  
    string y = "123";  
    y = invers(y);         // ok, da überladen  
    print("Ergebnis: " + y); // welches print?  
}
```

Der Compiler meldet schon einen Fehler. Deshalb wird der Quellcode der beiden Bibliotheken geändert, so dass die Namenskonflikte aufgelöst werden (**keine** gute Lösung):

```
#include <iostream>
using namespace std;
void X_print(string s) {
    cout << s;
}
double X_invers(double d) {
    return 1/d;
}
#include <iostream>
#include <string>
using namespace std;
void Y_print(string s) {
    cout << s << endl;
}
string Y_invers(string s) {
    string r="";
    for (int i=s.length()-1; i>=0; i--)
        r = r + s.at(i);
    return r;
}
```

```
#include "X.cpp"
#include "Y.cpp"
int main(){
    double x=2;
    x = X_invers(x);
    X_print("Ergebnis: ");
    cout << x << endl;

    string y = "123";
    y = Y_invers(y);
    Y_print("Ergebnis: " + y);
}
```

Um diese Umbenennung zu vermeiden, sind in C++ übergeordnete Gültigkeitsbereiche mittels Namespaces möglich.

Ein Namensraum wird definiert durch das **Schlüsselwort** `namespace` und die in geschweifte Klammern eingeschlossenen Elemente des Namensraums.

```
$ cat X.cpp
#include <iostream>
using namespace std;
namespace X {
    void print(string s) {
        cout << "X: " << s;
    }
    double invers(double d) {
        return 1/d;
    }
}
$
```

```
$ cat Y.cpp
#include <iostream>
#include <string>
using namespace std;
namespace Y {
    void print(string s) {
        cout << "Y: " << s << endl;
    }
    string invers(string s) {
        string r="";
        for (int i=s.length()-1; i>=0; i--)
            r = r + s.at(i);
        return r;
    }
}
$
```

```
$ cat main.cpp
#include "X.cpp"
#include "Y.cpp"
int main(){
    using namespace X;
    double x=2;
    x = invers(x);
    print("Ergebnis: ");
    cout << x << endl;

    string y = "123";
    y = Y::invers(y);           // qualifizierter Name
    Y::print("Ergebnis: " + y);

    { using Y::print;         // lokales Synonym
      y = "abc";
      print("Ergebnis: " + y);
    }

    x = 18;
    print("Ergebnis: ");
    cout << x << endl;
}
$
```

```
$ main
X: Ergebnis: 0.5
Y: Ergebnis: 321
Y: Ergebnis: abc
X: Ergebnis: 18
$
```

Abkürzungen

Wenn lange Namen für Namensräume verwendet werden, kann man durch:

```
namespace abc = NamenEinesNamespace;
```

eine **Abkürzung** definieren und dann einfach schreiben:

```
using namespace abc;
```

Zugriff auf Methode einer Basisklasse

Um innerhalb einer abgeleiteten Klasse auf eine Methode der Basisklasse gezielt zugreifen zu können, kann die `using`-Deklaration verwendet werden:

```
class Basis {
    protected:
        void f(int);
};

class Abgeleitet : public Basis {
    public:
        using Basis::f; // öffentliches Synonym für Basis::f()
        int h();
};

a
...
Basis einBasisObjekt;
einBasisObjekt.f(0);
Abgeleitet einAbgeleitetesObjekt;
einAbgeleitetesObjekt.f();
```

Fehler f() ist nicht public!

ok!

`private` Methoden von Basisklassen können mit dieser Technik **nicht** öffentlich gemacht werden!

2. Typumwandlung

Aus C kennen wir die implizite Typumwandlung und die explizite Typumwandlung mittels des cast-Operators.

Ein double soll in einen long gewandelt werden:

```
$ cat cast.cpp
#include <iostream>
using namespace std;
int main() {
    long l;
    double d=123456.12;
    long x = d; // double -> long erzeugt Compilerwarnung
    cout << x << endl;
}
$
```

```
g++ cast.cpp -o cast
cast.cpp: In function `int main()':
cast.cpp:6: warning: initialization to `long int' from `double'
cast.cpp:6: warning: argument to `long int' from `double'
$
```

Durch casten kann dies umgangen werden:

```
$ cat cast02.cpp
#include <iostream>
using namespace std;
int main() {
    long l;
    double d=123456.12;
    long x = (long) d; // double -> long
    cout << x << endl;
}
$
```

D.h. die **Typkontrolle des Compiler** wird zur Laufzeit „**ausgeschaltet**“. **Dies sollte vermieden werden!**

C++ stellt mehrere Typumwandlungsoperatoren zur Verfügung, mit der spezielle Effekte erzielt werden und nicht generell alles in alles wandelbar ist.

Diese **Operatoren** werden wir folgt notiert:

```
OperatorName<T> (Ausdruck)
```

mit der Bedeutung:

Der Wert von `Ausdruck` soll in den Typ `T` gewandelt werden.

static_cast

Der `static_cast` Operator ermöglicht es, vom **Compiler durchgeführte** Typumwandlungen **explizit** durchzuführen.

```
$ cat static_cast.cpp
#include <iostream>
using namespace std;
int main() {
    enum Wochentag {Sonntag, Montag, Dienstag, Mittwoch,
                   Donnerstag, Freitag, Samstag};
    Wochentag heute = Mittwoch;
    int i = Montag;      // implizite Typumwandlung nach int
    //heute = i;        // Fehler: Datentypen nicht zuweisungskompatibel
    heute = static_cast<Wochentag>(i); // korrekte Typumwandlung
    cout << heute << endl;
    heute = static_cast<Wochentag>(18); // undefiniert!
    cout << heute << endl;
}
$
```

Kann eigentlich nicht umgewandelt werden! Einige Compiler erlauben das aber!

- Unterstützt Ihr Compiler dies?
- Was wird ausgegeben?

Im Bereich OOP wird der `static_cast` Operator häufig im Zusammenhang mit **Zeigern auf Objekten** verwendet:

```
$ cat static_cast02.cpp
class Basis {
    public:
        int b;
        Basis() : b(0) {cout << "Basis: " << b << endl;}
        Basis(int p) : b(p) {cout << "Basis: " << b << endl;}
};
class Abgeleitet : Basis {
    public:
        int a;
        Abgeleitet() : a(0) {cout << "Abgeleitet: " << a << endl;}
        Abgeleitet(int p) : a(p) {cout << "Abgeleitet: " << a << endl;}
};
int main() {
    Abgeleitet ao(2);
    Basis *pb;
    Abgeleitet *pa = &ao;

    pb = pa; // implizite Typumwandlung
    //pa = pb; // verboten
    pa = (Abgeleitet*) pb; // nicht erwünschter C-Stil, aber möglich
    pa = static_cast<Abgeleitet*>(pb); // so sollte man das tun, aber was ist
    // Konsequenz?
}
$
```

Downcast nennt man diese Typumwandlung von einer Basisklasse zur abgeleiteten Klasse.

Achtung:

Ein **Downcast** ist **nicht erlaubt**, wenn die **Basisklasse virtuell** ist!

dynamic_cast

Der dynamic_cast Operator

```
dynamic_cast<T>(e)
```

verhält sich so wie der static_cast, wenn die Typprüfung von e statisch, also schon zur Compilezeit durchgeführt werden kann.

Folgendes Verhalten beschreibt den dynamic_cast:

- Die **Typprüfung** wird zur **Laufzeit** vollzogen, wenn sie nicht eindeutig zur Compilezeit bestimmt werden kann.
- Der Typ **T** muss dabei ein **Zeiger** oder eine **Referenz** auf eine Klasse sein.
- Ist das Argument **e** ein **Zeiger**, der **nicht** auf ein Objekt vom Typ **T** oder abgeleitet von T zeigt, wird als Ergebnis der Typumwandlung **NULL** zurück gegeben.
- Eine **Exception** (bad_cast) wird geworfen, wenn **e** eine **Referenz** ist, die **nicht** auf ein Objekt vom Typ **T** (oder von T abgeleitet) verweist.

Das folgende Programm zeigt den dynamic_cast:

```
$ dynamic_cast.cpp
#include <iostream>
using namespace std;
```

```
class Basis {
    public:
        virtual void f() {cout << "Basis" << endl;}
};

class Abgeleitet : public Basis {
    public:
        virtual void f() {cout << "Abgeleitet" << endl;}
};

Abgeleitet* g(Basis *pb) { // globale Fkt g() benutzt f()
    Abgeleitet *pa = dynamic_cast<Abgeleitet*>(pb);
    if (pa)
        pa->f();
    return static_cast<Abgeleitet*>(pb);
}

int main() {
    Basis bo;
    Abgeleitet ao;
    Basis *pbb = &bo;
    Basis *pba = &ao;
    Abgeleitet *perg;

    perg = g(pba); // Abgeleitet::f() wird aufgerufen
    perg = g(pbb); // Abgeleitet::f() wird nicht aufgerufen!
}
$
```

const_cast

Mittels `static_cast` oder `dynamic_cast` kann die **const-Eigenschaft** eines Objektes **nicht** eliminiert werden, mit dem C-Stil `cast` kann das erreicht werden.

Besser ist es, dazu den `const-cast` zu verwenden.

Achtung:

„casting the const away“ sollte nur in Ausnahmefällen verwendet werden!

```
int main() {
    const int i = 0;
    const int *ip = &i;
    // *ip = 1;    // Fehler: ip ist const Zeiger auf const int
    cout << *ip << endl;

    // int *iq = &i;    // Fehler: impliz. Konvertierung von 'const int*' in
    //                  // 'int*' nicht möglich

    int *iq = (int*) &i; // C-Stil cast ermöglicht Konvertierung
    *iq = 2;
    cout << *iq << endl;

    int *ir = const_cast<int*>(&i);    // expliz. Typumwandlung
    *ir = 3;
    cout << *ir << endl;
}
```