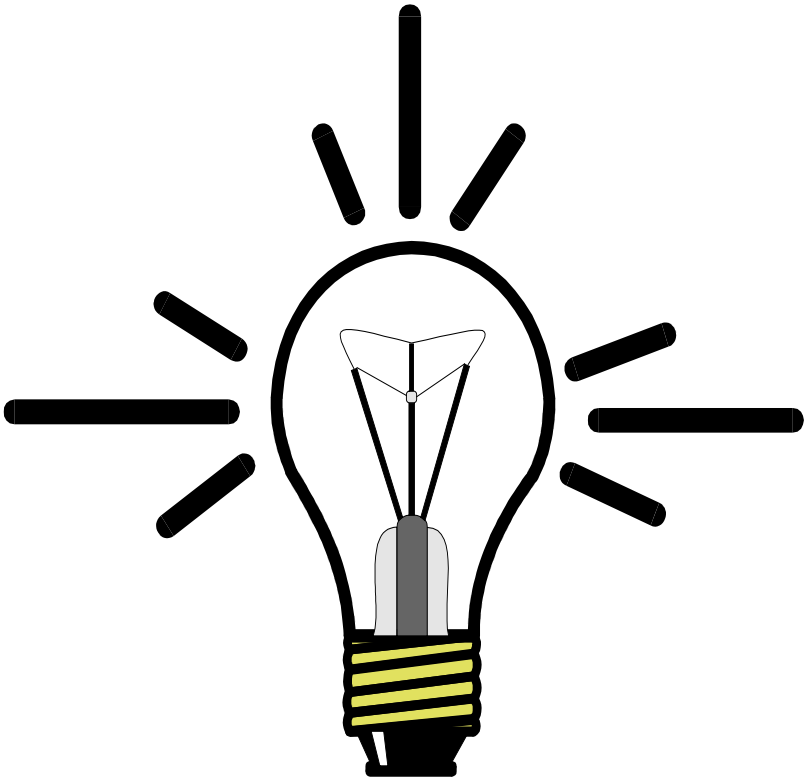
Wer C sagt, muss auch PlusPlus sagen!

Die Sprache
Der Kurs
Die Erleuchtung



Ergänzungsmanuskript für Fortgeschrittenen-Kurs C++ Programmierung

Kursübersicht

TAG 1,
- Namensbereiche
- Ausnahmebehandlung
- RTTI und die neuen Cast-Operatoren

TAG 2,
- IOStream-Bibliothek
- Strings

TAG 3,
- Templates

TAG 4,
- Standard Template Library (Teil 1)

TAG 5,
- Standard Template Library (Teil 2)

1 Vorwort

Dieses Manuskript ergänzt das Manuskript „Grundkurs C++ Programmierung“.

Seit 1998 ist C++ ein ANSI- und ISO-Standard. Es handelt sich um eine Weiterentwicklung des de-facto Standards von Bjarne Stroustrup. Neu sind Namensräume als frei definierbare globale Gültigkeitsbereiche. Darüber hinaus hat die C++-Laufzeitbibliothek ihre Standardisierung als Standard Library erhalten. Die in ihr definierte Klasse String ist komfortabler und weniger fehleranfällig als die aus C bekannten Zeichenketten.

Von besonderer Bedeutung ist die Standard Template Library (STL), die das Programmieren entscheidend erleichtert. Während die Entwicklung von C++ auf Bjarne Stroustrup basiert, geht die Entwicklung der STL im wesentlichen auf Alexander Stepanov und die Idee des generischen Programmierens zurück.

Templates bilden das grundlegende Konzept. Sie lassen sich als Schablonen für die Konstruktion von Funktionen und Klassen mit verschiedenen Typen verstehen. Das Template-Konzept hat sich auf alle Bereiche der Standard Library ausgewirkt.

Die Sprachelemente, die in den nun folgenden Kapiteln behandelt werden, sind noch vergleichsweise jung, aber trotzdem außerordentlich Mächtig und weittragend. Sie werden inzwischen von heutigen Kompilern im wesentlichen unterstützt.

H ein kurzer Überblick über die wichtigsten Neuerungen:

- Namespaces (Namensbereiche)
  Zur konfliktfreien parallelen Verwendung von Bibliotheken die namensgleiche Bezeichner enthalten.

- Exception Handling (Ausnahmebehandlung)
  Strukturierte Methode zur Fehlerbehandlung in Programmen, insbesondere bei Verwendung von Programmbibliotheken.

- RTTI (RUNTIME TYPE INFORMATION)
  Mechanismen zur Bestimmung des genauen Typs von Objekten zur Laufzeit und Absicherung von casts.

- Templates (Schablonen)
  Erstellung von (Typ-)parametrisierten Klassen- und Funktionsmustern.

2 Namensbereiche

Eines der Hauptziele im Rahmen von OOP ist wiederverwendbarer Code und die Nutzung von Software ICs. In der Praxis bedeutet dies, dass Programme in immer stärkerem Maße auf Bibliotheken aufbauen und außerdem die Zahl der verfügbaren Bibliotheken regelrecht explodiert. Dabei kommt es unglücklicherweise hin und wieder vor, dass in verschiedenen Bibliotheken globale Bezeichner gleichen Namens vorkommen, wodurch deren gleichzeitiger Einsatz erheblich erschwert, wenn nicht sogar unmöglich gemacht wird.

Was kann man im Konfliktfall tun? Die Bezeichner einer Bibliothek entsprechend abändern? Das ist nur möglich, wenn die Bibliothek im Quellcode vorliegt. Um Probleme mit älterer Software zu verhindern, die auf der Bibliothek aufbaut kann man nicht dass Original, sondern lediglich eine Kopie ändern. Damit wären dann Konsistenzprobleme bei nachfolgenden Verbesserungen der Bibliothek vorprogrammiert, und selbst wenn man diese Klippe umschiffen kann, hat man stets den doppelten Wartungsaufwand. Die angepasste Version könnte sich zudem mit wieder anderen Bibliotheken beißen...

Eine andere Möglichkeit wäre, eine der Bibliotheken lokal innerhalb eines speziellen Moduls einzubinden und (mit viel Aufwand) eine entsprechende Schnittstelle zu realisieren, die wiederum gepflegt werden muss.

Zur Vorsorge kann man auch allen Bezeichnern innerhalb einer Bibliothek ein Präfix voranstellen, aber auf Produkte von Fremdherstellern hat man natürlich diesbezüglich keinen Einfluss. Zudem ist die richtige Wahl der Präfixlänge nicht einfach. Ist es zu kurz, ist es auch mit der Eindeutigkeit nicht weit her; ist es zu lang, wird es dem Anwender schnell lästig. Alles in allem ein unbefriedigender Zustand.

Aus diesem Grunde werden sogenannte **Namensbereiche** (namespaces) eingeführt. Ein Namesbereich ist ein abgegrenzter Gültigkeitsbereich für globale Namen, der beliebigen Programmteilen übergestülpt werden kann.

```
namespace eineHuelle
{
      /* ...beliebiger Code */
}
namespace andereHuelle
{
      /* ...beliebiger Code */
}
```

C++ definiert die gesamte Standard-Bibliothek in einem eigenen Namensbereich, der **`std`** genannt wird. Alle Bezeichner gehören zu diesem Namensbereich und müssen außerhalb dieses Bereiches mit **`std::`** qualifiziert werden. Die älteren Versionen von C++ kannten dieses Konzept nicht. Die Headerdatei `<iostream.h>` kam ohne Namensbereiche aus. Das standardisierte C++ verwendet Headerdateien ohne die Endung `.h` z. B. **`<iostream>`.**

Die bekannten globalen Stream-Objekte `cin` und `cout` müssen dann mit dem Namensbereich `std` aufgerufen werden:

```
#include <iostream>

int main()
{
      std::cout << ?Hallo, Welt! ? << std::endl;
      return 0;
};
```

2.1 using-Direktive

Es gibt nun verschiedene Mechanismen, um auf Bezeichner aus einem Namensbereich zuzugreifen. Am einfachsten ist die Einblendung eines gesammten Namensbereiches mit Hilfe einer `using`-Direktive.

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

int main()
{
      cout << ?Hallo, Welt!? << endl;
      return 0;
};
```

Mit der using-Direktive können alle Bezeichner eines Namensbereiches ohne Qualifizierung des Namensbereiches angesprochen werden. Alle Namen sind damit allerdings auch innerhalb des Gültigkeitsbereichs bekannt, in den sie mit der using-Direktive aingeführt worden sind. Damit können wieder die bekannten Probleme mit Doppelbezeichnern auftreten.

Beispiel:

```
namespace fhte
{
      int x,y,z;
}

int x = 1;

void f()
{
      int y = 5;

      {
            using namespace fhte; // fhte wird innerhalb von {...}
                                  // zugreifbar

            ++x;  // Fehler: x ist mehrfach deklariert
            ++y;  // lokales y
            ++z;  // fhte::z wird verwendet
      }
```

```
      ++x; // globales x wird verwendet
      ++z; // Fehler: z ist hier wieder unbekannt
}
```

Die Wirkung von `using` ist transitiv, d.h. `using namespace A` macht alles sichtbar, was in A deklariert oder sichtbar gemacht wurde. Werden mehrere Namensbereiche auf diese Weise eingeblendet, schließt der Kompiler automatisch doppelte Bezeichner beider Bereiche von der Einblendung aus (d. h unabhängig von der Reihenfolge der Einblendungen) . Auf sie muss dann, genau wie ohne Einblendung, mit Hilfe des Skopusoperators (ähnlich wie auf Klassenelemente) zugegriffen werden. An dieser Stelle sei noch mal daran erinnert, dass auch eine Klasse einen Namensraum bildet, der in den Namensraum aller Basisklassen eingebettet ist.

Es ist erlaubt, Namensbereiche zu schachteln und entsprechend beim Zugriff den Scope-Operator zu kaskadieren. Funktionen innerhalb eines Namensbereiches können wie bei Klassen entweder sofort oder später (mit Qualifizierung) definiert werden. Sofort definierte Funktionen sind jedoch nicht automatisch `inline`.

```
namespace outer {
//...
 namespace inner {
   void f(void); // declaration
   void g(void) { .... } // definition
 }
}
void outer::inner::f(void) { ... } // definition
```

Gleichnamige Namensbereiche, die an verschiedenen Stellen des Programms (ja sogar in verschiedenen Modulen) auftreten, werden automatisch zu einem gemeinsamen Namensbereich zusammengefasst. In diesem Punkt unterscheiden sich Namensbereiche von Klassen.

```
namespace doppelt
{
  int f1(void);
  //...
}
namespace einfach
{
  //...
}
namespace doppelt
{
  int f2(void);
  //...
}

using namespace doppelt;  // macht f1() und f2() verfügbar
```

2.2 using-Deklaration

Mit einer `using`-Deklaration können einzelne Bezeichner eines Namensbereiches lokal eingeblendet werden.

```
using std::endl;
```

Damit wid `endl` im aktuellen Gültigkeitsbereich lokal gleichbedeutend mit `std::endl` .

Beispiel:

```
namespace fhte
{
      int x,y,z;
}

int x = 1;

void f()
{
      int y = 5;

      {
            using fhte::x;  // globales x wird überdeckt
            using fhte::y;  // Fehler: y ist mehrfach deklariert
            using fhte::z;

            ++x;        // fhte::x wird verwendet
            ++z;        // fhte::z wird verwendet
      }

      ++x;              // globales x wird verwendet
      ++z;              // Fehler: z ist hier wieder unbekannt
}
```

2.3 Anonyme Namensbereiche

Etwas gewöhnungsbedürftig ist die Behandlung von namenlosen Namensräumen, denn auch so etwas ist zulässig.

```
namespace { ... }
```

Anonyme Namensbereiche können verwendet werden, um `static` nur noch für die Lebensdauer einer Variablen einzusetzen. Die Beschränkung der Sichtbarkeit von Variablen auf eine Datei mit Hilfe von `static` kann durch anonyme Namensbereiche ersetzt werden:

```
// a.cpp
static int n;
void f()
{
      n = 1;
}
// b.cpp
```

```
extern int n;
void g()
{
      n = 10; // Fehler: Kein Zugriff auf static Variable in a.cpp
}
```

Dies kann ersetzt werden durch:

```
// a.cpp
namespace
{
      int n;
};
void f()
{
      n = 1;
}
// b.cpp
extern int n;
void g()
{
      n = 10; // Fehler: Kein Zugriff auf Variable n in a.cpp
}
```

2.3.1 Aliasnamen

Um etwas Schreibarbeit zu sparen können Aliasnamen[1] für Namensbereiche vergeben werden. Dies bietet sich im Beispiel für die zweite Funktion an.

```
namespace andere = andereHuelleMitGanzFurchtbarLangemNamen;
andere::doppelteFunktion();
```

Die Vergabe von Aliasnamen für die doppelten Bezeichner selbst ist natürlich nicht möglich, sonst könnte der Linker ja nicht mehr vereinen, was zusammengehört!

2.4 Koenig-Lookup

Es ist nicht notwendig, für Funktionen einen Namensbereich anzugeben, wenn die Argumente im Namensbereich der Funktion definiert sind. Beim Aufruf der Funktion wird sie auch in allen Namensbereichen der übergebenen Parameter gesucht. Diese Regel wird Koenig-Lookup genannt.

Beispiel:

```
#include <iostream>
namespace fhte
{
      class Punkt
      {
            int x;
      public:
            int getX();
      };
```

[1] Hier sei an **typedef** erinnert, mit dem ebenfalls lediglich Aliasnamen vereinbart werden

```
      int Punkt::getX()
      {
            return x;
      }
}

namespace fhte
{
      int f(Punkt &p)
      {
            return p.getX()*p.getX();
      }
}

int main()
{
      fhte::Punkt p;
      f(p);       // f(p) wird gefunden über das Argument
                  // funktioniert nicht in VCC 6.0
      return 0;
}
```

3 Ausnahmebehandlung

3.1 Was ist Exception Handling

Ausnahmebehandlung erlaubt es, Fehler und „unerwartetete“ Situationen in eigens für ihre Behandlung vorgesehenen Codeabschnitten, d. h. unabhängig vom „normalen“ Code, abzufangen und zu behandeln.

Ein defensiver Programmierstil gebietet es, auf Ausnahmen vorbereitet zu sein und zu verhindern, dass sie Fehler (z. B. Abstürze, fehlerhafte Ergebnisse, Programmabbruch) nach sich ziehen. Das erstellte Programm soll stabil sein. Daraus folgt, dass man in kritischen Programmteilen Code zur Erkennung und Behandlung von Fehlern vorsehen muss. Besonders kritisch sind hierbei Ein-/Ausgabeoperationen und der Umgang mit dynamischen Speicher, oder allgemein: der Umgang mit *(knappen oder geteilten) Ressourcen*[2]. Aber auch so simple Dinge wie eine Division durch 0 wollen abgefangen werden.

Eine der traditionellen Methoden zur Erkennung von Fehlern ist die Rückgabe eines Fehlercodes durch kritische Funktionen, entweder als direkter Rückgabewert oder über eine globale Variable[3]. Letzteres ist notwendig, weil der Wertebereich vieler (insbesondere mathematischer) Funktionen keine Lücken aufweist, die man zur Fehlersignalisierung nutzen kann. Der zurückgelieferte Fehlercode müsste streng genommen nach jedem Aufruf geprüft werden. Diese Maßnahmen sind jedoch sehr aufwendig, oftmals sogar geradezu ermüdend, und resultieren nicht selten in Code, der nicht mehr sonderlich leicht zu lesen ist. Dies führt leider häufig dazu, dass man die Fehlerbehandlung nicht konsequent durchhält oder auch bewusst einen Kompromiss eingeht. Wie weit man dabei gehen kann, muss jeweils anhand der Aufgabenstellung und deren Wichtigkeit (Beispiel, Hobby, kommerzielles Produkt, Regierungsauftrag) entschieden werden.

Aus diesem Widerspruch aus konsequenter Fehlerbehandlung einerseits und Wartbarkeit des resultierenden Programms andererseits ist die Idee des Exception Handlings entstanden.

Exception Handling stellt einen Mechanismus zur Verfügung, um normalen und fehlerbehandelnden Code übersichtlich zu trennen und Ausnahmesituationen sicher zu behandeln.

Im Zusammenhang mit Bibliotheken, die ja immer wichtiger werden, weist Stroustrup in [StrouCPL] noch auf ein weiteres Problem hin: Der Ersteller einer Bibliothek weiß sehr genau, wie er Ausnahmen entdecken kann. Er kann jedoch schwerlich eine

[2] Hier sei auf die von Stroustrup beschriebene Methode *„resource acquisition is initialization“* hingewiesen. Dabei wird eine Resource in ein Objekt verpackt: die Anforderung im Konstruktor, die Freigabe im Destruktor. Wird das Objekt zerstört (z. B. beim Verlassen des Blocks), erfolgt die Freigabe der Resource automatisch.

[3] Oftmals wird auch beides gemacht (z. B. UNIX Systemaufrufe). Der direkte Rückgabewert (-1 bei UNIX) zeigt an, dass etwas schiefgelaufen ist, und die globale Variable (UNIX: errno) enthält den genauen Fehlercode.

optimale Lösung für die Behandlung dieser Ausnahmen in allen Anwendungen, die auf der Bibliothek aufsetzen, implementieren. Der Anwendungsprogrammierer steht vor dem umgekehrten Problem. Er weiß zwar, wie er mit den Ausnahmen umzugehen hat, aber da er die Implementation der Bibliothek nicht kennt (und auch nicht kennen soll: Information Hiding) kann er sie, wenn überhaupt, nur unter Mühen entdecken. Auch hier bietet das Konzept des Exception Handling eine Lösung:

Exceptions ermöglichen einer Bibliothek, Ausnahmezustände an das aufrufende Programm zu melden und ggf. sogar noch Daten über die näheren Begleitumstände zu liefern.

Demnach kann man eine Exception als ein, durch eine Datenstruktur repräsentiertes Ereignis auffassen. Dabei gilt jedoch, dass Exceptions nur **synchron** als Resultat von Anweisungen im Programm auftreten. Sie sind also nicht mit Interrupts oder anderen **asynchronen** Ereignissen zu verwechseln!

3.2 Ausnahmebehandlung in C++

Das Prinzip der Ausnahmebehandlung lässt sich leicht erläutern:

- Spezielle Handler übernehmen die Behandlung der Ausnahmen. **`catch`** ist das zuständige neue Schlüsselwort.
- Tritt eine Ausnahme auf, so erzeugt der Programmierer eine Exception. **`throw`** ist das zuständige neue Schlüsselwort. Damit wird die Ausführung im aktuellen Block abgebrochen und die Kontrolle an einen Handler weitergereicht. Der Handler kann die Ausnahme bearbeiten. Anschließend wird die Ausführung nach dem Handler fortgesetzt.
- Mit dem Schlüsselwort **`try`** werden Anweisungen „probeweise“ durchgeführt. Dies geschieht in einem, durch einen Block gekennzeichneten Gültigkeitsbereich, den `try` definiert.
- Ausnahmen werden durch C++-Objekte beschrieben. Die Methoden der OOP stehen hierfür zur Verfügung.

Die Ausnahmebehandlung geschieht also mit 3 neuen Schlüsselwörtern `try`, `throw` und `catch`.

Mit Hilfe von `try` wird ein Block aus beliebigen Anweisungen gekennzeichnet, in dem Exceptions auftreten können.

Wird innerhalb des Blocks ein Ausnahmezustand erkannt, kann mit Hilfe von `throw` eine Exception „ausgeworfen“ werden. Eine Exception kann dabei ein beliebiges Objekt, auch von einem Standardtyp, sein. In der Regel wird man für Exceptions jedoch spezielle Klassen vereinbaren. Das kann auf so triviale Weise geschehen wie in folgendem Beispiel:

```
class simpleException { } // nur Meldung des Ereignisses
```

Es kann jedoch sinnvoll sein, eine Exception-Klasse mit Datenelementen zu versehen, die vor dem Auswurf einer Instanz mit Informationen zur Ursache der

Ausnahme initialisiert werden, um ihre Behandlung bzw. Lokalisierung zu unterstützen.

```
class detailedException
{
  public:
  enum cause_t { user_error, hardware_error, software_error } why;
  int where;

  detailedException( cause_t _why, int _where )
    : why(_why), where(_where) { }
}
```

Unmittelbar hinter dem `try`-Block folgen die Exception-Handler (mindestens einer). Sie werden gekennzeichnet durch das Schlüsselwort `catch`, gefolgt von einer Parameterliste mit dem Typ der zu behandelnden Exception und anschließendem Codeblock zu Realisierung der Ausnahmebehandlung (z. B. Fehlermeldung ausgeben, default Werte setzen und fortsetzen,...). Existieren mehrere Handler folgen diese unmittelbar aufeinander, **normaler Code zwischen den Handlern ist nicht erlaubt!**

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

class MyException
{
      double x1, x2;
public:
      MyException(double x1, double x2):x1(x1), x2(x2){}
      void print()
      {
            cout << "MyException aufgetreten:" << endl;
            cout << "x1= " << x1 << endl;
            cout << "x2= " << x2 << endl;
      }
};

double f(double x1, double x2)
{
      try
      {
            if (x2 == 0.0)
            {
                  throw MyException(x1, x2); // poblemspezifische
                                             // Ausnahme werfen
            }
      }
      catch(MyException & e)     // Ausnahme abfangen
      {
            e.print();           // Meldung ausgeben
            throw "Fehler aufgetreten\n";   // allgemeine Ausnahme
                                            // werfen;
      }
      return x1/x2;
}
```

```
int main()
{
      double bruch;
      try
      {
            bruch = f(81, 3);
            cout << "bruch= " << bruch << endl;
            bruch = f(3, 0);
            cout << "bruch= " << bruch << endl;
      }
      catch(...)       // default-Handler
      {
            cout << "Exception aufgetreten" << endl;
      }
      return 0;
}
```

Das Programm gibt aus:

```
bruch= 27
MyException aufgetreten:
x1= 3
x2= 0
Exception aufgetreten
```

Dieses Programm zeigt prototypisch die Ausnahmebehandlung. Die Klasse `MyException` dient zur Behandlung einer Division durch 0. Sie beschreibt den Fehler und gibt die notwendigen Informationen aus.
Die Methode `f()` löst eine problemspezifische Exception aus. Das Programm wird unterbrochen und nach einem passenden Handler gesucht. Der wird in `catch (MyException &e)` gefunden. Für das Objekt `e` wird mit `print()` eine entsprechende Meldung ausgegeben. Darüber hinaus wird mit `throw "Fehler aufgetreten\n";` eine allgemeine Exception geworfen, die erst mit dem default Handler `catch(...)` abgefangen wird.

Die Syntax des Exception Handling erinnert zum einen an die `switch` Anweisung, zum anderen an Funktionsaufrufe. Beide Vergleiche haben ihre Berechtigung.

- Der Code innerhalb des `try` Blocks liefert ähnlich zu einem `switch` die Bedingung, gemäß derer einer der Handler (oder auch keiner) angesprungen wird. Auch einen default Handler gibt es:

  ```
  catch(...) { }
  ```

  Er wird für alle Exceptions angesprungen, die von den anderen Handlern nicht behandelt werden. Im Unterschied zu `switch` sind jedoch keine `break` Anweisungen zwischen den Handlern nötig, und wenn im `try`-Block keine Exception auftritt, werden alle Handler übersprungen, **auch der default handler!**

- Der eigentliche Aufruf der Handler erfolgt wie der Aufruf einer Funktion mit einem Argument. Die Angabe eines Namens für die dem formalen Parameter entsprechende Exception ist jedoch **optional**. Zur Auswahl des passenden Handlers genügt ja der Typ. Soll allerdings innerhalb des Handlercodes auf

Datenelemente der übergebenen Exception zugegriffen werden, muss diese selbstverständlich einen Namen haben

Die Suche nach dem passenden Handler erfolgt von oben nach unten, d.h. **die Anordnung der Handler ist relevant**. Insbesondere muss der default Handler als letzter stehen. Auch seine Definition harmoniert mit der Syntax für Funktionsaufrufe: die Ellipse (...) passt auf alles. Doch die Gemeinsamkeiten zu Funktionsaufrufen gehen noch weiter:

Ein Handler für Exceptions einer Klasse A passt auch auf Exceptions aller von A abgeleiteten Klassen (erweiterte Zuweisungskompatibilität, vgl. Vererbung und Polymorphie).

Dabei ist jedoch zu beachten, dass im Falle von call-by-value die übergebene Exception auf den Basistyp „zurechtgestutzt“ (slicing) wird und der Originaltyp der Exception nicht mehr feststellbar ist. Damit ist auch kein Zugriff mehr auf spezielle Datenelemente der abgeleitenten Klasse mehr möglich und der Mechanismus der späten Bindung für Aufrufe virtueller Methoden steht nicht mehr zur Verfügung! Für polymorphe Exceptionobjekte ist daher call-by-reference vorzuziehen. Unabhängig davon ist es in den allermeisten Fällen sinnvoll, eine Klassenhierarchie für Exceptions zu definieren.

Beispiel:

```
class MathException { };
class OverflowException : public MathException { };
// ... weitere mathematische Ausnahmen
class DivbyZeroException : public MathException { };

try
{
  // ... kritische Rechenoperationen
}
catch( DivbyZeroException )
{
  // ... z. B. Korrektur durch Ergebnis = INT_MAX
}
catch( MathException )
{
  // fängt alle von MathException abgeleiteten Exceptions ab.
  // Muss an zweiter Stelle stehen, da sonst auch sämtliche
  // DivbyZeroExceptions abfangen würden!
}
catch ( ... )
{
  // fängt alle anderen Exceptions ab
}
```

Wie man sieht, können bei hierarchischer Definition gleichartige Exceptions einfach behandelt werden. Fügt man am Ende noch einen Handler für die Basisklasse ein, ist man auch in Zukunft sicher, falls jemand neue Exceptions ableitet.

Eine Exception gilt als erledigt, sobald ein Handler zu ihrer Bearbeitung gefunden und aufgerufen wurde. Stellt sich innerhalb des Handlers (z. B. anhand der in der

Exception übergebenen Informationen oder weil Korrekturmaßnahmen fehlschlagen) heraus, dass dieser die Exception nicht behandeln kann, so kann dieselbe Exception erneut ausgeworfen werden. Dabei wird die Originalexception weitergereicht und nicht etwa die (eventuell zurechtgestutzte) lokale Kopie. Auch kann der Handler ggf. andere Exceptions auswerfen.

```
try
{
  // ...beliebiger Code
}
catch( AnException e )
{
  // ...Behandlungsversuch
  if ( !erfolgreich) throw; // Originalexception erneut auswerfen
  // ...
  throw e;             // lokale Kopie neu auswerfen
  // ...
  // andere Exception auswerfen
  throw AnotherException( /* Initialisierungsdaten */ );
}
```

Jede Gruppe von Handlern ist nur für die Behandlung von Exceptions aus ihrem zugeordneten `try` Block verantwortlich. Daher werden alle innerhalb von Handlern (die ja außerhalb des `try` Blocks liegen) ausgeworfenen Exceptions nach außen an den nächsten umschließenden `try` Block weitergereicht. Die `try` Blöcke können also geschachtelt werden. Ihr Gültigkeitsbereich setzt sich in Funktionen, die in ihrem Inneren aufgerufen werden, bis zu beliebiger Tiefe fort. Dieser Mechanismus verhindert Endlosrekursionen beim Exception Handling und gestattet die Implementierung von mehreren Schichten zur Fehlerbehandlung. Ebenfalls können Exceptions nach Außen weitergereicht werden, für die kein Handler existiert.

```
void g(void)
{
  try
  {
    //...
    // zur Weitergabe an die aufrufende Funktion
    throw SomeException();
  }
  catch ( AnException )
  {
    //...
  }
  catch( SomeException )
  {
    // ...Cleanup[4]
    throw SomeException(); // keine Endlosrekursion !!!
  }
}

void main(void)
{
  try
  {
```

[4] empfielt sich, z. B. um vor dem Verlassen der Funktion noch dynamische Objekte freizugeben oder andere Aufräumarbeiten zu erledigen, die der Kompiler nicht automatisch macht

```
    g()
  }
  catch( AnException )
  {
    // wird nie erreicht, AnExceptions werden in g() erledigt
  }
  catch( SomeException }
  {
    // behandelt die Ausnahme aus g()
  }
}
```

Standard-Ausnahmeklassen

Alle Ausnahmeklassen leiten sich von der Basisklasse `std::exception` ab. Sie stellt die Methode `what()` zur Verfügung, mit der eine Beschreibung der Ausnahme ausgegeben werden kann.

Beispiel:

```
#include <iostream
#include <string
using namespace std;

void f1(string s, int i, char c)
{
      try
      {
            s.at(i) = c;  // throws out_of_range
      }
      catch (exception & e)
      {
         cerr << "Standard-Ausnahme: " << e.what() << endl;
      }
}

void f2(string s, int i, char c)
{
      try
      {
            s[i] = c;        // ERROR: undefined behavior
      }
      catch (...)
      {
         cerr << "sonstige Ausnahme" << endl;
      }
}

int main()
{
      try
      {
            string s1("FHTE");       // kann bad_alloc auslösen
            f1(s1, 4, 'a');
            f2(s1, 5, 'a');
      }
      catch (bad_alloc )
```

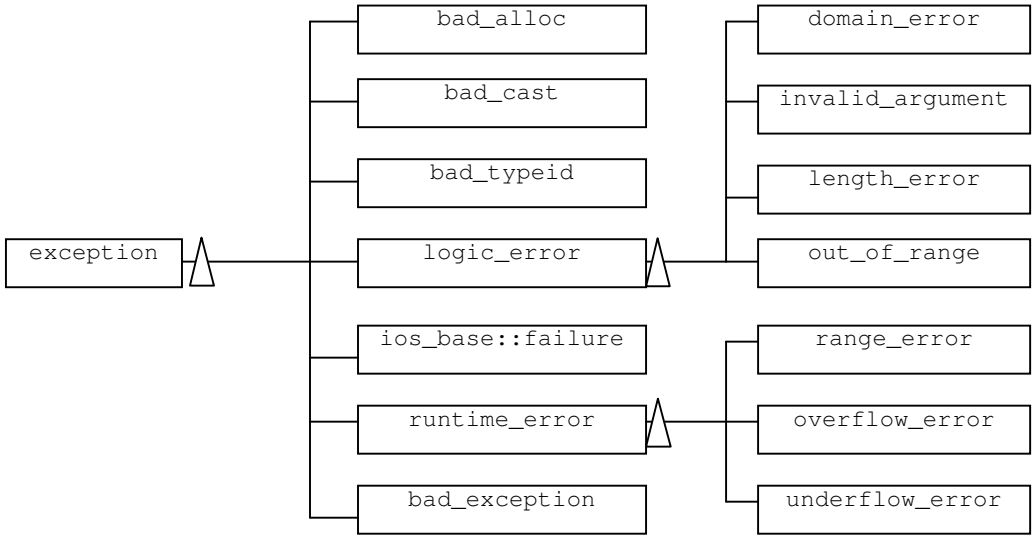
```
      {
            cerr << "kein Speicherplatz mehr" << endl;
            return -1;
      }
      catch (...)
      {
            cerr << "sonstige Ausnahme" << endl;
            return -1;
      }

      return 0;
}
```

Ergebnis:

```
Standard-Ausnahme: invalid string position
unerwartete sonstige Ausnahme
```

Hier wird die Klasse `string` verwendet (vgl. Kapitel 6). Die Funkktion `f1()` verwendet die Methode `at()` dieser Klasse; sie kann eine `out_of_range` Ausnahme werfen. Die Funktion `f2()` verwendet den Indexoperator `[]`. Indexüberschreitung führt zu einem Fehler mit undefiniertem Verhalten. Die Fehler werden mit entsprechenden `catch` Handlern abgefangen.

Die Standard-Bibliothk definiert eine Hierarchie von Standard-Ausnahmen:



Erläuterungen:

`exception`: Basisklasse.
`bad_alloc`: Speicherzuweisungsfehler; tritt bei Fehler von `new` auf.
`bad_cast`: Typumwandlungsfehler; tritt bei Fehler von `dynamic_cast` auf.
`bad_typeid`: Typumwandlungsfehler; tritt bei Fehler von `typeid` auf.
`logic_error`: tritt z. B. bei logischen Vorbedingungen auf.
`ios_base::failure`: I/O-Fehler.
`runtime_error`: nicht vorhersehbarer Fehler.
`bad_exception`: tritt bei Fehler von unerwarteter Ausnahme auf.
`domain_error`: Fehler des Anwendungsbereichs.
`invalid_argument`: tritt z. B. bei Initialisierungsfehlern auf.
`length_error`: Fehler in Standard-Bibliothek bei Größenüberschreitungen.
`out_of_range`: Fehler in Standard-Bibliothek bei Bereichsüberschreitungen.

`range_error`: Bereichsüberschreitung zur Laufzeit.
`overflow_error`: arithmetischer Überlauffehler.
`underflow_error`: arithmetischer Unterlauffehler.

Behandlung nicht abgefangener Ausnahmen

Existiert im aktuellen Block kein umschließendes `try` bzw kein passender Handler, wird die Suche rekursiv in den aufrufenden Funktionen fortgesetzt. Wird schließlich auch in der Funktion `main()` kein umschließendes `try` mit passendem Handler gefunden, wird die Funktion `terminate()` aufgerufen. Diese ruft in ihrer Defaultversion eine weitere Funktion namens `abort()` auf, was zur Beendigung des Programms führt. Dieses Verhalten ist eine konsequente Fortsetzung der Weitermeldung von Ausnahmezuständen nach außen (in diesem Fall an das Betriebsystem) und stellt eine grundlegende Veränderung des Umgangs mit Fehlern dar. Bisher führten unentdeckte Fehler i.d.R. nicht zum Programmende und traten erst durch Folgefehler an ganz anderer Stelle zutage. Dies war mit Schwierigkeiten bei der Fehlerlokalisierung verbunden und gab Programmen einen unberechenbaren touch.

Soll das Programm weiterlaufen, müssen also alle Exceptions abgefangen werden. Würde man auch nur einen Handler vergessen, würde das Programm irgendwann aussteigen. Und beim Einführen einer neuen Exception, etwa in eine Bibliothek, müsste sämtlicher Anwendercode angepasst werden. Nun zeigt sich, wie wichtig die Möglichkeit ist, Gruppen von Exceptions über ihre Basisklasse abzufangen. Andernfalls müsste man stets für alle möglichen Exceptions Handler vorsehen. Da sie nicht wie etwa `cases` bei `switch` gruppiert werden können, wäre oftmals redundanter Code die Folge.

```
switch ( a )
{
  // das geht...
  case 1:
  case 3:
  // ein Code für alle drei
  break;
}

try
{
  //...
}
// das dagegen nicht
catch( Ex1, Ex2 )
{
  //...
}
// und das auch nicht
catch( Ex1 )
catch ( Ex2 )
{
  //...
}
```

```
// aber so erwischen wir sie
catch( ExBase ) //Exbase sei Basisklasse von Ex1, Ex2
{
  //...
}
```

Ausnahmen in der Schnittstelle deklarieren

Es ist möglich die Exceptions, die eine Funktion auslösen kann, in ihre Deklaration aufzunehmen. Dabei sind drei Fälle zu unterscheiden:

- `void f(void) throw(Ex1, Ex2);` // nur Ex1 und Ex2 (und davon abgeleitete)
- `void g(void) throw();` // keine Exceptions
- `void h(void);` // beliebige Exceptions

Dabei sind auch diejenigen Exceptions zu beachten, die durch den Aufruf weiterer Funktionen auftreten können! Wenn g() die Funktion f() aufrufen würde, müsste sie also die Exceptions Ex1 und Ex2 selbst behandeln und dürfte sie nicht nach außen sichtbar werden lassen. Tritt eine nicht deklarierte Exception auf, wird die Funktion `unexpected`() aufgerufen, die in ihrer Standardimplementation ihrerseits `terminate()` aufruft. Diese zusätzliche Deklaration dient nicht der Unterscheidung von Funktionen bzgl. Überladung!

Behandlung nicht abgefangener Ausnahmen mit eigenen Funktionen

Wie schon für `_new_handler()` können auch für `terminate()` und `unexpected()` eigene Funktionen substituiert werden, indem `set_terminate()` bzw `set_unexpected()` mit einem entsprechenden Funktionszeiger aufgerufen werden. Die set-Funktionen liefern, wie schon `set_new_handler()`, einen Zeiger auf die bisherige Funktion zurück.

```
// Ersatz für void terminate(void)
void term_func(void) { /* ... */ };

// PFV ist Zeigertyp auf void (void) Funktion
typedef void (*PFV)(void);

int main(void) {
  PFV old_terminate;

  old_terminate = set_terminate( term_func ); // neue Fkt setzten
  //...
  set_terminate( old_terminate); // ggf. so alte Fkt. Zurücksetzen
return 0;
}
```

Die Funktion `unexpected()` kann ebenso ersetzt werden.

stack unwinding

Für die Fortsetzung des Programms nach der Behandlung einer Ausnahme gibt es prinzipiell zwei Möglichkeiten:

- Fortsetzung hinter *throw*
- Fortsetzung hinter dem Handler, der die Exception behandelt hat

In C++ hat man sich entschieden, das Program hinter dem Handler fortzusetzen. Daher werden vor dem Aufruf des Handlers alle automatischen Variablen im `try` Block aufgeräumt und ggf. deren Destruktoren aufgerufen (*stack unwinding*).

Tritt dabei (in einem Destruktor) eine weitere Exception auf, wird terminate() aufgerufen. Es kann also stets nur eine Exception aktiv sein.

An dieser Stelle eine Warnung: Da Destruktoren nur für vollständig konstruierte Objekte aufgerufen werden, können beim Auftreten von Exceptions in Konstruktoren von Klassen, die Daten auf dem Heap oder andere Ressourcen verwalten, Speicherlecks oder besetzte Ressourcen zurückbleiben. In diesen Fällen ist es besser, die Klasse so zu definieren, dass ein Objekt zunächst vollständig konstruiert wird, und die kritische Initialisierung durch separaten Aufruf einer Methode *init()* (außerhalb des Konstruktors) erledigt wird. In diesem Zusammenhang wird abermals auf die in [StrouCLP] beschriebene Methode *"resource aquisition is initialization"* verwiesen.

3.2.1 Ausnahmen bei Klassen

Beispiel:

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

class Bruch
{
      int z, n;
public:
      class DividingByZeroException
      {
      public:
            void print()
            {
                  cerr << "DividingByZeroException: n ist 0!\n";
            }
      };

      Bruch(int z, int n):z(z)
      {
            if (n!=0) this->n = n;
            else throw DividingByZeroException();
            cout << "Konstruktor: z/n= " << z <<"/"<< n << endl;
      }

      ~Bruch()
      {
            cout << "\nDestruktor: z/n= " << z <<"/"<< n << endl;
      }
};

void einlesen(int * z, int * n)
```

```
{
      cout << "z= "; cin >> *z;
      cout << "n= "; cin >> *n;
}

int main()
{
      try
      {
            int z, n;
            einlesen(&z, &n);
            Bruch b(z,n);
            einlesen(&z, &n);
            Bruch c(z,n);
      }
      catch(Bruch::DividingByZeroException & e )
      {
            e.print();
            return -1;
      }
      return 0;
}
```

Die Fehlerklasse `DividingByZeroException` wird als eingebettete Klasse deklariert. Im Konstruktor wird bei Nenner `n= 0` die Ausnahme geworfen. Soll das zweite Objekt `n=0` erhalten, dann wird die Ausnahme geworfen. Vorher wird aber für das statische erste Objekt der Destruktor aufgerufen.
Ausgabe:

```
z= 56
n= 7
Konstruktor: z/n= 56/7
z= 12
n= 0

Destruktor: z/n= 56/7
DividingByZeroException: n ist 0!
```

4 RTTI und die neuen Cast-Operatoren

4.1 Die „alten“ Casts (Rückblick und Statusaufnahme)

Casts waren bereits in C nichts Ungewöhnliches (obgleich heiß diskutiert). Sie stehen selbstverständlich auch in C++ weiter zur Verfügung, sowohl in der alten cast-Notation (`(int) 5.3`), als auch in einer neuen funktionalen Notation (`int(5.3)`) . Letztere trägt der Tatsache Rechnung, dass ein cast ein (unärer) Operator ist, den man in C++ mit einer entsprechenden Methode[5] überladen kann:

```
class Bruch {
  double dZaehler, dNenner;
public:
  Bruch( double IniZaehler=0, double IniNenner=1)
   : dZaehler(IniZaehler), dNenner(IniNenner) { }
operator double() { return dZaehler / dNenner; }
operator int() { return int( dZaehler / dNenner); }
};
```

Zu beachten ist an dieser Stelle, dass benutzerdefinierte Typkonvertierungen von Typ A nach Typ B in C++ auf zwei Wegen[6] implementiert werden können: Als cast-Operator in A (`A.operator B()`) oder als Konvertierkonstruktor in B (`B(const A&)`) . Cast-Operator und Konvertierkonstruktor bilden also ein ähnliches Paar wie Zuweisungsoperator und Kopierkonstruktor. Hier wie dort kann es zu Mehrdeutigkeiten kommen, wenn man seine ungewollt Klassen zu großzügig ausstattet. So ist der Konstruktor im obigen Beispiel wegen des Defaultwerts für `dNenner` gleichzeitig ein Konvertierkonstruktor von `double` nach `Bruch`!

Sowohl cast-Operatoren als auch Konvertierkonstruktoren werden vom Kompiler zur Realisierung von impliziten Typkonvertierungen, z. B. bei Zuweisungen oder Funktionsaufrufen (= Zuweisung **Argumente** an **Parameter**), benutzt. Dabei ist zu beachten, dass der Kompiler stets maximal eine implizite (benutzerdefinierte) Typkonvertierung vornimmt:

```
class B { };
class A { public: A(); A(const B&); }; // B->A
class C { public: operator B(); }; // C->B

void f(A);
void main(void) {
  A a;
  B b;
  C c;
  f(a); // OK
  f(b); // OK f(A(b))
  f(c); // Error f( A( (B) c ) ) zwei implizite Konvertierungen
  f( (B) c ); // OK, erste Konvertierung explizit
}
```

[5] Die Überladung mit einer globalen Operatorfunktion ist für casts nicht möglich
[6] Für Konvertierungen in Basis- oder Bibliothekstypen, für die keine Klassendeklaration zugänglich ist, bleibt nur Möglichkeit 2

Bei impliziten Casts gilt wie auch bei Mehrdeutigkeiten, dass der Kompiler nichts tut, was nicht einwandfrei ist. Darum ist zwar eine Zuweisung eines beliebigen Zeigers an `void*` zulässig (kann nicht dereferenziert und missbräuchlich verwendet werden), nicht jedoch umgekehrt. Bei Mehrdeutigkeit muss immer explizit gecasted oder mit dem Scope-Operator qualifiziert werden.

Neben der Unterscheidung zwischen implizit und explizit kann man Casts noch bzgl. Ihrer Wirkung unterscheiden:

- Umkonvertierender Cast (von Werten)
  Hier findet eine echt Umwandlung der betroffenen Daten statt. So wird z. B. ein (2 Byte) `int`-Wert, der in einem Ausdruck nach `float` gewandelt wird, umgesetzt in 4 (Byte) Vorzeichen, Exponent und Mantisse. Diese Art der Konvertierung funktioniert nur, wenn eine entsprechende Konvertierfunktion (Operator oder Konstruktor) definiert ist.

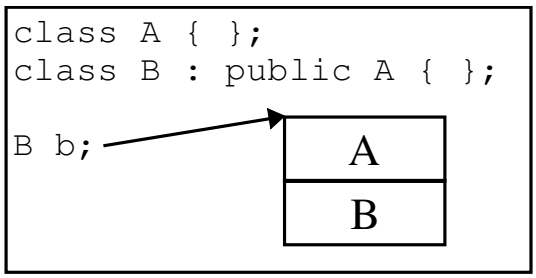
- Uminterpretierender Cast (von Zeigern und Referenzen)
  Hier wird für den Inhalt, auf den der Zeiger verweist, ein neuer Datentyp angenommen. Dabei bestehen ziemlich weitreichende Freiheiten. Eine explizite Konvertierungsfunktion ist nicht nötig. Das Resultat ist jedoch oft „ohne Gewähr“ (vgl. [ARM]) . So kann man z. B. mittels

  ```
  int l = 0x10203040;
  char *pc = (char*) &l;
  for (int i=0; i<4; i++, pc++)
    cout << ?Byte ? << i << ?: ? << *pc << endl;
  ```

  auf die einzelnen Bytes in einem Langwort zugreifen. Allerdings macht man sich hier zu Lasten der Portierbarkeit von der Rechnerarchitektur (*Little-Endian/Intel* vs. *Big-Endian/Motorola*) abhängig (aber vielleicht will man ja genau diesen Unterschied hier sehen). Ansonsten wäre es besser, wenngleich langsamer, durch Schieben und Ausmaskieren nacheinander auf die einzelnen Bytes zuzugreifen (oder wenigstens bedingte Kompilierung zu verwenden).

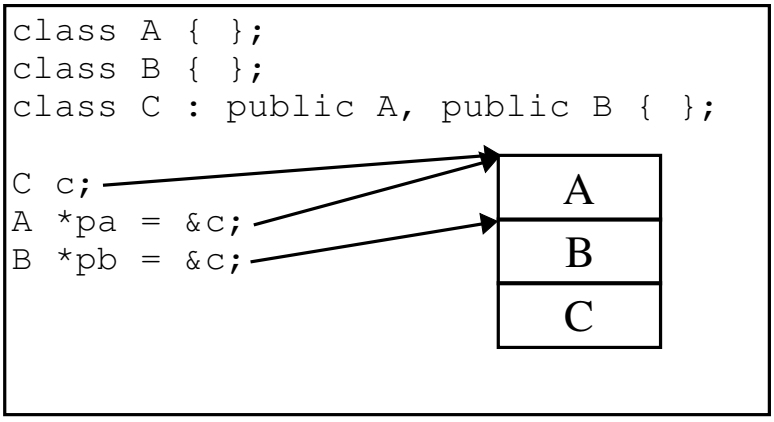
Wenn nicht sehr hardwarenah gearbeitet wird, sollten prinzipiell keine Annahmen über Besonderheiten der Hardware oder des Kompilers, die über die Sprachdefinition hinausgehen, gemacht werden.

Schaut man genauer hin, so erkennt man, dass die beiden Formen eigentlich identisch sind. Tatsächlich wird nämlich auch bei der uminterpretierenden Form ein Wert konvertiert, und zwar der des Zeigers.
Instanzen von abgeleiteten Klassen (bzw. Zeiger auf sie) können bekanntlich an Stelle ihrer `public` Vaterklassen (oder Zeigern auf sie) verwendet werden. Man spricht hier von einem (impliziten) **upcast** (nach oben bzgl. der Klassenhierarchie). Bei einfacher Vererbung sieht dies folgendermaßen aus:



Daten von Vater- und Kindklasse liegen hintereinander im Speicher, Anfangsadresse von Kindobjekt und Vater-Anteil im Kindobjekt starten an derselben Adresse. Bei der Konvertierung muss also der Zeiger nicht verändert werden; es wird einfach nur noch der Vater-Anteil betrachtet. Die Rückwandlung von Vater nach Kind ist nur mittels explizitem **downcast** möglich. Bei Instanzen macht er allerdings keinen Sinn: Bei einer Zuweisung einer Instanz an eine Instanz seiner Vaterklasse wird kopiert, und zwar nur der Anteil der Vaterklasse (für mehr ist ja kein Platz). Der Rest wird praktisch abgeschnitten (*slicing*) und steht zum Zeitpunkt einer Rückwandlung nicht mehr zur Verfügung. Dies ist bei Call-By-Value Schnittstellen grundsätzlich zu beachten!

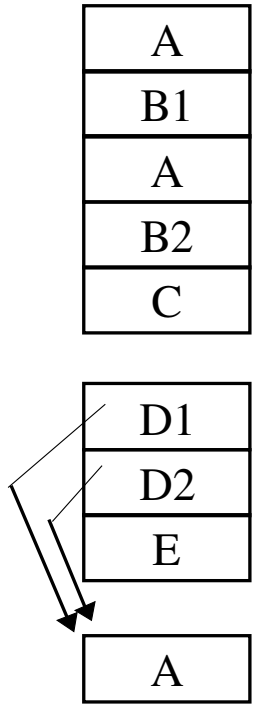
Etwas anders sieht die Sache bei Mehrfachvererbung aus. Hier liegt maximal eine „Erbschaft“ direkt an der Anfangsadresse des Objekts. Wird zu anderen Vorfahren gewandelt, muss also der Zeiger (um einen dem Kompiler bekannten Offset) verändert werden.



Bei der expliziten Rückwandlung wird (mit dem bekannten Offset) der alte Zeiger wieder rekonstruiert.

Problematisch wird das Ganze allerdings bei Vererbungsgraphen wie im folgenden Bild, und zwar unabhängig davon ob virtuell oder normal vererbt wird.

```
class A { };


class B1 : public A { };
class B2 : public A { };
class C : public B1, public B2 { };




class D1 : virtual public A { };
class D2 : virtual public A { };
class E : public D1, public D2 { };

C c;
E e;
A *pa1 = &c; // welches A?
A *pa2 = &e;
```



Zwar macht der upcast von C (mit Qualifizierung) oder E nach A keine Probleme. Der umgekehrte Weg ist jedoch in beiden Fällen nicht möglich. Es fehlt die Information über den Offset. Daumenregel: Der downcast funktioniert immer dann, wenn ein impliziter upcast eindeutig, und das Basisobjekt echt Teil des abgeleiteten Objekts ist.

4.2 Die neuen Cast-Operatoren

Der letzte Abschnitt hat gezeigt, dass casts zu verschiedenen Zwecken eingesetzt werden und nicht immer ganz unproblematisch sind. Im Zuge der RTTI-Erweiterungen von C++ wurden vier neue cast-Operatoren und ein neuer Operator zum dynamischen Abfragen von Typinformationen definiert, die in den folgenden Abschnitten beschrieben sind. Sie sollen zum einen die Lesbarkeit des Codes erhöhen, indem sie Sinn und Zweck des Casts verdeutlichen, und zum anderen einige der Probleme im Zusammenhang mit Casts lösen. Die Problematik des up- und downcasts steht dabei an erster Stelle. Die Syntax der neuen Casts weicht sowohl von der traditionellen als auch der funktionalen Notation ab:

`Syntax: neuer_cast<Zieltyp>(Argument)`

Der alte Cast bleibt nach wie vor gültig, nicht zuletzt aus Gründen der Abwärtskompatibilität,. Zumindest für zukünftige Programme gilt auf jeden Fall die starke Empfehlung, ihn nicht mehr zu verwenden und statt dessen je nach Ziel des Casts einen der neuen cast-Operatoren zu verwenden!

4.2.1 static_cast

Syntax: static_cast<T>(arg)

Die Typen von `T` und `arg` müssen zum Übersetzungszeitpunkt vollständig bekannt sein, eine forward Deklaration reicht also nicht mehr

(aus [StrouDes])

```
class X; // X ist ein unvollständig deklarierter Typ
class Y; // Y ist ein unvollständig deklarierter Typ

void f(X* px) {
  Y* p = (Y*) px; // erlaubt aber gefährlich
  p = static_cast<Y*>(px); // Fehler: X und Y sind nicht vollständig
deklariert
}
```

Dies liegt daran, dass dieser Cast wie der Name schon sagt vollständig zur Übersetzungszeit und nicht etwa während der Laufzeit (`dynamic_cast`, siehe unten) ausgeführt wird. Er ist in erster Linie gedacht als explizite Umkehrung einer impliziten Typumwandlung (etwa von einer abgeleiteten in eine Basisklasse), aber auch für die echte Konversion zwischen Werten verschiedener Typen (sofern entsprechende Umwandlungsfunktionen definiert sind).

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C { /* ... */ };

int main(void) {

  int i = 5;
  float f = 3.5;
  i = static_cast<int>(f);

  C c, *pc = &c;
  B b, *pb;
  A *pa;

  pa = &b;                      // OK implicit upcast
  pb = (B*) pa;                 // OK explicit downcast
  pb = static_cast<B*>(pa);     // OK explicit downcast

  pc = (C*) pa;                 // OK?!
  pc = static_cast<C*>(pa);     // Error
  return 0;
}
```

Er funktioniert immer dann, wenn eine implizite Typumwandlung von `T` nach `arg` existiert, und wenn für einen expliziten Cast in der Gegenrichtung zur Übersetzungszeit genügend Informationen verfügbar sind (d.h. nicht bei virtueller Vererbung oder mehrfachen Erben derselben Klasse). Damit kann er für diesen Anwendungsfall nicht mehr und nicht weniger, als sein traditioneller Vorläufer. Dafür kann sein Ergebnis als sicher betrachtet werden.

4.2.2 const_cast

Syntax: const_cast<T>(arg)

Dieser Cast entfernt eine `const-` oder `volatile-Deklaration` aus einem Typ.

Dass `const` durch einen Cast entfernt werden kann, wurde bereits an anderer Stelle erwähnt. Dies zu tun ist jedoch kein guter Stil. Der `const_cast` wurde eingeführt, um diese besondere Anwendung herauszustellen. **Gleichzeitig wurde die Fähigkeit, const zu entfernen, aus allen anderen neuen cast-Operatoren entfernt. Dies ist also nur mit const_cast möglich.**

Die Entfernung von const ist stets mit Vorsicht zu genießen. Dies zeigt auch das folgende Beispiel:

```
// const2.cpp
// Andreas Rau 10.09.96, abgeändert U.Harms 15.02.02

#include <iostream>
using namespace std;

// als Beispiel fuer eine Klasse
class Int
{
  int i;
public:
  Int(int _i) : i(_i) { }
  Int& operator=(const Int& x) { i = x.i; return *this; }
  operator int(void) const { return i; }
};

void Cheat(const Int kxci)
{
  cout << "Cheating..." << endl;
  cout << "#1 &kxci                  at " << (void*) &kxci << endl;
  cout << "#2 (Int) kxci             at " << &((Int) kxci) << endl;
  cout << "#3 (Int&) kxci            at " << &((Int&) kxci) << endl;
  cout << "#4 Int(kxci)              at " << &(Int(kxci)) << endl;
// E: Cant`t cast
//  cout << "const_cast<Int>(kxci)     at " << &const_cast<Int>(kxci) << endl;
  cout << "#5 const_cast<Int&>(kxci) at " << &const_cast<Int&>(kxci) << endl;
}

void cheat(const int ki)
{
  cout << "cheating..." << endl;
  cout << "#1 &ki                    at " << (void*) &ki << endl;
  cout << "#2 (int) ki               at " << &((int) ki) << endl;
  cout << "#3 (int&) ki              at " << &((int&) ki) << endl;
  cout << "#4 int(ki)                at " << &(int(ki)) << endl;
// E: Cant`t cast
//  cout << "const_cast<int>(ki)       at " << &const_cast<int>(ki) << endl;
  cout << "#5 const_cast<int&>(ki)   at " << &const_cast<int&>(ki) << endl;
}

int main(void)
{
  const Int kxci = 10;
  const ki = 10;

  Cheat(kxci);
  cout << "#1 Int = " << kxci << endl;
  ((Int) kxci) = 20;
  cout << "#2 Int = " << kxci << endl;
  ((Int&) kxci) = (30);
  cout << "#3 Int = " << kxci << endl;
```

```
  Int(kxci).operator=(40);
  cout << "#4 Int = " << kxci << endl;
  const_cast<Int&>(kxci) = 50;
  cout << "#5 Int = " << kxci << endl;

  cheat(ki);
  cout << "#1 int = " << ki << endl;
//  ((int) ki) = 20; // E: LValue required
//  cout << "#2 int = " << ki << endl;
  ((int&) ki) = (30);
  cout << "#3 int = " << ki << endl;
//  int(ki) = 40; // E: Multiple Declarations
//  cout << "#4 int = " << ki << endl;
  const_cast<int&>(ki) = 50;
  cout << "#5 int = " << ki << endl;
  return 0;
}
```

Die Ausgabe des Programms gibt Rätsel auf, wobei nicht klar ist, wieviel davon dem Kompiler anzulasten ist.

```
Cheating...
#1 &kxci                    at 0x23780fc6
#2 (Int) kxci               at 0x23780fbc
#3 (Int&) kxci              at 0x23780fc6
#4 Int(kxci)                at 0x23780fb2
#5 const_cast<Int&>(kxci) at 0x23780fc6
#1 Int = 10
#2 Int = 10
#3 Int = 30
#4 Int = 30
#5 Int = 50
cheating...
#1 &ki                      at 0x23780fc6
#2 (int) ki                 at 0x23780fc6
#3 (int&) ki                at 0x23780fc6
#4 int(ki)                  at 0x23780fc6
#5 const_cast<int&>(ki)     at 0x23780fc6
#1 int = 10
#3 int = 10
#5 int = 10
```

Betrachtet man die Adressen, so sieht man, dass teilweise temporäre non-const Objekte erzeugt werden, wobei die Regel nicht ganz klar wird. Beim anschließenden Versuch, den Cast für Zuweisungen zu missbrauchen, funktioniert aber nicht alles so wie gedacht. Die Klasse verhält sich zwar wie erwartet (Änderungen #2, #4 nur auf dem temporären Objekt), aber der Standardtyp tanzt aus der Reihe: obwohl alle Casts das Originalobjekt zurückliefern sollten, schlägt keine der Änderungen durch, und bei Fall #2 und #4 verschluckt sich der Kompiler.

Der `const_cast` von `const int` nach `int`, der nach Stroustrup ein undefiniertes Ergebnis liefern sollte, wird einfach abgelehnt.

Bleibt abschließend nur noch festzustellen, dass das Entfernen von `const` tunlichst zu vermeiden und im Fall der Fälle durch `const_cast` deutlich gemacht werden sollte.

4.2.3 reinterpret_cast

Syntax: reinterpret_cast<T>(arg)

Die Typen von `T` und `arg` müssen nicht vollständig bekannt sein, sollten aber in keiner Hierarchiebeziehung zueinander stehen. Im Gegensatz zu `static_cast` und `dynamic_cast`, deren stärken in up- und downcasts liegen, ist `reinterpret_cast` für die „Schmutzarbeit“ zuständig. Mit seiner Hilfe lässt sich so ziemlich alles in alles umwandeln, insbesondere Zeiger in Ganzzahltypen und umgekehrt. Im Unterschied zu `static_cast` geschieht dies jedoch ohne Gewähr! Man kann also gewissermaßen Äpfel in Birnen verwandeln, weiß aber bis zum Reinbeißen nicht, ob nicht doch eine Zitrone rausgekommen ist. Die einzige Ausnahme hiervon bildet die Rückkonvertierung in den Originaltyp von arg, die gesichert ist.

Sämtliche Umwandlungen, für die `reinterpret_cast` üblicherweise eingesetzt wird, sind höchst unsicher und stark implementierungsabhängig. Man sollte sie unbedingt durch die Verwendung von `reinterpret_cast` kenntlich machen und dabei noch mal kritisch überlegen, ob sie denn wirklich nötig sind.

***Beispiel**: reinterpret_cast in der Hierarchie*

```
// cast3.cpp
// Andreas Rau 10.09.96, abgeändert U.Harms 15.02.02

#include <iostream>
using namespace std;

class A1
{
  /* ... */
};
class A2
{
  /* ... */
};
class B : public A1, public A2
{
  /* ... */
};

int main(void) {
  A1 *pa1;
  A2 *pa2;
  B b, *pb;

  cout << "Basisklasse A1\n";
  pa1 = &b;
  pb = static_cast<B*>(pa1);
  if (pb == &b) cout << "static_cast ok\n";
  pb = reinterpret_cast<B*>(pa1);
  if (pb == &b) cout << "reinterpret_cast ok\n";
  else cout << "reinterpret_cast fail\n";

  cout << "Basisklasse A2\n";
```

```
  pa2 = &b;
  pb = static_cast<B*>(pa2);
  if (pb == &b) cout << "static_cast ok\n";
  pb = reinterpret_cast<B*>(pa2);
  if (pb == &b) cout << "reinterpret_cast ok\n";
  else cout << "reinterpret_cast fail\n";
  return 0;
}
```

Erzeugt die Ausgabe:

```
Basisklasse A1
static_cast ok
reinterpret_cast ok
Basisklasse A2
static_cast ok
reinterpret_cast fail
```

Der `reinterpret_cast` wandelt also einfach den Zeigertyp ohne irgendwelche Offsets zu verändern! Der Programmierer sagt sozusagen, `pa2` zeige auf ein `B`, und der Kompiler glaubt es einfach. Damit ist zwar einiges möglich, aber die volle Verantwortung liegt nun beim Programmierer.

Beispiel: *Umwandlung von Ganzzahltypen und Zeigern*

```
// cast4.cpp
// Andreas Rau 10.09.96, abgeändert U.Harms 15.02.02

#include <iostream>
using namespace std;

class A;
class B;

int main(void) {
 long l=100;
 float f[2]={50, 70};
 int i;
 char *pc;
 A *pa;
 B *pb;

 // cannot cast, 2 Standardtypen
 // hier static_cast für Konversionen verwenden
 // i = reinterpret_cast<int>(f[0]);
 i = static_cast<int>(f[0]);
 cout << "i=" << i << endl;

 pc = reinterpret_cast<char*>(&f[0]);
 l = reinterpret_cast<long>(pc);
 cout << "l  = " << hex << l << endl;
 cout << "pc = " << (void*) pc << endl;
 cout << "&f = " << (long) &f << endl;

 cout << "?!   " << *reinterpret_cast<float*>(l + sizeof(float));
 cout << endl;
 cout << "?!   " << *((float*)(l + sizeof(float))) << endl;
```

```
 pa = reinterpret_cast<A*>(l);
 pb = (B*) l;
 pb = reinterpret_cast<B*>(pa);
 pa = (A*) pb;
 return 0;
}
```

Der Vergleich mit dem alten cast zeigt, dass `reinterpret_cast` im Grunde nichts Neues kann. Er macht nur im Code explizit deutlich, dass sämtliche Gewährleistungsansprüche nichtig sind.

4.2.4 dynamic_cast

`Syntax: dynamic_cast<T>(arg)`

Bei diesem Cast-Operator muss `T` ein Zeiger oder eine Referenz auf eine definierte Klasse oder vom Typ `void*` sein. Sinn und Zweck ist die Umwandlung zur Laufzeit zwischen polymorphen Objekten in einer Klassenhierarchie, und zwar für die eingangs geschilderten Problemfälle, in denen der `static_cast` mit seinem Latein am Ende ist.

Regeln und Realisierung dieser Umwandlung sind jedoch nicht ganz trivial, weshalb im folgenden versucht wird, die Definition im Standard etwas zu erläutern

aus [StrouCpl]
(1) „Wenn `T` ein Referenztyp ist und `arg` ein Objekt mit `T` als zugreifbarer Basisklasse, liefert der Ausdruck eine Referenz auf das eindeutige Teilobjekt (vom Typ `T`) dieses Objekts zurück.“ Ein upcast also!

„In allen anderen Fällen muss `arg` eine Referenz oder ein Zeiger auf ein polymorphes Objekt sein, also auf ein Objekt, dessen Klasse mindestens eine virtuelle Funktion enthält.“
Hintergrund dieser Bedingung ist folgendes: Für die neue Funktionalität ist Zusatzinformation nötig. Laut der Philosophie von C++ soll aber Ballast für ein Sprachmittel nur dem aufgebürdet werden, der es auch benutzt. Er wird darum in der VMT untergebracht. Da in umfangreichen Hierarchien, in denen `dynamic_cast` benötigt wird, praktisch immer virtuelle Funktionen und damit eine VMT vorhanden ist, ist dies keine Einschränkung.

(2) „Wenn `T` gleich `void*` ist, muss `arg` ein Zeiger sein. Der Ausdruck liefert dann einen Zeiger auf das gesamte Objekt zurück“.
Klingt zunächst trivial, bedeutet aber, dass man quasi in Umkehrung zum obigen Fall einen Zeiger auf das umschließende Gesamtobjekt erhält, ohne innerhalb einer tiefen Klassenhierarchie dessen Typ zu kennen!

„In allen anderen Fällen wird zur Laufzeit geprüft, ob die angeforderte Typkonversion für `arg` zulässig ist. Ist sie es nicht, gilt der Cast als fehlgeschlagen. Bei Konversion in einen Zeigertyp ist das Ergebnis NULL, bei Konversion in eine Referenz wird die Exception bad_cast geworfen.“

Jetzt zum eigentlichen Problem der Laufzeitprüfung:
(3) „Wenn das Objekt, auf das `arg` zeigt, einer Basisklasse eines größeren Objekts angehört und das entsprechende Teilobjekt dieses (größeren) Objekts ist, wird im Fall, dass das größere Objekt vom Typ `T` ist, dieses zurück geliefert."
Dies entspricht also einem `downcast` von der Basisklasse `arg` (enthalten in `T`, vgl. Skizzen) nach unten auf ein Objekt der abgeleitete Klasse `T`. Dies muss nicht das wirkliche, komplette Objekt, sondern kann eine beliebige Zwischenstufe sein.

(4) „Andernfalls wird zunächst das komplette Objekt ausfindig gemacht, auf das `arg` zeigt. Besitzt dieses Objekt eine eindeutige Basisklasse vom Typ `T`, wird das Teilobjekt zurückgeliefert, das dieser Basisklasse entspricht. Gibt es keine Basisklasse vom Typ `T`, schlägt der Cast fehl."
D. h. Fahrstuhlfahren in der Klassenhierarchie: zuerst ganz runter zum echten Objekt, dann wieder hoch zu einer beliebigen Basisklasse. Damit kann man bei Mehrfachvererbung in einen anderen Teilbaum reincasten.

```
// cast5.cpp
// Andreas Rau 10.09.96, abgeändert U.Harms 15.02.02

#include <iostream>
using namespace std;

class A
{
  virtual void gaga(void) { }
};
class B
{
};
class C : public A, public B
{
};

int main(void) {
  C c;
  A *pa;
  B *pb;
  pa = &c;
  pb = &c; // implicit cast to base class
  cout << "original    : " << pb << endl;
  pb = dynamic_cast<B*>(pa);
  cout << "dynamic_cast: " << pb << endl;
  pb = (B*) pa;
  cout << "classic cast: " << pb << endl;
  return 0;
}
```

Die Ausgabe macht die Schwäche des alten Cast an dieser Stelle offenbar: obwohl der Kompiler sein o.k. gibt, liegt das Ergebnis voll daneben. Der `dynamic_cast` liefert also echt neue Funktionalität.

Ausgabe:
```
original    : 006BFDF8
dynamic_cast: 006BFDF8
```

```
classic cast: 006BFDF4
```

Bei einem fehlgeschlagenen Cast von Zeigern liefert `dynamic_cast` den Wert `NULL` zurück, beim erfolglosen Cast von Referenzen wird eine `bad_cast` Exception ausgeworfen. Dynamisches Casten von Instanzen ist nicht möglich bzw. sinnvoll.

4.3 RTTI - Typbestimmung zur Laufzeit

Die Existenz polymorpher Objekte und die Kompatibilität zwischen Zeigern und Referenzen auf solche Objekte stellt den Programmierer manchmal vor die Frage, womit er denn nun eigentlich gerade hantiert. Diese Frage lässt sich naturgemäß nur zur Laufzeit beantworten. *(run time type information (RTTI).* Bisher stand dazu allerdings kein eigenes Sprachmittel zur Verfügung, weswegen in praktisch jeder Bibliothek selbstgebastelte Lösungen zu finden sind, die zwar funktionieren, deren Verwendung aber an einige Bedingungen gebunden ist und die keineswegs untereinander kompatibel sind.

Virtuelle Methoden sind auch der Mechanismus, mit dessen Hilfe man in der Mehrzahl der Fälle die Notwendigkeit umgehen kann, den genauen Typ des Objekts kennen zu müssen! Das Laufzeitsystem wählt automatisch die richtige Methode aus.

Die entscheidende Frage ist oftmals nicht, welchen Typ ein Objekt genau hat, sondern zu welchem Typ es kompatibel ist, bzw. ob es die Eigenschaften (Methode) eines bestimmten Typs zur Verfügung stellt, oder nicht. Hat man diese Frage geklärt (meist mittels `dynamic_cast`), kann die Feinunterscheidung durch Aufruf einer virtuelle Methode erfolgen. Das im folgenden beschriebene neue Sprachmittel zur Ermittlung der Typinformation steht jedoch im Unterschied zu `dynamic_cast` nicht nur bei polymorphen Objekten, sondern immer zur Verfügung.

4.3.1 type_info und typeid()

Zur Vereinheitlichung der Typidentifikation zur Laufzeit wurde ein neuer unärer Operator **`typeid`** definiert. Angewandt auf einen **polymorphen**[7] Typ oder einen Ausdruck, dessen Ergebnis einen solchen Typ besitzt, liefert dieser eine konstante Referenz auf den Typ **`type_info`** zurück, der in **`<typeinfo>`** definiert ist. **Der Ausdruck wird dabei jedoch nicht ausgewertet, d.h. keine Seiteneffekte!**

Der Typ `type_info` speichert minimale, eindeutige Informationen zu jedem Typ **auf implementationsabhängige Art und Weise**. Es ist jedoch garantiert, dass für jede Implementation die Information eindeutig ist und für Vergleiche verwendet werden kann.

Instanzen von `type_info` können nicht kopiert, wohl aber auf Identität verglichen werden. Die Methode `name()` liefert einen **implementationsabhängigen** eindeutigen Namen für den Typ, z. B. zur Ausgabe am Bildschirm. Die Methode `before()` liefert eine **implementationsabhängige** Ordnungsrelation für Instanzen von `type_info`, die nichts mit der Vererbungshierarchie zu tun hat. Sie kann

[7] Einschränkung mit derselben Begründung wie bei `dynamic_cast`

vielmehr verwendet werden, um `type_info` Objekte zusammen mit benutzer-definierten Zusatzinformationen zu jedem Typ in einer Tabelle zu verwalten.

Zum Abschluss nun noch die Regeln zur Anwendung von `typeid()` und einige Beispiele zu seiner Verwendung:

- `typeid` ist polymorph: angewendet auf einen Zeiger oder eine Referenz auf ein polymorphes Objekt wird ein `type_info` für das tatsächliche Objekt zurückgeliefert. Ist der Zeiger NULL wird eine `bad_typeid` Exception erzeugt.
- `typeid` wertet keine Ausdrücke aus: Es wird stets der statische Typ des Ergebnisses zurückgeliefert, das zur Übersetzungszeit ermittelt wird. Seiteneffekte sind nicht möglich.
- `typeid` wertet innerhalb eines Ausdrucks keine Array-Indices aus. Mit der Definition „`T p[n];`“ ist `typeid(p)` äquivalent zu `typeid(T)`, d.h. bei einem Array aus polymorphen Zeigern wird stets der Basistyp zurückgeliefert.
- `typeid` ignoriert die obersten const/volatile (cv) Modifizierer

```
class D {/* ... */};
D d1;
const D d2;

typeid(d1) == typeid(d2);       // true
typeid(D)  == typeid(const D);  // true
typeid(D)  == typeid(d2);       // true
```

Das letzte Beispiel zeigt, dass `typeid` ähnlich wie `sizeof` (auch ein unärer Operator!) sowohl auf Typen als auch auf Ausdrücke/Objekte angewandt werden kann. Auch `sizeof` wertet keine Ausdrücke aus!

Bezüglich der Verwendung von `typeid` ist der obigen Einleitung nichts hinzuzufügen. Von übermäßigem Gebrauch wird abgeraten. Fälle, in denen er wirklich nötig ist, sind so selten, dass lange Zeit überlegt wurde, ob sich seine Einführung überhaupt lohnt!

Beispiel:

```
// cast6.cpp
// U.Harms 15.02.02
// In VCC 6.0 setze (um RTTI zu aktivieren) Kompilerschalter /GR
#include <iostream>
#include <typeinfo>

using namespace std;

class A
{
public:
     virtual int f(){return 1;}
};
class B:public A
{
public:
     virtual int f(){return 0;}
```

```
};
class C:public A
{
public:
     /* ..*/
};

int main()
{
     A a, *pa=&a;
     B b;
     C c;

     cout <<  typeid(a).name() << endl;
     cout <<  typeid(b).name() << endl;
     cout <<  typeid(c).name() << endl;

     pa = &b;
     cout <<  typeid(*pa).name() << endl;
     if (typeid(*pa)== typeid(B))
          cout << "Objekt ist vom Typ B" << endl;

     pa= &c;
     cout <<  typeid(*pa).name() << endl;
     if (typeid(*pa)== typeid(C))
          cout << "Objekt ist vom Typ C" << endl;
     return 0;
}
```

Ergebnis:

```
class A
class B
class C
class B
Objekt ist vom Typ B
class C
Objekt ist vom Typ C
```

5 Die IOStream-Bibliothek

Dieses Kapitel beschreibt die neuen I/O-Funktionen von C++.

Wie auch in C, sind die neuen Sprachelemente für Input und Output nicht Teil der Sprache C++, sondern werden als Bibliotheken geliefert. Dabei sind die alten I/O-Funktionen aus C weiterhin vorhanden.

Es empfiehlt sich aber, die neuen Elemente zu verwenden, da sie eine typsichere, flexible und effiziente Methode für den Input und Output von Zeichen, Integers, Fließkomma-Zahlen, Strings und eigene Datentypen zur Verfügung stellen.

5.1 Das Streamkonzept

Die Sprache C war schon immer sehr stark mit dem Betriebssystem UNIX verknüpft. Unter diesem Betriebssystem ist der Begriff der Datei und insbesondere des Streams (des Zeichenstroms d. h. eine Folge von Zeichen) nicht auf physikalische Dateien auf einem Datenträger beschränkt, sondern findet auch für andere Ein/Ausgabekanäle und Geräte wie den Bildschirm oder die Tastatur Verwendung.

Die neuen Bibliotheksfunktionen bleiben diesem Konzept treu und bauen es unter konsequenter Nutzung der Neuerungen von C++ weiter aus.

Die Schnittstelle zur Stream-Bibliothek ist in der Datei `<iostream>` definiert. Sie hält dabei Klassen bereit, die die Ein- und Ausgabe von Daten über Streams unterstützen.

Die Ein-/Ausgabe in den neuen Bibliotheken von C++ basiert nicht auf Dateien, die über Funktionen mit Daten beschickt werden (`printf(...) / stdin, scanf(...) / stdout`), sondern auf der direkten Übergabe von Daten bzw. Zeichen an Stream-Objekte. Die Bedeutung dieser Zeichen (Buchstaben, Zahlen, Bitmuster) spielt dabei für die Verarbeitung keine Rolle. Die Stream-Objekte kennen dabei diese Bedeutung nicht.

Innerhalb der neuen Headerdateien werden Klassen deklariert, mit deren Instanzen verschiedene Arten von Streams implementiert werden können. Die wichtigsten dieser Klassen sind **`istream`** zur Realisierung von Streams zur Eingabe (input) und **`ostream`** zur Realisierung von Streams zur Ausgabe (output) von Daten.

5.2 Streams und C++

Eine der grundlegendsten Ideen bei dem Entwurf von C++ ist die Möglichkeit, neue Typen zu entwerfen, die genauso bequem verwendet werden können, wie die eingebauten Typen.

Daraus ergibt sich, dass auch die Ein-/Ausgabe von eigenen Typen sich konsequent an die Ein-/Ausgabe der Standardtypen anlehnt. Erst durch die OOP- Konzepte von

C++ wird dies möglich (Klassenbildung, Vererbung, Überlagerung von Funktionen und Operatoren). Dabei beschäftigt sich die Stream-I/O ausschließlich mit der Umwandlung typbehafteter Objekte in Textzeichen-Folgen und umgekehrt.

Es war die Aufgabe des Programmierers der I/O-Bibliothek, die Verbindung von typbehafteten Objekten und im wesentlichen untypisierten Zeichenfolgen, herzustellen.

5.3 Verwendung von Streams für Bildschirm und Tastatur

Das vorliegende Kapitel soll die Fähigkeit vermitteln, Streams für die unformatierte Ein- und Ausgabe an Bildschirm und Tastatur verwenden zu können. Daneben soll ein Einblick gegeben werden, warum manche Vorgehensweise bei der Verwendung von Streams so und nicht anders definiert wurde.

Für die Kommunikation mit der Tastatur und dem Bildschirm werden innerhalb der Headerdateien 4 Stream-Objekte (für die char-Realisierung der Templateklassen) definiert:

`cin` als Instanz der Klasse **`istream`**
`cout` als Instanz der Klasse **`ostream`**
`cerr` als Instanz von **`ostream`**
`clog` als Instanz von **`ostream`**

Diese Streams `cin`, `cout` und `cerr` (`clog` ist eine gepufferte Version von `cerr` für Protokollausgaben, auf die hier nicht näheres eingegangen wird), entsprechen den aus C bekannten Dateien `stdin`, `stdout` und `stderr`.

Zum Austausch von Daten mit diesen (und anderen) Stream- Objekten wurden die Bitshift-Operatoren **<<** und **>>** zu Ausgabeoperatoren bzw. Eingabeoperatoren überladen. Sie bleiben selbstverständlich auch als Bitshift-Operatoren verfügbar. Der Kompiler erkennt ja anhand der übergebenen Parameter, welche Operator-Funktion er aufrufen muss.

Die neuen Operatoren sind aufgrund ihrer Richtung eindeutig hinsichtlich ihrer Bedeutung und für sämtliche Standarddatentypen bereits vordefiniert.

```
cin  >> Eingabevariable;
cout << Ausgabeausdruck;
```

Das Stream-Objekt `cin` übergibt ein Eingabedatum an die Variable `Eingabe-variable` (lvalue), während das Stream-Objekt `cout` ein Ausgabedatum von dem `Ausgabeausdruck` (lvalue oder rvalue) übernimmt.

5.3.1 Stream-Status

Um Fehler im Zusammenhang mit Streams feststellen zu können, besitzt jeder Stream (`istream` und `ostream`) einen Status.
Über diesen Status können Fehler und außergewöhnliche Zustände getestet werden.

Die Verwendung dieser Fehlersignalisierung wird hauptsächlich im Zusammenhang mit der Eingabe von Streams verwendet, weil hier die Eingabefehler des Anwenders abgefangen werden müssen.

Der aktuelle Zustand eines Streams kann durch folgende Elementfunktionen abgefragt werden:

```
  eof()    // gibt true bei end-of-file (ios::eofbit gesetzt)
  fail()   // gibt true bei Fehler (ios::failbit oder ios::badbit
           // gesetzt)
  bad()    // gibt true bei fatalem Fehler (ios::badbit gesetzt)
  good()   // gibt true, wenn Stream OK ist (ios::goodbit gesetzt)

..rdstate()     // gibt die gestzten Flags zurück
..clear()       // löscht alle Flags
..clear(flags)  // löscht alle Flags und setzt flags als Zustand
  setstate(flags)     // setzt zusätzlich flags als Zustand
};
```

Wenn der Status `bad()` oder `fail()` eingetreten ist, so ist im Stream ein Fehler aufgetreten. Der Unterschied zwischen beiden ist der, dass bei `fail()` eventuell nur ein Zeichen verloren gegangen ist. Wobei bei `bad()` für nichts mehr garantiert werden kann.

Bei dem Status `good()` oder `eof()` war die Eingabeoperation erfolgreich. Bei dem Status `good()` ist vermutlich die nächste Eingabeoperation erfolgreich.

Durch die Deklaration innerhalb der Klasse muss bei der Verwendung der Status-Bits der Klassenname **`ios`** gefolgt durch den Scope-Operator vorangestellt werden. z. B.

```
ios::goodbit
ios::badbit
ios::eofbit
ios::failbit
```

Um den Status der Eingabeoperation zu testen, steht folgende Möglichkeit zur Verfügung:

```
int s = cin.rdstate();

if (s & ios::goodbit) { // Alles OK
}
else if (s & ios::failbit) { // vielleicht Zeichen verloren gegangen
}
else if (s & ios::badbit) { // event. Formatierfehler
}
else if (s & ios::eofbit) { // End-of-File
```

```
}
```

Beispiel: Kopierschleife

```
void iocopy (istream &is, ostream &os) {
  char z;
  while (is >> z) os << z <<'\n';
}
```

Diese Funktion kopiert Zeichen eines Eingabezeichenstromes `is` in einen Ausgabezeichenstrom `os`.

Zu Bemerken ist bei diesem Beispiel, dass in der `while`-Anweisung der Stream getestet wird, der von `(is >> z)` gebildet wurde. Bei einem Status `good` wird hierbei ein Wert ungleich Null zurückgeliefert.

Erreicht wird dies wie folgt:
Eine Anweisung der Form

```
cout << "Hallo"
```

oder

```
cin >> z
```

liefert immer eine Referenz auf den Ein-/Ausgabestream zurück, damit die Anweisungen verkettet werden können. Eine Anweisung der Form

```
while (test) {...}
```

verlangt, dass `test` ein zählbarer Typ ist, z. B. `int, char` usw.

Ein Konstrukt der Form

```
while (cout << "Hallo") { .... }
```

bedeutet also, dass die Anweisung `cout << "Hallo"` z. B. in einen Integertyp umgewandelt wird. Explizite Typumwandlung in C++ ist gleichbedeutend mit Verwendung des Cast-Operators. Somit muss für den Datentyp `istream` und `ostream` ein Cast-Operator existieren. Das ist auch so. Der Cast Operator, der den Datentyp `istream` in einen Datentyp `int` castet, liefert also im Erfolgsfall eine Eins zurück, ansonsten den Wert Null.

5.3.2 Ausgabe

Die einheitliche Behandlung von Standard-Typen wie auch benutzerdefinierten Typen lässt sich durch Überschreiben des Operators `<<` ('schreibe in') erreichen. Dazu besteht noch die Möglichkeit, mehrere Objekte in einer einzelnen Anweisung auszugeben.

```
x = 123;
cerr << "x = " << x << '\n';
```

Dabei kann `x` ein Objekt vom Typ `float, int, char` usw. sein. Die Variable `x` kann auch ein eigener benutzerdefinierter Typ sein. Eine Komplexe Zahl würde beispielsweise so ausgegeben werden:

```
x = (1.2, 3)
```

Doch dazu weiter unten mehr.

Die obige Anweisung:

```
cerr << "x = " << x << '\n';
```

ist dabei äquivalent zu

```
cerr << "x = ";
cerr << x;
cerr << '\n';
```

Wie eine solche verkettete Anweisung mit benutzerdefinierten Typen realisiert wird, wird weiter unten noch beschrieben.

Wichtig an dieser Stelle ist noch die Priorität des Operators <<. Dieser Operator besitzt eine sehr geringe Priorität, weshalb Anweisungen wie:

```
cout << "a+b*c" << a+b*c << '\n'
```

zu keinen Problemen führen. Doch schon bei

```
cout << "a^b|c" <<  a^b|c << '\n'
```

gibt es Probleme. Die Verwendung von Klammern an dieser Stelle ist somit zwingend Vorschrift (ebenso bei Verwendung von << als Shift-Operator in einer Ausgabeanweisung).

```
cout << "a^b|c" <<  (a^b|c) << '\n'
```

5.3.2.1 Bindungsrichtung:

Bei einer Verkettung von Ausgaben über den Operator << ist natürlich die Reihenfolge der Ausgabe wichtig, d.h. der Anwender des Operators << muss wissen, in welcher Reihenfolge eine verkettete Anweisung ausgegeben wird.

Bei dem Kapitel über die Überlagerung von Operatoren wird erklärt, wie eine Anweisung der Form:

```
cout << "Hi Leute \n";
```

in einen Funktionsaufruf umgewandelt wird:

```
operator <<(cout, "Hi Leute \n");
```

Genau das selbe geschieht bei einer verketteten Anweisung.

Man überlege sich, wie folgende Anweisung mit dem Operator << in eine Funktions-Form gebracht wird.

```
cout << "Mir geht's gut... " << "Mir auch\n";
```

====>

```
operator<<(operator<<(cout, "Mir geht es gut..."), "Mir auch\n")
```

Zuerst wird die Anweisung in der inneren Klammer ausgeführt. Danach wird der Rückgabewert der 1. Anweisung verwendet, um die 2. Anweisung ausführen zu können.

Daraus folgt, dass mehrere hintereinander stehende Objekte in einer Ausgabeanweisung in der erwarteten Reihenfolge von links nach rechts ausgegeben werden. Für den weiter unten beschriebenen Eingabeoperator gilt das entsprechende!

5.3.2.2 Die Ausgabe von benutzerdefinierten Typen

Bei der Ausgabe von eigenen Typen muss man natürlich zuerst den Operator << überladen und zudem noch dafür sorgen, dass auch der eigene Typ in einer Ausgabeverkettung stehen kann.

Als Beispiel für eine solche Aufgabe wird wieder das Beispiel mit den Komplexen Zahlen betrachtet.

```
class Complex
{
      double re, im;
public:
      Complex(double _re = 0; _im = 0):re(_re), im(_im){}

      double getReal(void) { return re; };
      double getImag(void) { return im; };
// ...
};
```

Der Operator << könnte dann für die Klasse `Complex` versuchsweise folgendermaßen definiert werden:

```
friend ostream operator <<(ostream &s, Complex z) {
      return s << '(' << z.re << ',' << z.im << ')';
}
```

Jetzt kann schon der Inhalt einer Komplexen Zahl ausgegeben werden:

```
void main (void)
{
  Complex x(1.0,2.3);
  cout << "x = " << x << '\n';
}
```

Zu beachten ist hierbei, dass das erste Argument des überlagerten Operators << immer ein Objekt eines Ausgabestreams ist.

Bekanntermaßen ist das erste Argument eines überlagerten Operators der linke Operand. Bei einem binären Operator wird das zweite Argument durch den rechten Operand gebildet wird, wenn der Operator nicht lokal innerhalb einer Klasse überlagert wird, sondern wie hier außerhalb.

Der oben beschriebene Operator << der Klasse Complex kann natürlich noch nicht verkettet werden, weil er noch keinen L-Wert zurückgibt. Aber eine geringfügige Erweiterung zu

```
friend ostream &operator <<(ostream &s, Complex z) {
      return s << '(' << z.re << ',' << z.im << ')';
}
```

liefert den gewünschten Erfolg. Neu ist, dass als Rückgabewert eine Referenz auf das gewünschte Verkettungsobjekt (hier ein Objekt der Klasse ostream) zurückgeliefert wird.

Allgemein ist zu bemerken, dass die Definition eines neuen Ausgabeoperators für eigene Typen keine Änderungen der ostream-Klasse erfordert und keine Zugriffsrechte auf die Datenstruktur voraussetzt.

Daneben kann die Implementierung von ostream verändert werden, ohne dass Anwenderprogramme angepasst werden müssen.

5.3.3 Eingabe

Die Eingabe von Zeichenfolgen ist ähnlich aufgebaut wie die Ausgabe.

Die Klasse istream besitzt einen Eingabeoperator >> ('lese aus'). Ebenfalls kann für die Eingabe von benutzerdefinierten Typen der Operator >> überlagert werden. Auch ist es möglich, Anweisungen für die Eingabe zu verketten.

```
cin >> x1 >> x2 >> x3;
```

Die Trennung der verschiedenen Eingabefelder erfolgt dabei durch *whitespace*-Zeichen. Mit *whitespace*-Zeichen ist dabei das C-Standard-*whitespace* (blank, tab, newline, formfeed, carriage return) gemeint, welches als Ergebnis eines isspace() –Aufrufes aus <ctype> definiert ist.

Alternativ zum Operator >> stehen die get() –Funktionen zur Verfügung:

Die Funktion istream::get(char &c) liest ein einzelnes Zeichen aus dem Eingabestrom.

```
void main (void) // Zeichen-für-Zeichen Kopie
{
char c;
      while (cin.get(c)) cout << c;
}
```

Die Funktion istream::get(char *p, int n, char = '\n') liest maximal n Zeichen in einen durch p bezeichneten Puffer. In jedem Fall wird das Zeichen '\0' hinter das zuletzt gelesene Zeichen in den Puffer gesetzt, so dass höchstens n-1 Zeichen gelesen werden können. Das dritte Argument dieser Funktion spezifiziert einen Terminator, der den default Wert '\n' hat.

Die Funktion `istream::get(char *p, int n, char = '\n')` kann dazu verwendet werden, eine ganze Zeile in einen Puffer zu kopieren.

```
void fkt()
{
  char buf[100];

  //cin >> buf;       // gefährlich
  cin.get(buf, 99);   // sicher
}
```

`cin >> buf` ist deshalb kritisch, weil keine Überprüfung auf einen Überlauf vorhanden ist, und nach 99 Eingabezeichen der Puffer `buf` voll ist.

Als Gegenstück zur `istream::get(char &c)` Funktion existiert noch die Funktion `istream::putback(char &c)`. Diese Funktion dient dazu, ein Zeichen in den Eingabestrom zurückzuschreiben, dass dann das nächste zu lesende Zeichen wird.

```
// eatwhite löscht die whitespace-Zeichen aus einem Stream
istream & eatwhite(istream &is)
{
  char c;
  while (is.get(c)) {
    if (isspace(c)==0) {
      is.putback(c);
      break;
    }
  }
  return is;
}
```

Die Funktion `isspace(char c)` ist eine C-Standardfunktion, die prüft, ob das übergebene Zeichen ein *whitespace*-Zeichen ist oder nicht.

5.3.3.1 Eingabe von benutzerdefinierten Typen

Ähnlich wie die Ausgabe für benutzerdefinierte Typen kann auch die Eingabe von benutzerdefinierten Typen definiert werden:

```
friend istream &operator >>(istream &s, Complex &a)
{
  double re=0, im=0;
  char c=0;
  s >> c;
  if (c == '(') {
    s >> re >> c;
    if (c == ',') s >> im >> c;
    if (c != ')') s.clear(ios::badbit); // Fehlerstatus setzen
  }
  else {
    s.putback(c);
    s >> re;    // Es wurde nur der Realteil eingegeben
  }
  if (s) a = Complex(re, im);
  return s;
}
```

Bei diesem Beispiel wird die Methode `clear(iostate st = ios::goodbit)` von `ios` verwendet. Diese Funktion setzt den Stream-Status mit dem übergebenen Wert. Die Bezeichnung `clear()` scheint zunächst irreführend. Sie wurde deshalb so gewählt, weil sie am häufigsten dazu verwendet wird, einen Stream auf `good()` zurückzusetzen.

Zusammenfassend ein einfaches ***Beispiel*:**

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

int main (void) {
  int in1, in2, in3;

  cout <<"\nGib einen Integer-Wert fuer Tag, Monat und Jahr ein:\n";
  cin >> in1 >> in2 >> in3;
  cout << "Heute ist der " << in1 << "."<<in2<<"."<<in3;

  cout << "\nGib einen Integer-Wert fuer Tag, Monat und Jahr
ein:\n";
  cin >> in1;
  cin >> in2;
  cin >> in3;
  cout << "Heute ist der ";
  cout << in1;
  cout << "." ;
  cout <<in2;
  cout <<".";
  cout <<in3;
  return 0;
}
```

5.4 Formatierung

Bisher wurde kein Wert darauf gelegt, in welcher Form z. B. die Ausgabe stattfand. In vielen Fällen braucht man jedoch die Kontrolle darüber, in welcher Form und Länge eine Ausgabe erfolgen soll.

5.4.1 Formatieranweisungen für die Ausgabe

Speziell für die Ausgabe gibt es zwei Formatierungsoptionen.

- **`width(int)`**
  Legt die minimale Anzahl der Zeichen fest, die für die nachfolgende Ausgabe von Zahlen oder Strings benutzt wird.

  ```
  cout.width(4);
  cout << '(' << 9 << ')';
  ```

  Hier wird (9) in ein 4 Zeichen langes Feld ausgegeben.

  ```
  (9)
  ```

Wenn die Ausgabe der Zahl oder des Strings größer ist als die in `width()` angegebene Breite, dann wird trotzdem der komplette Wert ausgegeben und die Ausgabe nicht gekürzt.

- **`fill(char)`**
  Legt fest, welches Füllzeichen bei einer mittels `width()` angegebenen Ausgabebreite, verwendet wird.

```
cout.width(4);
cout.fill('*');
cout << '(' << 9 << ')';
```

  erzeugt als Ausgabe

```
***(9)
```

Der Aufruf dieser Optionen, beeinflusst nur die unmittelbar nachfolgende Ausgabe-Operation, so dass

```
cout.width(4);
cout.fill('*');
cout << '(' << 9 << ") (" << 9 << ')';
```

folgende Ausgabe erzeugt:

```
***(9) (9)
```

5.4.2 Status des Ein-/Ausgabeformats

Das Ein-/Ausgabeformat lässt sich über Flags steuern. Diese sind wie schon die stream Statuswerte als enums innerhalb von `ios` definiert.

Folgende Flags gibt es bei der Ein-/Ausgabe von C++:

- unverknüpfte Flags

`ios::skipws`	Whitespaces bei der Eingabe überspringen.
`ios::showbase`	Basis eines Integerwertes anzeigen
`ios::showpoint`	Dezimalpunkt und nachfolgende Nullen ausgeben
`ios::uppercase`	Großbuchstaben bei Ausgabe von hexadezimalen u. Gleitkommazahlen
`ios::showpos`	Ausgabe eines '+' bei positivem Zahlen.

Diese Optionen lassen sich über `flags()` setzen.

```
cout.flags(ios::showpos | ios::uppercase);
```

Hier werden die Flags `showpos` und `uppercase` gesetzt. Alle anderen Flags werden auf Null gesetzt.

Will man nur einzelne Flags setzen, ohne dass alle anderen Flags verändert werden, so muss man das Statuswort des Streams mit dem neuen Flag verknüpfen. Um das Statuswort des Streams zu erhalten genügt der Methodenaufruf `flags()`. Hier wird der Inhalt des Status des Streams zurückgeliefert.

```
ios::fmtflags oldFlags = cout.flags();
cout.flags(ios::showpos | ios::uppercase);
cout << hex << 61 << endl;
cout.flags(oldFlags);
cout << hex << 61 << endl;
```

Die Zeile

```
cout.flags (cout.flags() | ios::uppercase);
```

verändert lediglich das `uppercase` repräsentierende Bit. Diese ODER-Verknüpfung erfolgt bei Benutzung von `setf()` automatisch, d.h. obiger Aufruf entspricht

```
cout.setf(ios::uppercase);
```

Das Löschen solcher Flags geschieht analog mit der Methode `unsetf()`.

```
cout.setf(ios::uppercase);
cout << hex << 61 << endl;
cout.unsetf(ios::uppercase);
cout << hex << 61 << endl;
```

- verknüpfte Flags:

Feld positionieren (Flag-Feld: `adjustfield`)

`ios::left`	linksbündig,hinter dem Wert auffüllen
`ios::right`	rechtsbündig,vor dem Wert auffüllen
`ios::internal`	Vorzeichen linksbündig und Wert rechtsbündig;

Basis für Integerwerte (Flag-Feld: `basefield`)

`ios::dec`	Dezimal
`ios::oct`	Oktal
`ios::hex`	Hexadezimal

Fließkommaformat (Flag-Feld: `floatfield`)

`ios::scientific`	Expotentialdarstellung .dddddd Edd
`ios::fixed`	dddd.dd

Puffer entleeren

`ios::unitbuf`	nach jeder Ausgabeoperation
`ios::nonunitbuf`	nicht nach jeder Ausgabeoperation

Ein paar Optionen lassen sich nicht einfach über bestimmte Flags setzen. Dazu zählt z. B. die Basis für Integerwerte. Diese Optionen lassen sich nur durch die Angabe des Flag-Feldes setzen. Dafür existiert eine `setf()` Version mit 2 Parametern.

```
#include <iostream>
#include <iomanip>
using namespace std;
cout.fill('*');
cout.setf(ios::adjustfield, ios::left);
cout << setw(8) << 1.23 << endl;  // ergibt: 1.23****
```

sorgt dafür, dass die Ausgabe von Werten linksbündig erfolgt, während

```
cout.setf(ios::basefield, ios::hex);
```

die Ausgabe von hexadezimalen Werten erlaubt.
Hier noch ein Beispiel zu `ios::floatfield`:

```
cout.setf(ios::floatfield, ios::scientific);
cout << 0.123 << endl;  //ergibt: 1.230000e-001

cout.setf(ios::floatfield, ios::fixed);
cout << 1.0102E2 << endl;  //ergibt : 101.02
```

5.5 Manipulatoren

5.5.1 Verwendung von Manipulatoren

Das Setzen der Ausgabebreite mittels der `width()` Methode ist nicht gerade förderlich für die logische Verbindung zu der Ein-/Ausgabeoperation, da es sich nur für die direkt folgende Ausgabe auswirkt. Dagegen wirken Manipulatoren permanent, d.h. für alle folgenden Operatoren, bis die Formatierung explizit rückgesetzt wird.

Manipulatoren lassen außerdem das direkte Einfügen von Operationen in die Ein- / Ausgabeanweisung zu.

Folgende Manipulatoren werden in den Headerdateien `<iostream>` (Parameterlose Manipulatoren) und **`<iomanip>`** (Parametrisierte Manipulatoren) deklariert.

Manipulator	Parameter	Wirkung
`dec`		Darstellung Dezimal (Basis 10)
`hex`		Darstellung Hexadezimal (Basis 16)
`oct`		Darstellung Oktal (Basis 8)
`endl`		Ausgabe eines End of Line (`'\n'`)
`ends`		Ausgabe eines End of Strings (`'\0'`)
`flush`		Rücksetzen des Streams und Ausgabe des Puffers
`ws`		Whitespace überlesen
`setbase`	`int n`	Zahlen-Basis auf n setzen
`setfill`	`char c`	Füllzeichen definieren entspricht `fill()`
`setprecision`	`int n`	Anzahl Nachkommastellen definieren

`setw`	`int n`	Setzt die Feldbreite, entspricht `width()`
`setiosflags`	`fmtflag l`	Setzt das Formatflag `l`
`resetiosflags`	`fmtflag l`	setzt das Formatflag `l` zurück

Beispiele für Manipulatoren:

- Definieren der Konvertierungsbasis

```
int i = 36;

cout << dec << i << "  " << hex << i << "  ";
cout << oct << i << "  " << dec << i << endl;
```

Ausgabe:

```
36 24 44 36
```

- Spaltenorientierte Formatierung

```
long k1 = 123, k2 = 1234567890;

cout << setfill ('*') << setw (4) ;
cout  << k1 << "  " << k2 << endl;
```

Ausgabe:

```
*123  1234567890
```

5.5.2 Erzeugen eigener Manipulatoren

Natürlich ist es wünschenswert, eigene Manipulatoren zu definieren.

5.5.2.1 Manipulator ohne Parameter

Ziel:

```
cout << tu_was_einfaches << ...
```

Lösung:

Der Bezeichner `tu_was_einfaches` könnte an dieser Stelle ein Standard-Datentyp oder eine eigene Klasse mit überladenem Operator `<<` sein.

Für die Konstruktion eines eigenen einfachen Manipulators erzeuge man bei diesem Ansatz eine eigene Klasse.

Beispiel:

```
class SternKlasse
{
  friend ostream &operator << (ostream &os, SternKlasse &s)
  {
    return os << endl << "**********" << endl;
  }
```

```
};

SternKlasse Sterne;
cout << "Hallo Sterne" << Sterne;
```

erzeugt folgende Ausgabe:

```
Hallo Sterne
**********
```

Die Klasse `SternKlasse` besitzt einen Operator `<<`, der bei einer Anweisung der Form `cout << Sterne` ausgeführt wird.

5.5.2.2 Manipulator mit Parameter

Ziel:

```
cout << f(5) << ...
```

Ansatz:

- Die Funktion `f()` liefert als Rückgabewert irgend ein Objekt (z. B. Standardtyp)
- temporäres `f` Objekt: `f(5)` ist der Konstruktor der Klasse `f`: Liefert als Rückgabe ein Objekt der Klasse `f`
- `f.operator ()(5)`: Liefert als Rückgabe ein beliebiges Objekt, dass von `cout` verarbeitet werden kann.

Lösung:
Man erzeugt eine Klasse, deren Konstruktor einen Parameter erwartet.

Beispiel:

```
class SternKlasse
{
  int no_of_stars;
public:
  SternKlasse (int p=1) : no_of_stars(p) {}
  friend ostream &operator << (ostream &os, SternKlasse &s)
  {
    os << endl;
    for (int i=0; i<s.no_of_stars; i++) os << '*';
    return os << endl;
  }
};
cout << "Hello, world" << SternKlasse(12);
```

erzeugt folgende Ausgabe:

```
Hello, world
**********
```

Die Anweisung `SternKlasse(12)` erzeugt ein temporäres Objekt der Klasse `SternKlasse`. Dieses Objekt besitzt einen überladenen Operator `<<`, der sofort ausgeführt wird.

5.6 Positionierungsanweisungen

Es gibt mehrere Anweisungen, um den Positionszeiger eines Streams zu ermitteln oder zu setzen.

- Aktuelle Stream Position ermitteln

```
long istream::tellg();   // Ermitteln des Lesezeigers
long ostream::tellp();   // Ermitteln des Schreibzeigers
```

- Auf eine absolute Position gehen

```
enum ios::seekdir { beg, cur, end };

istream &ios::seekg (long pos, seekdir);
ostream &ios::seekp (long pos, seek_dir);
```

  Die Methoden `seekg()` und `seekp()` setzen den Positionierungszeiger auf die durch `pos` angegebene Stelle. Dabei wird relativ von der Position ausgegangen, die durch den 2. Parameter angegeben wird.

  Beispiel:

  1.) Positioniere auf einen bestimmten Rekord

```
Inputstream.seekg(10 * sizeof(record), ios::beg);
```

  Hier wird vom Beginn des Streams `Inputstream` (durch `ios::beg`) auf das 10. Element des Datentypes `record` positioniert.

  2.) Gehe auf nächsten Rekordanfang

```
Inputstream.seekg(sizeof(record), ios::cur);
```

  Hier wird von der aktuellen Position (durch `ios::cur`) auf den Anfang des nächsten Elements des Datentyps `record` positioniert.

  3.) Positioniere auf bestimmtes Byte vor dem Ende des Streams

```
Inputstream.seekg(-5, ios::end);
```

  Hier wird der Positionszeiger auf das 5. Byte vor dem Ende des Streams gesetzt.

5.7 Files und Streams

Die Verwendung von IO-Streams ist auch für Dateien möglich. Dabei werden zwei neue Klassen verwendet - **`ifstream`** beim Lesen aus einer Datei - **`ofstream`** beim Schreiben in eine Datei. Die Fähigkeiten beider Klassen sind dabei in der Klasse **`fstream`** verknüpft.

Das Öffnen einer Datei entspricht dabei dem Erzeugen eines zugeordneten Datei-Objektes.

Eine Datei wird für Schreibzugriffe geöffnet, indem ein Objekt der Klasse `ofstream` erzeugt und dem Konstruktor der Dateiname sowie der Eröffnungs-Modus übergeben wird.

```
ofstream OutFile1("output.txt");
ofstream OutFile2("test.txt", ios::app);
```

In ähnlicher Weise wird eine Datei für Lesezugriffe geöffnet, indem ein Objekt der Klasse `ifstream` erzeugt wird.

Die Eröffnungs-Modi können über eine Oder-Verknüpfung verknüpft werden.

Eröffnungs-Modi:

Modus	Bedeutung	Eröffnung zum
`in`	input	Lesen ab Dateianfang (default für `ifstream`)
`ate`	at end	Positionieren auf Dateiende
`binary`	binary	Bearbeiten ohne Aufbereitung
Modus		Eröffnung zum
`out`	output	Schreiben ab Dateianfang (default für `ofstream`)
`app`	append	Schreiben hinter Dateiende
`binary`	binary	Bearbeiten ohne Aufbereitung
`trunc`	truncate	Überschreiben des alten Inhalts

`fstream` Objekte können sowohl zum Lesen, als auch zum Schreiben geöffnet werden.

Beispiel: Einfache Dateibearbeitung

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

int main (int argc, char *argv[])
{
  char ch;

  ifstream FromFile(argv[1]);
  if (!FromFile) {
    cout << "Fehler: Kann Datei nicht öffnen" << endl;
    return 0;
  }

  ofstream ToFile("output.tmp");
  if (!ToFile) {
    cout << "Kann Datei nicht öffnen" << endl;
    return 0;
  }

     // Methode 1                |       // Methode 2
                                 |
  while (FromFile.get(ch)) {     |     while (FromFile >> ch) {
    ToFile.put(ch);              |       ToFile << ch;
  };                             |     };

  if ( !FromFile.eof() || ToFile.bad() ) {
```

```
    cout << "Fataler Fehler aufgetreten" << endl;
  };

  return 0;
}
```

Welcher Unterschied besteht hierbei zwischen der Methode 1 und Methode 2 ?
Bei der Methode 1 werden alle Zeichen kopiert, während die Methode 2 alle Whitespaces überspringt und somit nur Nutzzeichen kopiert.

2. Beispiel:

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

int main()
{
  fstream datei ("output.txt", ios::out | ios::trunc);
  datei.close(); // leere Datei jetzt vorhanden

  datei.open ("output.txt", ios::in | ios::out);
           // Datei ist zum Schreiben und Lesen geöffnet;

  datei << "Hallo wie geht's ?"; // in die Datei schreiben

  datei.seekg(0);     // Dateianfang suchen
  char buf[20];
  datei.getline(buf, 20);  // Zeile lesen
  cout << buf << endl;  // Zeile ausgeben

  datei.close();
};
```

5.8 Strings und Streams

Die vorgestellten Eigenschaften von I/OStreams können auch für Strings verwendet werden. Dafür gibt es in der Standardbibliothek Klassen, welche die Klasse **`strings`** (vgl Kapitel 6) als Streams verwenden. Diese String-Stream-Klassen stehen für das Lesen und Schreiben von `strings` zur Verfügung.
Darüber hinaus gibt es noch aus Kompatibilitätsgründen Stream-Klassen, die die aus C bekannten `char*` Strings (auch *C-Strings* genannt) als Streams einsetzen.

5.8.1 String-Stream-Klassen

Analog zu den Stream-Klassen für Dateien gibt es die folgenden Stream-Klassen für `strings`. Dabei interessiert hier nur die Spezialisierung für 8-bit-Zeichen:

- **`istringstream`** für das formatierte Lesen aus Strings *(input string stream)*
- **`ostringstream`** für das formatierte Schreiben in Strings *(ouput string stream)*
- **`stringstream`** für das formatierte Lesen und Schreiben in Strings
- **`stringbuf`** wird für das low-level Lesen und Schreiben in Strings verwendet. .

Die Eigenschaften dieser String-Stream-Klassen werden in der Headerdatei **<sstream>** definiert. Die neuen Methoden sind:

```
#include <sstream>
string str() const       // liefert den Inhalt des String-Streams
                         // als string Objekt zurück
void str(string &s)      // setzt den Inhalt des String-Streams
                         // auf das string Objekt s
```

Beispiel:

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

int main()
{

     stringstream s;

     s << "HZE an der FHTE" << endl;       // schreiben
     cout << s.str() << endl;              // lesen

     s << "Flandernstr. " << 33 << endl;   // anhängen
     cout << s.str() << endl;              // lesen

     s.seekp(29);                    // 33 suchen
     s << 101 << endl;               // mit 101 überschreiben
     cout << s.str() << endl;        // lesen

     s.str("73732 Esslingen");       // String-Stream neu schreiben
     cout << s.str() << endl;        // lesen

     return 0;
}
```

Ausgegeben wird:

```
HZE an der FHTE

HZE an der FHTE
Flandernstr. 33

HZE an der FHTE
Flandernstr. 101

73732 Esslingen
```

5.8.2 `char*` Stream-Klassen

In C können Ausgaben in einen eigenen Puffer erfolgen. Dies ist auch bei C++ und den IO-Streams möglich. Allerdings müssen hierzu die Klassen **ostrstream**, **istrstream** und **strstream** verwendet werden. Ihre Eigenschaften werden in der Headerdatei **<strstream>** definiert.

Hierbei wird der Stream nicht an ein File gekoppelt wie es im vorherigen Kapitel geschieht, sondern an ein Textzeichen-Array im Speicher.

So können z. B. Botschaften formatiert werden, die nicht unmittelbar ausgegeben werden sollen.

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

int main()
{
  char [80];
  ostrstream ost(p, 80);
  ost << "Hallo wie geht's ?" << ends;
  cout << p;
  return 0;
}
```

Der Vorteil gegenüber `sprintf(...)` und `sscanf(...)` von C besteht darin, das ein Objekt der Klasse `ostrstream` und `istrstream` seine Größe kennt, und gegebenenfalls in den Zustand `ios::fail()` wechselt.

Wenn ein IO-Stream Objekt in diesen Zustand verfällt, dann erfolgt keine Ein- bzw. Ausgabe mehr, bis die Methode `clear()` des Objektes aufgerufen wurde.

5.9 Klassenhierarchie der IOStream-Klassen

Die Abbildung auf der folgenden Seite zeigt die Klassenhierarchie der E/A-Stream-Klassen.

Bei allen Klassentemplates steht in der obersten Zeile der Template-Datentyp und in der folgenden Zeile die Realisierungen für `char` und `wchar_t`.

Von der Basisklasse **`ios_base`** leiten sich alle Streamklassen ab. Sie definiert alle Eigenschaften, die unabhängig von den beiden Realisierungen sind.
`basic_ios<>` definiert die vom Zeichentyp abghängigen Eigenschaften. Die erforderlichen Puffer werden als Objekte der Klasse **`basic_streambuf<>`** eingeführt. (Für Dateien leitet sich daraus `basic_filebuf<>` und für Stringklassen `basic_stringbuf<>` ab.)

`basic_istream<>` und **`basic_ostream<>`** leiten sich virtuell aus `basic_ios<>` ab. Sie sind für Ein- bzw. Ausgabestreams zuständig.
`basic_iostream<>` definiert Objekte, die sich zur Ein- und Ausgabe eignen.

Für die Streamklassen für *Dateien* stehen zur Verfügung:
`basic_ifstream<>` definiert zum Lesen von Dateien *(input file stream).*
`basic_ofstream<>` definiert zum Schreiben von Dateien *(output file stream).*
`basic_fstream<>` definiert zum Lesen und Schreiben von Dateien.

Für *Strings* treten **`basic_istringstream<>`** , **`basic_ostringstream<>`** und **`basic_stringstream<>`** an ihre Stelle.

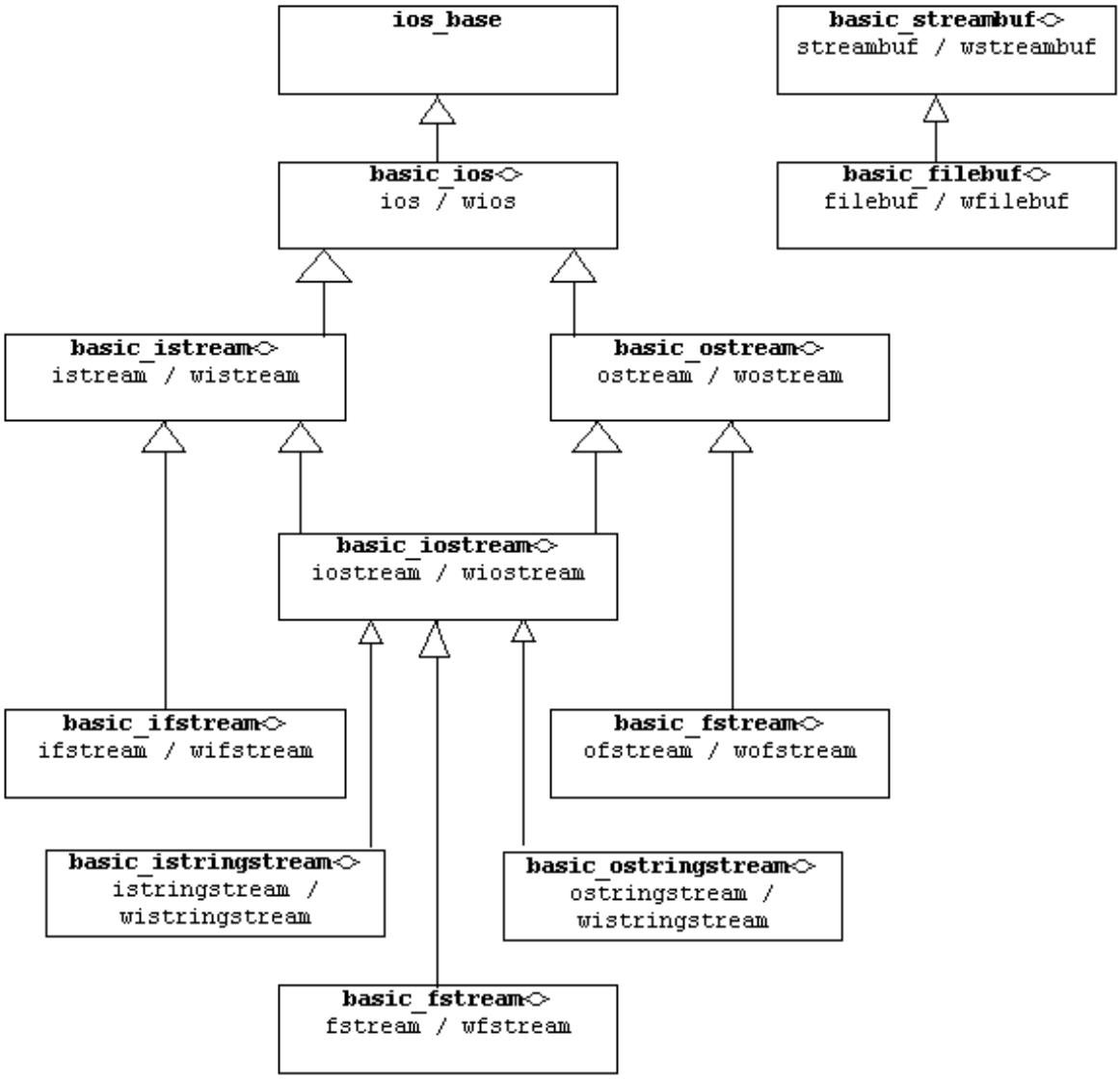


Abb.: Klassenhierarchie der IOStream-Klassen

6 Strings

In C++ gibt es keinen eigenen Datentyp für Zeichenketten. Sie werden realisiert durch char Arrays, die mit dem '\0'-Zeichen enden:

```
char st[]= {'F','H','T','E','\0'};
```

Diesen Zeichenkettentyp nennt man auch **C-Strings**. C-Strings sind unhandlich, da bereits für einfache Operationen wie kopieren und zusammenführen, die Bibliotheksfunktionen `strcpy()` und `strcat()` verwendet werden müssen:

```
char st1[10], st2[15], st3[30];

st1 = "HZE"; // Fehler! Zuweisung nur bei Initialisierung erlaubt!
st2 = "an der FHTE"; // Fehler!
st3 = st1 + st2; // Fehler! +-Operator nicht erlaubt!
```

Stattdessen muss formuliert werden:

```
char st1[10], st2[15], st3[30];

strcpy(st1, "HZE");
strcpy(st2, "an der FHTE");
strcpy(st3, st1);
strcat(st3, st2);
```

Weiterhin sind C-Strings sehr fehleranfällig! Der Programmierer hat beispielsweise bei der Anwendung von `strcpy()` selbst dafür zu sorgen, dass das Arrayende nicht überschritten wird.

Die Standardbibliothek stellt deshalb die Klasse **`string`** zur Verfügung. Implementiert ist sie als Templateklasse `basic_string<>` und verfügt deshalb über die Methoden eines Container-Typs. Weitere Einzelheiten dazu sind dem Abschnitt 6.3 zu entnehmen.

Mit Hilfe der Klasse `string` formuliert man die angeführten Zeichenkettenoperationen wie folgt:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string s1 = "HZE", s2, s3;
  s2 = " an der FHTE";
  s3 = s1 + s2;
  cout << s3 << endl;
  return 0;
}
```

6.1 Konstruktoren und Destruktor

Um mit der Klasse `string` zu arbeiten, hat man Objekte zu instantiieren. Dafür steht eine Reihe von Konstruktoren zur Verfügung:

6.1.1 Ausdruck	6.1.2 Resultat
`string s`	erzeugt einen leeren String `s`
`string s(str)`	`s` wird mit `str` initialisiert
`string s(str, n)`	`s` wird mit den Zeichen von `str` initialisiert, die die Indizes 0 bis `n` haben
`string s(str, n, strlen)`	`s` wird mit `strlen` Zeichen von `str` initialisiert, die mit dem Index `n` beginnen
`string s(cstr)`	`s` wird mit dem C-String `cstr` initialisiert
`string s(chars, n)`	`s` wird mit `n` Zeichen der Zeichenkette `chars` initialisiert
`string s(n, c)`	`s` wird `n`-fach mit dem Zeichen `c` initialisiert
`string s(n, m)`	`s` wird `n`-fach mit dem zu `m` gehörenden ASCII-Zeichen initialisiert
`s.~string()`	`s` wird gelöscht, d. h. alle Zeichen werden gelöscht und der Speicherplatz freigegeben

Beispiele:

```
string vorname;                 // vorname erhält Leerstring
string name("Meier");           // Initialisierung mit "Meier"
string name2(name);             // Initialisierung mit name
string s1(name,3);              // Initialisierung mit ?Mei?
string s2("Esslingen", 4, 3);   // Initialisierung mit "ing"
string s3("Esslingen",3);       // Initialisierung mit "Ess"
string s4(5, '*');              // Initialisierung mit "*****"
string s5(3, 21);               // Initialisierung mit "$$$"
```

6.2 Funktionen

6.2.1 Strings und C-Strings

Ein `string` Objekt kann nicht automatisch in einen C-String umgewandelt werden. Dafür steht die Funktion

```
c_str()
```

zur Verfügung:

Beispiel:

```
std::string s("105");
const char *p;
p = s.c_str()
int i = atoi(p);
```

Die `'\0'`-terminierte Zeichenkette darf nicht verändert werden, da nicht garantiert werden kann, dass diese Zeichenkette noch gültig ist, nach dem `s` verändert worden ist.

Die Funktion

```
copy()
```

lässt sich ebenfalls verwenden. Sie kopiert den Inhalt eines `string` Objektes in ein vorhandenes `char` Array. Allerdings wird das `'\0'`-Zeichen nicht angefügt:

Beispiel:

```
std::string s("Esslingen");
char buffer[80];
s.copy(buffer,80);          // kopiert maximal 80 Zeichen
s.copy(buffer, 80, 2);      // kopiert ab dem 3. Zeichen
```

6.2.2 size() und length()

Die Funktionen

```
size()
```

und

```
length()
```

geben die Anzahl der Zeichen bzw. die Stringlänge mit dem Typ `size_type` zurück. Beide Funktionen sind äquivalent.

Beispiel:

```
std::string s("Esslingen");
size_type i = s.size();    // i ergibt 9
size_type i = s.length();  // i ergibt 9
```

6.2.3 [..] und at()

Mit dem Index-Operator

```
[]
```

und/oder der Funktion

```
at()
```

lassen sich die einzelnen Zeichen des Strings lesen und schreiben. Beide geben das Zeichen zurück, dessen Index übergeben wird. Im Unterschied zu `[]` überprüft `at()` die Gültigkeit des übergebenen Index. Wenn `at()` mit einem unzulässigen Index aufgerufen wird, wird die `out_of_range` Ausnahme geworfen.

Beispiel:

```
std::string s1("0000"), s2("1234");
for (int i = 0; i < 4; i++)
     s1[i] = s2[i];  //
std::cout << s1;      // schreibt "1234"
s1[50];               // Fehler, undefiniertes Verhalten
s1.at(50);      // wirft out_of_range exception
s1[3] = s2.at(0);
```

6.2.4 Stringvergleich

Die üblichen Vergleichsoperatoren

```
==, !=, >, <, >=, <=
```

sind auch für Strings (und C-Strings) definiert.
Darüber hinaus kann man z. B. die Memberfunktion

```
int compare(size_type pos=0, size_type n=npos,
            const string& str); // pos und n gehören zu *this
```

verwenden. (`static const size_type string::npos =-1` gibt die Länge der längsten Zeichenkette an).

Beispiel:

```
std::string s1("FHTE"), s2("Esslingen");
std::cout << (s1 == s2); // Ergebnis: false bzw. 0
std::cout << (s1 > s2); // Ergebnis: true bzw. 1
std::cout << (s1 > "HTE"); // Ergebnis: false bzw. 0
std::string s3("gen"),
std::cout << s2.compare(6,3,s3);// Ergebnis: 0
std::cout << s2.compare(s2);    // Ergebnis: 0
std::cout << s2.compare(s1);    // Ergebnis: <0
std::cout << s3.compare(s1);    // Ergebnis: >0
```

6.2.5 Modifikatoren

Für die Umwandlung von Strings stehen eine Reihe von Member-Funktionen bereit:

1. Zuweisungen mit = und assign()

Beispiel:

```
const std::string s1("FHTE");
std::string s2;
s2 = s1;                    // s2 wird "FHTE" zugewiesen
s2 = "\nFlandernstr. 101"; // s2 erhält einen C-String
s2 = '!';                   // s2 erhält einzelnes Zeichen

s2.assign(s1);              // s2 wird "FHTE" zugewiesen
s2.assign(s1+s2,9,3);       // s2 erhält "der"
s2.assign(s1, 1, std::string::npos); // s2 erhält "HTE"

s2.assign("FHTE",5);        // s2 erhält das char Array:
                            // 'F', 'H', 'T', 'E', '\0'
s2.at(4);                   // Zugriff auf '\0' möglich
s2.assign(3, '§');          // s2 erhält '§', '§', '§',
```

1. Strings entleeren mit erase()

Beispiel:

```
std::string s1("FHTE");
s1 = ??;                    // leeren C-String zuweisen
s1.erase();                 // alle Zeichen löschen
s1.clear();                 // Inhalt leeren (nicht in
                            // VCC 6.0 implementiert)
```

1. Zeichen einsetzen, entfernen und ersetzen: append(), +=, insert(), replace()

Beispiel:

```
const std::string s1("FHTE");
std::string s2;

s2 += s1;                   // "FHTE" anhängen
s2 += "\nKanalstr. 33";     // C-String anhängen
s2 += '\n'                  // Zeichen anhängen

s2.append(s1);              // "FHTE" anhängen
s2.append(s1,1,2);          // "HT" anhängen
s2.append("an",3);          // 'a', 'n','\0' anhängen
s2.append(3, '?');          // '?', '?', '?' anhängen
s2.append('\n');            // nicht implementiert!
s2= "HZE";
s2.append("").append("an der FHTE");  // Verkettung

s2= "an der ";
s2.insert(7,s1);            // s2 enthält "an der FHTE"
s2.insert(0, "HZE "); // s2 enthält "HZE an der FHTE"
s2.insert(0, '\n');         // nicht implementiert
s2.insert(0, " ");          // ist stattdessen zu verwenden

s2.insert((std::string::size_type) 0,1, '\n'); // nur
                            // mit dem cast eindeutig;

s2 = "HZE an der FHTE";
s2.replace(0,3, "Student");     // "Student an der FHTE"
s2.erase(15);               // "Student an der "
s2.erase(7,3);              // "Student der "
s2.replace(0, 7, "Studentin");  // "Studentin der "
s2 += "FHTE";               // "Studentin der FHTE"

s2 = "FHTE";
s2.resize(6, '-');          // ergibt "FHTE--"
s2.resize(8);               // ergibt "FHTE--" mit Länge 8
s2.resize(4)                // ergibt "FHTE";
```

1. Teilstrings: substr()

Mit der Funktion `substr()` lassen sich Teilstrings eines Strings erzeugen:

Beispiel:

```
const std::string s1("HZE an der FHTE");
std::string s2;
s2 = s1.substr();           // s2 erhält Kopie von s1
s2 = s1.substr(4);          // s2 erhält "an der FHTE"
s2 = s1.substr(4,6);        // s2 erhält "an der"
s2 = s1.substr(s.find('F'));    // s2 erhält "FHTE"
```

6.2.6 Suchen mit find(), rfind(), find_first_of() usw.

Für das Suchen inerhalb von Strings stehen eine Reihe von Member-Funktionen bereit:

Beispiel:

```
const std::string s1("HZE an der FHTE\nKanalstr. 33");
std::string s2;
std::cout << s1.find("an");      // ergibt 4
std::cout << s1.find("an", 6);   // ergibt 17
s2 = s1.substr(s1.find("an",6)); // ergibt "analstr. 33"

std::cout << s1.rfind("an");     // ergibt 17, sucht
                                 // von rechts

std::cout << s1.find_first_of("an"); // ergibt 4
// sucht 1. Auftreten von des 1. Zeichens von "an"

std::cout << s1.find_last_of("an"); // ergibt 19
// sucht letztes Auftreten eines der Zeichen von "an"

std::cout << s1.find_first_not_of("an"); // ergibt 0
// sucht erstes Auftreten eines Zeichens nicht in "an"

std::cout << s1.find_last_not_of("an"); // ergibt 37
// sucht letztes Auftreten eines Zeichens nicht in "an"
```

6.3 Die Templateklasse basic_string<>.

Die Klasse **`string`** wird realisiert mit der Templateklasse **`basic_string<>`**:

```
namspace std{
     template<class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
     class basic_string;
}
```

Die Parameter sind der Zeichentyp `charT`, der Charakter (trait) der Zeichen und das Speichermodell.

Die C++ Standardbibliothek stellt zwei Spezialisierungen von `basic_string<>` bereit:

1. die Klasse **`string`** `mit den üblichen Zeichen vom Typ char`
   ```
   namspace std{
     typdef basic_string<char> string;
   }
   ```

1. die Klasse **`wstring`** `mit Zeichen des Type wchar_t`
   ```
   namspace std{
     typdef basic_string<wchar_t> wstring;
   }
   ```
   Damit lassen sich z. B. auch Unicode-Zeichensätze verwenden!

7 Templates

7.1 Motivation für Templates

Oft ergibt sich die Situation, dass die Logik eines Programmteils vollkommen unabhängig ist, von den enthaltenen Typen. Ein Standardbeispiel hierfür ist die Prozedur swap, die zwei Elemente eines beliebigen Typs vertauscht. Es ist extrem langweilig diese Prozedur für jeden Datentyp zu implementieren.

Beispiel ohne Templates:

```
// Datei: TEMPL1.CPP
#include <iostream>
using namespace std;

class Complex
{
private:
  int im, re;

public:
  Complex(int i, int r) : im(i), re(r) {};
  friend ostream &operator << (ostream &os, Complex &c);
  // ....
};

ostream &operator << (ostream &os, Complex &c)
{
  return os << "(" << c.im << "," << c.re << ")";
}

void swap(int &x, int &y)
{
  int temp;
  temp=x; x=y; y=temp;
}

void swap(Complex &x, Complex &y)
{
  Complex temp(0,0);
  temp=x; x=y; y=temp;
}


void main(void)
{
  int i1=10, i2=20;
  Complex c1(10, 40), c2(100, 10);

  cout << "i1=" << i1 << " i2=" << i2 << endl;  // i1=10 i2=20
  cout << "c1=" << c1 << " c2=" << c2 << endl;  // c1=(10,40) c2=(100,10)
  swap(i1, i2);
  cout << "i1=" << i1 << " i2=" << i2 << endl;  // i1=20 i2=10
  swap(c1, c2);
  cout << "c1=" << c1 << " c2=" << c2 << endl;  // c1=(100,10) c2=(10,40)
}
```

Was man gern hätte, sind generalisierte Funktionen, die unabhängig vom Datentyp formuliert werden, wobei der Kompiler die Funktion für jeden Datentyp automatisch in Maschinencode umsetzen soll. Der Programmierer wird dann entlastet, die Prozedur für jeden Datentyp manuell zu schreiben.

Was man also braucht, ist eine Schablone, mit der der Kompiler für jeden Datentyp automatisch die erforderliche Prozedur generieren kann. Dieses Problem ist nicht neu. Bereits im Rahmen der Programmiersprache Ada gibt es ein Sprachmittel für solche Schablonen. Dort heißt es „Generische Programmeinheiten“. C++ hat diese Idee aufgegriffen. In C++ heißen diese Schablonen „Templates“:

Ein anderes Beispiel ist eine Klasse für eine Tabelle zur Verwaltung von ints. In einem anderen Zusammenhang braucht man eine solche Tabelle für float. Hier ist es also das Problem, dass man generalisierte Daten braucht, d.h. eine Tabelle, die vom Datentyp unabhängige Daten verwaltet (Die Funktion zum Sortieren dieser Tabelle ist ein generalisierte Funktion).

Auch ohne Templates ist es möglich, Funktionen oder Objektklassen so zu gestalten, dass sie auf verschiedene Datentypen anwendbar sind. Hierzu werden vor allem Makros oder auch Zeigermechanismen eingesetzt.

```
// naiver Ansatz, unsicheres Makro,
// arbeitet auch mit Äpfel und Birnen
#define SWAP(T, x,y)                \
{ T temp;                           \
temp=x; x=y; y=temp; }

// typsicheres Makro, muss instantiiert werden
#define SWAP(typ)                  \
void swap(typ &x, typ &y) {        \
  typ temp;                        \
                                   \
  temp=x; x=y; y=temp;             \
}

// deklariert swap(...) mit int
SWAP(int);

// deklariert swap(...) mit complex
SWAP(complex);

void main(void)
{
  //...
  SWAP(int, i1, i2) // ok
  SWAP(complex, c1, c2) // ok
  SWAP(int, i1, c1) // Aua!!!
  swap(i1, i2);  // ruft swap(int, int) auf
  swap(c1, c2);  // ruft swap(complex, complex) auf
  swap(i1, c1);  // Kompiler entdeckt Typfehler
}
```

Oft bergen aber derlei Konstrukte schwer zu durchschauende Risiken, sind sehr umständlich anzuwenden oder nur schwer nachzuvollziehen - haben sie schon einmal ein Makro mit dem Debugger untersucht ?. Templates bieten im Gegensatz hierzu eine sichere und standardisierte Möglichkeit, um Funktionen und

Objektklassen unabhängig von den zur Anwendung kommenden Datentypen zu gestalten.

Der Einsatz von Schablonen ist für Funktionen und für Klassen möglich.

Der Begriff Template

Das Wort template steht im Englischen für die deutschen Wörter Form, Modell und auch Schablone, was dem Sinn der Verwendung dieses Wortes innerhalb der Terminologie der Programmiersprache C++ am nächsten kommt, denn Templates ermöglichen es dem Programmierer, einen sonst nötigen, manuellen Kopier- und Änderungsvorgang innerhalb des Programmquelltextes zu automatisieren.

Chronologie der Templates

Eine Implementierung des Template--Gedankens mit Hilfe von Makros wurde von Bjarne Stroustrup schon 1986 in [1] vorgestellt. Er nannte die so geschaffenen Gebilde generic classes also Grundklassen oder besser Klassenerzeuger. Eine Weiterentwicklung wurde durch seinen Beitrag “Parameterized Types for C++" zur USENIX C++ Conference 1988 in Denver vorgestellt. Seit Juli 1990 sind Templates Teil des ANSI C++ Standards und in [9] beschrieben.

7.2 Funktionstemplates

Eine Schablone kann nicht direkt aufgerufen werden. Man kann aber durch sie Funktionen erzeugen, die Schablonenfunktionen genannt werden.

7.2.1 Definition und Deklaration von Funktionstemplates

In C++ läßt sich eine Schablone durch das Schlüsselwort template definieren. Vor der Definition der Schablone (Schablone für Funktionen = Funktionsschablone) muss das Schlüsselwort template angegeben werden. Bei der Definition der Schablone wird offen gelassen, welchen Typ die Parameter haben. Beim Aufruf der Funktion, werden aktuelle Werte übergeben. Anhand des Typs der Werte wird erkannt, welche Version der Funktion benötigt wird. Das Offenlassen des Datentyps, der zur Laufzeit verwendet werden soll, wird dem Kompiler durch die Angabe eines generischen Datentyps mitgeteilt, welcher für die generischen Parameter und ggfs. den Rückgabewert verwendet wird. Die Angabe dieses Datentyps bzw. dieser Datentypen erfolgt in spitzen Klammern.

Bsp.

```
template <class typname>
```

```
typname min ( typname a, typname b)
{
  return (a < b) ? a: b;
}
```

Das Schlüsselwort class zeigt an, dass typname für einen noch zu spezifizierenden Typ steht. Beachten Sie, dass das Template min eine Typüberprüfung durchführt, ob beide übergebenen Objekte vom selben Typ sind.
Statt <class typname> kann auch <typename typname> angegeben werden.

Das Makro

```
#define min(i,j) ((i<j)? i:j)
```

hingegen führt keine Typprüfung durch.

Werden Makros so wie im Beispiel SWAP oben geschrieben, so hat man einen unvollkommenen Template-Ersatz in der Sprache C, der wenigsten eine Typprüfung auch für Makros erlaubt. Allerdings ist die Schreibweise sehr umständlich, jeder Typ muss von Hand eingeplant werden, eine automatische Generierung der Funktionen für verschiedene Datentypen findet nicht statt.

Definition einer Schablonenfunktion

Eine Schablonenfunktion wird vom Kompiler auf der Basis der Schablone erstellt. Der Kompiler definiert die Schablonenfunktion, wenn im Programmtext ein Aufruf der Schablone mit konkreten Parametern erfolgt.

So legt der Kompiler für das obige Beispiel eine Schablonenfunktion für den Datentyp int an, wenn die Schablone wie im folgenden mit int-Parametern aufgerufen wird:

```
int alpha;
int beta;
int gamma;

gamma = min (alpha, beta);
```

Es wird dann in der Schablone typname durch int ersetzt. Wie bereits erwähnt, erfolgt die Definition der Funktion und das Anlegen von Maschinencode erst dann, wenn der Kompiler an die Stelle des Aufrufs kommt.

Allgemeine Definition einer Funktionsschablone

Funktionsschablonen (nicht zu verwechseln mit Schablonenfunktionen!) werden folgendermaßen definiert:

```
template <class T1, ...., class Tn>
Ergebnistyp Funktionsname (T1 par_1, T1 par_2, ... Tn par_n)
{
  ....//Funktionsrumpf
}
```

Deklaration einer Funktionsschablone

Wird eine Schablone aufgerufen, obwohl die Definition der Schablone erst später erfolgt, so muss vor dem Aufruf eine Deklaration der Schablone erfolgen.
Die Deklaration sieht folgendermaßen aus:

```
template <class T1, ...., class Tn>
Ergebnistyp Funktionsname (T1 par_1, T1 par_2, ... Tn par_n);
```

7.2.2 Regeln für Funktionstemplates

(Ähnlichkeiten zu den Regeln für Funktionen sind **nicht** zufällig...)

- Für eine Funktion darf es nur genau eine Schablonen-Definition geben. Beim Aufruf einer Funktion werden nur die aktuellen Parameter zur Ermittlung der Schablone herangezogen, der Ergebnistyp der Funktion wird nicht berücksichtigt.
- Werden Funktionsschablonen überladen, so müssen sich die einzelnen Definitionen in der Parameterliste unterscheiden.

Beispiel:

```
// Datei: TEMPL4.CPP

template <class T>
static void swap(T &x, T &y)
{
  T temp;
  temp=x; x=y; y=temp;
}
```

Bei einem Funktionsaufruf wird das Templateargument (also der jeweilige Datentyp) nicht explizit angegeben, sondern der zur Anwendung kommende Datentyp wird durch den in C++ zur Verfügung stehenden Funktionsüberladungsmechanismus (function overloading) implizit erkannt, also einfach „swap(i1, i2)“.

- Alle generischen Datentypen in den spitzen Klammern der Schablonendefinition müssen mindestens einmal in der Parameterliste der Funktionsschablone verwendet werden. Können Sie sich denken, warum?

- Beim Funktionsaufruf muss die Zuordnung der Typen der aktuellen Parameter zu den generischen Typen eindeutig sein. Mit anderen Worten, in der Schablone gebe es zwei Übergabeparameter mit demselben generischen Datentyp. Dann darf die Schablone nicht mit zwei verschiedenen aktuellen Datentypen aufgerufen werden. Bei der Suche nach einem passenden Template finden keine impliziten Typumwandlungen statt!

```
// Datei: TEMPL5.CPP

template <class T>
void swap(T &x, T &y)
```

```
{
  T temp;
  temp=x; x=y; y=temp;
}

void main(void)
{
  int i1=10;
  char c1=20;
  swap(i1, c1);     // Kompiler meckert, weil er nicht
                    // swap(int, char)  findet
}
```

- Templates können explizit extrahiert werden:

```
 swap(int, int); // extrahiert swap für ints

 void main(void)
 {
 //...
 swap(i1, c1);    // jetzt existiert bereits eine Funktion,
                  // und c1 kann konvertiert werden
 }
```

- Die Namen der Datentypen in der Schablonen-Deklaration müssen nicht identisch mit den Namen in der Schablonen-Definition sein

```
template <class typname>
typname min( typname a, typname b);   // Deklaration

template <class T>                        // Definition
T min ( T a, T b)
{
  return (a < b) ? a: b;
}
```

Funktionsschablonen dürfen außer den Parametern mit generischen Datentypen auch noch andere Parameter mit konkreten Datentypen haben. Natürlich gilt auch hier, dass beim Funktionsaufruf die Zuordnung der Typen der aktuellen Parameter zu den generischen Typen eindeutig sein muss.

Beispiel für Spaßvögel: n-mal vertauschen von 2 Objekten:

```
// Datei: TEMPL6.CPP

template <class T>
void swap(T &x, T &y, int n)
{
  T temp;
  while (n--)
  {
    temp=x; x=y; y=temp;
  }
}

void main(void)
{
```

```
  int i1=10;
  int i2=20;

  swap(i1, i2, 10);     // 10 mal 2 ints vertauschen
}
```

- Werden Funktionsschablonen mit den Schlüsselwörtern extern, inline und static deklariert bzw. definiert, so muss die Angabe dieser Schlüsselwörter direkt vor dem Funktionsnamen erfolgen.

7.2.3 Explizite Qualifizierung von Templates

Oft ist es durch Mehrdeutigkeiten bei den Parametern nicht möglich, beim Aufruf das richtige Template eindeutig zu identifizieren. Ist bereits ein Template mit kompatiblen Typen instanziiert, kann dieses (erwünscht oder unerwünscht) aufgerufen werden. Um das gewünschte Template gezielt auszuwählen, bietet sich zunächst entsprechendes Casten der Parameter an. Besser ist es jedoch, das Template (wie bei Klassentemplates verbindlich vorgeschrieben) explizit durch Angabe seiner Template-Argumente zu qualifizieren.

z. B.

```
min (f, (float) i)      // min(float, float), sagt aber nicht alles

min<float>(f, i)        // min(float, float), als template markiert
```

7.2.4 Regeln für die Suche einer passenden Funktion

Folgende Reihenfolge gilt bei der Suche der passenden Funktion für einen gegebenen Aufruf und ist entscheident dafür, ob und welches Template aufgerufen wird.

1) Zuerst wird nachgeschaut, ob eine konkrete normale Funktion mit den entsprechenden Schnittstellen vorliegt. Die Überprüfung erfolgt an Hand der Parameterliste. Dabei muss allerdings eine exakte Übereinstimmung der aktuellen Parameter mit den Typen der formalen Parameter bestehen.

2) Existiert keine solche Funktion, so wird die Schablone aufgerufen. Dabei muss - wie Sie wissen - die Zuordnung der Typen der aktuellen Parameter zu den generischen Typen eindeutig sein.

3) Paßt auch dies nicht, dann wird geschaut, ob es eine konkrete Funktion gibt, die zwar in den Parametern nicht exakt mit dem Aufruf übereinstimmt, die es aber von den Parametern her erlaubt, dass sie gemäß dem Regelwerk der impliziten Typkonvertierung aufgerufen wird. Diese Funktion kann bereits zu einem früheren Zeitpunkt aus einem Template extrahiert worden sein

7.2.5 Spezialisierung / Überladung von Templates

Ein Template passt meistens nur für fast alle Typen. Für einige (z. B. char*) müssen oft Speziallösungen implementiert werden.

Regel 1) bietet den erforderlichen Einstiegspunkt, um Schablonen für Spezialfälle außer Kraft zu setzen, indem man eine spezielle Funktion explizit formuliert.

z. B.

```
void swap(char* s1, char* s2)
{
  char* st = new char[max(strlen(s1), strlen(s2)) + 1];
  strcpy(st, s1);
  strcpy(s1, s2);
  strcpy(s2, st);
  delete[] st;
}
```

Dies ist jedoch keine Spezialisierung des Templates im eigentlichen Sinne, sondern nutzt nur den obigen Mechanismus aus. Die so definierte Funktion wird immer übersetzt, auch wenn sie niemand aufruft (auch wenn sie der Linker später dann weglässt). Um ein Template explizit zu spezialisieren, gibt es eine eigene Notation:

```
void swap<char*>(char*, char*);
```

bzw. bei mehreren Template-Parametern

```
template <class T, class V>
void func(T, V);                    // allgemeines Template

template <class T>
void func<T, int>(T, int)           // Spezialisierung
```

7.2.6 Kompilerspezifische Besonderheiten

Je nach Kompiler kann die strenge Typprüfung der Parameter der Schablone etwas aufgeweicht sein. So kann es beispielsweise sein, dass const int als gleichwertig mit int behandelt wird.

7.3 Klassentemplates

Betrachten wir das folgende Beispiel eines Stacks aus Zeichen:

```
#include <iostream>

class Zeichenstack
{
private:
  int groesse;
  char * top;
  char * basis;
public:
```

```
  Zeichenstack (int g);       // Konstruktor
  ~Zeichenstack ();                 // Destruktor
  void push (char c);
  char pop ();
};

Zeichenstack::Zeichenstack (int g)   // Konstruktor
{
  top = basis = new char[groesse=g];
}

Zeichenstack::~Zeichenstack ()           // Destruktor
{
  delete [] basis;
}

void Zeichenstack::push (char c)
{
  *top++ = c;                              // entspricht: *top = c; top++;
}

char Zeichenstack::pop ()
{
  return *--top;
}

void main ()
{
  Zeichenstack stack1 (100);
  Zeichenstack stack2 (200);
  stack1.push ('a');
  stack2.push(stack1.pop());
  char zeichen = stack2.pop();
  std::cout << "\nDas Zeichen war: " << zeichen << std::endl;
}
```

Der Stack wird hier nicht als verkettete Liste implementiert, sondern als dynamisches Array von Zeichen. Eine entsprechende Datenstruktur kann man sich auch für int, float, double etc. schaffen. Allerdings ist die Schreibarbeit sehr lästig und beim Abschreiben hat man immer gute Chancen, Fehler zu produzieren, deshalb ist eine generische Klasse der bessere Ansatz. Eine generische Klasse ist nicht vollständig definiert. Bei den Definitionen innerhalb einer solchen Klasse kann anstatt eines festen Typs ein Bezeichner verwendet werden. Dieser Bezeichner ist Platzhalter für einen Typ. Erst bei der Definition eines konkreten Datenobjektes wird der Platzhalter durch eine konkrete Typangabe ersetzt.

Hier gleich die Lösung:

7.3.1 Definition und Deklaration von Klassentemplates

```
#include <iostream>

template <class T>
class Stack
{
private:
  int groesse;
```

```
  T * top;
  T * basis;
public:
  Stack (int g);        // Konstruktor

  ~Stack ()       // Destruktor
  {
    delete [] basis;
  };

  void push (T c);

  T pop ()
  {
    return *--top;
  }
};

template <class T>
Stack <T>::Stack (int g)   // Konstruktor
{
  top = basis = new T[groesse=g];
}

template <class T>
void Stack <T>::push (T c)
{
  *top++ = c;
}

void main ()
{
  Stack <int> Stack1 (100);
  Stack <int> Stack2 (200);
  Stack1.push (20);
  Stack2.push(Stack1.pop());
  int zahl = Stack2.pop();
  std::cout << "\nDie Zahl war: " << zahl << std::endl;

  Stack <double> Stack3 (100);
  Stack <double> Stack4 (200);
  Stack3.push (24.99);
  Stack4.push(Stack3.pop());
  double dzahl = Stack4.pop();
  std::cout << "\nDie Zahl war: " << dzahl << std::endl;
}
```

Es wird hier eine Klasse definiert, in der ein generischer Parameter benutzt wird:

```
template <class T>
class Stack {...};
```

T ist ein Parameter, der für einen Datentyp steht. Innerhalb der Klassendefinition kann T als Typangabe bei der Definition von Funktionen und Variablen verwendet werden. Die Entscheidung über den konkreten Typ findet erst bei der Definition eines Objektes der Klasse Stack statt.

Beachten Sie, dass der Name der Klasse dieser Schablonen-Definition Stack <T> ist. Dieser Name wird verwendet zur Definition von Memberfunktionen außerhalb der Klassenschablone (eine solche Funktion ist ja eine Funktionsschablone!):

```
template <class T>
T Stack <T>::pop ()
```

Bei der internen Definition von Methoden muss nichts spezielles beachtet werden, wie der Vergleich zwischen Konstruktor/Destruktor bzw. `push()/pop()` zeigt.

Um ein Klassenobjekt zu definieren, muss für den Parameter T der Schablonen-Definition ein konkreter Typ angegeben werden:

```
Stack <double> Stack3 (100);
```

Damit wird für das Datenobjekt Stack3 festgelegt, dass alle Definitionen mit T innerhalb der Klasse als double interpretiert werden.

Eine template-Klasse ist die Definition eines Musters für eine Gruppe von Klassen.

Der Kompiler erzeugt für jede Version der Klasse, die benötigt wird, den entsprechenden Maschinencode. Für jede Klasse werden die entsprechenden Objekte und Funktionen angelegt.

Bei der Instantiierung von Klassentemplates muss im Gegensatz zu Funktionstemplates der spezielle Typname explizit angegeben werden:

```
Stack<int> intStack;
```

7.3.2 Spezialisierung von Klassentemplates

Wie Funktionstemplates können auch Klassentemplates spezialisiert, d.h. für bestimmte Datentypen individuell implementiert werden. Man kann allerdings nicht nur einzelne Methoden der Klasse spezialisieren, sondern die Klasse muss als Ganzes für den bestimmten Datentyp implementiert werden.

Für eine explizite Spezialisierung muss man vor der Klassendeklaration template<> schreiben und den speziellen Template-Datentyp hinter dem Klassennamen angeben:

```
template<>
class Stack<std::string>
{
  ...
};
```

Entsprechend der Klassendeklaration muss auch jede Methode deklariert werden:

```
template<>
void Stack<std::string>::push (const std::string& elem)
{
  *top++ = elem;
}
```

Die spezialisierte Implementation muss nicht die gleichen Datenstrukturen wie das allgemeine Template verwenden, eine völlig andere Implementierung ist möglich.

Klassentemplates können auch partiell, d.h. teilweise spezialisiert werden. Das heißt, dass einer der allgemeinen typename- bzw. class - Parameter durch einen speziellen Typ ersetzt wird.

Das Klassentemplate

```
template <typename T1, typename T2>
class EineKlasse
{
  ...
};
```

kann z. B. folgendermaßen partiell spezialisiert werden:

```
// beide Typen sind gleich
template <typename T>
class EineKlasse<T,T>
{
  ...
};
```

oder

```
// der zweite Typ ist ein int
template <typename T>
class EineKlasse<T,int>
{
  ...
};
```

Die Klassen werden nun folgendermaßen instantiiert:

```
EineKlasse<int,double> dieklasse1;       // EineKlasse<T1,T2>

EineKlasse<double,double> dieklasse2;          // EineKlasse<T,T>

EineKlasse<double,int> dieklasse3;       // EineKlasse<T1,T2>
```

Passen mehrere Spezialisierungen gleich gut, so ist der Aufruf mehrdeutig und ein Fehler.

7.3.3 Default-Template-Parameter

Wie für Funktionsparameter können auch für Templateparameter Default-Werte definiert werden.

```
template <class T = int>
class Stack
{
  ...
};
```

Instantiiert werden die Klassen dann so:

```
Stack MeinIntStack;                        // Stack für int-Werte

Stack<double> MeinDoubleStack;             // Stack für double-Werte
```

7.3.4 Werte als Template-Parameter

Template-Argumente (sowohl bei Funktions- als auch bei Klassentemplates) können auch normale Werte sein.

Bsp:

```
template <class T, int SIZE>
class Stack
{
private:
  T elements[SIZE];
  T * top;
  T * basis;
public:
  Stack ();              // Konstruktor
// (ohne übergebenen Größen-Parameter)

  ~Stack ()       // Destruktor
  {
    delete [] elements;
  };

  ...
}
```

Dies hätte in diesem Fall den Vorteil, dass die Speicherverwaltung statisch ist, d.h. schneller als dynamische Speicherverwaltung.

Außer Typen dürfen Template-Argumente nur konstante Ausdrücke oder Adressen von Objekten oder Funktionen sein, auf die im Programm global zugegriffen werden kann. Die Argumente dürfen keine Objekte und nicht lokal sein.

7.4 Wie Templates den Präprozessor arbeitslos machen

Durch die Einführung der Templates ist er nun praktisch vollends entmachtet:

- Durch const wurde ihm das Monopol auf symbolische Definiton fester Werte entrissen
- inline Funktionen machten seinen direkt am Ort plazierten schnellen Makros typsichere Konkurrenz
- Templates graben der Variabilität seiner Makros typsicher das Wasser ab

Einzig für wilde Anwendungen, die echte Textersetzung unter Verwendung von „#“ und „##“ erfordern ist der Programmierer noch gezwungen, sich auf Abenteuer mit ihm einzulassen.)

8 Literatur

[ISO] Information Technology-Programming Languages-C++
ISO/IEC 14882-1998
ISO/IEC, 1998

[JosuttisCpp] Nicolai M. Josuttis
Objektorientiertes Programmieren in C++. Ein Tutorial für Ein-und Umsteiger.
Addison-Wesley, München, 2001 (2. Auflage)

[JosuttisSTL] Nicolai M. Josuttis
The C++ Standard Library. A Tutorial and Reference.
Addison-Wesley, Reading, 1999

[StrouCPL] Bjarne Stroustrup
Die C++ Programmiersprache
Addison-Wesley, München, 2000 (4. Auflage. Deutsche Übersetzung der Special Edition)

[StrouDes] Bjarne Stroustrup
Design und Entwicklung von C++
Addison-Wesley, München, 1994

[ARM] Margaret A. Ellis, Bjarne Stroustrup
The Annotated C++ Reference Manual (ARM)
Addison-Wesley, Reading, 1990