Notizen zu C++

Peter Thömmes
Version: 09. August 2003

p.thoemmes@surf25.de
www.notizen-zu-cpp.de

Inhaltsverzeichnis

1. Einordnung von C++

Um **1968** entwickelten Brian **Kernighan** und Dennis **Ritchie** bei AT&T (Bell Labs) die **prozedurale Programmiersprache C**. C enthält nur Makros, Zeiger, Strukturen, Arrays und Funktionen.

Bjarne **Stroustrup** stellte **1985** in seinem Buch "The C++ Programming Language" die darauf aufbauende **objektorientierte Sprache C++** vor, die er bei AT&T (Bell Labs) entwickelt hatte. C++ erweitert C um die Objektorientierte Programmierung (OOP), Referenzen (Referenzparameter statt Zeiger), Templates (Vorlagen, Schablonen) und Exceptions (Ausnahmen/Ausnahmebehandlung).

Alexander **Stepanov** und Meng **Lee** konnten **1994** das Ergebnis ihrer langjährigen Forschungsarbeiten im Bereich der Standard-Algorithmen (in den Hewlett-Packard Labs) erfolgreich als **STL** (Standard-Template-Library) in das ISO/ANSI-C++-Werk einbringen (Im July-Meeting 1994 stimmte das ANSI/ISO C++ Standards Committee diesem zu). Die STL ist je nach Software-Hersteller unterschiedlich implementiert. Es ist daher **nicht** immer alles bei jeder Implementierung möglich, was in der ISO/ANSI-STL-Beschreibung geschrieben steht.

Dieses Buch hat zum Ziel, dem Programmierer, der **C++** bereits kennt, zu helfen "sein C++" zu optimieren. Die einzelnen Kapitel können getrennt voneinander betrachtet werden. So kann der Programmierer bei der täglichen Arbeit, lernen seine Aufgaben sicher und effektiv zu lösen und vorhandenen Code zu verstehen oder zu verbessern.

Die enthaltenen Code-Beispiele sind plattformunabhängig, auch wenn z.B. bei der Ansteuerung relationaler Datenbanken mal ein spezielles Beispiel für Linux oder für Windows dabei ist.

Im Voraus sollte schon einmal gesagt werden, dass es ratsam ist sich im Bereich der dynamischen Heap-Speicherung gründlich der Funktionalität der **STL** zu bedienen. Was die STL bietet, ist in der Regel **kaum zu verbessern**, wenn es um dynamisches Heap-Management geht. Außerdem bleibt der Code **portabel**, denn die STL gibt es unter jedem Betriebssystem.

• Standardisierungs-Dokumente

DIS (Standard von ISO-ANSI)

Draft Proposed International Standard for Information Systems - Programming Language C++

→ Offizielle detailgenaue Beschreibung von C++

Doc No: X3J16/95-0087 WG21/N0687
Date: 28 April 1995
AT&T Bell Laboratories
ark@research.att.com

ARM (Buch von Margaret A. Ellis und Bjarne Stroustrup)

Annotated C++ Reference Manual

→ Ursprüngliche C++-Referenz vom April 1990

Addison-Wesley, ISBN 0-201-51459-1

• Version von Plattform und Compiler/Linker

Bevor man mit der Arbeit beginnt, ist es wichtig die genaue Version von Plattform und Compiler/Linker zu kennen, um gegebenenfalls alle Programmierer des Projektes auf den gleichen Stand (version, patches, service packs, option packs, ...) zu bringen. Die Beispiele in diesem Buch sollten allerdings unabhängig davon überall funktionieren.

Beispiele für die Erfragung der Version mittels Konsole (Kommandozeile):

UNIX-Plattform:

`more /etc/issue`
→ System-Name wie z.B.
`'Red Hat 7.2 (Enigma) ...'`

`uname -a`
→ Detaillierte Informationen über Kernel und Maschine wie z.B.
`'... 2.4.7-10 ... i686 ...'`

WINDOWS-Plattform:

`ver`
→ System-Name und Version wie z.B.
`'Microsoft Windows 2000 [Version 5.00.2195]'`

GNU C++ Compiler/Linker:

`g++ -v`
→ Bsp: `'... 2.95.3-5 ...'`

Microsoft C++ Compiler/Linker:

`cl`
→ Bsp: `'... Version 13.00.9466 for 80x86 ... 1984-2001'`

2. Grundsätzlicher Aufbau eines Projektes

2.1 Pro Klasse eine *.h und eine *.cpp-Datei

Man sollte in einem Projekt **für jede Klasse 2 Dateien** pflegen: Eine **Header-Datei** (`*.h`) mit der Deklaration und eine **Implementierungs-Datei** (`*.cpp`) mit dem Quell-Code der Implementierung der Klasse.

Ausnahme: **Abstrakte Klassen** und **Templates**
→ Hier gibt es keine Implementierung → **nur eine Header-Datei**.

Header-Datei (`*.h`):
Man sollte der Übersichtlichkeit wegen zunächst die public-Methoden aufführen, dann die restlichen Methoden und dann die Daten, die am besten immer nur private oder protected sind. Auf jeden Fall sollten die Member-Variablen der Klasse alle mit **`m_`** beginnen. Wenn es der Übersichtlichkeit dient, kann man die inline-Implementierung unterhalb der Deklaration, also außerhalb der Klasse anhängen. Makros sind zu vermeiden (besser `const-inline`-Templates) und statt `#define` sind Konstanten (`const`) zu verwenden. Auf keinen Fall ist der `include`-Wächter zu vergessen.

```
#ifndef _MYHEADER_H_       //include-Wächter
#define _MYHEADER_H_

#include ...

class ...; //forward-Deklaration

class MyClass
{

      //Member-Funktionen (Methoden):
      public:
            MyClass();
            ...
      protected:
            ...
      private:
            ...

      //Member-Variablen:
      private:
            enum{ NUM_ELEMENTS = 1};
            static const double m_dFACTOR;
            int m_nByteCnt;
            unsigned char m_abyElelements[NUM_ELEMENTS];
            ...
};
inline void MyClass::...
{
      ...
}
#endif //inlcude-Wächter
```

Implementierungs-Datei (`*.cpp`):
Die Implementierung einer Klasse beinhaltet den Code der Member-Funktionen. Man sollte darauf achten, dass der Code einer Member-Funktion nicht zu groß wird. Dadurch wird der Code übersichtlich und wartbar gehalten. Member-Funktionen mit 1000 Zeilen sollten auf jeden Fall zum verbotenen Bereich gehören. Member-Funktionen mit 50 Zeilen liegen auf jeden Fall im grünen Bereich.
Bei der Implementierung der Methoden ist Folgendes grundsätzlich immer zu beachten:

- Direkt am Anfang einer Methode immer prüfen, ob alle Parameter ok sind
  → ggf. return
  ```
  bool MyClass::Func(const string& strText) const
  {
        if(strText.empty())
              return false;
        ...
     return true;
  }
  ```
  → Hierdurch vermeidet man unnötige Konstruktionen, Speicherallokierungen und Destruktionen und erhöht die Performance

- Objekte immer erst dort instanziieren, wo sie gebraucht werden
  → Weniger Konstruktionen/Speicherallokierungen/Destruktionen (Performance)

- Instanziierungen von Objekten innerhalb von Schleifen vermeiden (besser vor der Schleife)
  → Weniger Konstruktionen/Speicherallokierungen/Destruktionen (Performance)

- Laufvariablen nach Möglichkeit im Schleifenkopf definieren
  ```
  for(unsigned short i = 0;i < 20;++i)
  {
        ...
  }
  ```
  → Hierdurch vermeidet man, dass eine Schleife innerhalb einer anderen Schleife mit der gleichen Variablen arbeitet, denn der Compiler meldet einen Fehler bei nochmaliger Verwendung der gleichen Definition.

- Prefix-Operatoren statt Postfix-Operatoren benutzen
  → Keine temporären Objekte (Performance)

- Wenn möglich Read-Only-Member-Funktionen benutzen
  → Klare Trennung zwischen den Methoden, welche das Objekt (bzw. den Wert des Objektes), zu dem sie gehören, nicht verändern und anderen Methoden

Grundsätzlich sollte man seine Implementierung immer **von vornherein** optimiert gestalten, denn wenn man es später tut, macht man sehr viel von der getanen Arbeit nochmal.

2.2 Benennung von Verzeichnissen, Dateien und Klassen

Man sollte eine Datei (`*.h` oder `*.cpp`) immer so benennen wie die Klasse, die darin zu finden ist:

Klasse **`MyClass`** → **`MyClass`**`.h`
`MyClass``.cpp`

Um die **selbst geschriebenen Klassen** besser von denen einer Bibliothek unterscheiden zu können, sollte man die eigenen Klassennamen **ohne Präfix** oder mit einem **eindeutigen Präfix** beginnen lassen ('**`My`**' oder '**`M`**' oder ...). Man sollte aber **nie das Präfix verwendeter Bibliotheken** (Bsp.: Microsoft: '`C`', Borland: '`T`') **für die eigenen Klassen nutzen**. Wenn man die eigene Klassenbibliothek als Firma auf dem freien Markt vertreibt, sollte man auf jeden Fall für die veröffentlichten Klassen ein **eindeutiges Präfix** benutzen. Hatte man bis dahin z.B. '`My`' als Präfix benutzt, dann ist das Suchen/Ersetzen sehr einfach, wohingegen das vorherige Arbeiten ohne Präfix einiges an Arbeit nach sich ziehen kann.

Um ohne eine mühselige Umbenennung der Klassen und Dateien auszukommen, wenn man selbst den Code in einem anderen Projekt wiederverwenden will, sollte man es tunlichst **vermeiden, den Projektnamen mit in die Namen der Klassen aufzunehmen**. Man sollte lediglich den **Haupt-Verzeichnisnamen** und den **Namen der ausführbaren Datei** entsprechend dem **Projektnamen** benennen.

Falsch: `Robot/RobotGUI.cpp`
`Robot/RobotGUI.h`

Richtig: `Robot/GUI.cpp`
`Robot/GUI.h`

Weiterhin kann es sehr hilfsreich sein, wenn man bei einer Spezialisierung durch public-Vererbung die abgeleiteten Klassen mit dem Basisklassen-Namen als Präfix versieht. Dies verschafft Übersicht und hilft beim Suchen.

Beispiel:

Parent: `View`
Child1: `FormView`
Child2: `Data1FormView`, `Data2FormView`, `Data3FormView`

Sucht man also nach allen 'Views' (`*View.h`, `View.cpp`), bekommt man alle Klassen. Sucht man hingegen nur nach 'FormViews' (`*FormView.h`, `*FormView.cpp`), erhält man nur alle ab Child1...
Member dieser Klassen benennt man angelehnt an die ungarische Notation sinnvollerweise mit einem entsprechenden Präfix.

Beispiel:

```
Data1FormView formviewData1;
Data2FormView formviewData2;
```

2.3 Zentrale Header-Datei (Settings.h)

Um für das Projekt spezifische Präprozessor-, Compiler-, Linker- und Bibliotheks-Optionen zu setzen, sollte man eine zentrale Header-Datei (**`Settings.h`**) haben. Hier kann man auch alle `#include`-Anweisung der verwendeten C++-Standard-Bibliotheken (Bsp.: `#include <stdio.h>`) unterbringen, um sie nicht immer wieder in allen anderen Dateien einfügen zu müssen. Außerdem sind hier alle **projektweit eindeutig zu haltenden globalen Konstanten** unterzubringen. Diese Datei muss dann in jedes Modul (`*.h` und `*.cpp`) eingebunden werden.

2.4 Der Code muss ohne Warnungen bauen

Man sollte sich angewöhnen, seinen Code immer so zu schreiben, dass der Compiler keine Warnungen beim Bau der Software bringt. Der Warning-Level des Compilers sollte dabei immer auf Maximum stehen, denn nur so erfüllt die Warnung ihren Zweck: sie soll etwas außergewöhnliches darstellen, und den Programmierer daran erinnern, dass er noch etwas an seinem Code ändern muss.

Grundsätzlich ist es immer möglich ohne Warnungen zu bauen. Eine Ausnahme ist der Bau mit fremdem Quell-Code. So kann zum Beispiel der Bau mit einer portablen STL sehr viele Warnungen des gleichen Typs generieren:

```
warning:
      identifier was truncated to '255' characters in the debug
      information.
warning:
      unreferenced inline function has been removed.
```

Um überhaupt noch die eigenen Warnungen zu finden, schaltet man dann am besten gezielt die STL-Warnungen ab. Wie dies zu tun ist, entnimmt man dem Handbuch des Compilers.

Beispiel für das Abschalten von Warnungen:

Die Datei `test.cpp` enthält eine leere `main`-Funktion und eine nicht genutzte statische Funktion:

test.cpp:

```
static bool GetTrue()
{
      return true;
}

int main()
{
      return 0;
}
```

GNU C++ Compiler:

Nach dem Bau mit höchsten Warning-Level (**`-Wall`**):

```
g++ test.cpp -Wall
```

ergibt sich folgende Warnung:

```
warning: 'bool GetTrue()' defined but not used
```

Das Abschalten der Warnung ist mit einer speziellen `-Wno`-Option möglich:

```
g++ test.cpp -Wall -Wno-unused
```

Microsoft Visual C++ Compiler:

Nach dem Bau mit höchsten Warning-Level (**`/W4`**):

```
cl test.cpp /W4
```

ergibt sich folgende Warnung:

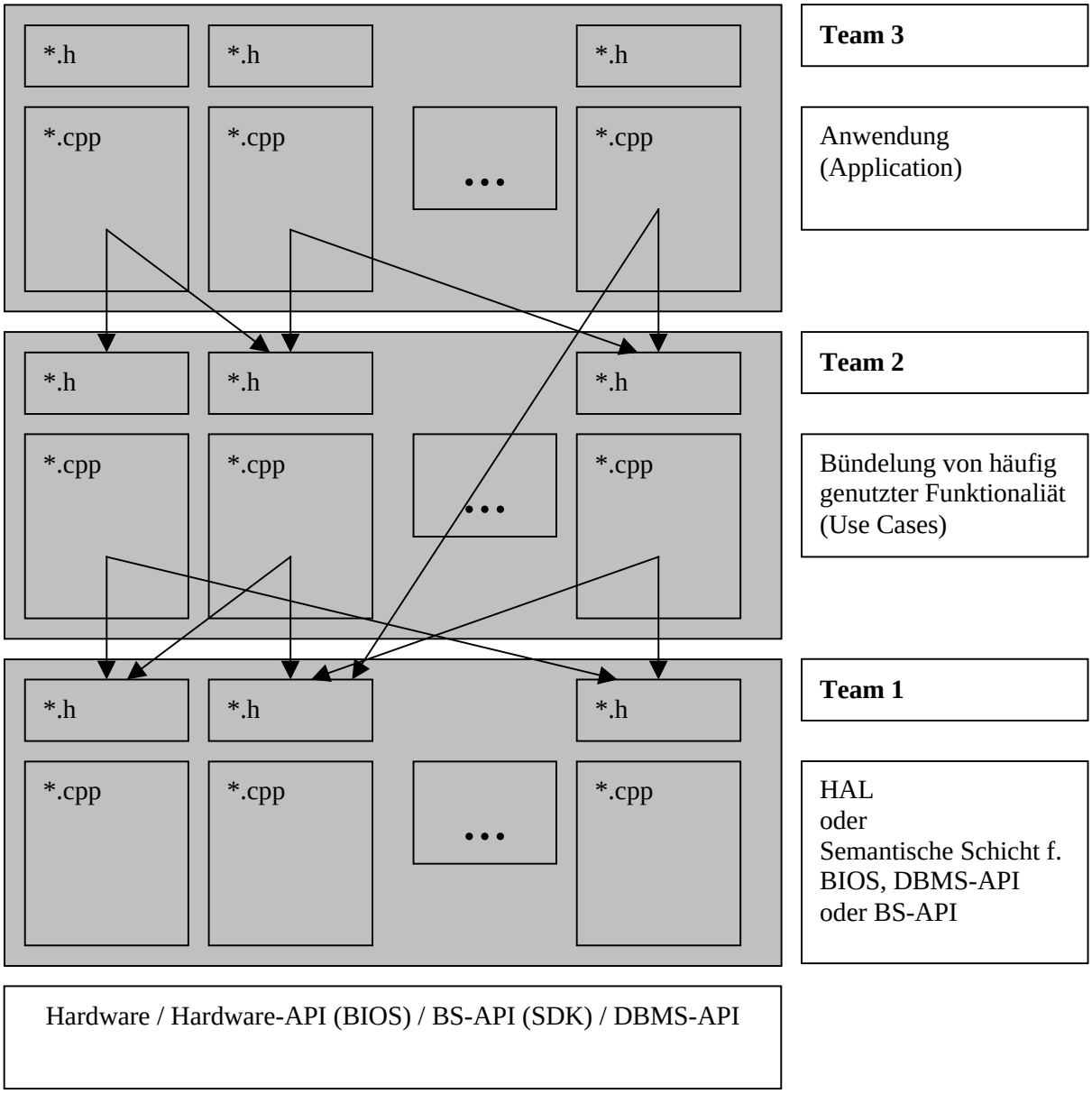
```
warning C4505:
  'GetTrue': unreferenced local function has been removed
```

Das Abschalten der Warnung ist mit einer speziellen `#pragma-Anweisung` möglich, die man vor die Funktion in den Code schreibt:

```
#pragma warning(disable:4505)
static bool GetTrue()
{
      return true;
}
```

2.5 Mehrere Schichten verwenden (horizontale Teilung)

Grundsätzlich sollte man komplexe Software-Projekte in mehrere Schichten (Layer) aufteilen:



Eine Schicht ist als Sammlung von Klassen zu verstehen, die nur von Klassen der darüberliegenden Schichten verwendet werden darf, d.h. es gibt nur in Klassen der darüberliegenden Schichten eine entsprechende `#include`-Anweisung.

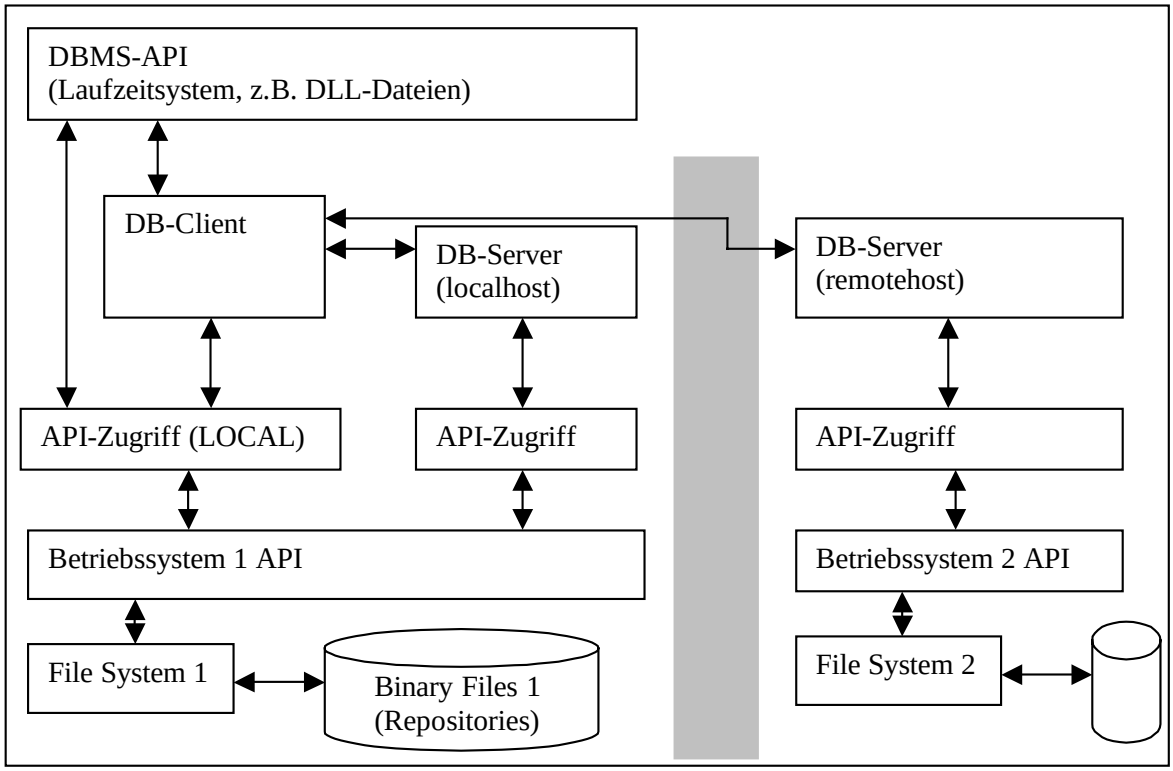
Anders ausgedrückt bedeutet dies, dass eine Klasse nur Klassen der darunterliegenden Schichten sieht und kennt. Wenn nicht genügend Funktionalität von einer unteren Schicht zur Verfügung gestellt wird, kann man auch schon mal eine Schicht überspringen und sich der darunterliegenden Funktionalität bedienen.

Die unterste Schicht hat nur die Hardware (bei BIOS-Programmierung) / das BIOS (bei Betriebssystem-Programmierung) / das BS-API (bei Anwendungs-Programmierung) oder das DBMS-API (bei Datenbank-Client-Programmierung) vor Augen und stellt damit ein HAL (Hardware-Abstraction Layer) bzw. eine semantische Schicht zur Verfügung. Um auf ein BS-API (Betriebssystem-API wie WIN32-API von Windows) zugreifen zu können, benötigt der Compiler das **SDK** (Software-Development-Kit) des Betriebssystems. Dort werden die API-Funktionen bekannt gemacht und können in einem Software-Projekt verwendet werden.

Man kann sich also bei der Programmierung einer Klasse im Schichten-Modell auf die richtige Verwendung der darunterliegenden Klassen beschränken und muss nicht das Gesamtproblem vor Augen haben. So ist es möglich, dass 3 Software-Teams mit unterschiedlich ausgerichteten Software-Experten relativ unabhängig voneinander an einem Gesamt-Projekt mit 3 Schichten entwicklen. Diese Unabhängigkeit der Teams endet an den jeweiligen Schnittstellen. Die Festlegung der Schnittstellen erfordert das Hineindenken in die Arbeit der anderen Teams.

Man kann ein Software-Konzept auf unterschiedlichen Detailstufen erarbeiten (Requirements, Specification, High Level Design und Detailed Level Design).

Die Grobstruktur (High-Level-Design) eines **Database Management Systems** könnte z.B. wie folgt aussehen:

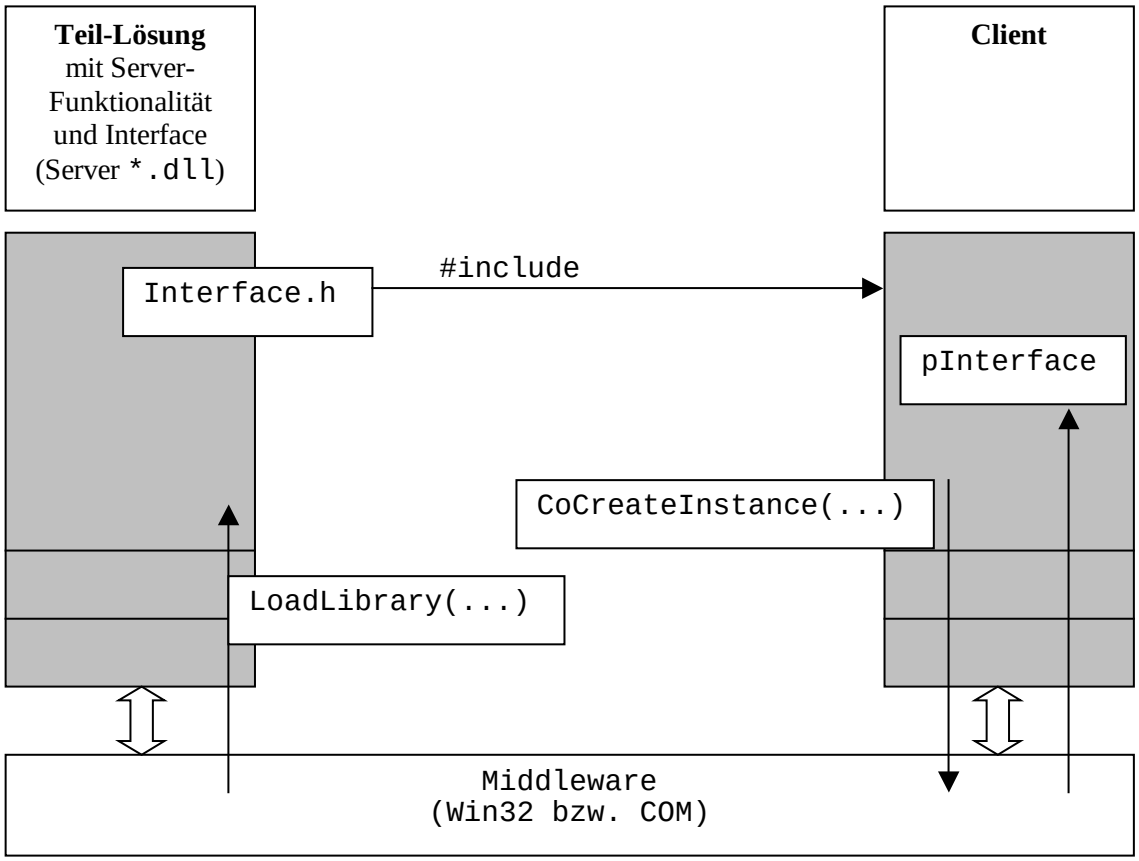


2.6 Client/Server-Modell verwenden (vertikale Teilung)

Man kann eine Anwendung nicht nur horizontal (in Schichten) sondern auch vertikal teilen. Man schreibt also für abgeschlossene Teilaufgaben jeweils eine Insel-Lösung (Server), welche von einem oder mehreren Clients genutzt werden kann. Die eigentliche Hauptanwendung kann hierbei als reiner Client implementiert werden, der sich nur der einzelnen Server bedient um seine Funktionalität zu implementieren. Wichtig ist die sinnvolle Festlegung der Interfaces der Server.

Beispiel:

Microsoft Inprocess-COM-Server unter 32-Bit-Windows (nur schematisch):

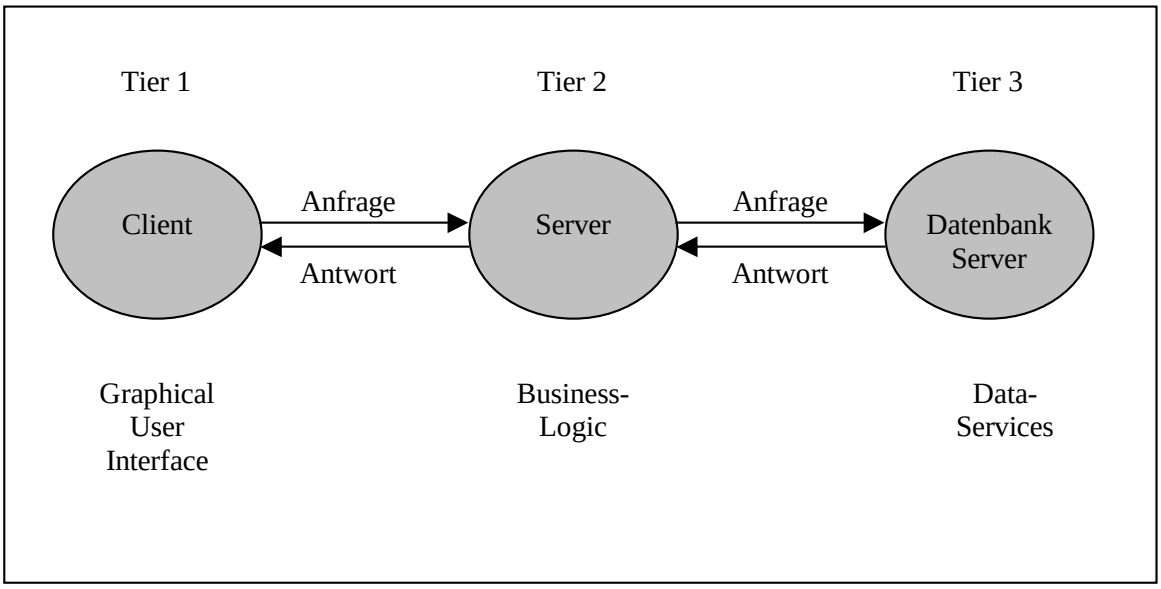


Man bindet die Header-Datei des Server-Interfaces ein und macht damit dessen Interface-Code bekannt. Nun kann man über die Middleware (Win32 bzw. COM) mittels eindeutiger ID (GUID) den Server starten, wozu die API-Funktion `CoCreateInstance()` bemüht wird. Dabei wird ein Zeiger auf das gewünschte Interface (`pInterface`) zurückgeliefert, mit dem man die Server-Methoden aufrufen kann. Dieses Verfahren wird vom Betriebssystem Windows unter dem Begriff COM (Component Object Model) implementiert. Der Inprocess-Server stellt sich für den Client als eine Software-Komponente in Form eines C++-Objektes dar und wird in dessen Prozessraum geladen (Heap-Objekt). Die vertikale Teilung kann bei komplexen Projekten auch so genutzt werden, dass ein Server intern selbst wieder als Client arbeitet, um andere Server zu nutzen.

Die Übertragung dieser Idee auf ein lokales Netzwerk (verteilte Anwendung) wird als DCOM bezeichnet (Distributed COM) und ist etwas komplizierter, vor allem werden dort die Server natürlich nicht in den Prozessraum des Clients geladen.

Beispiel:

Als abstraktes Beispiel für eine verteilte Anwendung (unabhängig von der verwendeten Middleware) sei die **3-Tier-Architektur** (vertikale Teilung in 3 wesentliche Einheiten) gezeigt:



Während der Client nur Code für die graphische Ein-/Ausgabe von Daten auf Seite des Anwenders enthält (Graphical User Interface), implementiert der Server die eigentliche Logik der zu lösenden Aufgaben (Business Logic). Der Datenbank-Server implementiert den Code für das Speichern/Abfragen von Daten in einer Datenbank (Data-Services).

Die Programmierung der 3 Einheiten (Tiers) geschieht von unterschiedlichen Experten. Den Client programmieren Spezialisten für graphische Benutzeroberflächen. Der Datenbank-Server wird (oder wurde) von Datenbank-Spezialisten programmiert. Der Server wird von Experten zur Lösung der spezifischen Aufgaben der Gesamt-Anwendung programmiert.

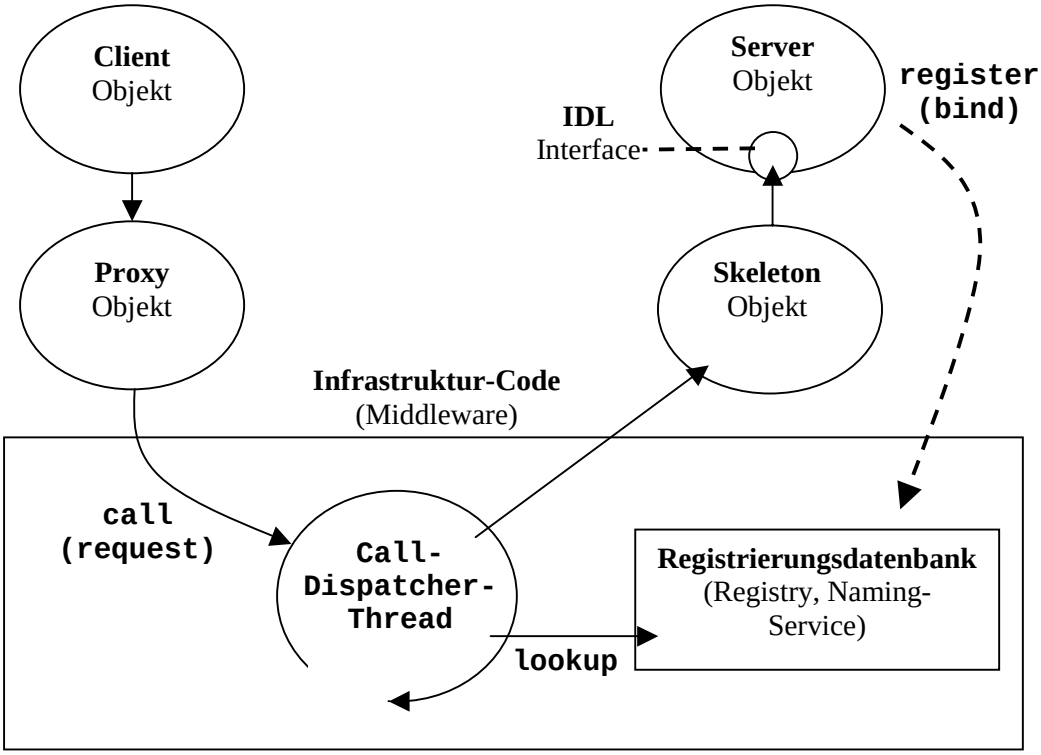
Das spezifische Wissen einer solchen Anwendung steckt somit komplett im Server mit der Business-Logic. Man kann sich also ein Datenbank-Management-System (DBMS) für Tier 3 kaufen, für Tier 1 eine graphische Oberfläche zu dem Server-Interface programmieren lassen und sich dann selbst auf die Kernaufgabe, nämlich die Implementierung der Business-Logic in Tier 2, konzentrieren.

2.7 Das Broker-Pattern (2-Tier-Architektur), CORBA, DCOM

2.7.1 Allgemeines

Möchte man Objekte nicht nur innerhalb eines Prozesses zur Verfügung stellen, sondern auch über die Prozess- oder gar Rechnergrenzen hinweg, dann wird dies in der Regel über das Broker-Pattern realisiert, wodurch man eine verteilte Umgebung (verteilte Anwendung) schafft.

Man benötigt dazu zumindest so etwas wie eine **Registrierungsdatenbank** (Registry, Naming-Service), also einen Datenbank-Server, der auf Anfrage den Pfad der Binärdateien einer Objekt-Definition (Klasse) liefert. Daneben benötigt man einen **Call-Dispatcher-Thread**, der auf eingehende Instanziierungs-Anfragen in der Registrierungsdatenbank nachschaut, wo sich die Binärdateien befinden, und ein Objekt der Klasse erzeugt. Dann liefert er ein Handle für das **IDL-Interface** an den Client zurück, womit dessen **Proxy** die Server-Objekt-Methoden über das Skeleton aufruft. Der Proxy wandelt dazu Methoden-Aufrufe in Byteströme um und das **Skeleton** (auch **Stub** genannt) baut daraus wieder Funktionsaufrufe zusammen. Den Vorgang nennt man auch **Marshalling**. Für die Rückgabewerte läuft der Vorgang anders herum ab.



Beispiele für die Implementierung des Broker-Patterns sind CORBA und DCOM. **CORBA** (Common Object Request Broker Architecture) wurde 1995 zum ersten Mal voll spezifiziert und ist verfügbar für unterschiedliche Plattformen, je nach Hersteller. Die Middleware, die auf jedem Rechner zu installieren ist, nennt sich **ORB** (Object Request Broker), wobei jeder Hersteller nochmals einen herstellerspezifischen Namen dafür hat. Die Kommunikation zwischen Proxy und Skeleton geschieht über das **IIOP** (Internet Inter-ORB Protocol), was über TCP/IP läuft und die ORBs der beiden Maschinen verbindet. Die Registrierungsdatenbank wird dort **Naming-Service** genannt. **DCOM**

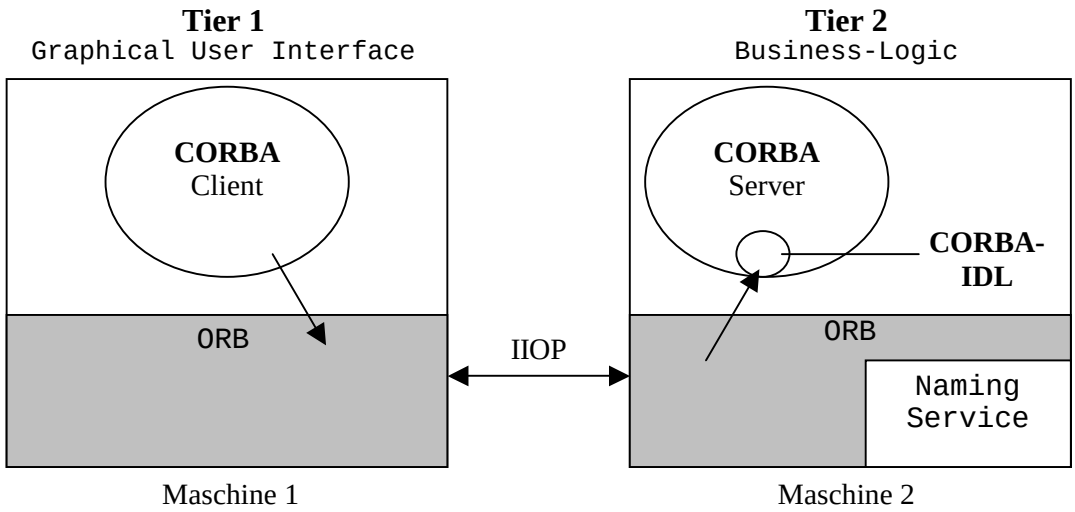
(Distributed Component Object Model) benutzt eine Middleware von Microsoft, die automatisch mit **Windows-NT** ausgeliefert wird, also praktisch Bestandteil des Betriebssystems ist. DCOM ist aber auch für andere Plattformen wie z.B. Linux (→ EntireX von Software AG) verfügbar.

Die Kommunikation zwischen Proxy und Skeleton (Stub) geschieht über **ORPC** (Object Remote Procedure Call), was ein Protokoll über DCE's RPC (Remote Procedure Call) ist. RPC läuft über TCP/IP. Als Registrierungsdatenbank wird die **Windows-NT-Registry** benutzt.

Auch die Sprache Java bietet eine Broker-Pattern-Implementierung an, die mit der virtuellen Maschine (JVM) ausgeliefert wird. Dort nennt sich das Ganze **Java/RMI** (Remote Methode Invocation) und kommuniziert über **JRMP** (Java Remote Method Protocol) über TCP/IP.

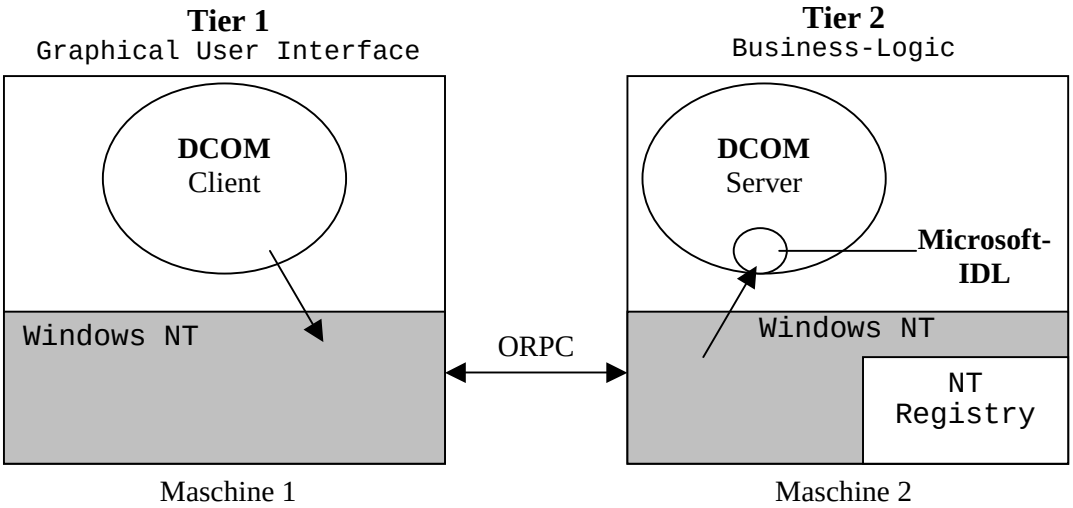
Als **Beispiel** sei die Trennung zwischen Graphical User Interface (GUI) und Business-Logic gezeigt:

CORBA:



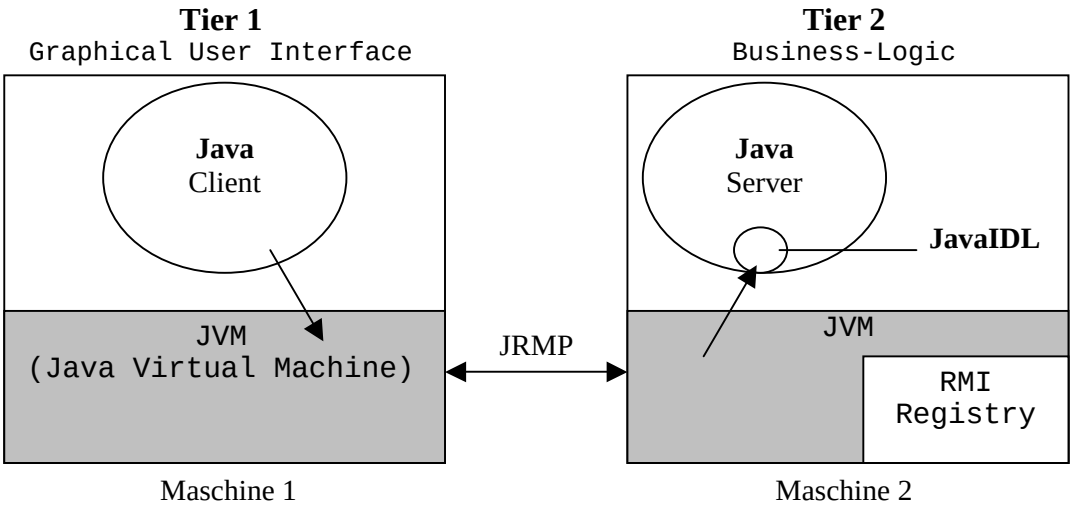
Zur Kompilierung von Proxy/Skeleton wird ein **CORBA-IDL-Compiler** benötigt.

DCOM:



Zur Kompilierung von Proxy/Stub wird der **MIDL-Compiler** von Microsoft benötigt.

Java/RMI:



Zur Kompilierung von Proxy/Skeleton wird ein **JavaIDL-Compiler** benötigt.

Man kann das Broker-Pattern natürlich auch **selbst realisieren**.

Zunächst muss man sich überlegen, wie eine Methode in einen Bytestrom verwandelt wird. Man könnte zum Beispiel sagen, daß das erste Byte eine eindeutige Nummer darstellt, die für eine bestimmte Funktion steht. Dann muss man irgendwie die Länge und Art der folgenden Parameter definieren usw. Damit hätte man dann so etwas wie **IDL** neu erfunden. Um komplexe Programme schreiben zu können, wäre es erforderlich eine automatische Umsetzung zu implementieren, also einen Compiler für das Proxy- und Skeleton-Objekt, den **IDL-Compiler**.

Hat man das, geht es an die Infrastruktur: Man könnte auf jedem Rechner einen Socket-Server laufen lassen, der auf einem bestimmten Port Anfragen für Objekt-Instanziierungen entgegennimmt und dann in einer Datenbank nachschaut, ob er dafür den Binär-Code vorliegen hat. Wenn ja, instanziiert er das Objekt und gibt ein Handle zurück, mit dem eine Kommunikation über einen Dispatcher-Thread auf einem anderen Port

stattfinden kann. Alle verfügbaren Objekte müssten hierzu in die Datenbank eingetragen werden (Registrierung). Man würde also die **Middleware** und die **Registrierungs-datenbank** neu erfinden.

Wie immer man es auch macht, das Prinzip wird sich sicherlich nicht wesentlich ändern.

Für **homogene Windows-NT-Umgebungen** macht es auf jeden Fall Sinn, **DCOM** einzusetzen. Der Aufwand der Vorbereitung der Middleware ist dann minimal mit hoher Erfolgsgarantie. In **UNIX-Umgebungen oder heterogenen Umgebungen** hat **CORBA** auf jeden Fall sein wichtigstes Einsatzgebiet. Dort konkurrieren viele ORB-Hersteller mit verschiedensten Angeboten, sowie Freeware-und Open-Source-Anbieter, wie z.B. TAO (The ACE ORB), omniORB und MICO.

2.7.2 IDL-Compiler

IDL-Code ist nicht sehr leicht portierbar, d.h. eine IDL-Datei, die man für DCOM erstellt hat, kann nicht einfach für CORBA wiederverwendet werden.

Beispiel:

DCOM:

```
[uuid(48221611-AB7E-4332-BC75-3D3302216856)]
dispinterface IDispIDLInterface
{
      properties:
      methods:
            [id(1)] short TestCall(BSTR bstrIN,BSTR* pbstrOUT);
};
```

CORBA:

```
interface IDLInterface
{
      short TestCall(in string szIN,out string szOUT);
};
```

An diesem einfachen Beispiel sieht man bereits einige wichtige Unterschiede zwischen der IDL-Implementierung von DCOM und CORBA:

ID für Interface-Funktionen:

DCOM definiert für jede Funktion eine eindeutige ID, d.h. eine Veränderung der ID erzwingt einen Neubau des Clients (als Nutzer des Interfaces), da die Helper-Funktionen in Proxy/Stub sonst nicht mehr aufeinander passen.

Strings:

Microsoft hat für DCOM speziell den Datentyp `BSTR` kreiert. `BSTR` ist eine Kombination aus der String-Definition von PASCAL (Längenangabe am Anfang) und der String-Definition von C (0x00 am Ende), d.h. `BSTR` hat beides. CORBA hingegen bildet einfach `string` auf `char*` ab (was leider zur Verwechslung mit `string` der STL führt).

Weiterhin ist das Einbinden der von der Kompilierung erzeugten Dateien in den C++-Code nicht standardisiert: Art, Anzahl und Namen der Dateien sind von IDL-Compiler zu IDL-Compiler verschieden. So hat sogar meist jeder Hersteller einer CORBA-Implementierung sein eigenes Verfahren. Man muss also in der jeweiligen Referenz nachlesen, welche Dateien generiert werden und in welche C++-Dateien sie einzubinden sind.

Beispiel für verschiedene CORBA-IDL-Compiler:

VisiBroker von Borland:

```
<Name>.idl
    ↓
idl2cpp      (IDL-Compiler)
    ↓
<Name>_c.hh
<Name>_c.cpp
<Name>_s.hh
<Name>_s.cpp
...
```

TAO von Douglas C. Schmidt:

```
<Name>.idl
    ↓
tao_idl      (IDL-Compiler)
    ↓
<Name>C.h
<Name>C.i
<Name>C.cpp
<Name>S.h
<Name>S.i
<Name>S.cpp
...
```

2.8 CORBA mit TAO (The ACE ORB) von Douglas C. Schmidt

2.8.1 Allgemeines

Dieses Kapitel erläutert den Einstieg in CORBA am praktischen Beispiel. Da es sehr speziell und umfangreich ist, soll der Leser wissen, dass er dieses Kapitel überspringen kann, wenn es momentan nicht Schwerpunkt seines Interesses ist.

TAO (**T**he **A**CE **O**RB) ist eine CORBA-Implementierung, die auf der plattform-übergreifenden Middleware ACE (**A**daptive **C**ommunication **E**nvironment) aufbaut. Es handelt sich dabei um ein Freeware-Open-Source-Produkt von Prof. Dr. Douglas C. Schmidt (**schmidt@cs.wustl.edu**). Der Umfang von ACE ist beachtlich, was zu langen Bauzeiten führen kann (mehrere Stunden auf einer 1 GHz CPU). Das heißt, man stößt den Bau am besten abends an und lässt ihn über Nacht laufen.

Ein paar wichtige Informationen zu CORBA:

- **`<name>_var`**

Hinter `<name>_var` definiert CORBA einen **Smart-Pointer** auf `<name>` (siehe: 'Smart-Pointer'). Es ist hilfreich, den Variablen-Namen mit **`sp`** beginnen zu lassen (siehe: 'Ungarische Notation'). Dadurch wird man bei seiner Benutzung daran erinnert, dass es ein Smart-Pointer ist und dass man diesen richtig anwendet, was z.B. die Verwendung der `in()`-Methode bei der Übergabe als Argument betrifft.
Beispiel:

```
CORBA::ORB_var spORB = CORBA::ORB_init(argc,argv);
CORBA::Object_var spRootPOAObj =
          spORB->resolve_initial_references("RootPOA");
PortableServer::POA_var spPOA =
          PortableServer::POA::_narrow(spRootPOAObj.in());
```

- **`string`**

`string` wird von CORBA einfach auf **`char*`** abgebildet. Es handelt sich also hierbei **nicht** um den `string` der STL, was zu Verwechslungen führen kann und die Einbindung des namespaces `std`

~~`using namespace std;`~~

verbietet.

- **Registrierung des Servers (IOR)**

Für die Registrierung des Servers benutzt CORBA ein String-Handle, d.h. eine fast 300 Zeichen lange ASCII-Zeichenkette, die etwa wie folgt aussieht:

```
IOR:010000001500000049444c3a...
```

Dieses String-Handle wird **IOR** (Interoperable Object Reference) genannt, was soviel bedeutet wie eine netzwerkweit eindeutige Referenz zu einem Objekt.

- **DII (Dynamic Invocation Interface)**

Die DII-Technik wird hier nicht behandelt.

Für den Vorgang des Registrierens (Server) bzw. des Erfragens einer Referenz (Client) gibt es 2 Verfahren:

- **IOR-Datei**

  Der Server erzeugt eine Datei mit dem Namen

  ```
  static const char IOR_FILE_NAME[] = "<name>.ior";
  ```

  und speichert sein IOR darin:

  ```
  CORBA::String_var strIOR
              = spORB->object_to_string(spIDLInterface.in());
  ofstream ofsIORFile(IOR_FILE_NAME);
  ofsIORFile << strIOR << endl;
  ofsIORFile.close();
  ```

  Dem Client muss diese Datei irgendwie zugänglich gemacht werden, das heißt er benötigt eine lokale Kopie davon oder die Datei muss über ein gemeinsames Filesystem "ge-shared" werden. Dann kann er sich das Handle besorgen:

  ```
  std::string strIORFile("file://");
  strIORFile += std::string(IOR_FILE_NAME);
  CORBA::Object_var spIORFile =
                 spORB->string_to_object(strIORFile.c_str());
  spIDLInterface = IDLInterface::_narrow(spIORFile.in());
  ```

- **Naming-Service (CosNaming)**

  Der Server kontaktiert einen Service über die Referenz "`NameService`"

  ```
  CORBA::Object_var spNameService =
     spORB->resolve_initial_references("NameService");
  CosNaming::NamingContext_var spNamingContext =
     CosNaming::NamingContext::_narrow(spNameService.in());
  ```

  Dieser Service muss natürlich zuvor gestartet und im Netzwerk zur Verfügung gestellt werden, d.h. er lauscht auf einen bestimmtem TCP/IP-Port, ob Anfragen eingehen.

  Anmerkung:

  CORBA benennt die allgemeinen Services für die Objekt-Verwaltung

  `Cos<ServiceName>`

  wobei **`Cos`** für **`Common object services`** steht.

  Der Server registriert sein IOR nun über die `bind()`-Methode hinter einem netzwerkweit eindeutigen Namen, wie z.B.

  "`de.surf25.thoemmes.p.CORBA.Server`":

```
CosNaming::Name ServerName(1);
ServerName.length(1);
ServerName[0].id =
  CORBA::string_dup("de.surf25.thoemmes.p.CORBA.Server");
spNamingContext->bind(ServerName,spIDLInterface.in());
```

Dem Client wird das IOR des Servers nun über den Naming-Service zugänglich gemacht. Er erfragt es mit der `resolve()`-Methode:

```
CORBA::Object_var spResolvedObj =
          spNamingContext->resolve(
                  "de.surf25.thoemmes.p.CORBA.Server");
spIDLInterface =
          IDLInterface::_narrow(spResolvedObj.in());
```

2.8.2 Code-Beispiel mit GNU C++-Compiler unter LINUX

Hier soll am Beispiel eine Minimal-Client/Server-Anwendung gezeigt werden, damit die gesagten Dinge etwas klarer werden. Der Einfachheit halber ist das Client- und Server-Projekt im selben Verzeichnis untergebracht und beides (inklusive IDL-Datei) wird mit derselben Make-Datei gebaut.

Vorbereitungen:

- Kopiere alle `tar.gz`-Archive von TAO in das Root-Verzeichnis `/` und extrahiere sie in der richtigen Reihenfolge, d.h. die Patches folgen dem Haupt-Archiv entsprechend der Patch-Nummer, hier gezeigt für TAO 1.2a mit Patch p1 bis ...:

  ```
  gunzip -c ACE+TAO-1.2a.tar.gz | tar xvf -

  gunzip -c TAO-1.2a_p1_patched_files.tar.gz | tar xvf -
  gunzip -c TAO-1.2a_p2_patched_files.tar.gz | tar xvf -
  ...
  ```

  Dadurch wird das Verzeichnis `/ACE_wrappers` angelegt und alle Dateien von ACE und TAO darin abgelegt.

- Setze die Umgebungs-Variable `ACE_ROOT`:

  ```
  export ACE_ROOT=/ACE_wrappers
  ```

- In `/ACE_wrappers/ace/` ist die Datei

  ```
  config.h
  ```

  zu erzeugen und mit folgenden 2 Zeilen (Reihenfolge beachten) abzuspeichern:

  ```
  #define ACE_HAS_STANDARD_CPP_LIBRARY 1
  #include "ace/config-linux.h"
  ```

- In `/ACE_wrappers/include/makeinclude/` ist die Datei `platform_<os>.GNU` unter neuem Namen zu kopieren:

  ```
  cp platform_linux.GNU platform_macros.GNU
  ```

- Nun ist in `/ACE_wrappers/ace/` die ACE-Umgebung zu bauen:

  ```
  make
  ```

  (dauert sehr, sehr lange – Stunden auf einer 1 GHz CPU)

- Setze die Umgebungs-Variable `TAO_ROOT`:

  ```
  export TAO_ROOT=/ACE_wrappers/TAO
  ```

- Baue in `/ACE_wrappers/TAO/` die TAO-Umgebung:

  ```
  make
  ```

- Baue in `/ACE_wrappers/TAO/orbsvcs/` die Services (Naming-Service):

  ```
  make
  ```

- Ermögliche das Laden der erzeugten dynamischen Bibliotheken (`*.so`) am besten in einem Shell-Login-Script, damit sie immer verfügbar sind:

  ```
  export LD_LIBRARY_PATH=$ACE_ROOT:$TAO_ROOT/TAO
  export LD_LIBRARY_PATH:$LD_LIBRARY_PATH:$TAO_ROOT/TAO/TAO_IDL
  ```

Projekt aufsetzen:

- Im Verzeichnis mit den eigenen C++-Entwicklungen (Bsp: `/home/Peter`) ist ein Verzeichnis für das CORBA-Projekt anzulegen (Bsp: `Client_n_Server`). Dort sind folgende (zunächst leere) Dateien zu erzeugen:

  ```
  IDLInterface.idl
  IDLInterfaceIUserImpl.cpp

  ServerApp.cpp
  Server.cpp
  Server.h

  ClientApp.h

  makefile
  ```

- Deklariere folgende statische Klasse in **`Server.h`**:

```
#ifndef _SERVER_H_
#define _SERVER_H_

#include <string> //STL

class Server
{
	public:
		static short Init(int argc,char* argv[]);
		static short TestCall(const std::string& strIN,
                                      std::string& strOUT);

	private:
		//c'tor / d'tor versteckt -> statische Klasse:
		Server();
		Server(const Server& Obj);
		~Server();
};

#endif
```

  Hier sieht man die Methode `TestCall()`, die später vom Client aufgerufen werden soll.

- Implementiere die Klasse in **`Server.cpp`**:

```
#include <stdio.h>
#include "Server.h"

short Server::Init(int argc,char* argv[])
{
	argc = argv[0][0]; //pseudo code to avoid warning
	return 0;
}

short Server::TestCall(
                const std::string& strIN,std::string& strOUT)
{
	printf(">>>SERVER: Client meldete sich -> [%s]\r\n",
               strIN.c_str());
	strOUT = "Gruss vom Server";
	return 0;
}
```

- Definiere das passende Interface für `TestCall()` in **`IDLInterface.idl`**:

```
interface IDLInterface
{
	short TestCall(in string szIN,out string szOUT);
};
```

- Implementiere das Interface in **`IDLInterfaceIUserImpl.cpp`**:

```
#include "IDLInterfaceI.h"

#include "Server.h"

IDLInterface_i::IDLInterface_i(void)
{
}

IDLInterface_i::~IDLInterface_i(void)
{
}

CORBA::Short IDLInterface_i::TestCall(
                      const char* szIN,
                      CORBA::String_out szOUT ACE_ENV_ARG_DECL)
ACE_THROW_SPEC((CORBA::SystemException))
{
      std::string strOUT;
      short nRet = Server::TestCall(std::string(szIN),strOUT);
      szOUT = CORBA::string_dup(strOUT.c_str());
      return nRet;
}
```

Hier sieht man, dass der Aufruf von `TestCall()` an die statische Klasse `Server` delegiert wird.

Für die weitere Vorgehensweise ist es wichtig zu wissen, dass der IDL-Compiler die Datei

`IDLInterfaceI.cpp`

als Vorlage (Template) für `IDLInterfaceIUserImpl.cpp` generiert. Wenn man also eine neue Interface-Funktion in der IDL-Datei `IDLInterface.idl` definiert, dann bekommt man dort den leeren Funktions-Rumpf für die Implementierung bereitgestellt. Man darf natürlich nicht `IDLInterfaceI.cpp` auch noch mitbauen.

- Implementiere die Server-Applikation **`ServerApp.cpp`**:

```
#include <stdio.h>

#include <ace/streams.h> //for ofstream
#include <orbsvcs/orbsvcs/CosNamingC.h> //for naming service

#include "IDLInterfaceI.h"
#include "Server.h"

static const char SERVER_NAME_TO_REGISTER[] =
                          "de.surf25.thoemmes.p.CORBA.Server";

static const char IOR_FILE_NAME[] = "./CORBAServer.IOR";

int main(int argc,char* argv[])
{
      //Parse Kommandozeilen-Parameter:
      bool bRegisterOnly = false;
      bool bUseNamingService = false;
      if(argc > 1)
      {
            for(long l = 0;l < argc;++l)
            {
                  if(!strcmp(argv[l],"-r")) //nur registrieren
                        bRegisterOnly = true;
                  if(!strcmp(argv[l],"-NS")) //NamingServ. nutzen
                        bUseNamingService = true;
            }
      }

      //Initialisiere die Middleware (ORB):
      CORBA::ORB_var spORB = CORBA::ORB_init(argc,argv);

      //Referenz z. Portable Object Adapter (RootPOA) besorgen:
      CORBA::Object_var spRootPOAObj =
            spORB->resolve_initial_references("RootPOA");
      PortableServer::POA_var spPOA =
            PortableServer::POA::_narrow(spRootPOAObj.in());

      //POA-Manager aktivieren:
      PortableServer::POAManager_var spPOAManager =
                                       spPOA->the_POAManager();
      spPOAManager->activate();
```

```
//Interface (servant) erzeugen:
IDLInterface_i IDLInterface;
if(bUseNamingService) //NamingServ.(bind/rebind,resolve)
{
      //Aktiviere das Interface (servant):
      IDLInterface_var spIDLInterface =
                                IDLInterface._this();

      //Besorge den Naming-Service/Context:
      CORBA::Object_var spNameService =
         spORB->resolve_initial_references("NameService");
      CosNaming::NamingContext_var spNamingContext =
         CosNaming::NamingContext::_narrow(
                                      spNameService.in());

      //Namen zum Registrieren zusammenbauen:
      CosNaming::Name ServerName(1);
      ServerName.length(1);
      ServerName[0].id =
            CORBA::string_dup(SERVER_NAME_TO_REGISTER);

      //Server registrieren:
      spNamingContext->bind(ServerName,
                            spIDLInterface.in());
}
else //IOR-file (object_to_string,string_to_object)
{
      //Interface (servant) aktivieren:
      PortableServer::ObjectId_var spOID =
                 spPOA->activate_object(&IDLInterface);

      //Über OID einen Zeiger auf das Interface besorgen:
      CORBA::Object_var spIDLInterface =
                      spPOA->id_to_reference(spOID.in());

      //IOR generieren (Interoperable Object Reference):
      CORBA::String_var strIOR =
            spORB->object_to_string(spIDLInterface.in());

      //IOR in Datei schreiben:
      ofstream ofsIORFile(IOR_FILE_NAME);
      ofsIORFile << strIOR << endl;
      ofsIORFile.close();
}

//Programm beenden, wenn nur Registr. erwünscht ("-r"):
if(bRegisterOnly)
{
      printf(">>>SERVER: Registered successfully\r\n");
      return 0;
}
```

```
    //Statische Klasse Server initialisieren:
    Server::Init(argc,argv);

    //Call-Dispatcher starten
    //-> Requests vom client akzeptieren (Message-Loop):
    spORB->run();

    //Aufräumen nachdem die Arbeit getan ist:
    spPOA->destroy(1,1);
    spORB->destroy();

    return 0;
}
```

Der Server kann mit **2 benutzerdefinierten Kommandozeilen-Parametern** aufgerufen werden:

`-r`	→	Nur registrieren
`-NS`	→	Naming-Service statt IOR-Datei nutzen (muss zuvor gestartet worden sein)

Weiterhin kann er mit den **von CORBA definierten Parametern** aufgerufen werden, was man leicht dem Aufruf `CORBA::ORB_init(argc,argv)` ansieht, denn diesem werden `argc` und `argv` mitgegeben. Um dieses Beispiel mit Naming-Service auf einem lokalen Rechner auszuprobieren, muss man deshalb dem Server noch den Parameter

```
-ORBInitRef NameService=
                      corbaloc:iiop:localhost:12345/NameService
```

mitgeben (hier beispielhaft für `NameServicePort=12345`).

- Implementiere die Client-Applikation **`ClientApp.cpp`**:

```
#include "IDLInterfaceC.h"

#include <string>
#include <ace/streams.h> //für ofstream
#include <orbsvcs/orbsvcs/CosNamingC.h> //für NamingService

static const char REGISTERED_SERVER_NAME[] =
                          "de.surf25.thoemmes.p.CORBA.Server";

static const char IOR_FILE_NAME[] = "./CORBAServer.IOR";

int main(int argc,char* argv[])
{
     //Kommandzeilen-Parameter parsen:
     bool bUseNamingService = false;
     if(argc > 1)
     {
          for(long l = 0;l < argc;++l)
          {
               if(!strcmp(argv[l],"-NS")) //NamingServ. nutzen
                    bUseNamingService = true;
          }
     }

     //Initialisiere die Middleware (ORB):
     CORBA::ORB_var spORB = CORBA::ORB_init(argc,argv);

     //Smart-Pointer aufs Interface (servant) besorgen:
     IDLInterface_var spIDLInterface;
     if(bUseNamingService) //NamingServ. (bind/rebind,resolve)
     {
          //Naming-Service/context besorgen:
          CORBA::Object_var spNameService =
             spORB->resolve_initial_references("NameService");
          CosNaming::NamingContext_var spNamingContext =
             CosNaming::NamingContext::_narrow(
                                        spNameService.in());

          //Registrierten Namen zusammenbauen:
          CosNaming::Name ServerName(1);
          ServerName.length(1);
          ServerName[0].id =
               CORBA::string_dup(REGISTERED_SERVER_NAME);

          //Interface vom Naming-Service besorgen:
          CORBA::Object_var spResolvedObj =
               spNamingContext->resolve(ServerName);
          spIDLInterface =
               IDLInterface::_narrow(spResolvedObj.in());
     }
```

```
        else //IOR-file (object_to_string,string_to_object)
        {
              //IOR-Datei auslesen:
              std::string strIORFile("file://");
              strIORFile += std::string(IOR_FILE_NAME);
              CORBA::Object_var spIORFile =
                    spORB->string_to_object(strIORFile.c_str());
              if(CORBA::is_nil(spIORFile.in()))
              {
                    printf(">>>CLIENT: IOR-Fehler [%s]\r\n",
                           IOR_FILE_NAME);
                    return -1;
              }
              spIDLInterface =
                         IDLInterface::_narrow(spIORFile.in());
        }
        if(CORBA::is_nil(spIDLInterface.in()))
        {
              printf(">>>CLIENT: Fehler\r\n");
              return -1;
        }

        //Server-Funktion 'TestCall()' aufrufen:
        printf(">>>CLIENT: Rufe jetzt 'TestCall()' auf...\r\n");
        CORBA::String_var spszAnswer = CORBA::string_dup("?");
        short nError = spIDLInterface->TestCall(
                                        "Gruss vom Client",
                                        spszAnswer.out());
        printf(
              ">>>CLIENT: Server antwortete -> #[%d] - [%s]\r\n",
              nError,
              spszAnswer.in());

        return 0;
}
```

Der Client kann mit **einem benutzerdefinierten Kommandozeilen-Parameter** aufgerufen werden:

`-NS` → Naming-Service statt IOR-Datei nutzen
(muss zuvor gestartet worden sein)

Weiterhin kann er mit den **von CORBA definierten Parametern** aufgerufen werden, was man wie beim Server dem Aufruf `CORBA::ORB_init(argc,argv)` ansieht. Bei lokaler Nutzung des Naming-Service ist dies der gleiche wie beim Server:

```
-ORBInitRef NameService=
                corbaloc:iiop:localhost:12345/NameService
```

- Zum Schluss nun noch die Make-Datei '**makefile**':

```
#-----------------------------------------------------------------------
# IDL-Dateien (mit dem Pojekt abzugleichen):
#-----------------------------------------------------------------------

IDL_FILES = \
           IDLInterface
IDL_SRC = \
           IDLInterfaceIUserImpl.cpp \
           IDLInterfaceS.cpp \
           IDLInterfaceC.cpp
IDL_PRECIOUS = \
           IDLInterfaceI.cpp IDLInterfaceI.i IDLInterfaceI.h \
           IDLInterfaceS.cpp IDLInterfaceS.i IDLInterfaceS.h \
           IDLInterfaceC.cpp IDLInterfaceC.i IDLInterfaceC.h
IDL_CLEAN = \
           IDLInterfaceI.* \
           IDLInterfaceS.* \
           IDLInterfaceC.*

#-----------------------------------------------------------------------
# Server- und Client-Dateien (mit dem Pojekt abzugleichen):
#-----------------------------------------------------------------------

SVR_BIN = server
SVR_SRC = ServerApp.cpp Server.cpp
SVR_OBJ = \
     IDLInterfaceC.o IDLInterfaceS.o IDLInterfaceIUserImpl.o \
     ServerApp.o Server.o

CLT_BIN = client
CLT_SRC = ClientApp.cpp
CLT_OBJ = \
     IDLInterfaceC.o \
     ClientApp.o

#-----------------------------------------------------------------------
# TAO - checks and flags:
#-----------------------------------------------------------------------

ifndef TAO_ROOT
  TAO_ROOT = $(ACE_ROOT)/TAO
endif

TAO_IDLFLAGS += -GI
TAO_IDLFLAGS += -Sc

# Wenn die TAO orbsvcs library nicht mit geuegend Komponenten gebaut
# wurde, wird der Bau nicht durchgefuehrt:

TAO_ORBSVCS := $(shell sh $(ACE_ROOT)/bin/ace_components --orbsvcs)
ifeq (Naming,$(findstring Naming,$(TAO_ORBSVCS)))
  BIN = $(SVR_BIN) $(CLT_BIN)
endif # Naming
```

```
#-----------------------------------------------------------------------
# TAO - includes:
#-----------------------------------------------------------------------

include $(ACE_ROOT)/include/makeinclude/wrapper_macros.GNU
include $(ACE_ROOT)/include/makeinclude/macros.GNU
include $(TAO_ROOT)/rules.tao.GNU
include $(ACE_ROOT)/include/makeinclude/rules.common.GNU
include $(ACE_ROOT)/include/makeinclude/rules.nonested.GNU
include $(ACE_ROOT)/include/makeinclude/rules.local.GNU
include $(TAO_ROOT)/taoconfig.mk

#-----------------------------------------------------------------------
# Bau der Software:
#-----------------------------------------------------------------------

# srcs:
SRC = $(IDL_SRC) $(SVR_SRC) $(CLT_SRC)

# includes:
CPPFLAGS += -I$(TAO_ROOT)/orbsvcs/orbsvcs

# libs:
LDFLAGS += -L$(TAO_ROOT)/orbsvcs/orbsvcs -L$(TAO_ROOT)/tao \
                -L$(TAO_ROOT)/orbsvcs/Naming_Service
LDLIBS = -lTAO_CosNaming -lTAO_Svc_Utils \
                -lTAO_IORTable -lTAO_PortableServer -lTAO

.PRECIOUS: $(IDL_PRECIOUS)

server: $(addprefix $(VDIR),$(SVR_OBJ))
     $(LINK.cc) $(LDFLAGS) -o $@ $^ $(VLDLIBS) $(POSTLINK)

client: $(addprefix $(VDIR),$(CLT_OBJ))
     $(LINK.cc) $(LDFLAGS) -o $@ $^ $(VLDLIBS) $(POSTLINK)

realclean: clean
     -$(RM) $(IDL_CLEAN)

#-----------------------------------------------------------------------
# Dependencies:
#-----------------------------------------------------------------------

# DO NOT DELETE THIS LINE -- g++dep uses it.
# DO NOT PUT ANYTHING AFTER THIS LINE, IT WILL GO AWAY.
```

Projekt bauen:

Nachdem man dieses Projekt erfolgreich aufgesetzt hat, kann man es mit

```
make
```

bauen und findet nach erfolgreichem Bau die ausführbaren Dateien

```
server
client
```

im gleichen Verzeichnis. Die Objektdateien (`*.o`) werden im Unterverzeichnis `./.obj/` abgelegt.

Projekt testen:

Da man sich die Argumente, mit denen die einzelnen Programme zu starten sind, natürlich schlecht auswendig merken kann, packt man sie am besten in einige Shell-Scripts:

'`Export_LD_LIBRARY_PATH.sh`':

```
export LD_LIBRARY_PATH=
     $ACE_ROOT:$ACE_ROOT/ace:$TAO_ROOT/TAO:$TAO_ROOT/TAO/TAO_IDL
```

'`RunNamingService.sh`':

```
export NameServicePort=12345
$TAO_ROOT/orbsvcs/Naming_Service/Naming_Service
                           -ORBEndPoint iiop://localhost:12345 &
```

'`StopNamingService.sh`' (hier wird ein wenig der Textprozessor '`awk`' bemüht):

```
NAMING_SERVICE_PID=
  `ps x | grep -w Naming_Service | grep -v grep | awk '{print $1}'`
echo Going to kill $NAMING_SERVICE_PID
kill -9 $NAMING_SERVICE_PID
```

'`RunServerUsingIOR.sh`':

```
./server &
```

'`RunServerUsingNS.sh`':

```
./server -NS -ORBInitRef NameService=
                      corbaloc:iiop:localhost:12345/NameService &
```

'`RunServerUsingIOR.sh`':

```
./client &
```

'`RunServerUsingNS.sh`':

```
./client -NS -ORBInitRef NameService=
                           corbaloc:iiop:localhost:12345/NameService
```

'`StopServer.sh`' (hier wird wieder '`awk`' bemüht):

```
SERVER_PID=`ps x | grep -w server | grep -v grep | awk '{print $1}'`
echo Going to kill $SERVER_PID
kill -9 $SERVER_PID
```

Das erfolgreich gebaute Projekt kann jetzt wie folgt getestet werden:

- **Test mit IOR:**

  Erst den Server, dann den Client starten:

  ```
  sh RunServerUsingIOR.sh
  sh RunClientUsingIOR.sh
  ```

  → man sollte folgende 2 Zeilen in der Ausgabe sehen:

  ```
  >>>SERVER: Client meldete sich -> [Gruss vom Client]
  >>>CLIENT: Server antwortete -> #[0] - [Gruss vom Server]
  ```

  Zum Schluss wird dann der Server wieder beendet:

  ```
  sh StopServer.sh
  ```

  Zur Kontrolle sollte man sich die vom Server generierte IOR-Datei `CORBAServer.IOR` einmal anschauen (`less ./CORBAServer.IOR`), die den IOR-String des Servers beinhaltet.

- **Test mit Naming-Service:**

  Erst den Naming-Service, dann den Server und dann den Client starten:

  ```
  sh NamingService.sh
  sh RunServerUsingNS.sh
  sh RunClientUsingNS.sh
  ```

  → man sollte wieder folgende 2 Zeilen in der Ausgabe sehen:

  ```
  >>>SERVER: Client meldete sich -> [Gruss vom Client]
  >>>CLIENT: Server antwortete -> #[0] - [Gruss vom Server]
  ```

  Zum Schluss wird dann der Server und der Naming-Service wieder beendet:

  ```
  sh StopServer.sh
  sh StopNamingService.sh
  ```

Kopie auf anderen Rechner:

Möchte man die Software auf einen anderen Rechner kopieren, dann darf man die dynamisch "ge-linkten" Dateien (`*.so`) unter `/ACE_wrappers` nicht vergessen.

2.8.3 Code-Beispiel mit Visual C++ 6.0-Compiler unter WINDOWS-NT

Hier soll im Prinzip das gleiche Beispiel wie im letzten Kapitel gezeigt werden, jedoch mit einem anderen Compiler und auf einer anderen Plattform. Es wird nur die Vorgehensweise skizziert, um damit die wesentlichen Unterschiede herauszuarbeiten.

Vorbereitungen:

- Kopiere alle `tar.gz`-Archive von TAO in das Verzeichnis `C:\` und extrahiere sie in der richtigen Reihenfolge, d.h. die Patches folgen dem Haupt-Archiv entsprechend der Patch-Nummer (benutze einen beliebigen Packer/Entpacker wie z.B. PowerArchiver). Dadurch wird das Verzeichnis `C:\ACE_wrappers` angelegt und alle Dateien von ACE und TAO darin abgelegt.

- In `C:\ACE_wrappers\ace\` ist die Datei

  ```
  config.h
  ```

  zu erzeugen und mit folgenden 2 Zeilen (Reihenfolge beachten) abzuspeichern:

  ```
  #define ACE_HAS_STANDARD_CPP_LIBRARY 1
  #include "ace/config-win32.h"
  ```

- Lade das Projekt

  ```
  C:\ACE_wrappers\ace\ace.dsw
  ```

  und baue

  ```
  Win32 Release
  ```

  und

  ```
  Win32 Static Release
  ```

  wonach folgende Ergebnisse zu finden sind:

  ```
  C:\ACE_wrappers\bin:
        ace.dll
        aced.dll

  C:\ACE_wrappers\ace:
        aces.lib
        acesd.dll
  ```

- **Lade das Projekt**

  ```
  C:\ACE_wrappers\tao\taoace.dsw
  ```

  und stelle in der IDE Folgendes ein:

```
Tools.Options.Directories...

      Executables:

            'C:\ACE_wrappers\bin\release'    (zuerst!!!)
            'C:\ACE_wrappers\bin\'
```

und baue alle Projekte **außer** `ACE_SSL` und `SSL_IOP` als

```
Win32 Release
```

- Lade das Projekt

```
C:\ACE_wrappers\TAO\orbsvcs\Naming_Service\Naming_service.dsw
```

und baue

```
Win32 Release
```

Projekt aufsetzen:

- Im Verzeichnis mit den eigenen C++-Entwicklungen (Bsp: `C:\CPPDev\Peter`) ist ein Verzeichnis für das CORBA-Projekt (Bsp.: `CORBA_TEST`) anzulegen. Am besten richtet man in 2 Unterverzeichnissen (`Server` und `Client`) 2 neue Projekte ein:

```
Server.dsp
Client.dsp
```

- Besondere Einstellungen der IDE:

```
Tools.Options.Directories...
      Executables:
            'C:\ACE_wrappers\bin'
      Include Files:
            'C:\ACE_wrappers'
            'C:\ACE_wrappers\tao'
            'C:\ACE_wrappers\tao\orbsvcs'
      Library Files:
            'C:\ACE_wrappers\ace'
            'C:\ACE_wrappers\tao\tao'
            'C:\ACE_wrappers\tao\orbsvcs\orbsvcs'
      Source Files:
            'C:\ACE_wrappers\ace'
            'C:\ACE_wrappers\tao\tao'
            'C:\ACE_wrappers\tao\orbsvcs\orbsvcs'
```

- Man muss darauf achten, dass man das Projekt mit '**Multithreaded runtime libraries**' baut.

- Man platziert die `*.idl`-Datei am besten in das `Server`-Projekt.

- Der IDL-Compiler wird mit

  ```
  tao_idl -Sc -GI IDLInterface.idl
  ```

  aufgerufen, was man entweder mittels BATCH-Datei oder über '**Custom-Build**' im `Server`-Projekt tun kann.

Projekt-Bau:

Der Bau des Projektes muss immer so geschehen, dass zunächst die IDL-Datei kompilert wird und dann Server und Client gebaut werden.

Kopie auf anderen Rechner:

Möchte man die Software auf einen anderen Rechner kopieren, dann darf man die dynamisch "ge-linkten" Dateien (`*.dll`) unter `C:\ACE_wrappers` nicht vergessen. In diesem Fall:

```
ace.dll
msvcrt.dll
TAO.dll
TAO_CosNaming.dll
TAO_PotableServer.dll

stlport_vc<no>.dll
```

2.9 UML (Unified Modeling Language)

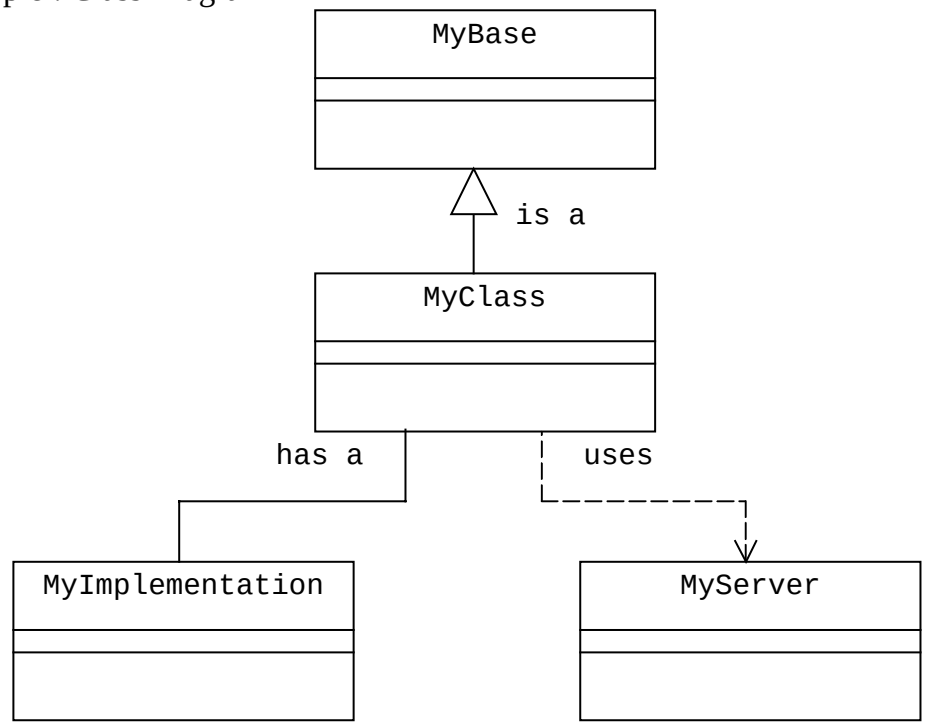
2.9.1 Allgemeines

UML (Unified Modeling Language) ist ein Standard zur graphischen Darstellung der Zusammenhänge in einer komplexen objektorientierten Software. UML führt die Notationen OMT (Object Modeling Technique von Rumbaugh), OOD (Object Oriented Design von Booch) und OOSE (Object Oriented Software Engineering von Jacobson) zusammen. C++-UML-Tools können aus der graphischen Darstellung des Class Diagrams C++-Code generieren, was zur Erstellung des Hauptgerüstes der Software sinnvoll sein kann. Bedenke: **Automatisch generierter Code ist nie besser als die vom Entwickler des Code-Generators verwendeten Konzepte**!

Es gibt verschiedene UML-Diagramme:

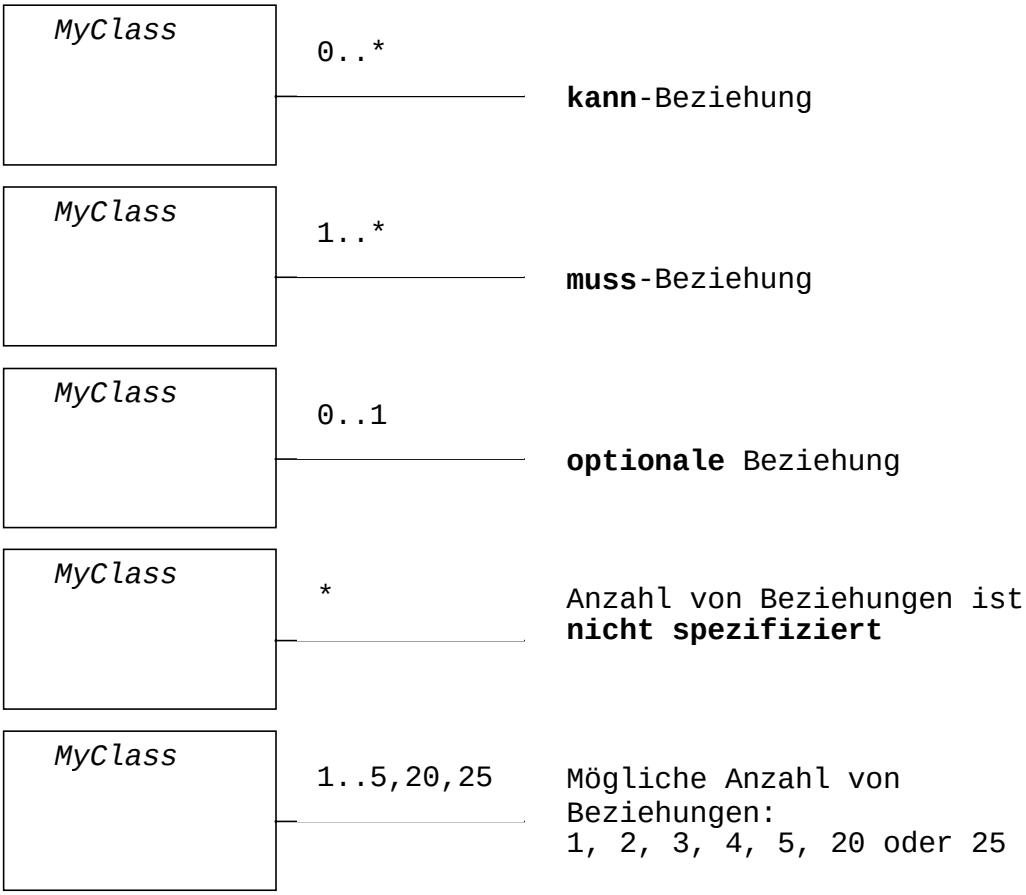
- Use Case Diagram:
  Was sind die Hauptfunktionen des Systems und wer sind deren Nutzer ?
- Activity Diagram:
  Wie laufen Aktivitäten im System ab (Ablaufdiagramme) ?
- Collaboration Diagram:
  Wer kommuniziert wie mit wem ?
- Sequence Diagram:
  Sequentielle Aufzeichnung der Aktionen
- Class Diagram:
  Klassen und ihre Beziehungen und Kardinalitäten
- State Transition Diagram:
  Zustandsübergänge eines Objektes

Beispiel: Class Diagram



2.9.2 Kardinalitäten nach UML

UML definiert **erlaubte Bereiche** für die Anzahl der an einer Beziehung beteiligten Objekte, sogenannte **Kardinalitäten**:



2.9.3 Frage nach den Klassen/Objekten

Bei der Konzeption taucht früher oder später immer folgende Frage auf: "Welche Klassen/Objekte benötige ich überhaupt ?" Je nach Komplexität der zu lösenden Aufgabe kann es sinnvoll sein, sich hierbei eines UML-Tools zu bedienen. Im Gegensatz zur prozeduralen Programmierung sucht man hier nicht nach den Funktionen, die in einem Fluss, gesteuert von Bedingungen/Entscheidungen, ausgeführt werden sollen, sondern nach **Instanzen**, die für die Ausführung bestimmter Aufgaben zuständig sind. Man fasst also Aufgaben zusammen, die

- zum gleichen Themengebiet gehören
  Beispiel: Layout-Algorithmen
  → Statische Klasse `LayoutAlgorithms`, die mit ihren Methoden den Code der Algorithmen für andere Klassen zur Verfügung stellt

oder

- sich mit der gleichen Art von Daten beschäftigen
  Beispiel: Zeichnen eines Graphen aus einer Menge von Koordinaten
  → Klasse `Graph` mit dem Methoden zum Zeichnen

Hat man festgelegt, welche Klasse wofür zuständig ist, dann muss man festlegen, welche **Ereignisse** es geben kann und von wem sie generiert werden (**Ursache**).

Beispiele für Ereignisse und ihre Ursache:

- Anwendungsstart durch den Benutzer
- Mausklick durch den Benutzer
- Call durch den Client
- Timer-Tick vom Betriebssystem
- ...

Danach legt man fest, durch wen (**Handler**) und wie (**Code**) die Ereignisse behandelt werden.

Beispiele für das Behandeln von Ereignissen:

- Anwendungsstart wird behandelt durch `main()`
  - Instanziierung und Initialisierung des Haupt-Objektes (`theApp`)
  - Aufruf der Methode `Run()` des Haupt-Objektes (Nachrichtenschleife)

```
int main()
{
      App theApp;
      theApp.Initialize();
      theApp.Run();
      return theApp.Uninitialize();
}
```

Pseudo-Code von `App::Run()`:

```
MsgParams Params;
while(1)
{
      if(GetMessage(Params) == "MSG_TERMINATE")
            break;
      TranslateMessage(Params);
      DispatchMessage(Params);
}
```

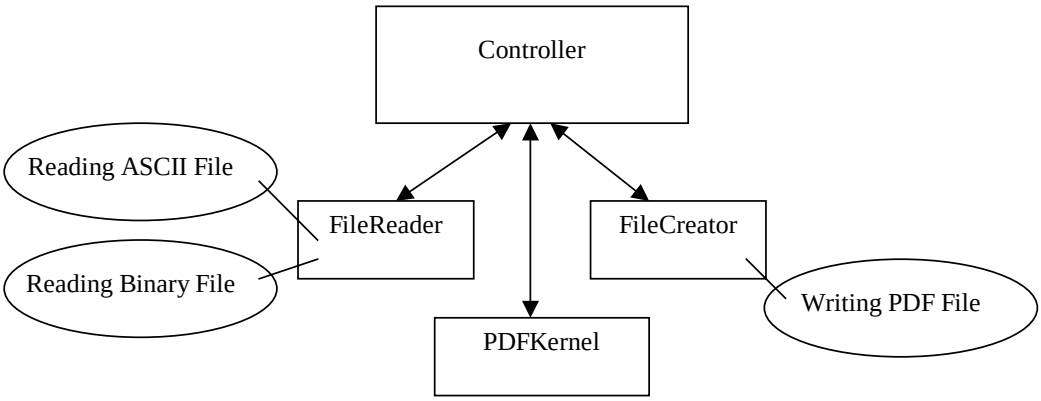
- Mausklick wird behandelt durch die Callback-Funktion `OnLeftMouseKlick()`
  - Dem Betriebssystem wird ein Zeiger auf die Funktion `OnLeftMouseKlick()` übergeben und per Handle bzw. eindeutiger ID ein Fensterbereich mit dieser Funktion verknüpft. Jeder Mausklick in diesen Fensterbereich führt dann zum Aufruf von `OnLeftMouseKlick()` durch das Betriebssystem.

Beispiel für die Instanzen einer Software (schematisch):

Aus mehreren Text-Dateien (ASCII) und mehreren Bild-Dateien (Binär) soll eine PDF-Datei erzeugt werden (ohne Graphical User Interface). Man findet beispielsweise folgende **Instanzen**:

`PDFGenerator:`

- Einer muss sich um alles kümmern (koordinieren): **Controller**
- Einer muss die Eingabe-Daten lesen: **FileReader**
- Einer muss die Eingabe-Daten decodieren & Ausgabe-Daten erzeugen: **PDFKernel**
- Einer muss die Ausgabe-Daten schreiben: **FileCreator**



Nachdem man diesen Ausgangspunkt gefunden hat, kann man an die Verfeinerung gehen, wie z.B. die Einführung einer Konfigurierung der Anwendung durch die Klasse `Configuration`. Bis hierhin jedoch findet man drei **große Klassen instanziiert als Member-Variablen in der Hauptklasse (`PDFGenerator`)**, die wiederum mit weiteren Klassen in Verbindung stehen, um ihre Aufgabe zu erfüllen. Neben der Findung dieser Klassen ist es wichtig zu wissen, welche **Ereignisse** zugelassen sind und von wem sie generiert werden (**Ursache**). Dann legt man fest, wer (**Handler**) diese Ereignisse wie (**Code**) behandelt. Im vorliegenden Fall wird ein Ereignis jedenfalls der Anwendungsstart durch den Benutzer sein, der von `main()` behandelt wird, indem `theApp` instanziiert, initialisiert und die Methode `Run()` aufgerufen wird. Wenn man nun schon mal eine grobe Vorstellung haben will, wie das Programm aussieht, dann könnte man es sich wie folgt vorstellen:

```
int main()
{
      PDFGenerator theApp; //Hauptklasse (Anwendung)
      theApp.Initialize();
      theApp.Run();
      return theApp.Uninitialize();
}
```

→ Fehlt nur noch die Deklaration und Implementierung der Klassen :-)

Natürlich wird so normalerweise nur der Prototyp aussehen. In der Praxis kann es sein, dass man sich in einem größeren System als Teilsystem wiederfindet, d.h. daß man sich irgendwie an eine bestehende Software dranhängen muss, sei es als Nutzer von Diensten und Informationen (COM-Client, CORBA-Client, ...) oder als Anbieter von Diensten und Informationen (COM-Server, CORBA-Server, ...) oder als zusätzlicher lokaler Prozess mit Zugriff auf Shared Memory oder als statisch oder dynamisch hinzugelinktes Erweiterungs-Modul (LIB-File, DLL-File, ...) oder ... .

3. Wichtige Begriffe und Sprachelemente

3.1 namespace und using

Problem:
Viele Programmierer → Viele Header-Dateien → Viele gleiche Namen

Abhilfe:
Jeder Programmierer vergibt ein eindeutiges Präfix für seine Namen. Statt dies von Hand zu tun, benutzt man namespace.

Statt:

```
const double peterdVERSION;
class peterMyClass
{
      ...
};
```

Besser:

```
namespace peter
{
      const double dVERSION;
      class MyClass
      {
      ...
      };
}
```

Nun muss der Nutzer natürlich auch dieses Präfix benutzen, um an seine Namen zu kommen:

a) **Explizite** Angabe des Präfixes der Namen:

```
const double peter::dVERSION = 2.0;
void peter::MyClass::peter::MyClass() {...}
```

b) **Implizite** Benutzung des Präfixes **für ausgewählte Namen (using)**:

```
using peter::dVERSION;
const double dVERSION = 2.0;
```

c) **Implizite** Benutzung des Präfixes **für alle Namen (using)**:

```
using namespace peter;
const double dVERSION = 2.0;
void MyClass::MyClass() {...}
```

Man sollte es jedoch **vorziehen**, eine **vertikale Teilung der Software mittels Client/Server-Modell** durchzuführen, was die Verwendung von gleichen Namen erlaubt.

3.2 Default-Konstruktor

Konstruktor ohne Argumente oder ausschließlich mit Default-Argumenten → Default-Initialisierung

Beispiel:

```
void MyClass::MyClass() : m_a(0),m_b(1),m_c(0),m_d(0)
{
}
```

oder (hier ist der normale Konstruktor auch schon enthalten):

```
void MyClass::MyClass(int a = 0,int b = 1,int c = 0,int d = 0)
     : m_a(a),m_b(b),m_c(c),m_d(d)
{
}
```

Anwendung:

```
MyClass Obj;
```

3.3 Copy-Konstruktor

Konstruktor zur Initialisierung eines Objektes mit Hilfe der Werte eines anderen Objektes → Es wird ein Objekt der gleichen Klasse als Argument übergeben.

Beispiel:

```
void MyClass::MyClass(const MyClass& Obj)
{
     if(this == &Obj)
          return *this;
     ...
     return *this;
}
```

Anwendung:

```
MyClass ObjA;
MyClass ObjB(ObjA);
```

3.4 explicit-Konstruktor

Konstruktor, der **nicht** zur impliziten Typ-Konvertierung herangezogen werden kann

→ Schlüsselwort `explicit`.

Hintergrund:

Normalerweise kann der Compiler **statt einem Objekt als Argument einer Funktion** auch den **Parameter dessen Konstruktors** entgegennehmen, vorausgesetzt der Konstruktor hat **nur 1 Argument**. Findet er eine solche Stelle, dann fügt er dort den Code für die Konstruktion des Objektes mit diesem Argument ein (implizite Typumwandlung).

Möchte man diese implizite Typkonvertierung abschalten, dann ist der Konstruktor mit `explicit` zu deklarieren.

Beachte: `explicit` darf <u>in der Implementierung nicht noch einmal vorangestellt</u> werden.

Beispiel:

```
class MyExplicitClass
{
      public:
            explicit MyExplicitClass(int);        //nur 1 Argum.
            explicit MyExplicitClass(double);     //nur 1 Argum.
      ...
};

class MyClass
{
      public:
            MyClass(int);         //nur 1 Argument
            MyClass(double);      //nur 1 Argument
      ...
};

void f(MyExplicitClass){...}

void g(MyClass){...}

void h(int i)
{
      f(i);      //nicht erlaubt
      g(i);      //-> MyClass Obj(i); g(Obj);
      ...
}
```

3.5 Zuweisungs-Operator

Der Operator = wird überschrieben:

```
class MyClass
{
      public:
            MyClass& operator=(const MyClass& Obj);
      ...
};
```

```
MyClass& MyClass::operator=(const MyClass& Obj)
{
     if(this == &Obj)
          return *this;
     ...
     return *this;
}
```

Somit ist Folgendes möglich:

```
MyClass ObjA;
MyClass ObjB;
ObjB = ObjA;
```

3.6 Abstrakte Klasse (= abstrakte Basisklasse)

Abstrakte Klasse = Klasse, die mindestens eine Methode enthält, welche **rein virtuell** ist → mindestens eine Methode ist mit "**`= 0`**" deklariert. Infolgedessen kann man kein Objekt von dieser Klasse erzeugen (instanziieren) und deshalb ist eine abstrakte Klasse immer nur als Basisklasse verwendbar: **Abstrakte Klasse = Abstrakte Basisklasse**.

Wenn die abstrakte Basisklasse nicht ausschließlich rein virtuelle Methoden enthält, dass heißt, sie dient nicht als Schnittstellen-Klasse (Abstract Mixin Base Class), sondern als echte Basis (Generalisierung), dann sollte der **Destruktor** immer **virtuell** sein und **zumindest leer implementiert** werden. Die Implementierung kann **nicht wirklich inline** geschehen, da virtual und inline sich gegenseitig ausschließen, wobei virtual dominiert:

```
class MyBase
{
public:
     virtual ~MyBase() {}
};
```

=

```
class MyBase
{
public:
     virtual ~MyBase();
};
MyBase::~MyBase() {}
```

Sie muss jedoch vorhanden sein, denn der **Compiler generiert auch für die Basisklasse immer einen Destruktor-Aufruf**! Da der Compiler die **inline-Implementierung zu einer normalen Funktion** macht, wenn er das Schlüsselwort **virtual** findet, kann man die leere Implementierung des Destruktors einfach inline mit in die Deklaration aufnehmen.

Eine **abstrakte Klasse mit ausschließlich rein virtuellen Methoden** (Schnittstellen-Klasse bzw. Abstract Mixin Base Class) hat keine Implementierung → dort gibt es **nur eine Header-Datei** (`*.h`) und keine Quellkode-Datei (`*.cpp`).

Beispiel:

```
class MyAbstractBase
{
      public:
            virtual ~MyAbstractBase() {} //Hack: nicht wirkl.inline
            virtual void Meth1() = 0; //rein virtuell
            void Meth2() { printf("Meth2()\n"); }
};

class MyAbstractMixinBase
{
      public:
            virtual void Meth2() = 0;
            virtual void Meth3() = 0;
            virtual void Meth4() = 0;
};
```

3.7 Default-Argumente

Um Default-Argumente zu definieren, setzt man in der Deklaration für alle Argumente ab einem bestimmten Argument einen Default-Wert an. Wenn man bspw. das dritte Argument per Default festlegt, dann muss auch das vierte, fünfte und alle weiteren per Default belegt werden:

```
void MyClass::MyClass(int Prm1 = 0,int Prm2 = 1);
void MyClass::SetObjVal(int Prm1,int Prm2,int Prm3 = 0,int Prm4 = 0);
```

Wenn nun beim Aufruf nur der erste Teil der Argumente (mindestens bis zum Anfang der Default-Argumente) angegeben wird, dann wird der Rest der Argumente mit den Default-Werten gespeist.

3.8 Unspezifizierte Anzahl von Argumenten

Hat man zum Zeitpunkt des Programmierens eine unspezifizierte Anzahl von Argumenten für eine Funktion, dann kann man dies mit `...` anzeigen:

Beispiel:

```
void format_string(string& strOUT,const char* szIN,...)
{
}
```

Nun ist der Programmierer selbst dafür verantwortlich, die folgenden Argumente in ihrer Art und Anzahl zu unterscheiden und auszuwerten.

Zur Formatierung eines Strings (`strOUT`) kann man zum Beispiel einen zweiten String (`szIN`) benutzen und so verschiedene Werte erst zur Laufzeit einfügen. Der zweite String dient als Template und bestimmt so die Art und Anzahl der folgenden Argumente (siehe: `printf()`). Zur Identifizierung von variablen

Teilen (Werten von Argumenten) benutzt man ein bestimmtes Zeichen (Bsp.: **%**), den sogenannten **Token**.

%d → String-Darstellung eines **Integers** wird eingebettet
Der Wert des Integers folgt als weiteres Argument
%s → **String** wird eingebettet
Der String folgt als weiteres Argument
%% → Das Zeichen **%** wird eingebettet
Es folgt hierfür kein weiteres Argument
...

Durch Parsen des Strings findet der Programmierer nun die enthaltenen Token (%) und deren Parameter (`d`, `s`, %, ...) und weiß dadurch, wie viele Argumente folgen und welcher Art diese sind. Mit `va_list()`, `va_start()`, `va_arg()` und `va_end()` kann man die Argumente entgegennehmen, wie folgende Implementierung zeigt:

```
inline void format_string(
          bool bAppend,string& strOUT,const char* szIN,...)
{
     //Initialisiere den Inhalt von strOUT:
     if(!bAppend)
          strOUT = "";
     //Initialisierung:
     va_list vaListVariableArguments;
     va_start(vaListVariableArguments,szIN);

     //Parse den String:
     int i = 0;
     double d = 0.0;
     void* pv = NULL;
     char szNumberString[128] = "";
     char szFormatString[64] = "";
     char byFSIdx = -1;
     short nFlag = 0; //keinen token gesehen
     while(*szIN)
     {
          switch(*szIN)
          {
               default:
                    if(!nFlag) //keinen token gesehen
                         strOUT.append(1,*szIN);
                    else
                         szFormatString[++byFSIdx] = *szIN;
                    break;
```

```
case '%':
      if(nFlag) //keinen token gesehen -> %%
      {
            strOUT.append(1,'%');
            nFlag = 0;
      }
      else
      {
            byFSIdx = -1;
            szFormatString[++byFSIdx] = '%';
            nFlag = 1; //token gesehen
      }
      break;
case 's':
      if(nFlag) //token gesehen
      {
            strOUT += va_arg(
                  vaListVariableArguments,char*);
            nFlag = 0;
      }
      else
      {
            strOUT.append(1,*szIN);
      }
      break;
case 'c':
case 'd':
case 'i':
case 'o':
case 'u':
case 'x':
case 'X':
      if(nFlag) //token gesehen
      {
            szFormatString[++byFSIdx] = *szIN;
            szFormatString[++byFSIdx] = 0x00;
            byFSIdx = -1;
            i = va_arg(
               vaListVariableArguments,int);
            sprintf(
               szNumberString,szFormatString,i);
            strOUT += szNumberString;
            nFlag = 0;
      }
      else
      {
            strOUT.append(1,*szIN);
      }
      break;
```

```
                    case 'e':
                    case 'E':
                    case 'f':
                    case 'g':
                    case 'G':
                          if(nFlag) //falls token gesehen
                          {
                                szFormatString[++byFSIdx] = *szIN;
                                szFormatString[++byFSIdx] = 0x00;
                                byFSIdx = -1;
                                d = va_arg(
                                   vaListVariableArguments,double);
                                sprintf(
                                   szNumberString,szFormatString,d);
                                strOUT += szNumberString;
                                nFlag = 0;
                          }
                          else
                          {
                                strOUT.append(1,*szIN);
                          }
                          break;
                    case 'p':
                          if(nFlag) //falls token gesehen
                          {
                                szFormatString[++byFSIdx] = *szIN;
                                szFormatString[++byFSIdx] = 0x00;
                                byFSIdx = -1;
                                pv = va_arg(
                                      vaListVariableArguments,void*);
                                sprintf(
                                  szNumberString,szFormatString,pv);
                                strOUT += szNumberString;
                                nFlag = 0;
                          }
                          else
                          {
                                strOUT.append(1,*szIN);
                          }
                          break;
                }
                ++szIN;
          }
          va_end(vaListVariableArguments);
    }
```

Zu diesem Beispiel:

Diese Implementierung nutzt `sprintf()`, da es nur ein Beispiel ist. Möchte man einen guten Ersatz für den Umweg über **`char[]`**

```
string strTest;
...
char szTest[1024] = "";
const char szHelloWorld[] = "Hello World";
short nNum = 5;
double dNum = 75.333;
char cHex = 0x4A;
void* pVoid = &cHex;
sprintf(
      szTest,
      "s: '%s'\nd: '%03d'\nf: '%3.1f'\nX: '0x%02X'\np: '%p'",
      szHelloWorld,
      nNum,
      dNum,
      cHex,
      pVoid);
strTest = szTest;
```

schaffen, also so etwas wie...

```
string strTest;
...
const char szHelloWorld[] = "Hello World";
short nNum = 5;
double dNum = 75.333;
char cHex = 0x4A;
void* pVoid = &cHex;
format_string(
      false,
      strTest,
      "s: '%s'\nd: '%03d'\nf: '%3.1f'\nX: '0x%02X'\np: '%p'",
      szHelloWorld,
      nNum,
      dNum,
      cHex,
      pVoid);
```

...dann muß man die Dinge, die `sprintf()` tut, selbst ausprogrammieren, um hinreichend kurze Laufzeiten der Funktion zu haben (was einiges an Arbeit bedeutet).

3.9 l-value und r-value

Es ist ein grundsätzlicher Unterschied, ob ein Objekt in einer Anweisung als l-value oder als r-value benutzt wird:

l-value = Left Value

→ Das Objekt steht auf der **linken** Seite eines **Gleichheitszeichens**. Es erwartet also sozusagen einen neuen Wert (Inhalt) in Form einer Zuweisung (**`operator=(...)`**).

r-value = Right Value

→ Das Objekt steht auf der **rechten** Seite eines **Gleichheitszeichens**. Es übergibt also seinen Wert (Inhalt) an ein anderes Objekt. Hierbei kann eine Typumwandlung geschehen, wenn ein entsprechender Operator definert wurde (Beispiel: `operator int()`).

3.10 Funktionszeiger

Funktion und dazu passenden Zeiger definieren:

```
int Func(int nParm)
{
     ...
}

int (*pFunc)(int nPrm);
```

→ Funktion über Zeiger aufrufen:

```
pFunc = Func;
pFunc(5);
```

Beispiel:

```
#include <stdio.h>

struct Helper
{
     public:
          static void Cut()
          {
               printf("Cut()\n");
          }
          static void Copy()
          {
               printf("Copy()\n");
          }
          static void Paste()
          {
               printf("Paste()\n");
          }
};

typedef void (*PMENU)();

int main()
{
     PMENU Edit[] = {&Helper::Cut,&Helper::Copy,&Helper::Paste};

     Edit[0](); //entspricht Aufruf von Helper::Cut()
     Edit[1](); //entspricht Aufruf von Helper::Copy()
     Edit[2](); //entspricht Aufruf von Helper::Paste()
     return 0;
}
```

3.11 union

3.11.1 Allgemeines

Eine `union` ist eine besondere Struktur, in der sich **alle Elemente an der gleichen Adresse** befinden, d.h. man hat **verschiedene Speicherstrukturen für ein und denselben Speicherplatz**. Der Speicherplatz ist natürlich so groß wie das größte Element der union. Dies ist auch der Grund dafür, dass in einer union keine dynamischen Elemente, wie Listen der STL, vorkommen dürfen, sondern **nur statische Elemente**.

Das Besondere an der `union` ist, dass der Compiler anhand des verwendeten Namens den zugehörigen Datentyp erkennt und benutzt.

Grundsätzlich gibt es 2 Arten der Anwendung einer `union`:

- Objekte unterschiedlichen Typs in eine Sequenz packen
  (Bsp.: STL-Container `list`)
- Mehrere Datenstrukturen für dieselben Daten verwenden
  (hardwareabhängig)

3.11.2 Objekte unterschiedlichen Typs in eine Sequenz packen (list)

Eine `union` kann manchmal sehr hilfreich sein, wenn man versucht, verschiedene Elemente in eine Sequenz (z.B. list) zu packen. Normalerweise ist dies nicht möglich, da eine Sequenz (STL-Container oder auch ein Array) nur für einen Typ gedacht ist.

Beispiel:

Es soll eine Liste von Teilen (Stückliste) einer bestimmten Baueinheit gespeichert werden:

```
#include <stdio.h>
#include <list>
using namespace std;

struct MyHousing
{
     double dSizeX;
     double dSizeY;
     double dSizeZ;
};

struct MyMiniSwitch
{
     double dMaxVoltage;
     double dMaxCurrent;
     unsigned long lLayoutCode;
};
```

```
struct MyLED
{
      unsigned long lRGBColor;   //0x00RRGGBB;
      double dForwardVoltage;
      double dCurrent1000mcd;
      double dSize;
};

struct MyResistor
{
      double dResistance;
      double dMaxPower;
      double dSize;
};

struct MyElement
{
      unsigned long lTag;
      union //anonymous union
      {
            struct MyHousing      Housing;        //lTag = 1
            struct MyMiniSwitch   MiniSwitch;     //lTag = 2
            struct MyLED          LED;            //lTag = 3
            struct MyResistor     Resistor;       //lTag = 4
      };
};
int main()
{
      list<MyElement> listElements;
      MyElement Element;
      Element.lTag = 1;
      Element.Housing.dSizeX = 3.4;
      Element.Housing.dSizeY = 4.4;
      Element.Housing.dSizeZ = 6.5;
      listElements.push_back(Element);

      Element.lTag = 3;
      Element.LED.lRGBColor = 0x00FF0000;        //red
      Element.LED.dForwardVoltage = 1.7;         //1.7V
      Element.LED.dCurrent1000mcd = 0.010;       //10 mA
      Element.LED.dSize = 5;                     //5 mm
      listElements.push_back(Element);

      printf("List of Elements:\n");
      list<MyElement>::iterator it;
```

```
	for(it = listElements.begin();it != listElements.end();++it)
	{
		switch((*it).lTag)
		{
			case 1:
				printf("Housing:\n");
				printf("   SizeX: %lf\n",(*it).Housing.dSizeX);
				printf("   SizeY: %lf\n",(*it).Housing.dSizeY);
				printf("   SizeZ: %lf\n",(*it).Housing.dSizeZ);
				break;
			case 3:
				printf("LED:\n");
				printf("   RGB-Color: %lf\n",
					(*it).LED.lRGBColor);
				printf("   Forward Voltage: %lf\n",
					(*it).LED.dForwardVoltage);
				printf("   Current at 1000mcd: %lf\n",
					(*it).LED.dCurrent1000mcd);
				printf("   Size: %lf\n",(*it).LED.dSize);
				break;
			default:
				printf("Not printable [lTag] = %ld:\n",
					(*it).lTag);
		}
	}
	return 0;
}
```

3.11.3 Mehrere Datenstrukturen für dieselben Daten(hardwareabhängig)

Mit Hilfe einer `union` kann man ein und dieselben Daten über mehrere Datenstrukturen ansprechen. So kann man bspw. den Speicherbereich über seriell empfangene Datenbytes sequentiell füllen und die Daten anschließend über eine Struktur ansprechen oder umgekehrt.

Beispiel:

Hier wird der Speicherbereich über eine `unsigned long` Variable gefüllt und dann Byte für Byte auf den Bildschirm gebracht. Man beachte hier besonders, dass das niederwertigste Byte (**LSB**) des **`unsigned long`** auf Intel-Rechnern (PC) **zuerst abgespeichert wird**, was der folgende Code beweist:

```
union ID
{
	unsigned char abyByteBuffer[4];
	unsigned long dwValue;
};
```

```
int main()
{
      ID id;
      id.dwValue = 0x04030201L;
      printf(
            "id.dwValue              [Hex]: %08X\n",
            id.dwValue);
      printf(
            "id.abyByteBuffer[0]..[3] [Hex]: %02X%02X%02X%02X\n",
            id.abyByteBuffer[0],
            id.abyByteBuffer[1],
            id.abyByteBuffer[2],
            id.abyByteBuffer[3]);
      return 0;
}
```

Auf einem **Intel-Rechner (PC)** findet man so als Ausgabe:

```
id.dwValue               [Hex]: 04030201
id.abyByteBuffer[0]..[3] [Hex]: 01020304
```

Man hat also byteweise die gleiche Signifikanz, aber insgesamt nicht. Dieses Problem is allgemein bekannt und wird in der Regel mit "little-endian" im Gegensatz zu "big-endian" bezeichnet:

- LSBF (**least significant byte first**) → "**little-endian**"-CPU-Systeme
  Bsp.: Microsoft-Windows-NT auf PC mit Intel-80x86-CPU
  LINUX auf PC mit Intel-80x86-CPU
  Digital-VMS auf VAX
- MSBF (**most significant byte first**) → "**big-endian**"-CPU-Systeme
  Bsp.: Apple-MacOS auf Macintosh mit Motorola-M68xxx-CPU
  Sun-Solaris (UNIX) auf Workstation mit Sparc-CPU
  HP-UX (UNIX) auf Workstation mit PA-Risc
  Java-JVM auf allen Plattformen unabhängig von der CPU
  TCP/IP auf Netzwerken

Das Betriebssystem **Linux** erlaubt je nach Kompilierung die eine oder die andere Betriebsart:

In `/usr/src/linux/asm/byteorder.h`:

```
#include <linux/byteorder/big_endian.h>
```

oder

```
#include <linux/byteorder/little_endian.h>
```

Dann gibt es noch die "**bi-endian**"-CPUs, welche **umschaltbar** sind. Hierzu gehören Merced, IA64, PowerPC, ... Was jedoch ein Rechnersystem benötigt **um wirklich damit arbeiten zu können**, ist eine **entsprechende Architektur** wie z.B. die HP/Intel **EPIC**-Architektur (Explicitly Parallel Instruction Computing). Gegenbeispiele: Die Apple-PowerMac-Architektur erlaubt nur den "big-endian"-Betrieb der "bi-endian"-PowerPC-CPU und die Sun-Sparc-Station-Architektur erlaubt nur den "big-endian"-Betrieb der "bi-endian"-Sparc-CPU.

Eine Sonderrolle spielen PDP-11-Systeme (NUXI) von Digital, wie man dem Beispiel entnehmen kann:

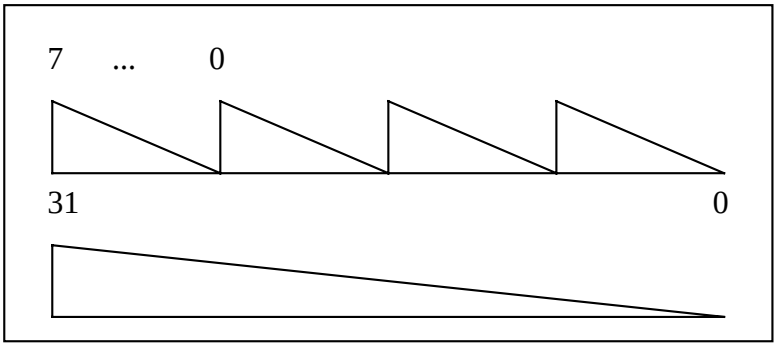
```
unsigned long lMyValue = 0x04030201L;
```

Speicherung:

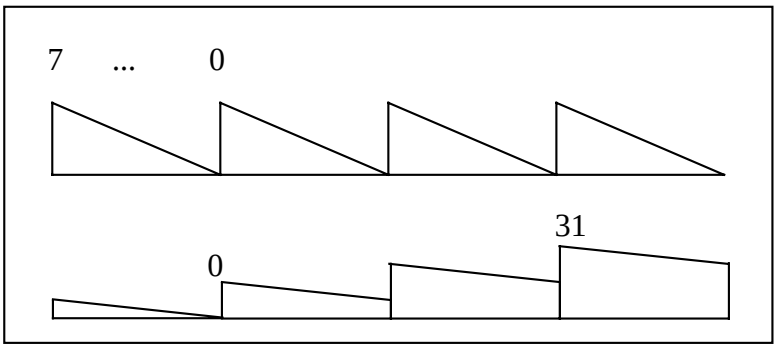
little-endian:	`01 02 03 04`	PC (WinNT), VAX (VMS)
big-endian:	`04 03 02 01`	Mac (MacOS), Sparc (Solaris), Java-JVM, TCP/IP
NUXI:	`03 04 01 02`	PDP-11 (NUXI)

Wenn man sich fragt, warum sich in der Netzwerk-Welt die "big-endian"-Technologie durchgesetzt hat, dann ist mit Sicherheit ein Grund, dass TCP/IP bereits früh ein Bestandteil von UNIX war (Berkeley-UNIX 4.2BSD 1983). Sicherlich ist aber auch ein Grund, dass "big-endian" für die Datenübertragung recht logisch erscheint, wenn man sich den Verlauf der Signifikanz ansieht:

`"big-endian": unsigned long`



`"little-endian": unsigned long`



Diese Tatsache sollte man immer bei der **Übertragung von Daten** beachten, wie z.B. über die **serielle Schnittstelle eines Rechners**. Die serielle Rechner-Schnittstelle ist in der Regel eine RS232-Schnittstelle, die Dateien oder sonstige Daten Byte für Byte versendet oder empfängt (z.B. zum und vom Modem). Man sollte für die Daten auf beiden Seiten der Übertragungsstrecke immer nur Byte-Puffer (Dateien) benutzen, also das erste versendete Byte `abySendByteBuffer[0]` auch wieder in das erste Byte auf der Empfangsseite `abyRecvByteBuffer[0]` schreiben. Danach kann man dazu übergehen, die Daten zu interpretieren und rechnerabhängige Zahlenformate (alles was größer als ein Byte ist) einzusetzen.

3.11.4 Bitfelder zum Abtasten von Byte-Streams (hardwareabhängig)

In einer Struktur können Speicherbereiche der Größe 1...32 Bit als Feld definiert werden. Der Compiler rundet die Gesamtgröße der Struktur auf das nächste volle Byte auf. **Normalerweise** hat man nur 3 Speichergrößen zur Verfügung:

```
struct Conventional
{
     unsigned char   Byte;           //1 Byte
     unsigned short  Word;           //2 Byte
     unsigned long   DoubleWord;     //4 Byte
};
```

Ein Bitfeld benutzt nur einen Teil eines Datentyps (Bsp.: 7 Bit). Um direkten Zugriff darauf zu haben, kann man eine Struktur mit Bitfeldern definieren. Möchte man einen '`unsigned long`'-Speicherbereich in einen 25-Bit- und einen 7-Bit-Bereich aufteilen, dann sieht das folgendermaßen aus:

```
struct MyBitfieldData
{
     unsigned long ID           :   25;    //25 Bit ID
     unsigned long Key          :    7;    //7 Bit Key
};
```

Hierbei gilt wieder das bereits im letzten Kapitel Gesagte: Es hängt von der verwendeten **Hardware** ab, wie die Bytes gespeichert werden ("**little-endian**" oder "**big-endian**"). Möchte man **von der Hardware unabhängig** werden, dann darf man nur **`unsigned char`** (Byte) verwenden. Aber selbst dann ist zu beachten, dass die Bitfeld-Definition auf der Grundlage von **Bytes** unterschiedlich ist:

Auf little-endian wird das niederwertigste Bit zuerst definiert:

Beispiel: Definition von

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Code						Flag	Valid

Little-Endian:

```
struct MyStreamByte
{
     unsigned char Valid        :    1;    //???? ???X
     unsigned char Flag         :    1;    //???? ??X?
     unsigned char Code         :    6;    //XXXX XX??
};
```

Big-Endian:

```
struct MyStreamByte
{
     unsigned char Code        :    6;   //XXXX XX??
     unsigned char Flag        :    1;   //???? ??X?
     unsigned char Valid       :    1;   //???? ???X
};
```

Beispiel für eine "**little-endian**"-Hardware (WinNT oder LINUX auf einem PC):

Mit Hilfe einer `union`, bestehend aus einem Byte-Puffer und einer Struktur mit Bitfeldern, lässt sich ein **Byte-Stream** (Datenstrom in Form von Bytes) leicht **abtasten**, wie das Beispiel zeigt:

```
struct MyStreamData
{
     unsigned char ID_LO        : 8;  //8 Bit ID-L
     unsigned char ID_HI        : 7;  //7 Bit ID-H (untere 7 Bit)
     unsigned char Valid        : 1;  //1 Bit Valid (oberstes Bit)
     unsigned char Flag         : 1;  //1 Bit Flag (unterstes Bit)
     unsigned char TextLen      : 7;  //7 Bit (obere 7 Bit,0..127)
     unsigned char szText[128];       //0..127 Zeichen Text + 0x00
};

union MyBitStream
{
     unsigned char abyStreamBuffer[131];
     MyStreamData StreamData;
};

int main()
{
     MyBitStream bs;
     bs.abyStreamBuffer[0] = 0x01; //ID_LO = 0000 0001
     bs.abyStreamBuffer[1] = 0x03; //Valid = 0 / ID_HI = 000 0011
     bs.abyStreamBuffer[2] = 0xFE; //TextLen = 0000 001 / Flag = 0

     bs.StreamData.szText[127] = 0x00;
     for(int i = 5;i < 131;++i)
          bs.abyStreamBuffer[i] = 'A' - 5 + i;

     size_t size = sizeof(bs.abyStreamBuffer);

     printf("BitStream bs.abyStreamBuffer [ %u Byte ]:\n",size);
     printf(    "ID: 0x%02X%02X\n",
                bs.StreamData.ID_HI,bs.StreamData.ID_LO);
     printf("Valid: %0ld\n",bs.StreamData.Valid);
     printf("Flag: %0ld\n",bs.StreamData.Flag);
     printf("TextLen: %0ld\n",bs.StreamData.TextLen);
     printf("szText: %s\n",bs.StreamData.szText);

     return 0;
}
```

3.11.5 Maske per Referenz anwenden

Wenn eine Funktion einen Byte-Strom analysieren soll, wobei sie einen Zeiger auf den Byte-Strom erhält, dann sollte man es aus Performance-Gründen vermeiden, den Byte-Strom nochmal in die `union` einzulesen/zu kopieren (wie im vorherigen Absatz gezeigt). Stattdessen sollte man den Byte-Strom mit der `union` referenzieren.

Beispiel:

```
void Func(const unsigned char* const pbyBuf,unsigned long dwLen)
{
	static const unsigned long dwMaskOffset = 5;
	if((dwLen + dwMaskOffset) < MyUnion::Len)
		return;

	//Ansetzen der Maske am richtigen Index):
	MyUnion& Mask = *((MyUnion*) &pbyBuf[dwMaskOffset]);
	if(Mask.m_bMember) //Anwendung der Maske
	{
		...
	}
}
```

3.11.6 Test-Funktion zum Testen der Maschine auf little- bzw. big-endian

Wie man unschwer aus dem oben Gesagten erkennen kann, lässt sich die `union` wegen ihrer Hardwareabhängigkeit genial nutzen, um die Maschine (CPU) auf little- bzw. big-endian zu testen:

```
bool IsBigEndianMachine()
{
	static short nIsBigEndian = -1;
	if(nIsBigEndian == -1)
	{
		union ID
		{
			unsigned char acByteBuffer[4];
			unsigned long dwValue;
		};
		ID id;
		id.dwValue = 0x44332211L;
		if(id.acByteBuffer[0] == 0x11)
			nIsBigEndian = 0; //little-endian (Intel-CPU)
		if(id.acByteBuffer[0] == 0x44)
			nIsBigEndian = 1; //big-endian (Sparc-CPU)
	}
	return (bool) nIsBigEndian;
}
```

```
int main()
{
     if(IsBigEndianMachine())
          printf("Big Endian\n");
     else
          printf("Little Endian\n");
     return 0;
}
```

3.12 extern "C" zum Abschalten der Namenszerstückelung

Der C++-Compiler vergibt jeder Funktion einen eindeutigen Namen. Dies ist erforderlich, da ja die Überladung von Funktionen möglich ist, also mehrere Funktionen mit gleichem Namen im Code stehen können. Man kann den Vorgang jedoch mit

`extern "C"`

(C-Linkage) unterdrücken.

In dem Fall wird die Funktion genau mit dem Namen eingebunden, der ihr vergeben wurde.

Beispiel:

```
extern "C" void func(unsigned char uiFlags);
```

Bemerkung:

Der Vorgang des name mangling ist nicht standardisiert und von Compiler zu Compiler unterschiedlich. Deshalb müssen Funktionen, die anderen Programmierern in Form von Binärcode (`*.obj`, `*.so`, `*.dll`, `*.ocx`) und dazugehöriger Header-Datei zur Verfügung gestellt werden, mit der Option **`extern "C"`** übersetzt werden.

4. Grundsätzliche Regeln beim Programmieren

4.1 Include-Wächter verwenden

Um wiederholtes Einfügen von Headern zu vermeiden, wird der Code von Header-Dateien eingefasst in folgende `#ifndef`-Anweisung:

```
#ifndef _HEADERNAME_H_
#define _HEADERNAME_H_
   ... //Header-Code
#endif
```

4.2 Kommentar // dem Kommentar /* */ vorziehen

Wenn man immer nur // zur Kommentierung benutzt, dann kann man zum Testen ganz einfach Blöcke mit /* */ auskommentieren.

4.3 Optimiere die Laufzeit immer gleich mit

Es ist falsch zu glauben, man könne im Nachhinein mit kleinen Code-Änderungen oder durch Compiler-Switches eine wesentliche Optimierung seines Codes hinsichtlich der Laufzeit erreichen. Wenn man sich nicht gleich die Arbeit macht mitzudenken, dann muss man für eine spätere Optimierung wesentliche Teile des Codes noch einmal neu schreiben.

4.3.1 Objekte erst dort definieren, wo sie gebraucht werden

Beachte:

Wenn man nie in den Sichtbarkeitsbereich eines Objektes gelangt, also irgendwo zwischen der öffnenden Klammer **{** davor und der schließenden Klammer **}** danach, dann erfolgt auch nicht die Konstruktion (bei der Definition) bzw. die Destruktion (bei der schließenden Klammer **}**).

→ Immer zuerst alle Eventualitäten abfangen und das Objekt erst dann, wenn es auch wirklich gebraucht wird, konstruieren, also ggf. in einem Block einer if-Anweisung.

Beispiel:

Statt:

```
TextParser Parser(strText);
unsigned long dwCmd = 0L;
if(strText.length() > 3)
{
      dwCmd = Parser.GetCmd()
}
```

Besser:

```
unsigned long dwCmd = 0L;
if(strText.length() > 3)
{
      TextParser Parser(strText);
      dwCmd = Parser.GetCmd()
}
```

Ein weiterer Vorteil dieser Vorgehensweise ist, dass man kleine logische Blöcke (inclusive der Variablen-Definitionen) erhält, die einem das Herunterbrechen einer zu groß gewordenen Funktion erleichtern. Man denke hier an den Ausdruck "gewachsener Code".

Aus diesen Gründen sollte man sich generell angewöhnen, Variablen immer erst dort zu definieren, wo man sie braucht.

4.3.2 Zuweisung an ein Objekt mit der Konstruktion verbinden

Wenn man ein Objekt konstruiert, dann will man ihm auch meist einen Wert zuweisen. Es ist dann effektiver, direkt über die richtigen Argumente des Konstruktors (Bsp.: Copy-Konstruktor) die Zuweisung durchzuführen, als zuerst den Default-Konstruktor aufzurufen und dann später die Zuweisung zu machen. Denn im zweiten Fall werden zunächst alle Member mit Default-Werten belegt und später nochmals mit anderen Werten. Eine sofortige Zuweisung bei der Konstruktion, wie

```
MyClass ObjB = ObjA;
```

wird vom Compiler in den Aufruf eines Copy-Konstruktors umgesetzt und ist damit identisch mit

```
MyClass ObjB(ObjA);
```

4.3.3 return, break und continue mit Geschick einsetzen

Es gibt einige typische "Anti-Effektivitäts-Regeln", denen man leider allzu oft begegnet. Hierzu gehören Regeln wie "Für jedes `if` ein `else`" oder "Für jeden Block `{...}` nur `einen` Ausgang". Der Profi jedoch sollte wissen, dass Schlüsselwörter wie `return`, `break` und `continue` die Basis einer laufzeitoptimierten Programmierung bilden, ja sogar Code einfacher lesbar machen können.

`return`:

Im Folgenden sind 2 Funktionen gezeigt: `Check1()` und `Check2()`. Beide Codes tun dasselbe. Den zweiten jedoch hatte ich wesentlich schneller nachgeschaut, da er immer wieder das gleiche Muster erkennen lässt.

```
#include <stdio.h>
#include <string>
using namespace std;
string GetDataBaseRecord(unsigned short nID)
{
      switch(nID)
      {
            case 0:
                  return string("002345");
            case 1:
                  return string("011345");
            case 2:
                  return string("012245");
            case 3:
                  return string("012335");
            case 4:
                  return string("012344");
            default:
                  break;
      }
      return string("");
}
int Check1()
{
   bool bSuccess = false;
   string strData = GetDataBaseRecord(0);
   if(strData.length()>=2)
   {
      if(strData[0] == strData[1])
      {
         strData = GetDataBaseRecord(1);
         if(strData.length()>=3)
         {
            if(strData[1] == strData[2])
            {
               strData = GetDataBaseRecord(2);
               if(strData.length()>=4)
               {
                  if(strData[2] == strData[3])
                  {
                     strData = GetDataBaseRecord(3);
```

```
                    if(strData.length()>=5)
                    {
                       if(strData[3] == strData[4])
                       {
                          strData = GetDataBaseRecord(4);
                          if(strData.length()>=6)
                          {
                             if(strData[4] == strData[5])
                             {
                                bSuccess = true;
                             }
                             else
                             {
                                bSuccess = false;
                             }
                          }
                       }
                       else
                       {
                          bSuccess = false;
                       }
                    }
                    else
                    {
                       bSuccess = false;
                    }
                 }
                 else
                 {
                    bSuccess = false;
                 }
              }
              else
              {
                 bSuccess = false;
              }
           }
           else
           {
              bSuccess = false;
           }
        }
        else
        {
           bSuccess = false;
        }
     }
     else
     {
        bSuccess = false;
     }
  }
  else
  {
     bSuccess = false;
  }
```

```
   if(bSuccess)
   {
      return 0;
   }
   else
   {
      return 1;
   }
   return 0;
}
int Check2()
{
      string strData = GetDataBaseRecord(0);
      if(strData.length() < 2)
            return 1;
      if(strData[0] != strData[1])
            return 1;

      strData = GetDataBaseRecord(1);
      if(strData.length() < 3)
            return 1;
      if(strData[1] != strData[2])
            return 1;

      strData = GetDataBaseRecord(2);
      if(strData.length() < 4)
            return 1;
      if(strData[2] != strData[3])
            return 1;

      strData = GetDataBaseRecord(3);
      if(strData.length() < 5)
            return 1;
      if(strData[3] != strData[4])
            return 1;

      strData = GetDataBaseRecord(4);
      if(strData.length() < 6)
            return 1;
      if(strData[4] != strData[5])
            return 1;
      return 0;
}
int main()
{
      int nError = Check1();
      nError = Check2();
      return nError;
}
```

Ich denke, man muss nicht viel zu dem Beispiel sagen?

Eine gute Regel:
Eine Funktion ist immer **so früh wie möglich zu verlassen**. Deshalb sollte man **bevorzugt überprüfen, welche Bedingungen nicht erfüllt sind** (angefangen bei den Argumenten), und dann ggf. mit `return` rausgehen.

`break` und `continue`:

Ebenso wie für `return` lassen sich endlos viele Beispiele für `break` und `continue` finden. Der Abbruch einer Schleife (`break`) ist immer dann sinnvoll, wenn weitere Schleifendurchgänge kein neues Resultat mehr bringen. Das sofortige Einleiten der nächsten Schleife (`continue`) ist immer dann angebracht, wenn die aktuelle Schleife kein neues Resultat mehr bringt. Somit ist auch hier bevorzugt (vor allem anderen) zu überprüfen, ob ein `break` oder `continue` angebracht ist.

Beispiel:

Statt:

```
list<unsigned short>::iterator it;
unsigned short i = 1;
bool bOk = true;
for(it = listNums.begin();it != listNums.end();++it,++i)
{
	if((*it) != 100)
	{
		if((*it) != i)
		{
			bOk = false;
		}
		else
		{
		}
	}
}
```

Besser:

```
list<unsigned short>::iterator it;
unsigned short i = 1;
bool bOk = true;
for(it = listNums.begin();it != listNums.end();++it,++i)
{
	if((*it) == 100)
		continue;

	if((*it) != i)
	{
		bOk = false;
		break;
	}
}
```

Sofort die nächste Schleife einleiten, wenn keine weitere Überprüfung nötig ist (`continue`). Sofort die ganze Schleife abbrechen (`break`), wenn das Ergebnis feststeht.

4.4 Laufvariable im Schleifenkopf definieren

Wenn man die Laufvariable immer **im** Schleifenkopf definiert, dann ist sichergestellt, dass man in jeder Schleife mit einer eigenen Laufvariablen arbeitet. Man kann dann nie fälschlicherweise die Laufvariable einer übergeordneten Schleife manipulieren, selbst wenn man den gleichen Namen verwendet:

```
#include <stdio.h>
#include <string>
using namespace std;
int main()
{
      string strText1("abcd");
      string strText2("Hello ");
      for(unsigned short i = 0;i < strText1.length();++i)
      {
            printf("\n%c.) ",strText1[i]);
            for(unsigned short i = 0;i < strText2.length();++i)
            {
                  printf("%c",strText2[i]);
            }
      }
      return 0;
}
```

4.5 Der Stack ist immer dem Heap (new/delete) vorzuziehen

Man sollte immer wenn es möglich ist den Stack benutzen, d.h. Objekte **nicht** mit `new` erzeugen. Dies sollte zumindest möglich sein, wenn es sich um **lokale** Objekte handelt. Im anderen Fall (Bsp.: Stacküberlauf) bietet jedoch die STL in der Regel immer irgendeinen passenden Container an (Bsp.: `string`, `list`, `set`, ...), der das Heap-Management übernehmen kann und selbst als Stack-Variable benutzt wird. So kann man mit `string` auch einfach das Problem der variablen Stringlänge, die sich erst zur Laufzeit ergibt, lösen ohne den Heap selbst zu managen. Dabei kann die Länge des Strings sogar noch ständig variiert werden, ohne dass man mit `new`/`delete` spielen muss.

Statt:

```
MyClass* pObj = new MyClass("Test"); //Heap -> delete erforderlich
char* sText = new char[len]; //Länge zur Laufzeit einstellen
...
delete pObj;
delete[] sText;
```

Besser:

```
MyClass Obj("Test"); //Stack -> kein delete erforderlich
string strText(""); //vollkommen dynamische Länge
...
```

Gründe, den Stack dem Heap vorzuziehen:

- Die **Destruktion** wird **automatisch** vorgenommen
- Automatischer Destruktoraufruf beim **stack-unwinding von** `try-catch`
- Es kann kein `delete` vergessen werden → keine Speicherlöcher
- Es kann nicht fälschlicherweise `delete` statt `delete`[] nach "`new char[]`" verwendet werden → keine Speicherlöcher durch vergessene Klammern `[] (`ohne `[]` löscht `delete` nur die erste Speicherstelle des `char[]`-Arrays)
- "`string`" hat im Gegensatz zu "`new char[]`" eine vollkommen dynamische Länge
- Stack-Variablen können vom Compiler **optimaler** genutzt werden
- Heap-Management sollte man generell immer der STL überlassen

4.6 protected nur bei Basisklassen

Wenn man nicht gerade eine Basisklasse schreibt, dann sollte man Methoden, die versteckt werden sollen (also die Implementierungs-Methoden), hinter **`private`** verstecken.
Macht man später eine Basisklasse daraus, dann kann man gezielt Methoden `protected` machen, um sie von der Child-Klasse aus nutzen zu können. Im Einzelfall sollte man sorgfältig prüfen, ob die Methode dazu virtual gemacht werden muss und ob sie in dem Fall von der Child-Klasse entsprechend überschrieben wird (ggf. ist ein Aufruf der Basisklassen-Methode in der Child-Klassen-Methode zu implementieren).

4.7 Keine Fehler beim Mischen von C- und C++-Code machen

Es ist Folgendes zu beachten, wenn man C-Code in ein C++-Projekt aufnimmt:

- Die *.obj-Dateien des C-Compilers müssen kompatibel mit denen des C++-Compilers sein

- Eine C++-Funktion, die von C-Code aufgerufen wird, muss mit **`extern "C"`** deklariert werden

```
extern "C" void func();
```

- Die `main()`-Funktion sollte sich in einer *.cpp-Datei befinden, die mit dem C++-Compiler übersetzt wird, da dieser dort die Konstruktion und Destruktion für statische Objekte des C++-Codes einfügt

- `new/delete` und `malloc/free` dürfen nicht gemischt verwendet werden

- Von C genutzte `struct`-Deklarationen müssen sich in `*.c`-Dateien befinden, die mit dem C-Compiler übersetzt werden, da der C++-Compiler `public`-Klassen daraus macht.

4.8 Ungarische Notation verwenden

Die ungarische Notation von Microsoft-Programmierer Charles Simonyi ist ein wirklich gutes Werkzeug, um den Überblick über die Variablen bei komplexen Projekten zu behalten. Die ungarische Notation kennzeichnet alle Variablen-Namen mit dem Typ, indem sie ein Präfix vergibt:

Stufe 1: Präfix unmittelbar vor dem Namen:

```
bBoolean    bool, BOOL → true,TRUE oder false,FALSE

cName       char → 8 Bit mit Vorz., -128...+127, 0..127 = ASCII-Std.
nName       short → 16 Bit mit Vorz., -32768...+32767
lName       long → 32 Bit mit Vorz., -2147483647...+2147483648
iName       int → auf 16 Bit Betriebssystem short, auf 32 Bit long

byName      unsigned char → 1 Byte, 0...255, 0x0...0xFF
wName       unsigned short → 2 Byte, 0...65535, 0x0...0xFFFF
dwName      unsigned long → 4 Byte, 0...4294967295, 0x0...0xFFFFFFFF
uiName      unsigned int →  auf 16 Bit Betriebssystem unsigned short
                            auf 32 Bit Betriebssystem unsigned long

sName       char[] → Zeichenkette, Vektor von char-Elementen
szName      char[] → Zeichenkette, die mit 0x00 endet (zeroterminated)
strName     string → string-Object (STL)

fName       float → 32 Bit Fließkomma, 7 Stellen
dName       double → 64 Bit Fließkomma, 15 Stellen

eName       enum → Aufzählung von int
vName       void -> kann alles sein
```

Stufe 2: Präfix vor dem Präfix von Stufe 1:

```
a...        Array (Vektor, Matrix), Bsp.: int anNumbers[256];
p...        Pointer, Bsp.: MyClass* pObj;
sp...       Smart-Pointer, Bsp.: MyClassPtr spObj;
```

Stufe 3: Präfix vor dem Präfix von Stufe 2:

```
g_...       Globale Variable
m_...       Member-Variable
```

Man sollte aber auch sonst auf einleuchtende Präfixe zurückgreifen:

```
list<MyClass>   listObjs;
set<MyClass>    setObjs;
```

Die Klassen und Strukturen eines Projektes sollte man sinnvoll bezeichnen und **nur bei Verkauf als Bibliothek an Dritte** ein herstellerbezeichnendes Präfix davorstellen (Bsp.: `class YString`). Nutzt man hingegen eine fremde Bibliothek, dann sollte deren Hersteller ebenfalls diese Regel beherzigen (Bsp.: Die Bibliothek 'MFC' von Microsoft benutzt das 'C': `class CString`). Ein Präfix für Klassennamen sollte jedoch wegen der Wiederverwendbarkeit von Code in anderen Projekten **nie projektabhängig** sein bzw. den Projektnamen beinhalten.

Beispiele:

```
m_szText:
     Member-Variable, nullterminierter String

dValue:
     Lokale Variable, double-Wert

byFlags:
     Lokale Variable, unsigned char-Wert (Byte)

m_mapByte2DoubleWord:
     Member-Variable, STL-Container map
     (mit unsigned char-Key und unsigned long-Value)

strText:
     Lokale Variable, STL-string-Objekt

m_setIDs:
     Member-Variable, STL-Container set
```

4.9 Eingebaute Datentypen nie hinter typedef verstecken

Es ist ein Irrglaube, dass man durch einfache Veränderung eines `typedef` in irgendeiner zentralen Header-Datei einen Datentyp umschalten kann. Man ändert ja dabei nichts an der Wertebereichsüberprüfung. Im Gegenteil: Man sollte sich immer genau überlegen, welcher Datentyp wo passt und diesen dann hart kodiert einsetzen und überprüfen.

Beispiel:

Funktioniert:

```
#include <stdio.h>

typedef unsigned long UNSIGN;
int main()
{
     UNSIGN a = 60000;
     UNSIGN b = 50000;
     UNSIGN c = 0;
     if(a > b)
     {
          if(a <= 60000)
               c = (UNSIGN) (a + b);
     }
     printf("c = %ld\n",c);
     return 0;
}
```

Funktioniert nicht:

```
#include <stdio.h>

typedef unsigned short UNSIGN;
int main()
{
      UNSIGN a = 60000;
      UNSIGN b = 50000;
      UNSIGN c = 0;
      if(a > b)
      {
            if(a <= 60000)
                  c = (UNSIGN) (a + b);
      }
      printf("c = %ld\n",c); //hier wird 44464 angezeigt
      return 0;
}
```

4.10 Implizite Typumwandlung ggf. abschalten

Es gibt verschiedene Arten von impliziter Typumwandlung:

- Konstruktoren mit genau einem Argument
- Zuweisungs-Operatoren = (Objekt ist l-value → bekommt neuen Wert)
- Operatoren für die implizite Typumwandlung
  (Objekt ist r-value → liefert seinen Wert)

Die Compiler benutzen diese Typumwandlungen automatisch, wenn sie passen. Hat man also einen Konstruktor mit genau einem Argument (welches kein Objekt vom Typ der betreffenden Klasse ist) geschrieben, dann kann es sein, dass der Compiler diesen Konstruktor (ungewollt) als Typumwandler benutzt.

Wenn der Compiler auf eine Zuweisung an einen fremden Datentyp stößt, sucht er nach einem Umwandler, welchen er zur Typumwandlung einsetzen kann. Dies tut er in der hier aufgeführten Reihenfolge, bis er einen passenden Umwandler gefunden hat:

1.) Konstruktor mit genau einem Argument

```
class MyClass
{
      public:
            MyClass(int nID) : m_nID(nID) {}
            ~MyClass() {}
      private:
            int m_nID;
};

void f(MyClass Obj)
{
      MyClass LocalObj = Obj;
}
```

```
int main()
{
	f(3);
	return 0;
}
```

Compiler:

f(i) — wird zu → **MyClass Obj(i); f(Obj);**

2.) Zuweisungs-Operator =

```
class MyClass
{
	public:
		MyClass() : m_nID(0) {}
		~MyClass() {}
		MyClass& operator=(const int nID)
		{
			m_nID = nID;
			return *this;
		}
	private:
		int m_nID;
};
int main()
{
	MyClass Obj;
	Obj = 5;
	return 0;
}
```

Compiler:

Obj = 5; — wird zu → **Obj.operator =(5);**

3.) Operator für die implizite Typumwandlung

```
class MyClass
{
	public:
		MyClass(int nID) : m_nID(nID) {}
		~MyClass() {}
		operator int() const { return m_nID; };
	private:
		int m_nID;
};
int main()
{
	MyClass Obj(4);
	int i = Obj;
	return 0;
}
```

Compiler:

`int i = Obj;` — `wird zu` → **`int i = Obj.operator int();`**

Es kann sinnvoll sein, die impliziten Typumwandlungen unwirksam zu machen und statt dessen explizit Umwandlungsfunktionen aufzurufen (z.B. `int AsInt(const MyClass& Obj)`).

Folgende Maßnahmen sind zum Abschalten der impliziten Typumwandlung erforderlich:

1.) Abschalten der impliziten Typumwandlung über Konstruktor mit genau einem Argument

→ Schlüsselwort **`explicit`** davor

2.) Abschalten der impliziten Typumwandlung über Zuweisungs-Operator =

→ Zuweisungs-Operator für Zuweisung von einem anderen Typ an die eigene Klasse hinter private verstecken

3.) Abschalten der impliziten Typumwandlung über Typumwandlungs-Operator

→ keinen Typumwandlungs-Operator definieren

4.11 inline nur bei sehr einfachen nicht-virtuellen Funktionen

4.11.1 Allgemeines

`inline` bietet sich an, wenn es einfach um das Zurückliefern eines Wertes geht.

Beispiel:

```
inline int MyClass::GetNum() { return m_nNum; }
```

Aber in allen anderen Fällen ist `inline` nicht zu empfehlen. Gründe hierfür gibt es genug:

- Bei zu **großen Funktionen** wird der Vorteil des Eliminierens des Funktionsaufrufes durch Einsetzen des Codes (also das, was `inline` macht) zu einem Nachteil, da die Hit-Raten für den **First-Level-Instruction-Cache des Prozessors** sich verschlechtern und Code des öfteren aus dem Second-Level-Cache oder Arbeitsspeicher geladen werden muss, was länger dauert.
- `inline` ist nur eine **Empfehlung an den Compiler**, die er nicht beachten muss. So **werden komplexe Funktionen** (mit Schleifen, Rekursionen, ...) bspw. in **`static`**-Funktionen für das jeweilige *.cpp-Modul, welches die Header-Datei mit der inline-Funktion einbindet, konvertiert. Jedes Modul hat somit eine eigene Kopie der

Funktion, wobei die Kopien zudem noch auf unterschiedlichen Speicherseiten im virtuellen Speicher des Prozesses liegen können → blätter, blätter, blätter.

- Bei **virtuellen Funktionen** wird `inline` vom Compiler ignoriert, da erst zur Laufzeit feststeht, welcher Code aufzurufen ist. **Es kann keine virtuelle `inline`-Funktion geben!**
- Eine **Änderung** einer `inline`-Funktion zieht in der Regel einen zeitaufwendigen Bau nach sich, da alle ***.cpp-Module**, die die Header-Datei mit der `inline`-Funktion nutzen, **neu kompiliert werden müssen**.
- In **Bibliotheken** ist `inline` mit Vorsicht zu genießen, denn es verhindert **binäre Updates** (*.dll → nur Kopieren erforderlich / *.obj → Kopieren und neues Linken erforderlich), da neue Header-Dateien entstehen (*.h → Kopieren, neues Kompilieren und neues Linken erforderlich).
- In `inline`-Funktionen kann man **keine Breakpoints** setzen, es sei denn der Compiler generiert eine echte DEBUG-Version der Software, aber dann ersetzt er alle `inline`-Funktionen durch `static`-Funktionen für das jeweilige Modul:

```
inline void Func()
{
      ...
}

static void (*FuncCall)() = Func;

Func();          //-> inline-Aufruf
FuncCall();      //-> kein inline-Aufruf
```

4.11.2 Widerspruch "virtual und inline": virtual dominiert inline

Wenn eine Methode als **`virtual`** deklariert wird, dann heißt das, dass **erst zur Laufzeit** über vfptr und vftable festgestellt wird, welcher Code auszuführen ist. Folglich kann man nicht zum Compiler sagen, dass er bei allen Funktionsaufrufen ein bestimmtes Stück Code einsetzen soll (**`inline`**). Tut man es doch, dann verzeiht der Compiler diesen Fehler, indem er inline ignoriert. Verwirrend bleibt der Code dann trotzdem.

Beispiel:

```
class MyBase
{
     public:
          virtual ~MyBase();
          virtual void Meth1() { printf("Hello"); } //Widerspruch
          virtual void Meth2();
};

inline MyBase::Meth2() { printf(" World\n"); } //Widerspruch

MyBase::~MyBase() {}
```

4.11.3 Basisklasse: Virtueller Destruktor als leere inline-Funktion

Da der Compiler bei **`virtual`**-Methoden immer die **`inline`**-Instruktion ignoriert, bietet es sich an, einen leeren virtuellen Destruktor einfach `inline` in die Deklaration aufzunehmen. Dies führt wohl kaum zu Verwirrungen, kann aber der Übersichtlichkeit dienen:

```
class MyBase
{
      public:
            virtual ~MyBase() {}  //Hack: nicht wirklich inline
            void Meth1() { printf("Meth1()\n"); }
};
```

4.12 Falsche Benutzung einer Klasse ausschließen

Man sollte die falsche Benutzung einer Klasse durch entsprechende Programmierung unterbinden. So kann man bspw. Zuweisung und Copy-Konstruktor hinter `private` verstecken, damit niemand eine Kopie des Objektes erzeugen kann (zumindest nicht ohne weiteres).

4.12.1 Kopie eines Objektes verbieten

Copy-Konstruktor und Zuweisungs-Operator = hinter `private` verstecken:

```
class MyClass
{
      ...
      private:
            MyClass(const MyClass& nID);
            MyClass& operator=(const MyClass& nID);
      ...
};
```

4.12.2 Konstruktion eines Objektes verbieten

Konstruktor und Destruktor hinter `private` verstecken:

```
class MyClass
{
     public:
          Func();
          ...
     private:
          MyClass();
          MyClass(const MyClass& Obj);
          ~MyClass();
     ...
};
```

Jetzt kann die Implementierung der Klasse lediglich als **statisches** Objekt genutzt werden.

Beispiel:

```
          MyClass()::Func();
```

4.13 Laufzeitschalter immer Compiler-Schaltern vorziehen

Wenn man der Einfachheit halber Compiler-Schalter statt Laufzeitschalter in den Code einbaut, dann nimmt man das Risiko in Kauf, dass der Anwender eines Tages kommt und doch eine andere Variante haben möchte → Das ganze Projekt ist nochmal aus dem Tresor zu holen, man muss sich nochmal einarbeiten, die Compiler-Schalter umdefinieren (dabei überprüfen, welche Kombinationen von Compiler-Schaltern erlaubt sind) und eine neue Version bauen und vor allem die **neue Version erst einmal testen**. Nicht zu unterschätzen ist das In-Umlauf-Bringen der neuen Version mit allen Konsequenzen (Versionsverwaltung, Handbuch, ...) → Man sollte erst gar nicht darüber nachdenken einen Compiler-Schalter zu verwenden, wenn ein Laufzeitschalter möglich ist.

Beispiel:

Statt:

```
#define OPT_SPEED
#include <stdio.h>
int main()
{
     #ifdef OPT_SPEED
          printf("Optimzed for Speed\n");
     #else
          printf("Optimzed for Memory\n");
     #endif
     return 0;
}
```

Besser:

```
#include <stdio.h>
int main(int argc,char* argv[])
{
      if(argc > 1) //more than just the exe's name
      {
            if(!strcmp(argv[1],"OPT_SPEED"))
            {
                  printf("Optimzed for Speed\n");
                  return 0;
            }
      }
      printf("Optimzed for Memory\n");
      return 0;
}
```

4.14 short statt bool als return-Wert bei Interface-Methoden

Bei Interface-Methoden ist eine Abänderung im Nachhinein meist mit sehr viel Aufwand (ggf. von vielen Programmierern) verbunden! Da es im Laufe einer Software-Implementierung leider allzu oft vorkommt, dass man dem `return`-Wert "`false`" oder auch dem `return`-Wert "`true`" noch weitere Attribute geben möchte, wie z.B. "ist zwar ok (`true`), aber am Limit", sollte man bei Interface-Methoden lieber gleich `short` als `return`-Wert einsetzen.

Beispiel:

Statt:

```
bool Analyse(const vector<char>& vectData,string& strDecoded);


      true  →    ok
      false →    not ok


            → if(!Analyse(vectData,strDecoded))
                  ... //error
```

Besser:

```
short Analyse(const vector<char>& vectData,string& strDecoded);

       0    →    ok
      -1    →    not ok, because of wrong format
      -2    →    not ok, because of wrong length
      -3    →    not ok, because of wrong CRC


            →    if(nError = Analyse(vectData,strDecoded))
                        ... //error
```

oder

→

```
short nError = Analyse(vectData,strDecoded);
switch(nError)
{
      case  0: //ok
            break;
      case -1: //wrong format
            ...
            break;
      case -2: //wrong length
            ...
            break;
      case -3: //wrong CRC
            ...
            break;
      default: //unknown error
            ...
            break;
}
```

5. Strings

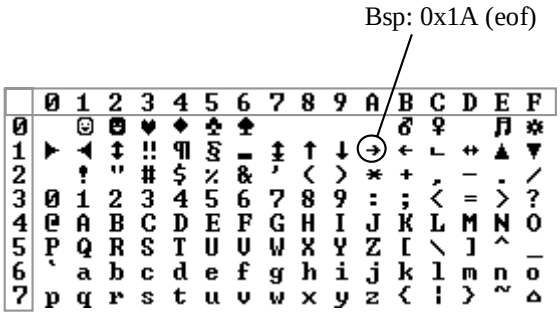
5.1 ASCII-Tabelle

Allgemeines:

Die Zeichen mit dem Code 0...127 sind weltweit eindeutig (7-Bit-ASCII-Code). Im 8-Bit-ASCII-Code (SBCS, Single Byte Character Set) sind die Zeichen oberhalb 127 in sogenannte Codepages für verschiedene Sprachen bzw. Sprachräume aufgeteilt. Ihre korrekte Darstellung ist nur mit Kenntnis der Codepage (Bsp.: DOS-Codepage 850) möglich. Neben 8-Bit-ASCII (SBCS) gibt es noch die MBCS-Codierung (Multiple Byte Character Set), welche teilweise mit 8-Bit- und teilweise mit 16-Bit-Codes arbeitet und den 16-Bit-UNICODE.

7-Bit-ASCII-Code:

Die Zeichen von 0x00 bis 0x1F werden als Sonderzeichen zur Steuerung benutzt. Dabei ist 0x00 das String-Ende ("\0") und 0x1A das Dateiende (eof). Weiterhin steht 0x0A für Zeilenumbruch (line feed, LF bzw. "\n") und 0x0D für Wagenrücklauf (carriage return, CR bzw. "\r"). Tabulator ("\t") ist 0x09 usw.
Als Protokoll für die Übertragung von ASCII-Zeichen wird gerne CR/LF genutzt, d.h. eine Sequenz von ASCII-Zeichen endet mit <0x0D><0x0A> bzw. "\r\n". Weiterhin sehr verbreitet ist das Protokoll STX/ETX, bei dem eine Sequenz von ASCII-Zeichen nach STX (0x02) beginnt und mit ETX (0x03) endet. Obwohl die Zeichen von 0x00 bis 0x1F eigentlich nur zur Steuerung dienen, hat man den meisten sichtbare Symbole zugeordnet.



8-Bit-ASCII-Code:

Ich zeige hier kurz die Standard-8-Bit-ASCII-Codes von DOS und Windows-NT. Standard bedeutet, die Codepage ist English (United States). Unter DOS entspricht dies "DOS-Codepage = 437". Unter Windows-NT ist es die Einstellung "User-Locale = **English (US)**", was folgenden Werten entspricht: wLanguage = 1033 (0x0409), wCodePage = 1200 (0x04B0).

DOS-Standard-8-Bit-ASCII:

DOS-Codepage = 437, English (US)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺	☻	♥	♦	♣	♠					♂	♀		♫	☼
1	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	⌂
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
F	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	

Windows-NT-Standard-8-Bit-ASCII:

wLanguage = 1033 (0x0409), wCodePage = 1200 (0x04B0)
User-Locale = English (US)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	□
8	€	□	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	□	Ž	□
9	□	‘	’	“	”	•	–	—	˜	™	š	›	œ	□	ž	Ÿ
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Anmerkung: **UNICODE** definiert hinter Code 0x0000...0x007F die gleichen Zeichen wie ASCII hinter 0x00...0x7F.

Beispiel: Sprach- und plattformunabhängige Übertragung von ACSII-Text-Codes:

Man stellt die Steuerzeichen (0x00...0x1F) und die Zeichen oberhalb 127 (0x7F...0xFF) dar, indem man sie in jeweils 2 Hex-Nibble konvertiert und als Hex-Zahl (einheitlicher ASCII-Code) anzeigt:

```
char HiNibble(unsigned char byByte)
{
      unsigned char byVal = (byByte & 0xF0) >> 4;
      if(byVal > 9)
            return (byVal – 10 + 'A');
      return (byVal + '0');
}

char LoNibble(unsigned char byByte)
{
      unsigned char byVal = byByte & 0x0F;
      if(byVal > 9)
            return (byVal – 10 + 'A');
      return (byVal + '0');
}

int main()
{
      unsigned char byByte = 0x14;
      printf("Byte: 0x%c%c\n",HiNibble(byByte),LoNibble(byByte));
      return 0;
}
```

Grundsätzlich bietet es sich an, für die Zeichenverarbeitung String-Objekte zu verwenden, wie z.B. `string` der STL.

5.2 string der STL

5.2.1 Allgemeines

Die STL definiert hinter `string` ein Template für den Typ `char`:

```
typedef basic_string<char> string;
```

Hierbei wird das Template `basic_string` benutzt:

```
template<  classT,
           class traits = string_char_traits<T>,
           class Allocator = allocator          >
class basic_string;
```

Durch die `string`-Beispiele in diesem Kapitel wird schon mal auf das große Kapitel STL vorgegriffen.

Hier das erste allgemeine Beispiel:

```
#include <stdio.h>
#include <string>
using namespace std;

int main()
{
	string strText("Hello"); //-> "Hello"

	//Umwandeln in char*:
	size_t len = strText.size() + 1;
	char* szText = new char[len];
	strcpy(szText,strText.c_str());
	delete[] szText;
	szText = NULL;

	//" " anhängen:
	strText.append(" ");

	//strText2 anhängen:
	string strText2("World");
	strText += strText2;

	//Ab Zeichen 6 "--- " einfügen:
	strText.insert(6,"--- ");

	//Ab Zeichen 8 szInsertStr einfügen:
	string szInsertStr("*-");
	strText.insert(8,szInsertStr.begin());

	//5 mal '>' davor:
	strText.insert(strText.begin(),5,'>');

	//Ab Zeichen 5 " #" einfügen:
	strText.insert(5," # ");

	//nNum dezimal ab Zeichen 7 einfügen:
	int nNum = 10101;
	char szNum[256];
	itoa(nNum,szNum,10); //radix = 10
	strText.insert(7,szNum);

	//Am Anfang "#0x :" einfügen:
	strText.insert(0,"#0x: ");

	//lNum hexadezimal ab Zeichen 3 einfügen:
	long lNum = 65535;
	ltoa(lNum,szNum,16); //radix = 16
	strText.insert(3,szNum);

	return 0;
}
```

5.2.2 *String formatieren mit Hilfe von sprintf()*

Die Formatierung von `string` geschieht normalerweise über die Zuhilfenahme eines `char[]`-Arrays und mittels der Funktion `sprintf()`:

```
#include <stdio.h>
#include <string>
using namespace std;
int main()
{
	char szTest[1024] = "";
	const char szHelloWorld[] = "Hello World";
	short nNum = 5;
	double dNum = 75.333;
	char cHex = 0x4A;
	void* pVoid = &cHex;
	sprintf(szTest,
		"s: '%s'\nd: '%03d'\nf: '%3.1f'\nX: '0x%02X'\np: '%p'",
		szHelloWorld,
		nNum,
		dNum,
		cHex,
		pVoid);
	string strTest = szTest;
	printf("%s",strTest.c_str());
	return 0;
}
```

Sollte man eine eigene Funktion, wie

```
	void format_string(string& strOUT,const char* szIN,...)
	{
	}
```

programmieren, dann ist darauf zu achten, dass diese nicht unter Zuhilfenahme von `sprintf()` implementiert wird, da sonst die Laufzeit wesentlich länger als die des oben gezeigten Codes wird.

Bemerkung:

Da es oft zur Verwirrung führt, seien hier nochmal die 2 verschiedenen Token **** und **%** aufgeführt:

**** ist ein Token für den Compiler

→ Ausdrücke werden bei der Kompilierung durch ASCII-Codes ersetzt.

Beispiele:

```
"\\"   →   "\"
"\n"   →   "<0x0A>"
"\r"   →   "<0x0D>"
```

% ist ein Token für `printf()`-Funktionen

→ Ausdrücke überleben die Kompilierung und werden zur Laufzeit ausgewertet.

5.2.3 Teil-Strings ersetzen mit string::replace() und string::find()

Mit `string::replace()` hat man die Möglichkeit, Teil-Strings durch andere Teil-Strings zu ersetzen. Die Methode `string::find()` liefert einem die Vorkommnisse des gesuchten Teil-Strings und `string::replace()` ersetzt sie dann.

Beispiel:

```
#include <stdio.h>
#include <string>
using namespace std;

int main()
{
     char szOld[] = "First Line [LF]Second Line[LF]Third Line[LF]";

     string strNew(szOld);
     int pos = 0;
     do
     {
          pos = strNew.find("[LF]");
          if(pos != string::npos)
               strNew.replace(pos,strlen("[LF]"),"\n");
     }
     while(pos != string::npos);

     printf("Old String:\n\n%s\n\n",szOld);
     printf("New String:\n\n%s\n",strNew);

     return 0;
}
```

5.2.4 Zeichen löschen mit string::erase() und einfügen mit string::insert()

`string::erase()` und `string::insert()` sind komfortable Methoden, um Zeichen aus einem String herauszulöschen oder einzufügen.

Beispiel:

```
          string strTest("Hello");              //"Hello"
          string strNew = strTest.erase(0,1);   //"ello"
          strNew = strTest.erase(0,2);          //"lo"
          strNew = strTest.erase(1,1);          //"l"
          strNew = strTest.insert(0,"Hel");     //"Hell"
          strNew = strTest.insert(4,"o");       //"Hello"
```

5.2.5 Umwandlung in Zahlen mit strtol() und der Methode string::c_str():

Wenn man `string`-Objekte als Argumente für String-Manipulations-Funktionen benutzt, dann muss man die Konvertierungsmethode `c_str()` von string benutzen.

Beispiel:

```
#include <stdio.h>
#include <string>
using namespace std;

int main()
{
      string strHexNumber("0x1FA5");
      string strDecNumber("59");
      string strOctNumber("18");

      //Umwandeln in Zahl:
      char* pWrongChar = NULL;
      long lNum = strtol(strHexNumber.c_str(),&pWrongChar,16);
      if(*pWrongChar != 0x00)
            printf("Error: Wrong character [ %c ]\n",*pWrongChar);
      lNum = strtol(strDecNumber.c_str(),&pWrongChar,10);
      if(*pWrongChar != 0x00)
            printf("Error: Wrong character [ %c ]\n",*pWrongChar);
      lNum = strtol(strOctNumber.c_str(),&pWrongChar,8);
      if(*pWrongChar != 0x00)
            printf("Error: Wrong character [ %c ]\n",*pWrongChar);

      strOctNumber= "17";
      lNum = strtol(strOctNumber.c_str(),&pWrongChar,8);
      if(*pWrongChar != 0x00)
            printf("Error: Wrong character [ %c ]\n",*pWrongChar);

      return 0;
}
```

5.2.6 Teil eines anderen Strings anhängen mit string::append()

Manchmal muss man genau einen bestimmten Teil eines Strings an einen bestehenden String (kann auch ein leerer String sein) anhängen. Hierzu lässt sich `string::append()` hervorragend verwenden.

Beispiel:

```
#include <stdio.h>
#include <string>
using namespace std;
```

```
int main()
{
	string strText1("Nice ");
	string strText2("This day");

	//3 Zeichen von strText2 anhängen, beginnend beim Index 5:
	strText1.append(strText2,5,3);

	printf("%s\n",strText1);
	return 0;
}
```

5.2.7 Konfigurationsdateien parsen mit string::compare() und string::copy()

Um die Anwendung von `string::compare()` und `string::copy()` zu zeigen, eignet sich das Beispiel einer einfachen Konfigurationsdatei, wie sie z.B. unter Linux verwendet wird, um den ersten Ethernet-Adapter zu konfigurieren (`/etc/sysconfig/network-scripts/ifcfg-eth0`). Die Datei '`ifcfg-eth0`' hat etwa folgendes Format:

```
DEVICE=eth0
BOOTPROTO=static
IPADDR=175.44.5.40
BROADCAST=172.26.2.255
NETMASK=255.255.0.0
GATEWAY=175.44.0.1
ONBOOT=yes
```

Es werden also die Namen der Parameter linksbündig mit Gleicheitszeichen in die Datei geschrieben und der Wert wird ohne Lücke (SPACE) darangehängt. Die Datei lässt sich nun einfach parsen:

```
#include <stdio.h>
#include <string>
using namespace std;
bool GetLine(FILE* pFile,string& strLine)
{
	static bool bEOFReached = false;
	if(bEOFReached)
		return false;
	strLine = "";
	char szChar[2] = "?";
	while((szChar[0] = (char) fgetc(pFile)) != EOF)
	{
		if(szChar[0] == 0x0D) //CR ("\r")
			continue;
		if(szChar[0] == 0x0A) //LF ("\n")
			break;
		strLine += szChar;
	}
	if(szChar[0] == EOF)
		bEOFReached = true;
	return true;
}
```

```
short ParseLine(const string& strLine,
                pair<string,string>& pairParamValue)
{
     static const string strDevice("DEVICE=");
     static const string strBootProto("BOOTPROTO=");
     static const string strIPAddr("IPADDR=");
     static const string strBroadcast("BROADCAST=");
     static const string strNetmask("NETMASK=");
     static const string strGateway("GATEWAY=");
     static const string strOnBoot("ONBOOT=");

     string strParam("?");
     char szValue[1024] = "";

     if(!strLine.compare(0,strDevice.length(),strDevice))
     {
          strLine.copy(   szValue,
                          strLine.length()-strDevice.length(),
                          strDevice.length());
          szValue[strLine.length()-strDevice.length()] = 0x00;
          strParam = strDevice;
     }
     else if(!strLine.compare(0,strBootProto.length(),strBootProto))
     {
          strLine.copy(   szValue,
                          strLine.length()-strBootProto.length(),
                          strBootProto.length());
          szValue[strLine.length()-strBootProto.length()] = 0x00;
          strParam = strBootProto;
     }
     else if(!strLine.compare(0,strIPAddr.length(),strIPAddr))
     {
          strLine.copy(   szValue,
                          strLine.length()-strIPAddr.length(),
                          strIPAddr.length());
          szValue[strLine.length()-strIPAddr.length()] = 0x00;
          strParam = strIPAddr;
     }
     else if(!strLine.compare(0,strBroadcast.length(),strBroadcast))
     {
          strLine.copy(   szValue,
                          strLine.length()-strBroadcast.length(),
                          strBroadcast.length());
          szValue[strLine.length()-strBroadcast.length()] = 0x00;
          strParam = strBroadcast;
     }
     else if(!strLine.compare(0,strNetmask.length(),strNetmask))
     {
          strLine.copy(   szValue,
                          strLine.length()-strNetmask.length(),
                          strNetmask.length());
          szValue[strLine.length()-strNetmask.length()] = 0x00;
          strParam = strNetmask;
     }
```

```
	else if(!strLine.compare(0,strGateway.length(),strGateway))
	{
		strLine.copy(	szValue,
				strLine.length()-strGateway.length(),
				strGateway.length());
		szValue[strLine.length()-strGateway.length()] = 0x00;
		strParam = strGateway;
	}
	else if(!strLine.compare(0,strOnBoot.length(),strOnBoot))
	{
		strLine.copy(	szValue,
				strLine.length()-strOnBoot.length(),
				strOnBoot.length());
		szValue[strLine.length()-strOnBoot.length()] = 0x00;
		strParam = strOnBoot;
	}
	else
	{
		pairParamValue.first = "";
		pairParamValue.second = "";
		return -1;
	}

	pairParamValue.first = strParam;
	pairParamValue.second = string(szValue);

	return 0;
}
int main()
{
	static const char szFileName[] = "ifcfg-eth0";
	FILE* pFile = fopen(szFileName,"r");
	if(!pFile)
		return -1;

	string strLine;
	pair<string,string> pairParamValue;

	short nError = 0;

	printf("Content of '%s'...\r\n",szFileName);
	while(GetLine(pFile,strLine))
	{
		nError = ParseLine(strLine,pairParamValue);
		if(!nError)
			printf(	"%s%s\r\n",
				pairParamValue.first.c_str(),
				pairParamValue.second.c_str());
		else
			printf("ERROR\r\n");
	}

	fclose(pFile);
	pFile = NULL;

	return 0;
}
```

5.2.8 Worte sortieren mit set<string>

Der einfachste Weg eine Ansammlung von Worten Leerzeichen oder zu sortieren ist, sie in einen STL-Container vom Typ `set<string>` zu schreiben:

Beispiel:

```
#include <stdio.h>
#include <set>
#include <string>
using namespace std;

int main()
{
      string strName1("Peter");
      string strName2("Claudia");
      string strName3("Anna-Lena");
      string strName4("Markus");

      set<string> setSortedNames;
      setSortedNames.insert(strName1);
      setSortedNames.insert(strName2);
      setSortedNames.insert(strName3);
      setSortedNames.insert(strName4);

      printf("Namen - aufsteigend sortiert:\r\n");
      set<string>::iterator it;
      for(it = setSortedNames.begin();
          it != setSortedNames.end();
          ++it)
      {
            printf("%s\r\n",(*it).c_str());
      }

      printf("\r\nNamen - absteigend sortiert:\r\n");
      set<string>::reverse_iterator rit;
      for(  rit = setSortedNames.rbegin();
            rit != setSortedNames.rend();
            ++rit)
      {
            printf("%s\r\n",(*rit).c_str());
      }
      return 0;
}
```

Als **Wort** bezeichnet man hier eine zusammenhängende Zeichenkette, die keines der folgenden Zeichen enthält:

```
" "    (SPACE, 0x20),
"\t"   (TAB, 0x09)
"\r"   (RETURN, CR, 0x0D)
"\n"   (NEWLINE, LF, 0x0A)
```

5.2.9 *Strings zuschneiden mit string::replace() und string::resize()*

Um `string` von rechts zuzuschneiden gibt es `resize()`, für das Zuschneiden von links geht man den Weg über die Benutzung von `replace()`:

```
#include <stdio.h>

#include <string>
using namespace std;

int main()
{
      string strTest("0123456789");
      printf("strTest .....................: [%s] length = %d\r\n",
             strTest.c_str(),
             strTest.length());

      //Links zuschneiden:
      strTest.replace(0,4,"");
      printf(
            "After 'replace(0,4,\"\")' ......: [%s] length = %d\r\n",
            strTest.c_str(),
            strTest.length());

      //Rechts zuschneiden:
      strTest.resize(3,0x00);
      printf("After 'resize(3)' ............: [%s] length = %d\r\n",
             strTest.c_str(),
             strTest.length());

      return 0;
}
```

5.3 string streams der STL

5.3.1 *Allgemeines*

Die wohl bekanntesten streams sind die iostreams `cin` und `cout`. Programmiert man jedoch graphische Benutzeroberflächen, so verlieren diese streams etwas an Bedeutung. Die string streams können jedoch trotzdem wertvolle Gehilfen sein. So ist z.B.

```
istringstream
```

sehr hilfsreich beim Parsen eines Texts nach den enthaltenen Worten. Und

```
ostringstream
```

ist eine große Hilfe bei der automatischen Lokalisierung mit `locale`-Objekten, die die Darstellung von Zeit, Währung und anderen Zahlen beeinflussen.

5.3.2 Text mit istringstream nach enthaltenen Worten parsen

Ein Text soll nach den darin enthaltenen Worten geparst werden. Eine der einfachsten Implementierungen dazu kann man mit `istringstream` vornehmen:

```
#include <sstream> //string streams
#include <list>
#include <string>
using namespace std;

int main()
{
      string strText("Hello world");
      istringstream issText(strText);
      string strWord;
      list<string> listWords;
      while(issText >> strWord)
            listWords.push_back(strWord);
      printf("The string '%s' contains the words...\r\n",strText);
      list<string>::iterator it;
      for(it = listWords.begin();it != listWords.end();++it)
            printf("   %s\r\n",*it);
      return 0;
}
```

6. Zeitermittlung

6.1 Weltweit eindeutiger Timestamp (GMT), Jahr-2038

Wenn man Daten über den gesamten Globus austauscht, dann ist es manchmal wichtig eine weltweit eindeutige Zeit zu verwenden. Zum einen, um Abläufe zu synchronisieren, und zum anderen, um Zeitangaben unabhängig vom Ort zu ermitteln bzw. auszuwerten. Die GMT (**Greenwich Mean Time**) ist eine solche Zeit, die mit Hilfe der System-Uhr ermittelt wird. Um eine genaue Zeitangabe zu erhalten ist es wichtig, eine Synchronisierung der eigenen System-Uhr mit einer Normal-Uhrzeit vorzunehmen. Dazu kann man das Network Time Protocol (NTP) verwenden, d.h. man startet einen NTP-Client als Hintergrund-Prozess, der bei einem NTP-Server (Daemon) regelmäßig die Zeit erfragt und die eigene System-Uhr nachstellt.

Beispiel:

```
#include <stdio.h>
#include <time.h>

int main()
{
      long lTime = time(NULL);
      tm tmGMT = *gmtime(&lTime);

      char szTimeStamp[128] = "";
      sprintf(   szTimeStamp,
                 "%04d-%02d-%02d %02d:%02d:%02d",
                 (tmGMT.tm_year + 1900),
                 tmGMT.tm_mon + 1,
                 tmGMT.tm_mday,
                 tmGMT.tm_hour,
                 tmGMT.tm_min,
                 tmGMT.tm_sec);

      printf("Timestamp: %s\r\n",szTimeStamp);

      return 0;
}
```

Problem:

`time(NULL)` liefert die Zeit in **Sekunden, gezählt vom 1. Januar 1970, 00:00 Uhr**. Leider wird der Wert zudem noch als **`long`** (`-2147483647...+2147483648`) zurückgeliefert, was bedeutet, dass es nach 68 Jahren (`+2147483648 / (365.25 * 86400)`), also **im Jahre 2038** zu einem Überlauf kommt.

Dies ist, nebenbei bemerkt, auch das Jahr in dem die 16-Bit-Zähler (`unsigned short`) des Modifizierten Julianischen Datums (**MJD**; **Tage, gezählt vom 17. November 1858**) überlaufen, was ich in der Tat als seltsamen Zufall empfinde. 16-Bit-MJD wird z.B. in den MPEG-2-Streams von DVB (Digital Video Broadcast = Digitales Fernsehen) zur Spei-

cherung der Startzeiten von Events (Fernsehsendungen) benutzt, und muss über einen Plausibiltäts-Check vor dem Überlauf im Jahre 2038 geschützt werden

```
dwMJD = wMJD; //in eine 32-Bit-Variable kopieren
if(dwMJD < 51604) //vor dem Jahr 2000 -> Überlauf der 16 Bit
      dwMJD += 0x10000;
```

was die Verwendbarkeit um 142 Jahre, also bis zum Jahr 2180 verlängert.

→ `time(NULL)`:

Man sollte auf andere Zeitermittlungsfunktionen zurückgreifen, die in der Regel vom SDK des Betriebssystems angeboten werden. Dabei ist immer darauf zu achten, das nicht intern der Aufruf `time(NULL)` gekapselt wird.

→ MJD:

Statt 16 Bit (`unsigned short`) sollte man 32 Bit (`unsigned long`) verwenden.

6.2 Automatische Lokalisierung der Zeitdarstellung (strftime)

Die automatische Lokalisierung der Zeitdarstellung geschieht, wie die Zeitermittlung selbst, über das System. Unschön hierbei ist, dass die Auswahl der sogenannten "locale" mit einem String erfolgt, der **plattformabhängig** ist. Man kann aber anhand von `defines` abfragen, um welche Plattform es sich handelt, und die jeweils richtigen Strings benutzen. Beispiel:

```
#include <stdio.h>
#include <time.h>
#include <locale.h>

//-------------------------------------------------------------------
// C-Locales fuer verschiedene Plattformen:
//-------------------------------------------------------------------
// Codeset ISO8859-1   --- Euro -->   Codeset ISO8859-15
//-------------------------------------------------------------------

#if defined(_WIN32) || defined(_WIN64) //-> <Language>_<Country>.<Codepage>
      static const char LOCALE_ENGLISH_US[]     = "English_USA.1252";
      static const char LOCALE_ENGLISH_UK[]     = "English_UK.1252";
      static const char LOCALE_GERMAN_DE[]      = "German_Germany.1252";
      static const char LOCALE_FRENCH_FR[]      = "French_France.1252";
      static const char LOCALE_ITALIAN_IT[]     = "Italian_Italy.1252";
      static const char LOCALE_SPANISH_ES[]     = "Spanish_Spain.1252";
#elif defined(__osf__) //-> <languagecode>_<COUNTRYCODE>.<ISOCodesetNo>
      static const char LOCALE_ENGLISH_US[]     = "en_US.88591";
      static const char LOCALE_ENGLISH_UK[]     = "en_UK.88591";
      static const char LOCALE_GERMAN_DE[]      = "de_DE.885915";
      static const char LOCALE_FRENCH_FR[]      = "fr_FR.885915";
      static const char LOCALE_ITALIAN_IT[]     = "it_IT.885915";
      static const char LOCALE_SPANISH_ES[]     = "es_ES.885915";
#elif defined(SNI) //-> <Languagecode>_<COUNTRYCODE>.<ISOCodesetNo>
      static const char LOCALE_ENGLISH_US[]     = "En_US.88591";
      static const char LOCALE_ENGLISH_UK[]     = "En_UK.88591";
```

```
      static const char LOCALE_GERMAN_DE[]      = "De_DE.885915";
      static const char LOCALE_FRENCH_FR[]      = "Fr_FR.885915";
      static const char LOCALE_ITALIAN_IT[]     = "It_IT.885915";
      static const char LOCALE_SPANISH_ES[]     = "Es_ES.885915";
#elif defined(_AIX) //-> <languagecode>_<COUNTRYCODE>.ISO<ISOCodesetNo>
      static const char LOCALE_ENGLISH_US[]     = "en_US.ISO8859-1";
      static const char LOCALE_ENGLISH_UK[]     = "en_UK.ISO8859-1";
      static const char LOCALE_GERMAN_DE[]      = "de_DE.ISO8859-15";
      static const char LOCALE_FRENCH_FR[]      = "fr_FR.ISO8859-15";
      static const char LOCALE_ITALIAN_IT[]     = "it_IT.ISO8859-15";
      static const char LOCALE_SPANISH_ES[]     = "es_ES.ISO8859-15";
#elif defined(__linux__) //-> <languagecode>_<COUNTRYCODE>
      static const char LOCALE_ENGLISH_US[]     = "en_US";
      static const char LOCALE_ENGLISH_UK[]     = "en_UK";
      static const char LOCALE_GERMAN_DE[]      = "de_DE@euro";
      static const char LOCALE_FRENCH_FR[]      = "fr_FR@euro";
      static const char LOCALE_ITALIAN_IT[]     = "it_IT@euro";
      static const char LOCALE_SPANISH_ES[]     = "es_ES@euro";
#elif defined(__hpux) //-> <languagecode>_<COUNTRYCODE>.iso<ISOCodesetNo>
      static const char LOCALE_ENGLISH_US[]     = "en_US.iso88591";
      static const char LOCALE_ENGLISH_UK[]     = "en_UK.iso88591";
      static const char LOCALE_GERMAN_DE[]      = "de_DE.iso885915";
      static const char LOCALE_FRENCH_FR[]      = "fr_FR.iso885915";
      static const char LOCALE_ITALIAN_IT[]     = "it_IT.iso885915";
      static const char LOCALE_SPANISH_ES[]     = "es_ES.iso885915";
#else //-> <languagecode>
      static const char LOCALE_ENGLISH_US[]     = "en";
      static const char LOCALE_ENGLISH_UK[]     = "en";
      static const char LOCALE_GERMAN_DE[]      = "de";
      static const char LOCALE_FRENCH_FR[]      = "fr";
      static const char LOCALE_ITALIAN_IT[]     = "it";
      static const char LOCALE_SPANISH_ES[]     = "es";
#endif

int main()
{
      long lTime = time(NULL); //Überlauf im Jahr 2038

      char szTime[128] = "";

      setlocale(LC_ALL,LOCALE_ENGLISH_US);
      strftime(szTime,128,"%x",localtime(&lTime));
      printf("Time: %s\r\n",szTime);

      setlocale(LC_ALL,LOCALE_GERMAN_DE);
      strftime(szTime,128,"%x",localtime(&lTime));
      printf("Zeit: %s\r\n",szTime);

      return 0;
}
```

Vorteile:

- Die Zeitdarstellung für verschiedene Sprachen/Länder passt sich automatisch dem System an
- Man muss die Implementierung für die Lokalisierung nicht selbst pflegen

Nachteile:

- Keine einheitliche Lösung für den String-Parameter von `setlocale()`
- Die Zeitdarstellung hängt vom System ab und kann je nach System unterschiedlich aussehen

Wenn man sich den Aufwand betrachtet, der getrieben werden muss, um den richtigen String-Parameter für `setlocale()` zu finden, dann ist es wohl ein berechtigter Aufwand, eine eigene Routine zu schreiben, die die Zeitdarstellung korrekt formatiert.

Beispiel:

```
#include <stdio.h>
#include <time.h>

#include <string>
using namespace std;

void print_time(const tm& tmTime,const string& strLocale)
{
	if(!strLocale.compare("en_US"))
	{
		printf(	"Time: %04d-%02d-%02d %02d:%02d:%02d\r\n",
			(tmTime.tm_year + 1900),
			tmTime.tm_mon + 1,
			tmTime.tm_mday,
			tmTime.tm_hour,
			tmTime.tm_min,
			tmTime.tm_sec);
	}
	else if(!strLocale.compare("de_DE"))
	{
		printf(	"Zeit: %02d.%02d.%04d %02d:%02d:%02d\r\n",
			tmTime.tm_mday,
			tmTime.tm_mon + 1,
			(tmTime.tm_year + 1900),
			tmTime.tm_hour,
			tmTime.tm_min,
			tmTime.tm_sec);
	}
}

int main()
{
	long lTime = time(NULL); //Überlauf im Jahr 2038
	tm tmGMT = *gmtime(&lTime);

	print_time(tmGMT,"en_US");
	print_time(tmGMT,"de_DE");

	return 0;
}
```

7. Konstantes

7.1 const-Zeiger (C-Funktionen)

C kennt im Gegensatz zu C++ keine Referenzparameter und benutzt Zeiger als Argumente, wenn eine Funktion einen Parameter manipulieren soll. Um innerhalb einer C-Funktion einen Parameter manipulieren zu können, muss der Aufrufer die zu übergebende Variable mit dem Operator & **referenzieren**, d.h. der Operator & ermittelt die Speicheradresse und liefert sie zurück, weist sie also somit dem Zeigerparameter zu. Innerhalb der Funktion kann der Wert nun mit Hilfe des Operators * gelesen oder beschrieben werden, d.h. er wird über den Operator * **dereferenziert**:

- **Zeiger auf Speicherstelle (Lesen/Beschreiben einer Variablen):**

  Funktion:
  ```
  void f(int* pParam)
  {
        *pParam++; //Inhalt verändern nach derefenzieren
  }
  ```
  Aufruf:
  ```
  int i;
  f(&i); //referenzieren der Variablen
  ```

- **Zeiger auf Zeiger (Lesen/Beschreiben eines Zeigers):**

  Funktion:
  ```
  void f(CDC** ppDC)
  {
        **ppDC = GetDC(); //Inh.verändern nach derefenzieren
  }
  ```
  Aufruf:
  ```
  CDC* pDC;
  f(&pDC); //referenzieren des Zeigers
  ```

Man kann nun aber auch durch **const** verhindern, dass verschiedene Sachen manipuliert werden:

```
void f(const char* p){...}      //konstante Daten

void f(char* const p){...}      //konstanter Zeiger

void f(const char* const p){...}//konstante Daten & konstanter Zeiger
```

Zu allem Überfluss ist auch noch eine andere Schreibweise erlaubt:

```
const char* p   <=== identisch ===>   char const* p
```

Regel, die immer gilt: **Was links von * steht, hält die Daten konstant!**

7.2 const-Referenzen (C++-Funktionen)

7.2.1 Allgemeines

In C++ werden **Referenzparameter** als Argument benutzt, wenn eine Funktion die Inhalte der Argumente verändern darf:

```
void f(MyClass& Obj)
{
      Obj.SetID(8);
}
```

Wenn die Funktion die Parameter jedoch nicht verändern darf, dann sollte man die Parameter per **`const`-Referenz** übergeben und nicht per Wert (was die Alternative wäre). Der große Vorteil liegt darin, dass beim Aufruf der Funktion nicht extra eine Kopie in eine für die Funktion lokale Speicherstelle erfolgen muss.

`const`-Referenz:

```
void func(const MyClass& Obj)
{
      if(Obj.GetID()==3)
      {
            ...
      }
}
```

Es sollte also im Idealfall in einem C++-Programm nur 2 Arten von Argumenten geben:

- Referenzen
- `const`-Referenzen

7.2.2 STL-Container als const-Referenzen verlangen const_iterator

Wenn man innerhalb einer Methode einen STL-Container (`list`, `set`, ...) per `const`-Referenz anspricht, dann kann man darüber nur iterieren, wenn man einen `const_iterator` benutzt.

Beispiel:

```
#include <stdio.h>
#include <list>
using namespace std;
```

```
class MyClass
{
      public:
            MyClass() {}
            unsigned long Count(
                              const list<unsigned short>& listValues)
            {
                  unsigned long dwLen = 0L;
                  list<unsigned short>::iterator it; //nicht ok
                  for(  it = listValues.begin();
                        it != listValues.end();
                        ++it)
                  {
                        ++dwLen;
                  }
                  return dwLen;
            }
};

int main()
{
      list<unsigned short> listValues;
      listValues.push_back(1);
      listValues.push_back(2);
      listValues.push_back(3);
      MyClass Obj;
      Obj.Count(listValues);
      return 0;
}
```

Der Compiler bringt am Zuweisungsoperator `=` einen Fehler: "**`no acceptable conversion`**". Man muss hier `const_iterator` statt `iterator` einsetzen. Richtig wäre also:

```
      list<unsigned short>::const_iterator it; //ok
```

7.3 Read-Only-Member-Funktionen

7.3.1 Allgemeines

Eine weitere Variante der Anwendung von `const` ist das **Schützen der Member-Variablen einer Klasse**. Wenn eine Methode wie folgt definiert wird, dann kann sie die Member-Variablen (mit Ausnahme der **mutable-Member**) eines Objektes der Klasse nicht verändern (**Read-Only**):

```
class MyClass
{
      public:
            void func(int i) const;
      ...
};

void MyClass::func(int i) const
{
      ...
}
```

7.3.2 mutable-Member als interne Merker (Cache-Index) verwenden

Member-Variablen, die mit **`mutable`** deklariert sind, können jedoch auch von Read-Only-Member-Funktionen geändert werden, und zwar ohne dass mit `const_cast` die Konstantheit weggecastet werden muss.

Beispiel:

```
#include <stdio.h>
#include <list>
using namespace std;

class MyIDStore
{
      public:
            MyIDStore() : m_nCacheIndex(-1) {}
            ~MyIDStore() {}
            bool ReadIDs(list<int>& listIDs,int nIndex = -1) const;
      private:
            mutable int m_nCacheIndex;
};
```

```
bool MyIDStore::ReadIDs(list<int>& listIDs,int nIndex) const
{
	if(nIndex == -1)
		return false;

	static list<int> listIDsCache;

	if(m_nCacheIndex != -1)
	{
		if(m_nCacheIndex == nIndex)
		{
			listIDs = listIDsCache;
			return true;
		}
	}
	m_nCacheIndex = nIndex;
	listIDsCache.clear();
	switch(nIndex)
	{
		case 0:
			listIDsCache.push_back(0);
			listIDsCache.push_back(1);
			listIDsCache.push_back(2);
			break;
		case 1:
			listIDsCache.push_back(3);
			listIDsCache.push_back(4);
			listIDsCache.push_back(5);
			break;
		default:
			listIDsCache.push_back(-1);
	}
	listIDs = listIDsCache;
	return true;
}

int main()
{
	MyIDStore IDStore;
	list<int> listCurrentIDs;

	IDStore.ReadIDs(listCurrentIDs,1);//-> Cache wird für 1 gefüllt
	IDStore.ReadIDs(listCurrentIDs,1);//-> Cache wird gelesen
	return 0;
}
```

7.3.3 Zeiger bei Read-Only-Member-Funktion besonders beachten

Es gibt eine versteckte Tatsache, die bei Read-Only-Member-Funktionen beachtet werden muss: Wenn sich unter den Member-Variablen Zeiger befinden, dann werden zwar diese Zeiger von Read-Only-Member-Funktionen nicht berührt (Ausnahme: mutable), jedoch kann der **Inhalt der entsprechenden Speicherstellen sehr wohl verändert** werden.

Beispiel:

```
#include <stdio.h>

class MyString
{
	public:
		MyString(const char* const szStr)
		{
			strcpy(m_szStr,szStr);
			pStr = &m_szStr[0];
		}
		void NewFirstChar(const char& c) const;
	private:
		char  m_szStr[256];
		char* pStr;
};

void MyString::NewFirstChar(const char& c) const
{
	*pStr = c;	//-> erlaubter Schreibzugriff trotz Read-Only
//	m_szStr[0] = c; //-> kompiliert nicht
}
int main()
{
	MyString Str("Hello");
	Str.NewFirstChar('F'); //-> funktioniert -> "Fello"
}
```

7.4 const-return-Wert

Es kann sinnvoll sein, den **Rückgabewert einer Funktion** mit `const` zu definieren:

```
const Multiply& Multiply::operator*(const Multiply& rhs)
{
	m_dValue *= rhs.m_dValue;
	return *this;
}
```

In diesem Fall vermeidet man die Manipulation des Rückgabewertes, bevor dieser in einer nicht-`const`-Variablen gespeichert wurde.

Beispiel:

```
class Multiply
{
	public:
		Multiply(double dValue = 0.0) : m_dValue(dValue) {}
		~Multiply() {}
		const Multiply& operator*(const Multiply& rhs);
	private:
		double m_dValue;
};
```

```
const Multiply& Multiply::operator*(const Multiply& rhs)
{
     m_dValue *= rhs.m_dValue;
     return *this;
}

int main()
{
     Multiply a(4.0);
     Multiply b(5.0);

     Multiply c(0.0);
     c = a * b;            //-> ok

     Multiply d(0.0);
     d = (a * b) = c;      //-> Compiler meldet Fehler
     return 0;
}
```

7.5 const statt #define verwenden

7.5.1 Globale Konstanten

Ein mit `#define` definierter Bezeichner wird vom Präprozessor ersetzt und taucht in keiner **Symboltabelle zum Debuggen** auf. Außerdem spuckt der Compiler keine **Fehlermeldung** aus, die sich auf den Bezeichner beziehen würde.

Man sollte Konstanten immer mit **`const`** und nicht mit `#define` definieren!

```
Statt:

     #define ASPECT_RATIO 1.653

Besser:

     const double dASPECT_RATIO = 1.653;
```

Achtung bei **Zeigern**! → Immer auch den **Zeiger** mit **`const`** versehen!

```
Statt:

     #define NAME "Pit"

Besser:

     const char* const szNAME = "Pit";

     //oder (synonym)

     const char szNAME[] = "Pit";
```

Beispiel:

```
#include <stdio.h>

const char szTEXT1[] = "Text 1";
const char* const szTEXT2 = "Text 2";
const char* szTEXT3 = "Text 3"; //pointer ist nicht const

int main()
{
      printf("%s\n%s\n%s\n",szTEXT1,szTEXT2,szTEXT3);
      ++szTEXT3; //nicht möglich mit szTEXT1 oder szTEXT2
      printf("%s\n%s\n%s\n",szTEXT1,szTEXT2,szTEXT3);
      return 0;
}
```

7.5.2 Lokale Konstanten einer Klasse

Wen man **lokale Konstanten** für den Sichtbereich innerhalb einer **Klasse** definieren will, dann sollten diese mit **`static`** deklariert werden, da nur eine Kopie für alle Objekte benötigt wird.

```
class MyClass
{
      private:
            static const int m_nVALUE;
            static const int m_dVALUE;
            ...
};

const int MyClass::m_nVALUE = 5;
const int MyClass::m_dVALUE = 5.0;
```

Wenn man bereits in der Deklaration den Wert einer Konstanten benötigt, dann gibt es die Möglichkeit, dies über den **`enum`-Hack** zu tun:

```
class MyClass
{
      ...
      private:
            enum { m_nTURNS = 5 };
            ...
};
```

7.6 const-inline-Template statt MAKRO (#define) verwenden

Nie Funktionen in Form von MAKROs implementieren:

Beispiel:

```
#define maximum(a,b) ((a) > (b) ? (a) : (b))

int x = 5;
int y = 2;
int z = maximum(++x,y); //-> danach: x = 7!!!, y = 2
```

Die letzte Zeile wird vom Präprozessor umgesetzt in

```
int z = ((++x) > (y) ? (++x) : (y));
```

→ x wird **zweimal** inkrementiert, da es größer als y ist, womit man einen **echten Fehler** hat.

Besser:
Man schreibt eine **konstante `inline`-Funktion**. Damit sie **typunabhängig** wird (wie das MAKRO), implementiert man sie als **Template**:

```
Statt:

#define maximum(a,b) ((a) > (b) ? (a) : (b))

Immer:

template<class T>
inline const T& maximum(const T& a,const T& b)
{
     return ((a) > (b) ? (a) : (b));
};
```

Natürlich sollte man auch mal einen Blick in die **STL** werfen, da dort schon Funktionen wie `max()` implementiert sind.

Es gibt aber auch Dinge, für die MAKROs unentbehrlich sind:

- **Debug-Funktionen verstecken (`#define _DEBUG`)**
  → nur im DEBUG-Bau vorhanden

  Beispiel:

  ```
  #ifdef _DEBUG
     #define TRACE(szText) OutputDebugString(szText);
  #else
     #define TRACE(szText)
  #endif
  ```

```
void OutputDebugString(const char* szStr)
{
   ...; //Ausgabe in ein DEBUG-Fenster des Debuggers
}
int main()
{
   TRACE("Test\n");
   return 0;
}
```

- **Variablennamen zusammenbauen**

```
#define GET_WORD(w) \
   (unsigned short) \
   ( (((unsigned short) ##Value##_HI) << 8) + \
                        ##Value##_LO)
int main()
{
   unsigned char Value_HI = 0xAA;
   unsigned char Value_LO = 0x55;
   unsigned short wValue = GET_WORD(Value);
   return 0;
}
```

8. Globales (static-Member)

8.1 static-Member

8.1.1 Allgemeines

`static`-Member-Variablen verhalten sich komplett anders als andere:

Deklaration:

```
class MyClass
{
	...
	private:
		static double m_dValue;
		static const double m_dMAXVAL;
	...
};
```

→ **keine Speicherallokierung** (wie bei normalen Variablen)

Implementierung:

```
double MyClass::m_dValue; //wird mit 0 initialisiert
const double MyClass::m_dMAXVAL = 3.5;
```

→ **Allokierung von Speicher**
→ **Initialisierung mit 0 bzw. NULL**

Die Variable wird nur einmal für alle Objekte angelegt! Man kann also von allen Objekten einer Klasse auf eine zentrale Information zugreifen (Bsp.: Objekt-Zähler), die global für alle Objekte der Klasse existiert.

8.1.2 Zugriff, ohne ein Objekt zu instanziieren

Auf `static`-Variablen kann **immer**, auch ohne dass ein Objekt der Klasse existiert, zugegriffen werden:

```
if(MyClass::m_dValue == 0.2)
{
	...
}
```

Grund:
Der Compiler fügt unmittelbar hinter "`main(){`" Code für die Allokierung von Speicher und die Initialisierung von statischen Variablen ein (**statische Initialisierung**). Unmittelbar vor Verlassen der `main()`-Funktion fügt er Code für die **statische Destruktion** dieser Variablen ein.

8.2 Vorsicht bei static-Variablen in nicht-statischen Methoden

`static`-Variablen, die **in Methoden definiert** werden, sind nur lokal sichbar, überleben aber die gesamte Programmlaufzeit, da sie sich im statischen Speicher befinden. Man sollte diese lokalen `static`-Variablen immer mit größter Vorsicht einsetzen.

Beispiel:

```
void MyNonStaticClass::Update()
{
      static bool bFirstCall = true;
      if(bFirstCall)
      {
            ... //einmalige Initialisierung
            bFirstCall = false;
      }
      ... //
}
```

Wenn man die Methode beim ersten instanziierten Objekt aufruft, ist noch alles klar. Instanziiert man ein weiteres Objekt, dann wird für dieses die einmalige Initialisierung für den ersten Aufruf nicht mehr durchgeführt. Das kann fatale Folgen haben, vor allem, wenn es sich hier um eine Klasse zur Darstellung von Sub-Fenstern einer graphischen Anwendung handelt.

Das Beispiel zeigt, dass die lokale `static`-Variable ein wunderbares Instrument ist, um eine Erstinitialisierung zu steuern. Allerdings kann dies zu argen Problemen bei nicht-statischen Methoden führen. Bei statischen Methoden stellt die Vorgehensweise kein Problem dar.

Ein anderer Anwendungsfall, bei dem dies zu beachten ist, sind lokale `static`-Variablen, die als **lokaler Cache** dienen. Dies ist ebenfalls ein schönes Werkzeug bei statischen Methoden; bei nicht-statischen Methoden ist jedoch darauf zu achten, dass der Cache **von allen Objekten gemeinsam genutzt** wird.

Beispiel:

```
string MyClass::GetItem(long lTypeNum)
{
      static hash_map<long,string> hmapSharedTable;
      static bool bSharedFlagNotInitialized = true;
      if(bSharedFlagNotInitialized)
      {
            for(long l = 0;l < 100;++l)
                  hmapSharedTable[l] = ...;
            bSharedFlagNotInitialized = false;
      }
      hash_map<long,string>::iterator it
            = hmapSharedTable.find(lTypeNum);
      if(it != hmapSharedTable.end())
            return (*it).second;
      return string("");
}
```

8.3 static-Variable in static-Methode statt globaler Variable

Wenn einem Lesezugriff auf eine globale Variable erst eine sinnvolle Initialisierung zur Laufzeit vorauszugehen hat, ist die richtige Reihenfolge der Nutzung der Variablen zwingend erforderlich. Man erreicht dies durch Kapselung in einer `static`-Methode. Dabei implementiert man die globale Variable als `static`-Variable.

Beachte: `static`-Variablen werden genau dann zum ersten Mal initialisiert, wenn der Programmlauf zum ersten Mal an der Definition der Variablen vorbeiläuft.

Beispiel zu diesem Problem:

```
#include <stdio.h>

FILE* fTextFile;

void OpenFile()
{
      fTextFile = fopen("HelloWorld.txt","w");
}

void WriteChar(char c)
{
      fputc(c,fTextFile);
}

void CloseFile()
{
      fclose(fTextFile);
}

int main()
{
      OpenFile();

      WriteChar('H');
      WriteChar('e');
      WriteChar('l');
      WriteChar('l');
      WriteChar('o');
      WriteChar(' ');
      WriteChar('W');
      WriteChar('o');

      CloseFile();          //verursacht ein Problem

      WriteChar('r');
      WriteChar('l');
      WriteChar('d');

      return 0;
}
```

Abhilfe mittels `static`-Methode:

```
#include <stdio.h>

class MyFile
{
	public:
		~MyFile()
		{
			if(theTextFile())
				fclose(theTextFile());
		}
		void WriteChar(char c)
		{
			fputc(c,theTextFile());
		}
	private:
		static FILE* theTextFile()
		{
			static FILE* fTextFile = NULL;
			if(!fTextFile)
				fTextFile = fopen("HelloWorld.txt","w");
			return fTextFile;
		}
};

int main()
{
	MyFile file;

	file.WriteChar('H');
	file.WriteChar('e');
	file.WriteChar('l');
	file.WriteChar('l');
	file.WriteChar('o');
	file.WriteChar(' ');
	file.WriteChar('W');
	file.WriteChar('o');
	file.WriteChar('r');
	file.WriteChar('l');
	file.WriteChar('d');

	return 0;
}
```

Wenn man von Anfang an immer alle globalen Variablen als `static`-Variablen in `static`-Methoden packt, dann kann man im Nachhinein sehr einfach Änderungen der Initialisierung erzwingen. Dabei ändert sich noch nicht einmal das Interface.

Beispiel:

Vorsorgliche Kapselung:

```
#include <stdio.h>

class MyTools
{
      public:
            static unsigned long& theCounter();
};

unsigned long& MyTools::theCounter()
{
      static unsigned long dwCnt = 0L;
      return dwCnt;
}

int main()
{
      unsigned long dwTest = ++MyTools::theCounter();
      dwTest = ++MyTools::theCounter();
      dwTest = ++MyTools::theCounter();
      dwTest = ++MyTools::theCounter();
      return (int) dwTest;
}
```

Anfangsstand des Zählers wird zur Laufzeit ermittelt:

```
unsigned long GetStartCounter() //hier nur Simulation!
{
      return 20L;
}

unsigned long& MyTools::theCounter()
{
      static unsigned long dwCnt = 0L;
      static bool bOk = false;
      if(!bOk)
      {
            dwCnt = GetStartCounter();
            bOk = true;
      }
      return dwCnt;
}
```

8.4 Lokale statische Arrays durch Main-Thread instanziieren

Wenn man eine Funktion schreibt, die später durch mehrere Threads ausgeführt wird, ist es durchaus üblich, der Funktion einen eindeutigen Index (Thread-Zähler) als Parameter mitzugeben. Im Code benutzt man dann statt einfachen Variablen jeweils ein Array von Variablen. Jeder Thread benutzt nur das Array-Element, welches er durch den ihm zugeteilten Index erreicht. Ein Problem hierbei ist, dass ein lokales statisches Array vom ersten Thread, der durch den lokalen Code läuft, instanziiert wird (die Default-Konstruktoren werden genau dann aufgerufen). Wenn man diesen ersten Aufruf nicht durchführt, bevor mehrere Threads konkurrierend auf das Array zugreifen, kann es zum Crash kommen (`access-violation`). Man muss also dafür sorgen, dass die Instanziierung des statischen Arrays abgeschlossen ist, bevor die Threads gestartet werden. Dies geschieht durch einen einmaligen Aufruf der Funktion, die das statische Array kapselt, durch den Main-Thread, und zwar bevor die Sub-Threads gestartet werden.

Beispiel mit GNU C++-Compiler und GNU Pth (Portable Threads) unter LINUX:

Vorbereitungen:

a) Herunterladen der Quell-Code-Dateien von GNU Pth:

http://www.gnu.org/software/pth/
ftp://ftp.gnu.org/gnu/pth/

z.B. Version 2.0.0:

```
pth-2.0.0.tar.gz
```

b) Entpacken:

```
tar -xvzf pth-2.0.0.tar.gz
```

c) Konfigurieren:

```
./configure --prefix=/usr/local
```

d) Bauen:

```
make
```

e) Testen:

```
make test
```

f) Installieren:

```
make install
```

Und hier das Code-Beispiel:

```
#include <stdio.h>

#include <pthread.h> //Threads

#include <string>
using namespace std;

class MyClass
{
	public:
		static void Initialize();
		static string GetText(unsigned long dwID);
		static void* MyThread(void* pdwID);

	private:
		enum{ MAX_NUM_THREADS = 100 };
		MyClass();
		MyClass(const MyClass& Obj);
		~MyClass();

		static string& arrTexts(int i);
};

void MyClass::Initialize()
{
	arrTexts(0); //hier werden alle Objekte des Arrays konstruiert
}

string MyClass::GetText(unsigned long dwID)
{
	return arrTexts(dwID);
}

void* MyClass::MyThread(void* pdwID)
{
	unsigned long dwID = (unsigned long) *((unsigned long*) pdwID);

	char szText[128] = "";
	sprintf(szText,"Thread[%ld] was executed",dwID);
	arrTexts(dwID) = szText; //Zugriff auf ein Array-Element
	pthread_exit(0);
}

string& MyClass::arrTexts(int i)
{
	static string astr[MAX_NUM_THREADS];
	if(i < MAX_NUM_THREADS)
		return astr[i];
	return astr[0];
}
```

```
int main()
{
      //Initialisierung durch Main-Thread:
      MyClass::Initialize();

      unsigned long adwID[2];
      memset(adwID,0,sizeof(adwID));

      pthread_t Thread[2];

      pthread_attr_t ThreadAttr;
      int nError = pthread_attr_init(&ThreadAttr);
      if(nError)
            return -1;
      nError = pthread_attr_setdetachstate(
                           &ThreadAttr,PTHREAD_CREATE_JOINABLE);
      if(nError)
      {
            pthread_attr_destroy(&ThreadAttr);
            return -1;
      }

      //Starten der Threads:
      adwID[0] = 0;
      nError = pthread_create(   &Thread[0],
                                 &ThreadAttr,
                                 MyClass::MyThread,
                                 (void*) &adwID[0]);
      adwID[1] = 1;
      nError = pthread_create(   &Thread[1],
                                 &ThreadAttr,
                                 MyClass::MyThread,
                                 (void*) &adwID[1]);

      //Warten bis die Threads fertig sind:
      void** ppStatus = NULL;
      pthread_join(Thread[0],ppStatus);
      pthread_join(Thread[1],ppStatus);
      pthread_attr_destroy(&ThreadAttr);

      //Ausgabe des Resultats:
      printf("%s\r\n",MyClass::GetText(0).c_str());
      printf("%s\r\n",MyClass::GetText(1).c_str());

      printf("Press <ENTER>...");
      getchar();
      return 0;
}
```

Beispiel mit Visual C++-Compiler unter WINDOWS-NT:

```
#include <stdio.h>

#include <windows.h> //Threads

#pragma warning(disable:4514)
#pragma warning(disable:4786)
#include <string>
using namespace std;

class MyClass
{
	public:
		static void Initialize();
		static string GetText(unsigned long dwID);
		static long GetTermFlag(unsigned long dwID);
		static unsigned long __stdcall MyThread(void* pdwID);

	private:
		enum{ MAX_NUM_THREADS = 100 };
		MyClass();
		MyClass(const MyClass& Obj);
		~MyClass();

		static string& arrTexts(int i);
		static long& arrTermFlags(int i);
};

void MyClass::Initialize()
{
	arrTexts(0); //hier werden alle Objekte des Arrays konstruiert
	arrTermFlags(0);
	memset(&arrTermFlags(0),0,MAX_NUM_THREADS * sizeof(long));
}

string MyClass::GetText(unsigned long dwID)
{
	return arrTexts(dwID);
}

long MyClass::GetTermFlag(unsigned long dwID)
{
	return arrTermFlags(dwID);
}

unsigned long __stdcall MyClass::MyThread(void* pdwID)
{
	unsigned long dwID = (unsigned long) *((unsigned long*) pdwID);

	char szText[128] = "";
	sprintf(szText,"Thread[%ld] was executed",dwID);
	arrTexts(dwID) = szText; //Zugriff auf ein Array-Element
	::InterlockedIncrement(&MyClass::arrTermFlags(dwID));
	return 0;
}
```

```
string& MyClass::arrTexts(int i)
{
      static string astr[MAX_NUM_THREADS];
      if(i < MAX_NUM_THREADS)
            return astr[i];
      return astr[0];
}
long& MyClass::arrTermFlags(int i)
{
      static long al[MAX_NUM_THREADS];
      if(i < MAX_NUM_THREADS)
            return al[i];
      return al[0];
}
int main()
{
      //Initialisierung durch Main-Thread:
      MyClass::Initialize();

      unsigned long adwID[2];
      memset(adwID,0,sizeof(adwID));
      unsigned long adwThreadID[2];
      memset(adwThreadID,0,sizeof(adwThreadID));
      void* ahThread[2];
      memset(ahThread,NULL,sizeof(ahThread));

      //Starten der Threads:
      adwID[0] = 0;
      ahThread[0] = ::CreateThread(   NULL,
                                      0L,
                                      MyClass::MyThread,
                                      (void*) &adwID[0],
                                      0L,
                                      &adwThreadID[0]);

      adwID[1] = 1;
      ahThread[1] = ::CreateThread(   NULL,
                                      0L,
                                      MyClass::MyThread,
                                      (void*) &adwID[1],
                                      0L,
                                      &adwThreadID[1]);

      //Warten bis die Threads fertig sind:
      while(!MyClass::GetTermFlag(0) || !MyClass::GetTermFlag(1))
            Sleep(250);
      if(ahThread[0])
            ::CloseHandle(ahThread[0]);
      if(ahThread[1])
            ::CloseHandle(ahThread[1]);

      //Ausgabe des Resultats:
      printf("%s\r\n",MyClass::GetText(0).c_str());
      printf("%s\r\n",MyClass::GetText(1).c_str());
      printf("Press <ENTER>...");
      getchar();
      return 0;
}
```

8.5 Globale Funktionen: Nutze virtuelle Argument-Methoden

Wenn man innerhalb globaler Funktionen virtuelle Methoden der übergebenen Argumente aufruft, dann hat man die Möglichkeit, die globale Funktion **für die gesamte Klassen-Hierarchie zu nutzen**.

Man kann so auch eine **argumentgesteuerte Funktion** implementieren, denn je nach Typ des Argumentes wird ja eine andere Implementierung der virtuellen Methode aufgerufen.

Beispiel:

```
class Base
{
	public:
		virtual ~Base() {} //Hack: nicht wirklich inline
		virtual long GenerateCompareValue() const;
};

inline bool IsLess(const Base& lhs,const Base& rhs)
{
	if(lhs.GenerateCompareValue() < rhs.GenerateCompareValue())
		return true;
	return false;
}
```

9. Referenz statt Zeiger (Zeiger für C-Interface)

Wo ein Zeiger ist, ist in der Regel auch ein **`new`** (bzw. `new[]`) und hoffentlich ein **`delete`** (bzw. `delete[]`). Wenn Zeiger an Funktionen übergeben werden, d.h. es existiert mehr als eine Kopie des Zeigers, dann taucht automatisch die zu lösende Frage auf, wer das `delete` ausführen soll und vor allem wann er dies zu tun hat oder tun darf. Wenn nämlich der eine `delete` auf den Zeiger anwendet und der andere noch mit dem Zeiger arbeitet, entsteht logischerweise ein Problem.

Diese Probleme sollten mit C++ der Vergangenheit angehören, den es gibt ja **Referenzen** (und die STL, welche das komplette Heap-Management kapselt, also alles, was mit `new` und `delete` zu tun hat). Hier ein paar wichtige Unterschiede zwischen Referenz und Zeiger:

- **Auf eine Referenz kann man kein `delete` anwenden**
- **Referenzen zeigen immer auf ein gültiges (nicht gelöschtes) Objekt**
- **Eine Referenz zeigt immer nur auf ein und dasselbe Objekt**
- **Der Wert NULL für eine Referenz ist nicht definiert**

Beispiel:

```
MyClass Object;
MyClass& Obj1;              //-> Compiler-Fehler
MyClass& Obj2 = Object;     //-> ok
```

Man sollte **nie einer Referenz einen Zeiger zuweisen**, da dieser NULL sein kann und eine Referenz mit dem Wert NULL nicht definiert ist.

Beispiel:

```
void g(MyClass& Obj)
{
     MyClass LocalObj = Obj;
}
int main()
{
     MyClass* pObj = NULL;
     g(*pObj); //-> access violation zur Laufzeit
     return 0;
}
```

Die Besonderheit, dass eine Referenz zur Laufzeit immer auf ein gültiges Objekt zeigen muss, erzwingt Folgendes: Eine **Referenz, welche Member-Variable ist**, muss immer bereits in der **Initialisierungsliste** des Konstruktors mit einem gültigen Wert belegt werden.

Beispiel:

```
#include <stdio.h>
#include <string>
using namespace std;

class StringManipulator
{
      public:
            StringManipulator(string& strText) : m_rstrText(strText)
            {
            }
            void OverwriteText(const string& strNewText)
            {
                  m_rstrText = strNewText;
            }
      private:
            StringManipulator& operator=(
                                    const StringManipulator& Obj);
            string& m_rstrText;
};

int main()
{
      string strText("Hello");
      StringManipulator Manipulator(strText);
      Manipulator.OverwriteText("World");
      return 0;
}
```

Man sollte immer, wenn es nur irgendwie geht, Referenzen statt Zeiger benutzen!

Einzige Ausnahme:
Man arbeitet mit einer C-Schnittstelle. Zum Beispiel ist das gesamte Programmier-Interface (API) von Windows in C definiert. Dort muss man z.B. mit Zeigern arbeiten, wenn man auf das Interface eines COM-Objektes zugreift (`Release()` nicht vergessen). Weiterhin kommt man dort um Zeiger nicht herum, wenn man Strings (sogenannte OLE-Strings oder BSTRs) zu anderen Prozessen sendet oder von dort empfängt (`SysFreeString()` nicht vergessen).

10. Funktionen, Argumente und return-Werte

10.1 Argumente sollten immer Referenzen sein

10.1.1 const-Referenz statt Wert-Übergabe (Slicing-Problem)

Man sollte die `const`-Referenz immer der Wert-Übergabe vorziehen!
Ausnahme: einfache Datentypen wie `char`, `short`, `long`.

Statt:

```
void Foo(MyClass Obj)
{
      ...
}
```

Immer:

```
void Foo(const MyClass& Obj)
{
      ...
}
```

Gründe:

- Kein Zeitverbrauch für Speicherallokierungen/-freigaben
- Kein Zeitverbrauch für Konstruktor-/Destruktoraufrufe
- Kein Zeitverbrauch für das Kopieren
- Kein unnötiger Speicherverbrauch durch das Anlegen einer lokalen Kopie
- Der Compiler verbietet das Ändern des Inhaltes, was bei einer Wertübergabe wegen der lokalen Kopie erlaubt ist und zur Verwirrung führen kann. Bei einer `const`-Referenz auf einen STL-Container muss man einen `const_iterator` benutzen.
- Kein Slicing-Problem

Das **Slicing-Problem**:

Wenn man bei einer Wert-Übergabe einen Wert vom Typ `MyClass` als Argument definiert, aber dann ein davon abgeleitetes Objekt übergibt, wird dieses in seine Basisklasse umgewandelt (Upcast). Dabei gehen natürlich einige Eigenschaften verloren, so z.B. überschriebene virtuelle Funktionen, an deren Stelle dann die entsprechende Funktion der Basisklasse aufgerufen wird.

Beispiel:

```
class MyWindow
{
      public:
            virtual ~MyWindow() {} //Hack: nicht wirklich inline
            virtual void Display() const;
};
void MyWindow::Display() const { printf("MyWindow\n"); }

class MyScrollWindow : public MyWindow
{
      public:
            void Display() const { printf("MyScrollWindow\n"); }
};

void ShowWin(MyWindow Win)
{
      Win.Display();
}

int main()
{
      MyScrollWindow W;
      ShowWin(W);
      return 0;
}
```

Hier wird nun nicht `MyScrollWindow::Display()` sondern `MyWindow::Display()` aufgerufen.

10.1.2 Referenz statt Zeiger

Man sollte die Referenz immer dem Zeiger vorziehen!

Statt:

```
      void Foo(MyClass* pObj)
      {
            ...
      }
```

Immer:

```
      void Foo(MyClass& Obj)
      {
            ...
      }
```

Grund:
Es taucht nie die Frage auf, wer das `delete` ausführen soll und vor allem wann er dies zu tun hat oder tun darf. Wenn nämlich innerhalb einer Funktion `delete` auf einen übergebenen Zeiger anwendet wird und der Aufrufer nach dem Funktionsaufruf noch mit dem Zeiger arbeitet, entsteht ein Problem.

Angenehmer Nebeneffekt:
Der Aufrufer der Funktion muss nicht wie bei C nachschauen, welche Objekte er referenzieren (&) muss, sondern übergibt einfach alle Parameter wie bei einer Wert-Übergabe. Muss die Funktion geändert werden (es wird aus einer const-Referenz eine Referenz), dann bleibt der Code des Aufrufers unangetastet.

10.2 Argumente: Default-Parameter vs. überladene Funktion

Es gibt 2 Alternativen, eine unterschiedliche **Anzahl** von Argumenten beim Aufruf einer Funktion zu behandeln:

- **Default-Werte → gleicher Code für die Behandlung**

In der Klassen-Deklaration definiert man **für alle Parameter ab dem N-ten Parameter** Default-Werte

Beispiel: Default-Werte ab dem 2. Parameter

```
class MyClass
{
      public:
         void func(int arg1,int arg2 = 0,int arg3 = 1,int arg4 = 0);
      ...
};
```

- **Überladene Funktionen → unterschiedlicher Code für die Behandlung**

Für jede Anzahl von Argumenten wird **eine eigene Funktion** (Name bleibt gleich) implementiert

Beispiel:

```
class MyClass
{
      public:
            void func(int arg1);
            void func(int arg1,int arg2);
            void func(int arg1,int arg2,int arg3);
            void func(int arg1,int arg2,int arg3,int arg4);
      ...
};
```

Hier sollte man **für gemeinsamen Code** eine `private` Funktion schreiben, die von allen aufgerufen wird.

10.3 Überladen innerhalb einer Klasse vs. über Klasse hinweg

10.3.1 Allgemeines

Es ist etwas anderes, ob man innerhalb einer Klasse eine Methode überlädt oder ob man eine Methode der Basisklasse überlädt.

Überladen innerhalb einer Klasse:

Wenn man innerhalb einer Klasse eine Methode überlädt, so entsteht eine Koexistenz von mehreren Funktionen gleichen Namens, sobald sich ihre **Parameter hinsichtlich Art und/oder Anzahl** unterscheiden.

Überladen einer Methode der Basisklasse:

Wenn man eine Methode der Basisklasse überlädt, dann geschieht dies ohne Rücksicht auf die Parameter. Es genügt, den gleichen Funktionsnamen zu benutzen. Das heißt aber auch, dass eine überladene Methode der Basis folgendermaßen geändert werden kann:

- Die Art/Anzahl der Argumente kann sich ändern
- Der return-Wert kann sich ändern

```
class MyBase
{
   public:
         virtual ~MyBase() {} //Hack: nicht wirklich inline
         virtual int Foo(int i);
};
int MyBase::Foo(int i) { ...; return 0; }

class MyClass : public MyBase
{
   public:
         int Foo(double i) { ...; return 0;}
};

int main()
{
   MyBase BaseObj;
   BaseObj.Foo(6); //-> ruft MyBase::Foo() auf

   MyClass Obj;
   Obj.Foo(8.0); //-> ruft MyClass::Foo() auf

   return 0;
}
```

Man sollte ein solches Vorgehen jedoch vermeiden. Der Compiler bringt hier in der Regel eine Warnung.

10.3.2 Nie Zeiger-Argument mit Wert-Argument überladen

Wenn man eine Überladung vornimmt, wobei die **Art** der Argumente der Funktionen unterschiedlich ist (was man vermeiden soll), dann sollte man nicht zugleich Zeiger und Werte für ein Argument zulassen.

Beispiel:

```
void f(int i);
void f(MyClass* pObj);
```

→ `f(0)` ruft **immer** `f(int i=0)` auf, obwohl auch der `NULL`-Zeiger gemeint sein könnte.

10.4 return: Referenz auf *this vs. Wertrückgabe

Wie man das Ergebnis einer Methode zurückgibt, hängt davon ab, ob ein lokal in der Methode erzeugtes Objekt oder das Objekt, zu dem die Methode gehört, zurückgeliefert werden soll.

10.4.1 Lokal erzeugtes Objekt zurückliefern: Rückgabe eines Wertes

Will man ein Objekt zurückliefern, welches man in einer Methode erzeugt hat, kann man keine Referenz (und keinen Zeiger) darauf zurückgeben, denn das lokal erzeugte Objekt wird ja wieder zerstört, wenn man die Funktion verlässt. Man gibt hier einen **Wert** oder einen **const-Wert** zurück (also eine Kopie des lokal erzeugten Objektes).

Beispiel:

```
class MyClass
{
	public:
		double MyClass::Calculate(const double& dInput)
		{
			double dReturn = dInput * 10;
			return dReturn; //-> Wert
		}
};

int main()
{
	MyClass Obj;
	double d = Obj.Calculate(7.0);
	return 0;
}
```

*10.4.2 Objekt der Methode zurückliefern: Referenz auf *this*

Wenn eine Methode das Objekt verändert, zu welchem sie gehört, und als Ergebnis eben dieses Objekt zurückliefern soll, sollte die Methode immer eine **Referenz auf `*this`** zurückliefern!

Beispiel: Operator für die Zuweisung

```
class MyClass
{
      public:
            MyClass(int nID = 0) : m_nID(nID) {}
            ~MyClass() {}
            MyClass& operator=(const MyClass& Obj)
            {
                  if(this == &Obj)
                        return *this;
                  m_nID = Obj.m_nID;
                  return *this;
            }
      private:
            int m_nID;
};

int main()
{
      MyClass Obj1(1);
      MyClass Obj2(2);
      Obj1 = Obj2;
      return 0;
}
```

10.4.3 Keine Zeiger/Referenzen auf private-Daten zurückliefern

Gründe:

- Wenn man die privaten Daten ordentlich kapseln will, darf man keine Zeiger oder Referenzen darauf zurückliefern, denn sonst kann der Aufrufer direkt darauf zugreifen.
- Wenn das Objekt zerstört wird, dann sind die Daten ungültig, obwohl noch jemand einen Zeiger oder eine Referenz darauf haben kann.
- Eine Methode, die Read-Only definiert ist (→ `const`), könnte einen Zeiger auf private Daten zurückliefern und somit indirekt doch das Schreibrecht implementieren.

Ausnahmen:

Operatoren, die einen direkten Zugriff auf die Daten ermöglichen sollen.

Beispiel: Der Zugriffs-Operator `[]`

Der Zugriffs-Operator `[]` muß eine Referenz auf die `private`-Daten zurückliefern:

```
class MyString
{
      public:
            MyString(const char* const szStr)
            {
                  strcpy(m_szStr,szStr);
            }
            char& operator[](int pos)
            {
                  return m_szStr[pos];
            }
      private:
            char m_szStr[256];
};

int main()
{
      MyString szStr("Hallo");
      char c = szStr[3];
      szStr[3] = 'X';
      c = szStr[3];
      return 0;
}
```

Bemerkung:

Um doch noch den "reinen" String im Inneren eines String-Objektes zurückliefern zu können, implementieren manche Bibliotheken in ihre String-Klasse eine Methode namens **`GetBuffer()`**, welche zugleich einen internen **Referenzzähler** hochzählt. Wenn der Nutzer des zurückgelieferten Strings mit dem String fertig ist, muss er dann **`ReleaseBuffer()`** aufrufen, um den Referenzzähler wieder zurückzudrehen. Dadurch wird verhindert, dass der String aus dem Speicher gelöscht wird, bevor der letzte Nutzer sich abmeldet. Löscht also der Besitzer das String-Objekt, wird nicht unbedingt der Speicher des internen Strings freigegeben, es sei denn der Referenzzähler geht dabei auf Null. Andernfalls wird zunächst nur ein Lösch-Flag gesetzt. Erst wenn der letzte Nutzer des internen Strings sich abmeldet (Referenzzähler geht auf Null) wird der Speicher freigegeben. Hierbei ist also sehr genau darauf zu achten, dass nie ein `ReleaseBuffer()` vergessen wird, da man sonst ein klassisches Speicherloch produziert.

10.5 return-Wert nie an referenzierendes Argument übergeben

Beispiel:

```
class MyMember
{
	public:
		MyMember(int nValue = 0) : m_nValue(nValue) {}
		~MyMember() {}
		int GetValue() const
		{
			return m_nValue;
		}
		void SetValue(int nValue)
		{
			m_nValue = nValue;
		}
	private:
		int m_nValue;
};

class MyClass
{
	public:
		MyClass(int nID = 0) : m_ID(nID) {}
		~MyClass() {}
		MyMember GetID()
		{
			return m_ID; //return-Wert
		}
	private:
		MyMember m_ID;
};

void SetValue(MyMember& Member,int nValue) //referenzierendes Arg.
{
	Member.SetValue(nValue);
}

int main()
{
	MyClass Obj(8);
	SetValue(Obj.GetID(),2); //funktioniert nicht wirklich!!!

	return 0;
}
```

Wenn in `SetValue()` schreibend auf `Member` zugegriffen wird, dann wird **nur eine temporäre Kopie** (Rückgabe-Wert von `MyClass::GetID()`) verändert. Es existiert keine wirkliche Verbindung zu `Obj`. Somit bleibt der Wert von `Obj` unverändert.

11. Smart-Pointer

11.1 Allgemeines

Smart-Pointer sind Objekte, die sich wie Zeiger verhalten, aber verschiedene Dinge automatisieren. Eine Smart-Pointer-Klasse wird sinnvollerweise als Template realisiert, damit man sich nicht auf einen einzigen Typ (auf den der Zeiger zeigen soll) festlegt.

11.2 Smart-Pointer für die Speicher-Verwaltung

11.2.1 Eigenschaften des Smart-Pointers für die Speicherverwaltung

- **Konstruktion**

Es wird ein interner Zeiger auf ein zuvor mit `new` allokiertes Objekt gespeichert.

- **Destruktion**

Das Objekt, auf das der interne Zeiger zeigt, wird mit `delete` zerstört. Dies ist wohl die wichtigste Eigenschaft der Smart-Pointer für die Speicher-Verwaltung, denn hierdurch werden Speicherlücken (Memory-Leaks) vermieden, wenn eine Exception geworfen wird. Grund: Im Rahmen des Stack-Unwinding nach dem Werfen einer Exception werden alle lokal konstruierten Objekte destruiert und somit wird hier automatisch das `delete` aufgerufen.

- **Zuweisung =**

Hier wird eine **einfache Kopie** des Zeigers vorgenommen (im Gegensatz zur '**deep copy**', wo auch eine Kopie vom Objekt angelegt wird). Die **Eigentümerschaft** wird dabei **an den Empfänger** der Kopie übertragen, wodurch dieser für das `delete` verantwortlich wird.

- **Dereferenzierung ***

Hier wird implementiert, wie das Objekt (auf welches der interne Zeiger zeigt) zurückgeliefert wird. Es wird also auf das Objekt zugegriffen.
Wenn der Objekt-Inhalt (Lesezugriff) über Server oder Datenbankzugriffe ermittelt werden muss, kann **lazy fetching** zum Einsatz kommen, d.h. es wird zunächst nur ein leeres Objekt der entsprechenden Klasse im Speicher erzeugt. Greift man dann lesend auf Member-Variablen zu, wird der entsprechende Wert ermittelt und erst dann im Objekt gespeichert (Cache).

11.2.2 Was zu beachten ist

- **Falls der Typ eine Klasse ist, muss der Zugriffsoperator definiert werden**

```
T* operator->() { return m_pT; }
```

- **Überprüfung auf NULL über implizite Typumwandlung implementieren**

  Um eine Abprüfung auf NULL zu ermöglichen, muss eine implizite Typumwandlung nach `bool` implementiert werden:

```
operator bool(); //-> if(pObj)...
                 //   (implizite Typumwandlung nach bool)
```

  Man kann statt dessen auch eine Methode

```
bool IsValid();
```

  mit dem gleichen Code zur Verfügung stellen.

- **Nie Typumwandlungs-Operator für Umwandlung in echten Zeiger implementieren**

  NIE: `operator T* { return m_pT; }`

  Hierdurch kann der Compiler eine implizite Umwandlung in einen normalen Zeiger vornehmen, was schlimme Folgen haben kann, zum Beispiel, was den Aufruf von `delete` betrifft. Außerdem kommt ein Compiler-Fehler wegen Mehrdeutigkeit, wenn man die Überprüfung auf NULL mit dem Operator `!` durchführt (`'operator !' is ambiguous`).

- **Transform()-Methode für Downcasts implementieren**

  Für **downcasts** sollte man folgende Methode implementieren:

```
template<class C>
bool Transform(SmartPtr<C>& Obj)
{
      T* pT = dynamic_cast<T*>(Obj.GetPtr());
      if(!pT)
            return false;
      m_pT = pT;
      return true;
}
```

11.2.3 Code-Beispiel

```
class MyBaseString
{
	public:
		MyBaseString()
		{
			strcpy(m_szStr,"");
		}
		MyBaseString(const char* const szStr)
		{
			strcpy(m_szStr,szStr);
		}
		virtual ~MyBaseString() {} //Hack: nicht wirklich inline
		virtual void PrintIt();
		char& operator[](int pos)
		{
			return m_szStr[pos];
		}
	protected:
		char m_szStr[256];
};
void MyBaseString::PrintIt()
{
	printf("Base: %s\n",m_szStr);
}

class MyString : public MyBaseString
{
	public:
		MyString() : MyBaseString() {}
		MyString(const char* const szStr) : MyBaseString(szStr)
		{
		}
		void PrintIt() { printf("Derived: %s\n",m_szStr); }
};

template<class T>
class SmartPtr
{
	public:
		SmartPtr(T* pT = NULL) : m_pT(pT) {}
		SmartPtr(SmartPtr<T>& Obj)
			:	m_pT(Obj.GetPtr()) //Eigentümer
		{
			Obj.Release();  //Obj von Eigentümerschaft befreien
		}
		~SmartPtr() { delete m_pT; }
		void Release()
		{
			m_pT = NULL; //von Eigentümersch.befreien
		}
		T* GetPtr()
		{
			return m_pT;
		}
```

```
            template<class C>
            bool Transform(SmartPtr<C>& Obj)
            {
                  T* pT = dynamic_cast<T*>(Obj.GetPtr());
                  if(!pT)
                        return false;
                  m_pT = pT;
                  return true;
            }
            bool IsValid()
            {
                  if(m_pT != NULL)
                        return true;
                  return false;
            }
            operator bool() { return IsValid(); }
            SmartPtr<T>& operator=(SmartPtr<T>& Obj)
            {
                  if(this == &Obj)
                        return *this;
                  if(m_pT)
                        delete m_pT;    //alten Speicher freigeben
                  m_pT = Obj.GetPtr();  //Eigentümerschaft übernehmen
                  Obj.Release();  //Obj von Eigentümersch.befreien
                  return *this;
            }
            T* operator->() { return m_pT; }
            T& operator*() { return *m_pT; }
      private:
            T* m_pT;
};

int main()
{
      //Eigentümerschaft abgeben:
      SmartPtr<MyString> pStr1(new MyString);
      SmartPtr<MyString> pStr2 = pStr1;
      pStr2->PrintIt();
      pStr1->PrintIt();//->access violation zur Laufzeit (m_pT==NULL)

      //Downcast:
      SmartPtr<MyString> pStrA(new MyString);
      SmartPtr<MyBaseString> pBaseStrA(new MyBaseString);
      pBaseStrA->PrintIt();           //-> MyBaseString::PrintIt()
      if(pBaseStrA.Transform(pStrA))  //-> Downcast
            pBaseStrA->PrintIt();     //-> MyString::PrintIt()

      //Upcast:
      SmartPtr<MyBaseString> pBaseStrB(new MyBaseString);
      SmartPtr<MyString> pStrB(new MyString);
      pStrB->PrintIt();               //-> MyString::PrintIt()
      if(pStrB.Transform(pBaseStrB))  //-> Upcast schlägt fehl !!!
            pBaseStrB->PrintIt();
      return 0;
}
```

11.2.4 Smart-Pointer immer per Referenz an eine Funktion übergeben

Bei der Wertübergabe eines Smart-Pointers wird eine Kopie per Zuweisungsoperator gemacht und die Eigentümerschaft an das Smart-Pointer-Objekt der Funktion übergeben, deren Destruktor beim Verlassen der Funktion das Objekt, auf welches der interne Zeiger zeigt, zerstört!

Dies gilt auch für den **`auto_ptr`** der STL (Smart-Pointer-Template für die Speicher-Verwaltung).

Beispiel:

```
class MyString
{
      public:
            MyString() { strcpy(m_szStr,""); }
            MyString(const char* const szStr)
            {
                  strcpy(m_szStr,szStr);
            }
            void PrintIt() { printf("%s\n",m_szStr); }
            char& operator[](int pos)
            {
                  return m_szStr[pos];
            }
      protected:
            char m_szStr[256];
};

#include <autoptr.h>

template<class T>
void func(auto_ptr<T>& pObj)
{
      pObj->PrintIt();
}


int main()
{
      auto_ptr<MyString> pStr(new MyString("Hello"));
      func(pStr);
      return 0;
}
```

11.2.5 Empfehlungen

Grundsätzlich ist es so, dass das **Debugging** bei Verwendung von Smart-Pointern sich schwieriger gestaltet als beim Einsatz von normalen Zeigern. Auf der anderen Seite befreien Smart-Pointer einen bei richtiger Implementierung (siehe auto_ptr der STL) automatisch von Speicherlücken (Memory-Leaks).

In 2 Fällen ist es in der Regel lohnend, Smart-Pointer einzusetzen:

a) Smart-Pointer bei Code im try-catch-Block:

Man kann sich bei der Speicher-Verwaltung entweder für eine **manuelle** Lösung entscheiden, wobei man **alle Heap-Zeiger vor dem try-Block definiert** und alle konstruierten Heap-Objekte mit **delete hinter dem catch-Block** löscht, oder für eine **automatische** Lösung, wobei man Smart-Pointer (Bsp.: auto_ptr der STL) für die Speicher-Verwaltung benutzt. Die automatische Methode erspart viel Arbeit und ist auf jeden Fall sicherer!

Statt:

```
class MyString
{
      public:
            MyString(const char* const szStr)
            {
                  if(szStr[0] == 0)
                        throw "string is empty!";
                  strcpy(m_szStr,szStr);
            }
            char& operator[](int pos)
            {
                  return m_szStr[pos];
            }
      protected:
            char m_szStr[256];
};

int main()
{
      MyString* pStr1 = NULL;
      MyString* pStr2 = NULL;
      try
      {
            pStr1 = new MyString("Hello");
            pStr2 = new MyString("");
      }
      catch(const char* const szError)
      {
            printf("Error: %s\n",szError);
      }
      if(pStr1)
            delete pStr1;
      if(pStr2)
            delete pStr1;
      return 0;
}
```

Besser:

```
class MyString
{
	public:
		MyString(const char* const szStr)
		{
			if(szStr[0] == 0)
				throw "string is empty!";
			strcpy(m_szStr,szStr);
		}
		char& operator[](int pos)
		{
			return m_szStr[pos];
		}
	protected:
		char m_szStr[256];
};

#include "autoptr.h"

int main()
{
	try
	{
		auto_ptr<MyString> pStr1(new MyString("Hello"));
		auto_ptr<MyString> pStr2(new MyString(""));
	}
	catch(const char* const szError)
	{
		printf("Error: %s\n",szError);
	}
	return 0;
}
```

b) Code mit vielen return-Anweisungen:

Bei einer Speicher-Verwaltung über Smart-Pointer braucht man sich über nichts Gedanken zu machen. Tätigt man hingegen die `delete`-Aufrufe manuell, dann muss man vor jedem `return` auf alle gültigen lokalen Zeiger `delete` anwenden, was den Code aufbläht und sehr unschön aussieht.

11.3 Smart-Pointer für andere Zwecke

Smart-Pointer können natürlich auch zum Automatisieren anderer Dinge als einer Speicher-Verwaltung eingesetzt werden. Beliebtes Einsatzgebiet ist die Aktivierung (`QueryInterface()`) bzw. Freigabe (`Release()`) von COM-Interfaces unter Windows.

12. new/delete

12.1 Allgemeines zu new

Das eingebaute Sprachelement **new** bewirkt durch

```
MyClass* pObj = new MyClass(...);
```

folgende, **durch den Compiler generierte Dinge**:

a) Speicher auf dem Heap besorgen:

```
void* pMem = operator new(sizeof(MyClass));
```

b) Objekt im Speicher konstruieren:

```
MyClass* pObj = static_cast<MyClass*>(pMem);
```

c) Konstruktor aufrufen (kann der Programmierer nicht explizit):

```
pObj->MyClass(...);
```

Es gibt also einen wesentlichen Unterschied zwischen **new** und **operator new**:

operator new kann man überladen und somit die Speicherreservierung beeinflussen,
new kann man nicht überladen!

Es gibt 3 Arten der new-Benutzung:

1.) Man will Speicher besorgen und ein Objekt (pObj) auf dem Heap konstruieren:
→ **new**

```
MyClass* pObj = new MyClass(...);
```

2.) Man will nur Speicher (pBuf) auf dem Heap besorgen:
→ **operator new**

```
const size_t BYTE_SIZE = 1024;
void* pBuf = operator new(BYTE_SIZE);
```

3.) Man hat bereits Speicher (pBuf) auf dem Heap und möchte ein Objekt genau dort konstruieren:
→ **Placement-new**

```
pObj = new (pBuf) MyClass(...);
```

Beachte: Der Programmierer kann einen Konstruktor nicht explizit aufrufen!

12.2 Allgemeines zu delete

Das eingebaute Sprachelement **`delete`** bewirkt durch

```
delete pObj;
```

folgende, **durch den Compiler generierte Dinge**:

a) Destruktor aufrufen:

```
pObj->~MyClass();
```

b) Speicher freigeben:

```
operator delete(pObj);
```

Es gibt also einen wesentlichen Unterschied zwischen **`delete`** und **`operator delete`**:

`operator delete`
kann man überladen und somit die Speicherfreigabe beeinflussen

`delete`
kann man nicht überladen!

Es gibt 2 Arten der delete-Benutzung:

1.) Man will ein Objekt (`pObj`) zerstören und dessen Speicher auf dem Heap freigeben:

→ **`delete`**

```
delete pObj;
```

2.) Man will nur Speicher (`pBuf`) auf dem Heap freigeben:

→ **`operator delete`**

```
operator delete(pBuf);
```

12.3 Beispiel für new/delete

```
#include <new.h>
class MyString
{
	public:
		MyString(const char* const szStr)
		{
			strcpy(m_szStr,szStr);
		}
		char& operator[](int pos)
		{
			return m_szStr[pos];
		}
	protected:
		char m_szStr[256];
};

int main()
{
	//new:
	MyString* pStr1 = new MyString("Hello");		//new
	void* pBuf = operator new(1024);		//operator new
	MyString* pStr2 = new (pBuf) MyString("World");	//Placement-new

	//delete:
	delete pStr1;		//delete
	operator delete(pBuf);	//operator delete

	return 0;
}
```

Beachte: Placement-`new` benötigt kein gesondertes `delete`!

12.4 Allgemeines zu new[]/delete[]

12.4.1 new[]

Das eingebaute Sprachelement **`new[]`** bewirkt durch

```
MyClass* arrObjs = new MyClass[10];
```

folgende, **durch den Compiler generierte Dinge** (Pseudo-Code):

a) Speicher auf dem Heap besorgen (gekapselt im **`operator new[]`**):

```
void* pMem = operator new(10 * sizeof(MyClass));
```

b) Objekt im Speicher konstruieren und **Default-**Konstruktor aufrufen:

```
MyClass* pObj = NULL;
for(int i = 0;i < 10;++i)
{
      pObj = static_cast<MyClass*>(pMem + i * sizeof(MyClass));
      pObj->MyClass(); //Default-Konstruktor
      if(i == 0)
            arrObjs = pObj;
}
```

12.4.2 delete[]

Das eingebaute Sprachelement **`delete[]`** bewirkt durch

```
delete[] arrObjs;
```

folgende, **durch den Compiler generierte Dinge** (Pseudo-Code):

Destruktoren aufrufen und Speicher freigeben (gekapselt im **`operator delete[]`**):

```
MyClass* pObj = NULL
for(int i = 0;i < sizeof(arrObjs)/sizeof(MyClass);++i)
{
      pObj = arrObjs[i];
      pObj->~MyClass();
      operator delete(pObj);
}
```

Würde man nur `delete` (ohne `[]`) aufrufen, dann würde nur das erste Objekt (`arrObjs = &arrObjs[0]`) destruiert und sein Speicher freigegeben werden.

12.5 Mit Heap-Speicher arbeiten

```
#include <new.h>
const size_t BYTE_SIZE = 1024;
void* pBufStart = operator new(BYTE_SIZE);

void* pBuf = pBufStart;

... //mit pBuf arbeiten

operator delete(pBufStart);
```

Man sollte `malloc`/`free` in C++ vermeiden. Einen Spezialfall findet man unter **Windows**, wo **globaler Heap** für die Kommunikation über die **Zwischenablage** (Copy&Paste, Drag&Drop) benutzt wird, wozu es die API-Funktionen **`GlobalAlloc()`** und **`GlobalFree()`** gibt.

12.6 Heap-Speicher als Shared Memory

Wenn bereits ein Puffer (`pBuf`) existiert, dann kann man beliebig oft beliebig verschiedene Objekte dort platzieren, solange diese nicht größer als der reservierte Speicherbereich sind.

Beispiel:

```
#include <new.h>
#include <list>
using namespace std;

class MyString
{
	public:
		MyString(const char* const szStr)
		{
			strcpy(m_szStr,szStr);
		}
		~MyString() { printf("Destruction\n"); }
		char& operator[](int pos)
		{
			return m_szStr[pos];
		}
		void PrintIt() { printf("%s\n",m_szStr); }
	protected:
		char m_szStr[256];
};

int main()
{
	//Heap als Shared-Memory reservieren:
	const size_t BYTE_SIZE = 1024 * 1024; //1 MB
	void* pSharedMemory = operator new(BYTE_SIZE);
	void* pMem = pSharedMemory;

	//Shared-Memory als String-Objekt nutzen:
	MyString* pStr = new (pMem) MyString("Hello");

	//Shared-Memory als Liste mit Integern benutzen:
	list<int>* plistInts = new (pMem) list<int>;
	plistInts->push_back(1);
	plistInts->push_back(2);
	plistInts->push_back(3);

	//Shared-Memory wieder als String-Objekt nutzen:
	pStr = new (pMem) MyString("World");

	//Shared-Memory wieder freigeben:
	operator delete(pSharedMemory);

	return 0;
}
```

12.7 new/delete statt malloc/free

`new`/`delete` sind in C++ immer statt `malloc`/`free` zu benutzen.

Beispiel: String-Arrays

`new`

→ Alle Strings werden vollständig durch `new` initialisiert, da nicht nur der Konstruktor des Arrays, sondern **auch die Konstruktoren aller String-Objekte** aufgerufen werden:

```
string* arrStrings = new string[10];
```

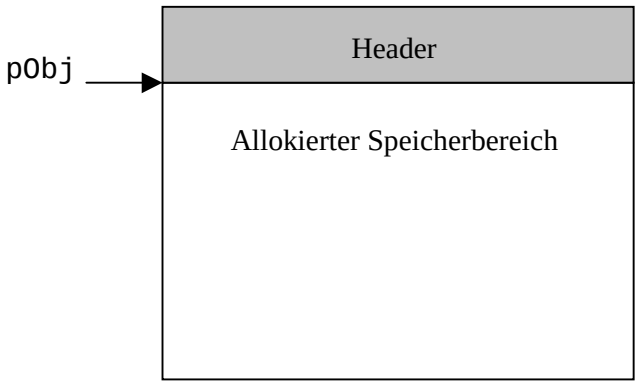
`delete`

→ Die **Destruktoren aller String-Objekte** werden aufgerufen:

```
delete[] arrStrings;
```

Arbeitsweise:

`new` allokiert nicht nur Speicher für das Objekt, sondern auch einen Header mit Informationen über die Größe des allokierten Bereiches. Diese Information wird von `delete` benötigt um den Speicher wieder freizugeben (man fragt sich nämlich immer, wie `delete` es schafft, allein aufgrund eines Zeigers die Größe des Bereiches, der freizugeben ist, zu bestimmen).



Wenn man nun den Speicher immer blockweise reserviert

```
void* pBlock = operator new(1024)
```

und auch blockweise wieder freigibt

```
delete pBlock
```

dann hat man nur einen Header pro Block und kann über Listen mit Blöcken und Zeigern eine schnelle Speicher-Verwaltung per Zeiger implementieren (siehe STL-Implementierungen).

Beispiel:

```
#include <new.h>

class MyListItem
{
	public:
		MyListItem() : m_pNext(NULL),m_nID(IncCounter()) {}
		long GetID() { return m_nID; }
		void SetNext(MyListItem* pNext) { m_pNext = pNext; }
		MyListItem* GetNext() { return m_pNext; }
		static long IncCounter();
	private:
		MyListItem* m_pNext;
		long	m_nID;
};

long MyListItem::IncCounter()
{
	static long lCounter = 0;
	return ++lCounter;
}

int main()
{
	//Block im Speicher allokieren:
	void* pBlock = operator new(1024*1024); //1 MB

	//Zeiger für den Speicher:
	MyListItem* pMem = (MyListItem*) pBlock;

	//Zeiger für die Liste:
	MyListItem* pRoot = NULL;
	MyListItem* pPrevItem = NULL;
	MyListItem* pItem = NULL;

	//Liste mit 100 Items generieren:
	pItem = pRoot = new (pMem) MyListItem();
	pMem += sizeof(MyListItem);
	pPrevItem = pItem;
	for(;;)
	{
		pItem = new (pMem) MyListItem();
		pMem += sizeof(MyListItem);
		pPrevItem->SetNext(pItem);
		pPrevItem = pItem;
		if(pItem->GetID() == 100)
			break;
	}
```

```
	//IDs der Items der Liste auslesen:
	pItem = pRoot;
	long l = 0L;
	for(;;)
	{
		l = pItem->GetID();
		if(l == 100)
			break;
		pItem = pItem->GetNext();
	}

	//Speicher wieder freigeben:
	delete pBlock;

	return 0;
}
```

12.8 Zusammenspiel von Allokierung und Freigabe

`new`/`new[]` bewirkt:

- Speicherallokierung
- Konstruktoraufruf(e)

`delete`/`delete[]` bewirkt

- Destruktoraufruf(e)
- Speicherfreigabe

ACHTUNG!
Da `new` eine Information über den Objekttyp vorfindet

```
string* arrStrings = new string[256];
```

ruft **`new` immer alle Konstruktoren** auf, also bei einem Array (`new[]`) auch die der einzelnen Elemente. Da `delete` diese Information nicht vorfindet, sondern nur den reinen Zeiger, **nimmt `delete` an, es handele sich nur um ein einzelnes Objekt**:

```
delete arrStrings; //ruft nur Destruktor von arrStrings[0] auf
```

Deshalb muss man **bei Arrays `delete[]`** benutzen:

```
delete[] arrStrings;
```

Einzel-Objekte:

```
MyClass* pMyObj = new MyClass;
...
delete pMyObj;
```

Arrays:

```
MyClass* arrMyObj = new MyClass[100];
...
delete[] arrMyObj;
```

Vorsicht:
Durch ein `typedef` kann ein Array unter Umständen wie ein Einzel-Objekt aussehen!

Beispiel:

```
#include <new.h>
class MyString
{
	public:
		MyString()
		{
			sprintf(m_szStr,"String %ld",IncCounter());
		}
		char& operator[](int pos) {	return m_szStr[pos]; }
		void PrintIt() { printf("%s\n",m_szStr); }
		static long IncCounter();
	protected:
		char m_szStr[256];
};

long MyString::IncCounter()
{
	static long lCounter = 0;
	return ++lCounter;
}

int main()
{
	MyString* arrStrings = new MyString[256];
	int i = 0;
	for(;;)
	{
		arrStrings[i].PrintIt();
		if(++i == 256)
			break;
	}
	delete[] arrStrings;
	return 0;
}
```

12.9 Eigener new-Handler statt Out-Of-Memory-Exception

new wirft eine Exception (Out-Of-Memory-Exception), wenn kein Speicher allokiert werden konnte! Hierzu: **new** kann (abgesehen von Placement-new) auf 3 verschiedene Arten aufgerufen werden:

```
... = new Type;       //→ Aufruf des Default-Konstruktors
... = new Type(Obj);  //→ Aufruf des Copy-Konstruktors
... = new Type[n];    //→ Array: Aufruf von n Default-Konstruktoren
```

Man kann das **Werfen der Exception abschalten**, wenn man einen **eigenen new-Handler** angibt. Hierzu dient die Funktion **_set_new_handler()**, die einen Zeiger auf einen new-Handler (_PNH) erwartet und einen Zeiger auf den alten new-Handler zurückliefert:

```
typedef int (__cdecl * _PNH)( size_t );
_PNH _set_new_handler(_PNH pNewHandler);
```

Man sollte für solche Fälle **beim Programmstart einen Speicherblock blockieren (lock)**, um ihn an dieser Stelle als letzten Rettungsanker freizugeben.

Beispiel:

```
#include <new.h>
bool NoMemoryAvailable(bool bOutOfMemory = false)
{
	static bool bNoMoreMem = false;
	if(bOutOfMemory)
		bNoMoreMem = true;
	return bNoMoreMem;
}
bool LockMemory(bool bLock,size_t size = 0)
{
	static void* pLockedMemory = NULL;

	bool bRetMemoryReleased = false;
	if(bLock)
	{
		if(!pLockedMemory)
			pLockedMemory = operator new(size);
	}
	else
	{
		if(pLockedMemory)
		{
			operator delete(pLockedMemory);
			pLockedMemory = NULL;
			bRetMemoryReleased = true;
			NoMemoryAvailable(true);
		}
	}
	return bRetMemoryReleased;
}
```

```
int MyNewHandler(size_t size)
{
      if(LockMemory(false)) //falls Platz geschaffen werden konnte
      {
            printf("Warnung: Der Speicherplatz wird knapp!\n");
            return 0;
      }
      printf("FEHLER: Der Speicherplatz ist zu knapp!!!\n");
      return 1;
}

#include <list>
using namespace std;
int main()
{
      //Speicher für den Notfall blockieren:
      LockMemory(true,2 * sizeof(long[9999999]));

      //Eigenen new-Handler setzen:
      _set_new_handler(MyNewHandler);

      //Speicherüberlauf auf dem Heap herbeiführen:
      list<long*> listPtr;
      for(;;)
      {
            if(NoMemoryAvailable()) //falls der Speicher zu Ende geht
                  break;
            long* p = new long[9999999];
            listPtr.push_back(p);
      }

      //Speicher auf dem Heap wieder freigeben:
      list<long*>::iterator it;
      for(it = listPtr.begin();it != listPtr.end();++it)
            delete[] (*it);
      listPtr.clear();

      return 0;
}
```

12.10 Heap-Speicherung erzwingen/verbieten

12.10.1 Heap-Speicherung erzwingen (protected-Destruktor)

Um die Heap-Speicherung (`new`) zu erzwingen, **versteckt man den Destruktor**. Damit man die Klasse noch als Basisklasse verwenden kann, wird der Destruktor jedoch **nicht hinter private** versteckt, sondern hinter protected. Nun erlaubt der Compiler keine direkte Konstruktion mehr, da die **automatische Destruktion nicht mehr möglich** ist (diese wird ja beim Verlassen des Gültigkeitsbereiches ausgeführt), weil der Destruktor von außerhalb nicht mehr aufgerufen werden kann. Das Objekt kann also **nur noch mit `new`** erzeugt werden. Für die manuelle Zerstörung des Objektes mit `delete` wird nun eine **`Destroy()`**-Methode bereitgestellt.

Beispiel:

```
class MyMember
{
      public:
            MyMember(int nValue = 0) : m_nValue(nValue) {}
            int GetValue() { return m_nValue; }
            void Destroy() { delete this; }
      protected:
            ~MyMember() {}
      private:
            int m_nValue;
};

class MyClass
{
      public:
            MyClass(int nID = 0) : m_pID(new MyMember(nID)) {}
            ~MyClass() { m_pID->Destroy(); }
            int GetID() { return m_pID->GetValue(); }
      private:
            MyMember*  m_pID;
            MyMember   m_ID; //-> Fehler: Destruktor nicht erreichbar
};

int main()
{
      MyClass Obj1;
      int i = Obj1.GetID();
      MyClass Obj2(4);
      int j = Obj2.GetID();
      return 0;
}
```

12.10.2 Heap-Speicherung verbieten (private operator new)

Um die Heap-Speicherung zu verbieten, versteckt man **`operator new`** und **`operator new[]`** hinter `private`:

```
class MyMember
{
      public:
            MyMember(int nValue = 0) : m_nValue(nValue) {}
            int GetValue() { return m_nValue; }
            ~MyMember() {}
      private:
            static void* operator new(size_t size);
            static void* operator new[](size_t size);
            int m_nValue;
};
```

```
class MyClass
{
	public:
		MyClass(int nID = 0) : m_ID(nID) {}
		~MyClass() {}
		int GetID() { return m_ID.GetValue(); }
	private:
		MyMember m_ID;
};

int main()
{
	MyClass Obj1;
	int i = Obj1.GetID();
	MyClass Obj2(4);
	int j = Obj2.GetID();
	MyMember* p = new MyMember(8); //-> Compiler-Fehler
	return 0;
}
```

Besonderheit:

Wenn ein Objekt mit verstecktem `operator new` als **Member eines anderen Objektes** verwendet wird und dieses andere Objekt mit `new` auf dem Heap konstruiert wird, dann wird auch das Member-Objekt dorthin konstruiert:

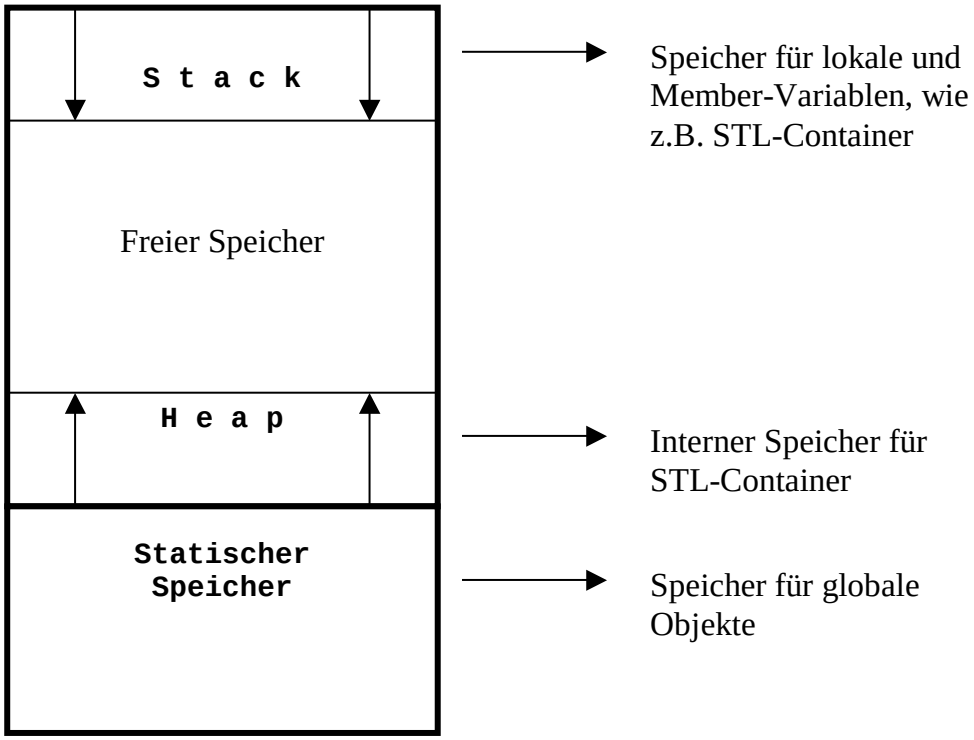
```
	MyClass* p = new MyClass(8);
```

Allerdings wird es behandelt wie ein Stack-Objekt (Stack im Heap).

13. Statische, Heap- und Stack-Objekte

13.1 Die 3 Speicher-Arten

Es gibt 3 Lokalitäten, wo Objekte gespeichert werden können:



a) Statischer Speicher

→ Speicher mit einer **Struktur, die während der gesamten Programmlaufzeit konstant bleibt**. Jedes Objekt und jede Variable behält ihren festen Platz. Hier werden **statische** und **globale** Objekte/Variablen gespeichert. Wenn statische Objekte/Variablen lokal (in einer Methode) verwendet werden, dann sind sie außerhalb der Methode unsichtbar.

```
int g_iFlags; //globale Variable

class MyStaticClass //statisches Objekt
{
     public:
          static long AddItem(const string& strItem);
          static void DeleteItem(long lHandle);
     private:
          MyStaticClass();
          MyStaticClass(const MyStaticClass& Obj);
          ~MyStaticClass();
};
```

b) Heap

→ Dynamischer Speicher, der **manuell verwaltet** wird (**`new`** /**`delete`**)

```
MyClass* pObj = new MyClass;
MyClass* pObj = new MyClass(...);
MyClass* pObj = new MyClass[256];
```

c) Stack

→ Dynamischer Speicher, der **vom Compiler verwaltet** wird. Die lokale Definition eines Objektes oder einer Variablen wird vom Compiler in **`push`**-Befehle für die Stackverwaltung der CPU umgesetzt. Beim Verlassen des Gültigkeitsbereiches (z.B. am Ende einer Methode) entfernt der Compiler die Objekte/Variablen wieder vom Stack (**`pop`**-Befehle).

Neben den lokalen Objekten/Variablen werden auch die Member-Variablen von Objekten vom Compiler auf den Stack "gepushed".

```
class MyClass
{
      public:
         MyClass(list<unsigned long>& rlistParentHandles)
            : m_rlistParentHandles(rlistParentHandles)
         {
         }
         ...
      private:
         unsigned long m_dwFlags;
         list<unsigned long> m_listHandles;
         list<unsigned long>& m_rlistParentHandles;
};
void MyClass::Foo()
{
      set<unsigned long> setValues;
      int i = 0;
      ...
}
```

Der **statische Speicher** verändert seine Größe während des Programmlaufs nicht. **Heap** und **Stack** hingegen sind **dynamisch** und teilen sich den freien Speicher. Da weder die maximale Größe des Stacks noch die maximale Größe des Heaps zu Programmbeginn feststeht, liegt ihre Startadresse an entgegengesetzten Enden des freien Speichers und sie wachsen aufeinander zu.

Entscheidungshilfen:

- **Global** verfügbare Objekte sind als **statische Objekte** zu implementieren, **nicht** als Heap-Objekte.
- **Lokale** und **Member**-Variablen (z.B. STL-Container) gehören auf den **Stack**.
- Die Verwendung von **`new/delete`** (Heap-Speicher) sollte der **STL** vorbehalten bleiben. In Ausnahmefällen ist es ratsam mit dem Smart-Pointer `auto_ptr` der STL zu arbeiten.

Im Folgenden wird zusammenfassend gezeigt, wie man Klassen schreibt um gezielt eine der 3 Speicherarten zu **erzwingen**.

13.2 Statische Objekte (MyClass::Method())

- **Konstruktor**, **Copy-Konstruktor** und **Destruktor** sind hinter **`protected`** versteckt
- Nur **`static`-Methoden**

Beispiel:

```
class MyClass
{
      public:
            static void Method() { printf("Method()\n"); }
      protected:
            MyClass();
            MyClass(const MyClass& Obj);
            ~MyClass();
};

int main()
{
      MyClass::Method();
      return 0;
}
```

13.3 Heap-Objekte (pObj->Method())

- **Destruktor** ist hinter **`protected`** versteckt
- **`public`-`Destroy()`-Methode**

Beispiel:

```
class MyClass
{
      public:
            MyClass() {}
            void Method() { printf("Method()\n"); }
            void Destroy() { delete this; }
      protected:
            ~MyClass() {}
};

int main()
{
      MyClass* pObj = new MyClass;
      pObj->Method();
      pObj->Destroy();
      return 0;
}
```

13.4 Stack-Objekte (Obj.Method())

- **`operator new`** ist hinter **`private`** versteckt
- **`operator new[]`** ist hinter **`private`** versteckt

Beispiel:

```
class MyMember
{
      public:
            MyMember() {}
            ~MyMember() {}
            void Method() { printf("Method()\n"); }
      private:
            static void* operator new(size_t size);
            static void* operator new[](size_t size);
};

int main()
{
      MyMember Obj;
      Obj.Method();
      return 0;
}
```

14. Programmierung einer Klasse

14.1 Allgemeines

14.1.1 Fragen, die beim Entwurf einer Klasse beantwortet werden sollten

- In welchen Speicher sollen die Objekte?
  → Statisch: Verstecken von Konstruktor und Destruktor; nur `static`-Methoden
  → Stack: Verstecken von `operator new()` und `operator new[]()`
  → Heap: Verstecken des Destruktors und Implementieren von `Destroy()`-Funktion
  → Stack oder Heap: Konstruktor und Destruktor sind `public`

- Soll die Objektanzahl (1..*) kontrolliert werden?
  → Objekt-Manager als `friend`-Klasse, `Create()`-Funktion oder `static`-Klasse?

- Darf ein Konstruktor mit genau 1 Argument (falls vorhanden) zur automatischen Typumwandlung herangezogen werden?
  → falls nein: `explicit` vor den Konstruktor schreiben

- Wie geschieht die Werte-Zuweisung?
  → Implementieren oder Verstecken von Copy-Konstruktor u. Zuweisungs-Operator `=`

- Soll die Klasse als Basisklasse dienen?
  → auf jeden Fall virtuellen Destruktor definieren

- Welche Methoden sollen von außen zugänglich sein (Interface)?
  → als `public` deklarieren (alles andere ist `private` oder evtl. `protected`)

- Wer soll (ausnahmsweise!) Zugriff auf die privaten Daten haben?
  → ggf. Objekt-Manager als `friend`-Klasse einbeziehen
  → ggf. globale Operatoren als globale `friend`-Funktionen einbeziehen

- Was sind die Beschränkungen für die Werte?
  → Parameterüberprüfungen

- Welche geerbten Basisklassen-Methoden müssen überschrieben werden?
  → ggf. ist die Basisklassen-Methode explizit aufzurufen (Zustandsmaschine)

- Welche Methoden sollen spezialisierbar sein, wenn die Klasse als Basisklasse dient?
  → virtuelle vs. nicht virtuelle Funktionen

- Welche Standard-Operatoren und -Funktionen sind zu deaktivieren (bzw. zu verstecken)?
  → als `private` deklarieren und <u>nicht implementieren</u>

- Soll die Klasse für verschiedene Daten-Typen der enthaltenen Member definiert werden?
  → ggf. als Template definieren (Bsp.: Liste)

14.1.2 Die wesentlichen Methoden einer Klasse sind zu implementieren

Eine Klasse sollte in der Regel immer folgende Methoden definieren:

Auf jeden Fall definieren oder gezielt verstecken:

- Default-Konstruktor (keine Argumente und ausschließlich Default-Argumente)
- Copy-Konstruktor
- **Virtuellen** Destruktor (→ vftable → RTTI → `dynamic_cast` auf jeden Fall möglich)
- `operator=()`

Definieren, falls die Objekte in STL-Containern verwendet werden sollen:

- Globale Operatoren `operator==()`, `operator!=()` und `operator<()`

Definieren, falls ein indizierter Zugriff möglich sein soll:

- `operator[]()`

14.1.3 Durch den Compiler automatisch generierte Methoden beachten

Der Compiler generiert automatisch folgende Methoden für eine Klasse:

- Default-Konstruktor:
  Falls **überhaupt kein Konstruktor** definiert wurde

- Destruktor (nicht virtuell):
  Falls nicht definiert

- Copy-Konstruktor (bitweise Kopie):
  Falls Kopie per Konstruktor im Code vorkommt oder eine Zuweisung in Verbindung mit einer Konstruktion.

- `operator=()` (bitweise Kopie):
  Falls Zuweisung im Code vorkommt (nicht in Verbindung mit einer Konstruktion)

- `operator&()`:
  Falls entsprechende Referenzierung im Code vorkommt

Beispiel:

```
class Empty
{
};

int main()
{
      Empty Obj1; //-> Compiler gener.Default-Konstrukt. & Destruktor
      Empty Obj2(Obj1); //-> Compiler generiert Copy-Konstruktor
      Obj2 = Obj1; //-> Compiler generiert Zuweisungs-Operator
      Empty* pConstObj = &Obj1; //-> Compiler generiert Adreßoperator
      return 0;
}
```

Wenn man **vermeiden** will, dass die Methoden automatisch generiert werden, dann deklariert man sie im **`private`**-Bereich (es genügt bereits, wenn dies in der Basisklasse geschehen ist).

Zum automatisch generierten Copy-Konstruktor:

Der automatisch generierte Copy-Konstruktor weist die Werte aller Member-Variablen mit **Ausnahme** der **`static`-Member** zu. Bestehen die Member wiederum aus Objekten, dann wird dazu deren Copy-Konstruktor oder `operator=()` aufgerufen. Wenn jedoch beides nicht vorhanden ist, dann werden wiederum einfach alle Member-Variablen zugewiesen. Es handelt sich also insgesamt um einen rekursiven Vorgang, der jeweils so weit in die Tiefe geht, bis er auf einen Copy-Konstruktor oder einen `operator=()` oder einen eingebauten elementaren Datentyp (→ bitweise Kopie) stößt.

Ausnahmefälle, in denen Copy-Konstruktor und `operator=()`nicht generiert werden:

- Wenn sich unter den Member-Variablen Referenzen oder `const`-Datenelemente befinden
- Wenn die Basisklasse die Methoden abgeschaltet hat (also im `private`-Abschnitt versteckt hat)

```
class Empty
{
   private:
      Empty(const Empty& Obj);   //-> nein !!!
   private:
      MyClass          m_Obj1;    //-> ok
      MyClass*         m_pObj3;   //-> ok
      const MyClass    m_Obj2;    //-> nein !!!
      MyClass&         m_pObj4;   //-> nein !!!
};
```

14.1.4 inline-Funktionen ggf. hinter die Deklaration schreiben

Statt die Implementierung kleiner Funktionen innerhalb der Klassen-Deklaration vorzunehmen, kann man sie **hinter die Deklaration in Form von `inline`-Funktionen** anhängen. Sie bleiben natürlich in der Header-Datei, jedoch wird die Deklaration der Klasse für den Nutzer überschaubarer.

```
class MyClass
{
     public:
          void Func();
     ...
};
inline void MyClass::Func()
{
   ...
}
```

Ausnahme: **Leere Funktionen**:

```
class MyClass
{
     public:
          MyClass() {}
     ...
}
```

14.1.5 Nie public-Daten verwenden

Man sollte Daten immer `private` oder `protected` deklarieren!

Gründe:

- **Kapselung** → Sicherheit durch kontrollierbaren Zugriff

- **Nur-Lese-Zugriff** durch Implementierung von **Read-Only**-Member-Funktionen möglich

  Beispiel:

```
class MyClass
{
      public:
            int ReadData() const { return m_nData; }
            ...
      private:
            int m_nData;
            ...
};
```

- **Variablen können ohne Interface-Änderung durch einen Algorithmus ersetzt werden**

  Die hierbei sinnvollen Algorithmen sind solche, bei denen die Zeit eine Rolle spielt:
  → Lazy-Fetching, Lazy-Evaluation, ...

  Beispiel:
  Wenn man die mittlere Geschwindigkeit eines Vorganges zur Verfügung stellen will, dann gibt es 2 Alternativen, diesen Wert bereitzustellen:

  - **Automatisch aktualisierte Variable bereitstellen:**
    Man aktualisiert den Wert per Timer innerhalb des Objektes (im Hintergrund) durch eine gleitende Mittelwertbildung und greift beim Lesen nur auf die Variable zu:
    Vorteile:
    - schneller Lesevorgang
    - wenig Speicherbedarf für die Abtastwerte (kleiner Ringpuffer)

    Nachteil:
    - Performance-Verlust, wenn der Wert nur selten gelesen wird

  - **Algorithmus bereitstellen:**
    Man aktualisiert den Wert erst beim Lesen (Lazy Evaluation) durch eine Berechnung über alle Abtastwerte:
    Vorteil:
    - Kein Performance-Verlust, wenn der Wert nur selten gelesen wird

    Nachteile:
    - langsamer Lesevorgang
    - hoher Speicherbedarf für die Abtastwerte (großer Ringpuffer)

14.1.6 Mehrdeutigkeiten (ambigious) erkennen

Problem:

Wenn der Compiler durch automatische implizite Typumwandlung mehr als eine passende überladene Funktion für ein Argument findet, dann streikt er: "`Error - ambigious`".

Beispiel:

```
void f(int i)
{
     int j = i % 2;
}

void f(char c)
{
     char k = c;
}

int main()
{
     double d = 2.3;
     f(d); //-> Fehler: ambigious:
           //   Compiler weiß nicht in was er umwandeln soll
     return 0;
}
```

Abhilfe: → `static_cast`

Zum obigen Beispiel:

```
int main()
{
     double d = 2.3;
     f(static_cast<int>(d));
     f(static_cast<char>(d));
     return 0;
}
```

Problem:

Verwendet man bei der Merfachvererbung gleiche Namen für die Methoden in den Basisklassen, dann kommt es ebenfalls zur Mehrdeutigkeit.

Beispiel:

```
class MyBase1
{
      public:
            MyBase1(int nID = 0) : m_nID(nID) {}
            void Do() { printf("MyBase1\n"); }
      private:
            int   m_nID;
};

class MyBase2
{
      public:
            MyBase2(int nID = 0) : m_nID(nID) {}
            void Do() { printf("MyBase2\n"); }
      private:
            int   m_nID;
};

class MyClass : public MyBase1, public MyBase2
{
};

int main()
{
      MyClass Obj;
      Obj.Do();  //-> Error - ambigious

      return 0;
}
```

Abhilfe: → Expliziter Zugriff auf die Methode einer Basisklasse

Zum obigen Beispiel:

```
class MyClass : public MyBase1, public MyBase2
{
      public:
            void Do() { MyBase1::Do(); }
};
```

14.2 Der Konstruktor

14.2.1 Kein new im Konstruktor / Initialisierungslisten für Member

Regel:

Ein Destruktor wird nur dann aufgerufen, wenn alle Aktionen im Konstruktor **ohne Exceptions** ausgeführt wurden. Eine Exception kann hier bspw. wegen Speichermangel auftreten (Out-Of-Memory).

Möchte man teilweise allokierten Speicher für Member-Variablen nicht ungenutzt herumliegen lassen, dann darf man den **Speicher nicht innerhalb eines Konstruktors allokieren**.

Man geht wie folgt vor:

Man setzt die **Zeiger auf jeden Fall auf einen definierten Wert**. Dies kann man erreichen indem man vor der Konstruktion eine sogenannte **Initialisierungsliste** abarbeiten lässt. Dann allokiert man den Speicher erst wenn er gebraucht wird (new) und gibt ihn im Destruktor wieder frei (delete). Wird der Speicher sofort benötigt, dann kann man new in die Initialisierungsliste einbauen:

Falsch:

```
MyClass::MyClass()
{
      m_pRect = new Rect(0,0,1,1);
      m_pCircle = new Circle(1,1);
}

MyClass::~MyClass()
{
      delete m_pRect;
      delete m_pCircle;
}
```

Richtig:

```
MyClass::MyClass() : m_pRect(NULL),m_pObj(new CClass(1,1))
{
}

MyClass::~MyClass()
{
      delete m_pRect;
      delete m_pCircle;
}

MyClass::Func()
{
      if(!m_pRect)
            m_pRect = new CRect(0,0,1,1);
      ...
}
```

Die **Initialisierungliste** ist auf jeden Fall bevorzugt vor der Initialisierung im Konstruktor zu benutzen, da Folgendes gilt:

a) **`const`-Member-Variablen** können nicht innerhalb des Konstruktors initialisiert werden.
b) **Referenz-Member-Variablen** können nicht innerhalb des Konstruktors initialisiert werden.

Funktioniert nicht:

```
class MyClass
{
     public:
          MyClass(int nID = 0)
          {
               m_nID = nID;
               m_cID = nID;    //-> Fehler
               m_rID = &m_nID; //-> Fehler
          }
     private:
          int        m_nID;
          const int  m_cID;
          int&       m_rID;
};

int main()
{
     MyClass Obj(7);
     return 0;
}
```

Funktioniert:

```
class MyClass
{
     public:
          MyClass(int nID = 0)
               :    m_nID(nID),
                    m_cID(nID),
                    m_rID((int&) MyClass::m_nID)
          {}
     private:
          int        m_nID;
          const int  m_cID;
          int&       m_rID;
};

int main()
{
     MyClass Obj(7);
     return 0;
}
```

Regel:

> **Immer die Initialisierungsliste zum Initialisieren bei der Konstruktion verwenden!**
>
> **Einzige Ausnahme:**
> Initialisierung vieler Variablen eines einfachen Datentyps mit dem gleichem Wert.
>
> Beispiel:
> `a = b = c = d = e = f = g = h = i = 0;`

ACHTUNG!
Die **Reihenfolge**, in der die Initialisierungsliste abgearbeitet wird, hat nichts mit der Reihenfolge der Aufführung der Variablen dort zu tun. Sie orientiert sich an der **Reihenfolge der Deklaration in der Klasse**. Weiterhin gilt: Basisklasse vor abgeleiteter Klasse, ... usw.

Beispiel: Bei beiden Klassen ist die Reihenfolge der Initialisierung gleich:

```
class MyClass1
{
	public:
		MyClass1(int nID = 0)
			:	m_nID(nID),
				m_cID(nID),
				m_rID((int&) MyClass::m_nID)
		{}
	private:
		int		m_nID;	//1
		const int	m_cID;	//2
		int&		m_rID;	//3
};
class MyClass2
{
	public:
		MyClass2(int nID = 0)
			:	m_rID((int&) MyClass::m_nID),
				m_nID(nID),
				m_cID(nID)
		{}
	private:
		int		m_nID;	//1
		const int	m_cID;	//2
		int&		m_rID;	//3
};
```

Grund:
Könnte man die Reihenfolge selbst bestimmen, also eine andere vorgeben, als bereits durch die Deklaration feststeht, dann müsste der Compiler sich die geänderte Reihenfolge zusätzlich merken, damit er die Zerstörung in entsprechend umgekehrter Reihenfolge zur Konstruktion durchführen kann. Das macht er aber nicht.

14.2.2 Keine virtuellen Methoden im Konstruktor aufrufen

Innerhalb eines Konstruktors sollte man keine virtuellen Methoden des Objektes aufrufen, denn erst nach kompletter Duchführung der Konstruktion des Objektes steht eindeutig fest, wohin der Methodenaufruf geleitet werden muss (`vftable`).

14.2.3 Arrays mit memset() initialisieren

Statt einer Schleife zum Initialisieren ist auf jeden Fall `memset()` vorzuziehen, da es effektiver ist:

Statt:

```
int aInt[256];
for(unsigned int i = 0;i < sizeof(aInt)/sizeof(int);++i)
     aInt[i] = 0;
```

Besser:

```
int aInt[256];
memset(aInt,0,sizeof(aInt));
```

14.3 Der Destruktor

14.3.1 Generalisierung ("is-a"): Basisklasse soll virtuellen Destruktor haben

Gründe:

- **`RTTI bzw. dynamic_cast`**

Durch das Hinzufügen von virtuellen Funktionen wird dem Objekt ein **vfptr** hinzugefügt. Dieser Zeiger zeigt auf ein **Array mit Funktions-Zeigern**, die sogenannte **vftable**. Da das Vorhandensein einer vftable Bedingung dafür ist, dass ein Laufzeit-Typ-Info-Objekt (RTTI-Object) an das Objekt angehängt werden kann, ermöglicht bereits allein die Tatsache, dass der Destruktor virtuell ist, dass man ein **`dynamic_cast`** (sicheres Downcasten zu abgeleiteten Klassen) durchführen kann.

- **`delete`**

Wenn man versucht, ein **Objekt über einen Zeiger auf seine Basisklasse** (dynamischer Typ des Zeigers = abgeleitete Klasse, statischer Typ des Zeigers = Basisklasse) mit `delete` zu **löschen**, dann ist das Verhalten undefiniert, wenn der Destruktor der Basisklasse nicht virtuell ist.

Beachte:

- Man sollte zumindest eine **leere Funktion** für den virtuellen Destruktor implementieren.

- Der Compiler generiert immer auch den Aufruf des Destruktors der Basisklasse!

- `virtual` und `inline` schließen sich gegenseitig aus.

  → **`inline`** wird durch **`virtual`** unwirksam
  → Der virtuelle Destruktor kann einfach `inline` in die Deklaration geschrieben werden

Beispiel:

```
class MyBase
{
      public:
            virtual ~MyBase() {} //Hack: nicht wirklich inline
            void Meth1() { printf("Meth1()\n"); }
};
```

14.4 Zuweisung per operator=()

14.4.1 Keine Zuweisung an sich selbst

Aus mehreren Gründen ist eine Zuweisung eines Objektes an sich selbst nicht erlaubt:

- Effizienz

- Die alten Werte dürfen nicht freigegeben werden (wie es bei einer normalen Zuweisung geschieht)

  → Man sollte am Anfang des Operators folgenden Code finden:

```
MyClass& MyClass::operator=(const MyClass& Obj)
{
      if(this == &Obj)
            return *this;
      ...
      return *this;
}
```

*14.4.2 Referenz auf *this zurückliefern*

Der Zuweisungs-Operator muss eine Referenz auf `*this` zurückliefern!

Gründe:

- Verkettung muss möglich sein:

  Beispiel: `x = y = z = 0;`

  → Der Rückgabewert des einen Operators ist gleich dem Argument des anderen:

  z ← 0
  y ← z
  x ← y

- Zeigerkonflikt muss vermieden werden:

Referenz auf das Argument ist nicht erlaubt, da sonst der Empfänger nicht eine Referenz auf das Objekt mit dem zugewiesenen Wert bekommt, sondern eine Referenz auf das Objekt, von dem die Werte übernommen wurden.

14.4.3 Alle Member-Variablen belegen

Es müssen die Werte **aller** Member-Variablen zugewiesen werden, **auch die der Basisklassen**!

Beispiel:

```
class MyBase
{
      public:
            MyBase(const int nID = 0) : m_nID(nID) {}
            virtual ~MyBase() {} //Hack: nicht wirklich inline
            void SetID(int nID) { m_nID = nID; }
            int GetID() const { return m_nID; }
      private:
            int m_nID;
};
```

```
template<class T>
class MyClass : public MyBase
{
	public:
		MyClass(const int nID = 0,const int nValue = 0)
			: MyBase(nID),m_nValue(nValue) {}
		~MyClass() {}
		MyClass& operator=(const MyClass& Obj)
		{
			if(this == &Obj)
				return *this;
			SetID(Obj.GetID());		//Basisklasse
			SetValue(Obj.GetValue());	//Eigene Member
			return *this;
		}
		void SetValue(int nValue) { m_nValue = nValue; }
		int GetValue() const { return m_nValue; }
	private:
		int m_nValue;
};

int main()
{
	MyClass<int> Obj1(4);
	MyClass<int> Obj2(6);
	Obj2 = Obj1;
	return 0;
}
```

14.5 Indizierter Zugriff per operator[]()

Der `operator[]()` muss als Ergebnis immer eine **Referenz** zurückliefern, denn diese wird benötigt um nicht nur lesend, sondern **auch schreibend** auf die per Index adressierte Speicherstelle zugreifen zu können.

Beispiel:

```
class MyIntArr
{
	public:
		MyIntArr(const int nInit = 0)
		{
			memset(m_arrInt,0,sizeof(m_aInt));
		}
		int& MyIntArr::operator[](int pos)
		{
			if((pos) && (pos < 256))
				return m_arrInt[pos];
			return m_arrInt[0];
		}
```

```
            const int* GetArray() const { return m_arrInt; }
            MyIntArr& operator=(const MyIntArr& Obj)
            {
                  if(this == &Obj)
                        return *this;
                  const int* parrInt = Obj.GetArray();
                  for(int i = 0;i < 256;++i)
                        m_arrInt[i] = *(parrInt++);
                  return *this;
            }
      private:
            int m_arrInt[256];
};

int main()
{
      MyIntArr A(8);
      A[3] = 5;
      return 0;
}
```

14.6 Virtuelle Clone()-Funktion: Heap-Kopie über pBase

Mittels virtueller Clone()-Funktion kann man Heap-Kopien über den Basisklassen-Zeiger machen. Man kann also eine Funktion implementieren, die über den Basisklassen-Zeiger (statischer Typ) operiert, aber jederzeit eine Kopie des dynamischen Typs erzeugen kann.

Beispiel:

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

class MyBase
{
      public:
            virtual MyBase* Clone() const = 0;
            virtual ~MyBase();
            void SetID(int nID) { m_nID = nID; }
            int GetID() const { return m_nID; }
      private:
            int m_nID;
};

MyBase::MyBase() {}
```

```
class MyClass : public MyBase
{
	public:
		explicit MyClass(int nID = 0)
		{
			SetID(nID);
			strcpy(m_szType,"Child");
		}
		MyClass(const MyClass& Obj)
		{
			SetID(Obj.GetID());
		}
		~MyClass() {}
		MyBase* Clone() const
		{
			return new MyClass(*this);
		}
	private:
		char m_szType[256];
};

void f(const list<MyBase*>& listObjs)
{
	list<MyBase*> listCloneObjs;

	list<MyBase*>::const_iterator it1;
	for(it1 = listObjs.begin();it1 != listObjs.end();++it1)
		listCloneObjs.push_back((*it1)->Clone()); //Kopie

	list<MyBase*>::iterator it2;
	for(it2 = listCloneObjs.begin();
	    it2 != listCloneObjs.end();
	    ++it2)
	{
		//Kopie untersuchen:
		printf("Obj-ID: %d\n",(*it2)->GetID());

		//Kopie löschen:
		delete (*it2);
		(*it2) = NULL;
	}
}

int main()
{
	list<MyBase*> listMainObjs;

	MyClass Obj1(1);
	MyClass Obj2(2);

	listMainObjs.push_back(&Obj1);
	listMainObjs.push_back(&Obj2);

	f(listMainObjs);

	return 0;
}
```

14.7 Objektanzahl über private-Konstruktor kontrollieren

Wenn man den Konstruktor hinter `private` versteckt, dann kann man nur noch über eine `friend`-Klasse (Objekt-Manager) oder über einen `static`-Pseudo-Konstruktor Objekte der Klasse erzeugen.

14.7.1 Objekte über eine friend-Klasse (Objekt-Manager) erzeugen

Beispiel:

```
class MyObjManager;

class MyClass
{
      friend MyObjManager;
      private:
            MyClass(int nID = 0) : m_nID(nID) {} //verstecken
            int m_nID;
};

class MyObjManager
{
      public:
            MyObjManager() {}
            MyClass* CreateMyClassObj(int nID = 0)
            {
                  if(IncNumObjs())
                        return new MyClass(nID);
                  return NULL;
            }
            static bool IncNumObjs();
      private:
            enum{ nMaxObjs = 10 };
};

bool MyObjManager::IncNumObjs()
{
      static int nNumMyClassObjs = 0;

      if(nNumMyClassObjs == nMaxObjs)
            return false;
      ++nNumMyClassObjs;
      return true;
}

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

```
int main()
{
	int i = 0;
	list<MyClass*> listObjPtr;

	MyObjManager ObjMan;
	for(;;)
	{
		MyClass* pObj = ObjMan.CreateMyClassObj(++i);
		if(pObj != NULL)
		{
			printf("Obj No. %d -> ok\n",i);
			listObjPtr.push_back(pObj);
		}
		else
		{
			printf("Obj No. %d -> *failure*\n",i);
			break;
		}
	}

	list<MyClass*>::iterator it;
	for(it = listObjPtr.begin();it != listObjPtr.end();++it)
		delete (*it);

	return 0;
}
```

14.7.2 Objekte über eine statische Create()-Funktion erzeugen

Beispiel:

```
class MyClass
{
	public:
		static MyClass* CreateObj(int nID = 0)
		{
			if(IncNumObjs())
				return new MyClass(nID);
			return NULL;
		}
		static bool IncNumObjs();
	private:
		MyClass(int nID = 0) : m_nID(nID) {} //verstecken
		enum{ nMaxObjs = 10 };
		int m_nID;
};
bool MyClass::IncNumObjs()
{
	static int nNumMyClassObjs = 0;

	if(nNumMyClassObjs == nMaxObjs)
		return false;
	++nNumMyClassObjs;
	return true;
}
```

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

int main()
{
	int i = 0;
	list<MyClass*> listObjPtr;
	for(;;)
	{
		MyClass* pObj = MyClass::CreateObj(++i);
		if(pObj != NULL)
		{
			printf("Obj No. %d -> ok\n",i);
			listObjPtr.push_back(pObj);
		}
		else
		{
			printf("Obj No. %d -> *failure*\n",i);
			break;
		}
	}
	list<MyClass*>::iterator it;
	for(it = listObjPtr.begin();it != listObjPtr.end();++it)
		delete (*it);
	listObjPtr.clear();

	return 0;
}
```

14.7.3 Genau 1 Objekt erzeugen (Code und/oder Tabelle)

Beispiel:

```
class MyTable
{
	public:
		static void InitTable();
		static int GetItem(int pos);
	private:
		MyTable() {} //verstecken
		static enum{ nNumItems = 3 };
};

int MyTable::GetItem(int pos)
{
	if((pos <0) || (pos >= nNumItems))
		return 0;

	static int arrTable[nNumItems];
	static bool bNotInitialized = true;
```

```
      if(bNotInitialized)
      {
            for(int i = 0;i<nNumItems;++i)
                  arrTable[i] = i;
            bNotInitialized = false;
      }

      return arrTable[pos];
}
int main()
{
      int i = MyTable::GetItem(2);
      return 0;
}
```

14.8 Klassen neu verpacken mittels Wrapper-Klasse

Wenn man eine oder mehrere existierende Klassen unverändert verwenden will, sie aber mit einem neuen Interface versehen möchte, dann verpackt man sie in eine andere Klasse, eine sogenannte Wrapper-Klasse. Diese Klasse macht weiter nichts, als Aufrufe der eingepackten Klasse(n) zu kapseln oder zu bündeln, ohne neue Funktionalität in Form einer Implementierung hinzuzufügen. Aus Sicht der Wrapper-Klasse findet also mehr oder weniger eine Delegation der Funktionsaufrufe an die eingepackte Klasse (die eigentliche Implementierung) statt.

Motivation für die Anwendung einer Wrapper-Klasse:

- Das Interface einer existierenden Klasse soll verbessert oder an eine neue Umgebung angepasst werden.

- Die Nutzung der Funktionalität einer existierenden Klasse soll vereinfacht oder gebündelt werden.

- Die Nutzung von existierendem C-Code soll in ein objektorientiertes Konzept eingebettet werden.

  Es kann z.B. vorkommen, dass ein Hardware-Hersteller für den Betrieb seiner Hardware nur ein C-Interface bereitstellt und neben der Header-Datei nur eine vorkompilierte Binärdatei mitliefert. Damit schützt er seinen Code vor fremden Augen und verhindert, dass man diesen abändern kann. Man muss also das C-Interface mittels Wrapper-Klasse in sein objektorientiertes Konzept einbetten.

- Man trennt in einer Architektur Kernel-Code (z.B. reines C++) von Infrastruktur-Code (z.B. Server-Interface).

  In diesem Fall kann man den Kernel (die gewrappte Klasse) portabel halten, indem man dort nur reines C++ (natürlich inclusive STL) verwendet. In der Wrapper-Klasse jedoch benutzt man Bibliotheken zur Implementierung einer Infrastruktur wie z.B. einer Client/Server-Architektur.

15. Richtiges Vererbungs-Konzept

15.1 Allgemeines

15.1.1 Nie von (nicht-abstrakten) Klassen ohne virtuellen Destruktor erben

Da im Destruktor der abgeleiteten Klasse Heap-Speicher freigegeben werden kann, sollte der Destruktor der Basisklasse virtuell sein und somit die Destruktion an die abgeleitete Klasse weiterleiten.

Beispiel:

```
class MyBaseArray
{
	public:
		MyBaseArray() { memset(m_arrInt,0,sizeof(m_arrInt)); }
		virtual ~MyBaseArray() {} //Hack: nicht wirklich inline
		virtual MyBaseArray* GetHeapArrayExt();
	protected:
		int m_arrInt[256];
};
MyBaseArray::MyBaseArray* GetHeapArrayExt() { return NULL; }

class MyArray : public MyBaseArray
{
	public:
		MyArray() : m_pHeapArrayExt(new MyBaseArray()) {}
		~MyArray() { delete m_pHeapArrayExt; }
		MyBaseArray* GetHeapArrayExt()
		{
			return m_pHeapArrayExt;
		}
	private:
		MyBaseArray* m_pHeapArrayExt; //Heap-Erweiterung
};

int main()
{
	MyBaseArray* pHeapArray = new MyArray();
	MyBaseArray* pHeapArrayExt = pHeapArray->GetHeapArrayExt();
	delete pHeapArray; //nur ok, weil Basisdestruktor virtuell ist
	return 0;
}
```

15.1.2 Nie den Copy-Konstruktor-Aufruf der Basisklasse vergessen

Vergisst man den Aufruf des Copy-Konstruktors der Basisklasse im Konstruktor, Copy-Konstruktor oder Zuweisungsoperator der abgeleiteten Klasse, dann werden die `private` Member-Variablen der Basisklasse nicht definiert belegt, was zu Problemen führen kann.

Beispiel:

```
class MyBase
{
      public:
            explicit MyBase(const int nID = 0) : m_nID(nID) {}
            MyBase(const MyBase& Obj) : m_nID(Obj.GetID()) {}
            virtual ~MyBase() {} //Hack: nicht wirklich inline
            void SetID(int nID) { m_nID = nID; }
            int GetID() const { return m_nID; }
      private:
            int m_nID;
};

class MyClass1 : public MyBase
{
      public:
            MyClass1(int nID,int nValue) : MyBase(nID),m_nValue(nValue) {}
            MyClass1(const MyClass1& Obj)
                  : MyBase(Obj.GetID()),m_nValue(Obj.GetValue()) {}
            ~MyClass1() {}
            void SetValue(int nValue) { m_nValue = nValue; }
            int GetValue() const { return m_nValue; }
      private:
            int m_nValue;
};

class MyClass2 : public MyBase
{
      public:
            MyClass2(int nID,int nValue) : m_nValue(nValue) {}
            MyClass2(const MyClass2& Obj) : m_nValue(Obj.GetValue()) {}
            ~MyClass2() {}
            void SetValue(int nValue) { m_nValue = nValue; }
            int GetValue() const { return m_nValue; }
      private:
            int m_nValue;
};

int main()
{
      MyClass1 Obj1(1,2);
      MyClass1 ObjClone1(Obj1);
      printf(     "Obj1-ID = %d\n   ObjClone1-ID = %d\n",
                  Obj1.GetID(),
                  ObjClone1.GetID());
      MyClass2 Obj2(1,2);
      MyClass2 ObjClone2(Obj2); //Fehler (Basisklassen-Member nicht belegt)
      printf(     "Obj2-ID = %d\n   ObjClone2-ID = %d\n",
                  Obj2.GetID(),
                  ObjClone2.GetID());
      return 0;
}
```

15.1.3 Statischer/dynamischer Typ und statische/dynamische Bindung

Der Compiler bindet **nicht-virtuelle Funktionen** statisch, d.h. ihr Code steht bereits nach der Kompilierung fest. **Virtuelle Methoden** werden dynamisch gebunden. Das bedeutet, dass erst zur Laufzeit feststeht, welcher Code zu verwenden ist (vftable).

Als **statischen Typ** eines Zeigers bezeichnet man den Typ, auf den er bei der Definition zeigt:

```
MyBase* pObj = NULL; //statischer Typ: MyBase
```

Als **dynamischen Typ** bezeichnet man den Typ des Objektes, auf den der Zeiger zur Laufzeit zeigt:

```
pObj = new MyClass(); //dynamischer Typ: MyClass
```

Der Aufruf **virtueller Funktionen** wird über den **vfptr** an die **vftable** geleitet und von da an den dynamischen Typ weitergegeben → **dynamische Bindung**. Der Aufruf **nicht-virtueller Funktionen** wird an den **statischen Typ** weitergeleitet → **statische Bindung**.

Beachte: **Default-Parameter werden immer der statischen Bindung entnommen!**

Beispiel:

```
class MyBase
{
	public:
		virtual MyBase() {} //Hack: nicht wirklich inline
		void PrintStaticType() { printf("MyBase\n"); }
		virtual void PrintDynamicType();
};
void MyBase::PrintDynamicType() { PrintStaticType(); }

class MyClass : public MyBase
{
	public:
		MyClass() {}
		void PrintDynamicType() { printf("MyClass\n"); }
};

int main()
{
	MyBase* pObj = new MyClass();
	pObj->PrintStaticType();
	pObj->PrintDynamicType();
	return 0;
}
```

15.1.4 Nie die Default-Parameter virtueller Funktionen überschreiben

Default-Werte der Argumente virtueller Methoden werden der statischen Bindung eines Objektes entnommen, so dass eine **Redefinition in überschriebenen virtuellen Methoden unwirksam** wird.

Beispiel:

```
class MyBase
{
     public:
          virtual MyBase() {} //Hack: nicht wirklich inline
          virtual void PrintNo(int nNo = 1);
};
void MyBase::PrintNo(int nNo = 1) { printf("No=%d\n",nNo); }
class MyClass : public MyBase
{
     public:
          MyClass() {}
          void PrintNo(int nNo = 2) { printf("No = %d\n",nNo); }
};
int main()
{
     MyBase* pObj = new MyClass();
     pObj->PrintNo(); //Default-Parameter von MyBase (1)
     return 0;
}
```

15.1.5 public-, protected- und private-Vererbung gezielt verwenden

- **Default-Zugriff auf Klassen bzw. Strukturen**

Klassen: Default-Zugriff ist **`private`**

```
class MyClass                 class MyClass
{                             {
     ...              =            private:
};                                       ...
                              };
```

Strukturen: Default-Zugriff ist **`public`**

```
struct MyClass                class MyClass
{                             {
     ...              =            public:
};                                       ...
                              };
```

- **Was wird wie geerbt?**

  Generell erbt man **immer alle `public`- und `protected` Member** der Basisklasse. Das Schlüsselwort bei der Vererbung (public, protected oder private) gibt jedoch die **höchst zulässige Öffentlichkeit** vor:

  - **`public`-Vererbung:**

    ```
    class MyClass : public MyBase
    {
          ...
    };
    ```

    → `MyClass` erbt alle **`public`**- und **`protected`**-Member von `MyBase`, und zwar als **`public`**- und **`protected`**-Member (alles bleibt unverändert)

  - **protected-Vererbung:**

    ```
    class MyClass : protected MyBase
    {
          ...
    };
    ```

    → `MyClass` erbt **`public`**- und **`protected`**-Member von `MyBase`, und zwar als **`protected`**-Member (`public` wird also zu `protected`)

  - **`private`-Vererbung:**

    ```
    class MyClass : private MyBase
    {
          ...
    };
    ```

    → `MyClass` erbt **`public`**- und **`protected`**-Member von `MyBase`, und zwar als **`private`**-Member (alles wird zu `private`)

  Beispiel:

  ```
  class MyBase
  {
        public:
              virtual MyBase() {} //Hack: nicht wirklich inline
              void PrintNo(int nNo) { printf("No = %d\n",nNo); }
  };

  class MyClass : protected MyBase
  {
        public:
              MyClass() {}
              void Print(int No) { PrintNo(No); }
  };
  ```

```
int main()
{
      MyClass Obj;
      Obj.PrintNo(2); //Fehler, da PrintNo() protected
      Obj.Print(2);   //Ok
      return 0;
}
```

- **Logische Bedeutung der Vererbungstechniken**

  - **`public`-Erbung einer Generalisierung ("is-a"):**

    ```
    class MyClass : public MyBase
    {
          ...
    };
    ```

    → Ein `MyClass`-Objekt **ist ein** spezielles `MyBase`-Objekt

  - **`public`-Erbung einer abstrakten Interface-Spezifikation ("implements"):**

    ```
    class MyClass : public MyMixin
    {
          ...
    };
    ```

    → Ein `MyClass`-Objekt **implementiert** das `MyMixin`-Interface

  - **`private`-Erbung einer Implementierung ("contains"):**

    ```
    class MyClass : private MyImpl1
    {
          ...
    };
    ```

    → Ein `MyClass`-Objekt **beinhaltet** ein `MyImpl1`-Objekt

  - **Member (`private`) einbetten ("has-a"):**

    ```
    class MyClass
    {
          ...
          private:
                MyMember m_Member;
    };
    ```

    → Ein `MyClass`-Objekt **hat ein** `MyMember`-Objekt als Member

- **Interface über (`private`) Zeiger nutzen ("uses")**

```
class MyClass
{
      ...
      private:
            MyInterface* m_pInterface;
};
```

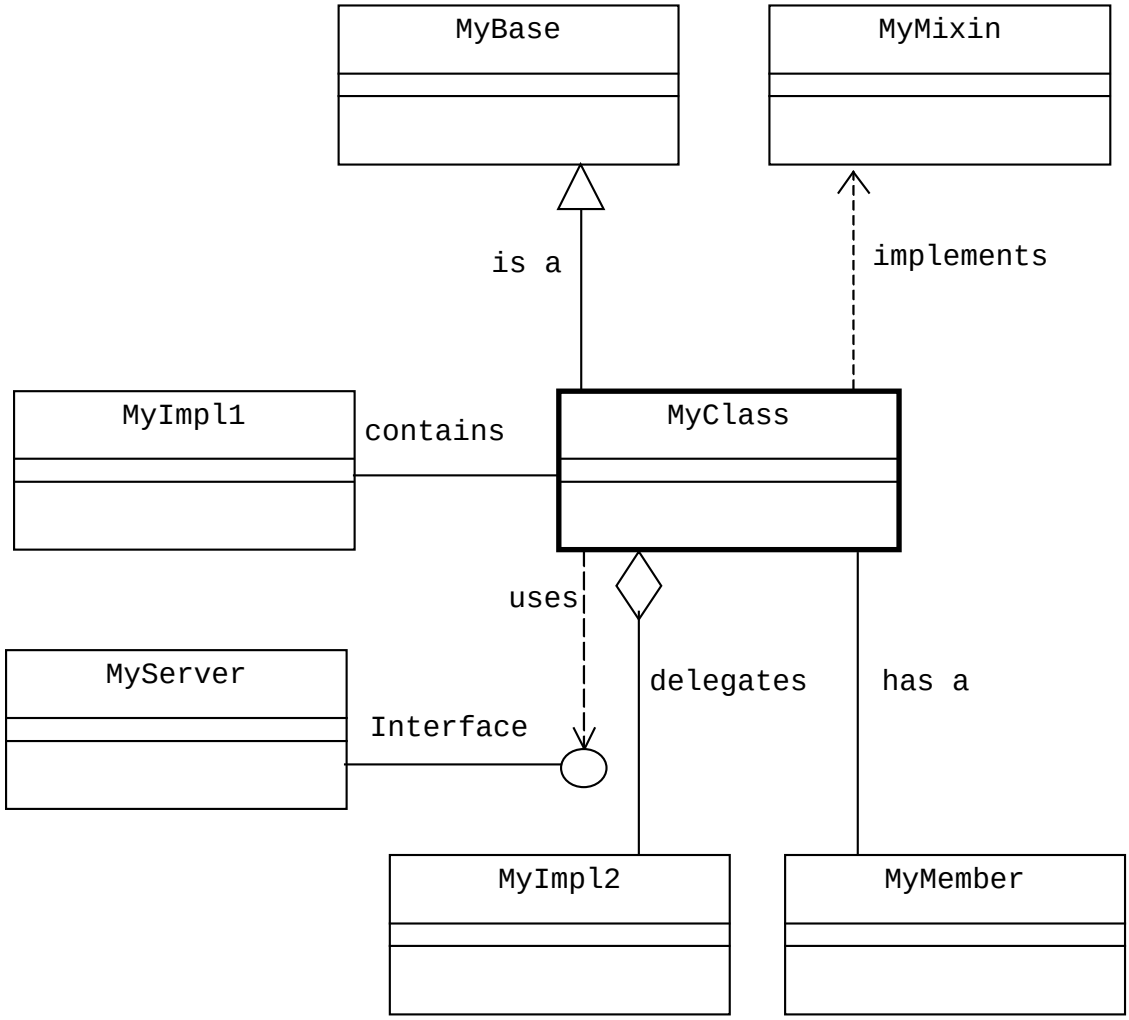
→ Ein `MyClass`-Objekt **benutzt ein** `MyInterface`-Objekt, welches in der Regel in ein anderes Objekt eingebettet ist (`MyServer::XInterface::Func()`)

- **Implementierung über (private) Zeiger nutzen ("delegate", "aggregate")**

```
class MyClass
{
      ...
      private:
            MyImpl2* m_pImpl2;
};
```

→ Ein `MyClass`-Objekt **delegiert Aufrufe an ein** `MyImpl2`-Objekt

Das Ganze in UML:



15.1.6 Rein virtuell / virtuell / nicht-virtuell

Rein virtuelle Methoden = Spezifikation

```
virtual void Func() = 0;
```

→ Funktionen, die die **Art des Aufrufes** festlegen (Schnittstellen-Spezifikation)

Virtuelle Methoden = Generalisierte Implementierung

```
virtual void Func() { printf("To implement!\n"); }
```

→ **Funktionen, die die Art** des Aufrufes festlegen und eine **allgemeine Implementierung** (Basis-Implementierung) bereitstellen, die durch Überschreiben spezialisiert werden kann oder muss.

Nicht-virtuelle Methoden = Statische Implementierung

```
void Func() { printf("Implementation ok\n"); }
```

→ Funktionen, die eine **verbindliche Implementierung** festlegen.

15.1.7 Rein virtuelle Methoden, wenn keine generalisierte Implem. möglich

Lässt sich keine gemeinsame verallgemeinerte Basis-Implementierung finden, dann wird nur die Schnittstelle spezifiziert, also eine rein virtuelle Methode in der Basisklasse angelegt.

Beispiel:

```
class MyBaseItem
{
	public:
		~MyBaseItem() {}
		virtual MyBaseItem() {} //Hack: nicht wirklich inline
		virtual void PrintColor() = 0; //rein virtuell
};

class MyItem : protected MyBaseItem
{
	public:
		MyItem(const char* const szColor)
		{ strcpy(m_szColor,szColor); }
		void PrintColor()
		{
			printf("Farb-Code: %s\n",m_szColor);
		}
	private:
		char m_szColor[256];
};
```

```
int main()
{
      MyItem Obj1("Red");
      MyItem Obj2("Blue");
      MyItem Obj3("Green");

      Obj1.PrintColor();
      Obj2.PrintColor();
      Obj3.PrintColor();

      return 0;
}
```

15.2 Spezialisierung durch public-Vererbung ("is a")

Spezialisierung → Ein Objekt der abgeleiteten Klasse **ist ein** ("is a") Objekt der Basisklasse, wobei die Funktionalität spezialisiert und in der Regel die Datenmenge erweitert wurde.

Die Basisklasse sollte also ein **verallgemeinertes Konzept** der abgeleiteten Klasse darstellen:

- Ein Basisklassen-**Zeiger** (Statischer Typ = Basisklasse) kann auch auf ein Objekt der abgeleiteten Klasse zeigen (Dynamischer Typ = abgeleitete Klasse). Es ist dabei **kein Downcast** der Zeiger notwendig (`dynamic_cast`).
- Ein Basisklassen-**Referenz**-Argument kann auch ein Objekt der abgeleiteten Klasse referenzieren.
- Alle Manipulationen, die mit einem Basisklassen-Objekt durchgeführt werden können, können auch mit einem Objekt der abgeleiteten Klasse durchgeführt werden.

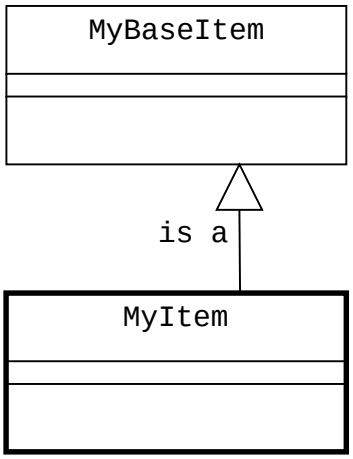
Beachte:

Man sollte **Default-Parameter von virtuellen Funktionen** nie überschreiben, da die Redefinition sowieso nicht genutzt wird. Grund: Der Compiler setzt die Default-Werte beim Kompilieren statisch ein.

Beispiel:

```
      class MyBaseItem
      {
            public:
                  MyBaseItem() {}
                  virtual ~MyBaseItem() {}
                  virtual void PrintObjInfo() const;
      };

      void MyBaseItem::PrintObjInfo() const
      {
            printf("Type: MyBaseItem\n");
      }
```

```
class MyItem : public MyBaseItem
{
      public:
            MyItem(const char* const szColor)
            { strcpy(m_szColor,szColor); }
            void PrintObjInfo() const
            { printf("Type: MyItem\nColor: %s\n",m_szColor); }
      private:
            char m_szColor[256];
};

void PrintInfo(MyBaseItem& Obj)
{
      Obj.PrintObjInfo();
}

void PrintInfo(MyBaseItem* pObj)
{
      pObj->PrintObjInfo();
}

int main()
{
      MyBaseItem Obj1;
      PrintInfo(&Obj1);
      PrintInfo(Obj1);

      MyItem Obj2("Green");
      PrintInfo(&Obj2);
      PrintInfo(Obj2);

      return 0;
}
```



15.3 Code-Sharing durch private-Vererbung ("contains")

Bei `private`-Vererbung baut man vorhandenen Code ein. Es geht hierbei rein um die Technik der Implementierung und nicht um das Herstellen einer Beziehung zwischen Basis- und abgeleiteter Klasse, wie bei der Spezialisierung durch `public`-Vererbung.

Eine typische Anwendung ist der **generische Pointer-Stack**. Man kann ihn generisch halten, indem man **`void*`**-Pointer verwendet. Hier wird dafür gesorgt, dass es gemeinsamen Code für verschiedene Typen von Zeigern gibt → siehe Templates. Somit wird eine Code-Aufblähung (template-induced-code-bloat) vermieden:

```
class MyGenericPtrStack
{
	public:
		void initialize() {	m_pTop = NULL; }
		void push(void* p)
		{
			m_pNode = new Node(p,m_pTop);
			m_pTop = m_pNode;
		}
		void* pop()
		{
			void* ret = NULL;
			if(m_pTop)
			{
				ret = m_pTop->m_p;
				m_pNode = m_pTop->m_pPrev;
				delete m_pTop;
				m_pTop = m_pNode;
			}
			return ret;
		}
		void clear()
		{
			while(m_pTop)
			{
				m_pNode = m_pTop->m_pPrev;
				delete m_pTop;
				m_pTop = m_pNode;
			}
		}
	private:
		struct Node
		{
			void* m_p;
			Node* m_pPrev;
			Node(void* p,Node* pPrev) : m_p(p),m_pPrev(pPrev) {}
		};
		Node* m_pTop;
		Node* m_pNode;
};
```

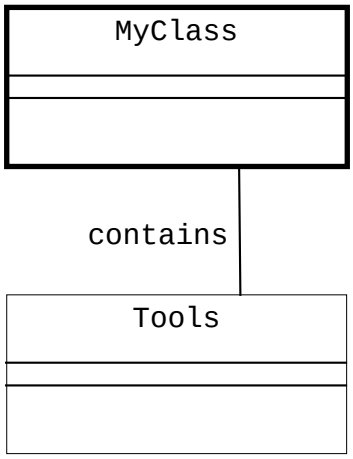
```
template<class T>
class MyPtrStack : private MyGenericPtrStack
{
	public:
		MyPtrStack() { initialize(); }
		~MyPtrStack() { clear(); }
		void Push(T pObj) { push(static_cast<void*>(pObj)); }
		T Pop() { return static_cast<T>(pop()); }
		void Clear() { clear(); }
};

int main()
{
	char* szText1 = "Hello";
	char* szText2 = "World";

	MyPtrStack<char*> PtrStack;

	PtrStack.Push(szText1);
	PtrStack.Push(szText2);

	char* p = NULL;
	p = PtrStack.Pop();
	printf("Text2 = %s\n",p);
	p = PtrStack.Pop();
	printf("Text1 = %s\n",p);

	return 0;
}
```

Allgemein gilt: Dort, wo auf **gemeinsamen Code** zugegriffen werden soll (Code-Sharing), ist die **private**-Vererbung einzusetzen.

Weiteres Beispiel:

```
class Tools
{
	public:
		void SetMode(int nMode = 0) { m_nMode = nMode; }
		void Print(const char* const szStr)
		{
			if(m_nMode)
				printf("\n*******\n%s\n*******\n",szStr);
			else
				printf("\n~~~~~~~\n%s\n~~~~~~~\n",szStr);
		}
	private:
		int m_nMode;
};
```

```
class MyClass : private Tools
{
     public:
          MyClass() { SetMode(1); }
          void PrintIt(const char* const szStr) { Print(szStr); }
};

int main()
{
     MyClass Obj;
     Obj.PrintIt("Hello");
     return 0;
}
```

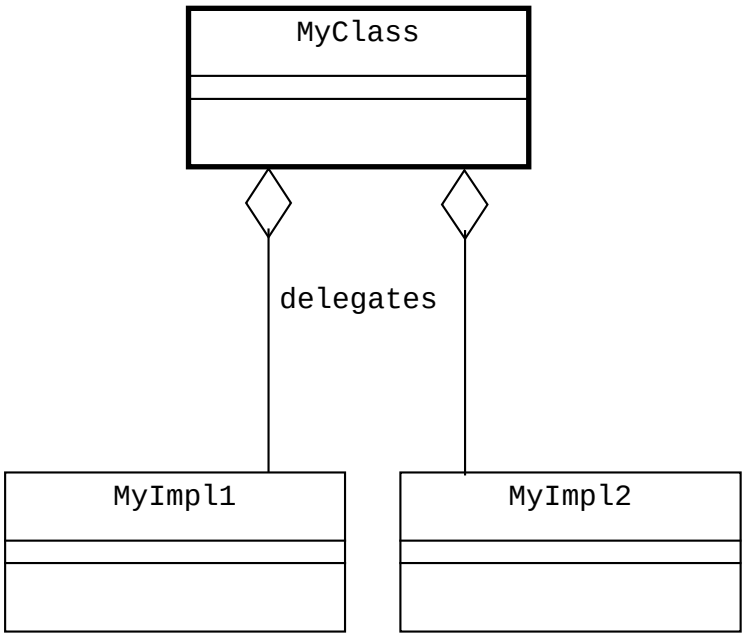


15.4 Composition statt multiple inheritance

Wenn man die Eigenschaften verschiedener Klassen in einem Objekt vereinen will, dann sollte man diese Klassen nicht als Basisklassen einer Mehrfach-Erbung (multiple inheritance) ansetzen, denn Mehrfach-Erbung ist allein wegen der Mehrdeutigkeit der Namen schon kritisch. Statt dessen sollte man in die Klasse mit den vereinten Eigenschaften **Zeiger auf die Implementierungs-Klassen** deklarieren. Dieses Konzept nennt man **Composition** (Zusammenfügung).

Beispiel:

```
class MyImpl1;
class MyImpl2;

class MyClass
{
     public:
          MyClass() : m_pImpl1(NULL),m_pImpl2(NULL) {}
          void Do();
          void Verify();
     private:
          MyImpl1* m_pImpl1;
          MyImpl2* m_pImpl2;
};

class MyImpl1
{
     public:
          MyImpl1() {}
          void Do() { printf("MyImpl1::Do() ist done!\n"); }
};

class MyImpl2
{
     public:
          MyImpl2() {}
          void Verify() { printf("MyImpl2::Verify() ist done!\n");
}
};

void MyClass::Do() { m_pImpl1->Do(); }
void MyClass::Verify() { m_pImpl2->Verify(); }

int main()
{
     MyClass Obj;
     Obj.Do();
     Obj.Verify();
     return 0;
}
```



15.5 Schnittstellen (AbstractMixinBaseClass) public dazuerben

Möchte man neben einer Klasse mit Implementierung noch weitere Klassen erben, dann ist dies eine relativ ungefährliche Anwendung, wenn es sich bei den weiteren Klassen um **abstrakte Basisklassen mit ausschließlich rein virtuellen public-Methoden** (= Schnittstellen-Klassen = **Abstract Mixin Bases Classes**) handelt.

Beispiel für eine Schnittstellen-Klasse (Abstract Mixin Base Class):

→ Abstrakte Basisklasse mit ausschließlich rein virtuellen public-Methoden:

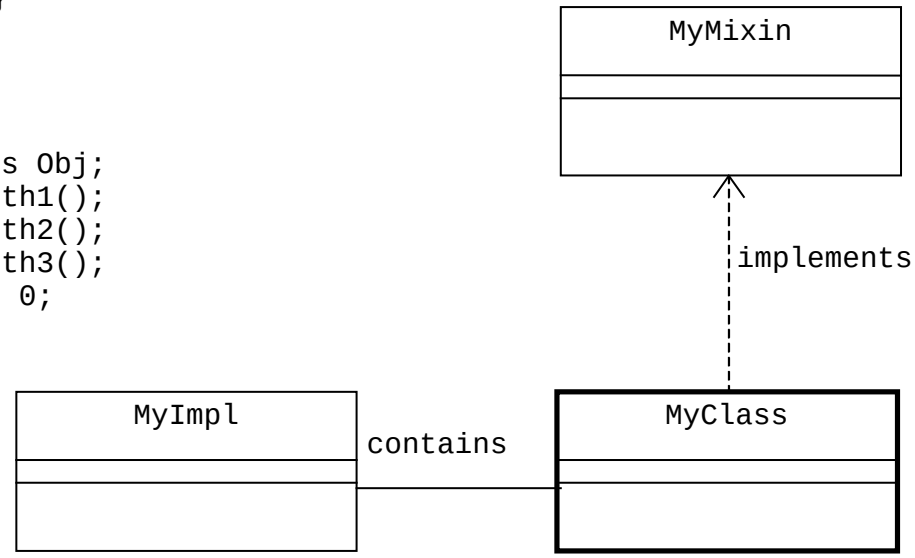
```
class MyMixin
{
     public:
          virtual int Meth1() = 0;
          virtual int Meth2() = 0;
          virtual int Meth3() = 0;
};
```

Beachte:

Es gibt hierin **keine** Member-Variablen!

In der abgeleiteten Klasse werden die rein virtuellen Schnittstellenfunktionen implementiert, indem die Methoden der Implementierung aufgerufen werden, die bspw. von einer bestehenden Implementierung mit private geerbt wurden:

```
class MyMixin
{
      public:
            virtual int Meth1() = 0;
            virtual int Meth2() = 0;
            virtual int Meth3() = 0;
};

class MyImpl
{
      public:
            int Func1()
            {
                  printf("Func1()\n");
                  return 1;
            }
            int Func2()
            {
                  printf("Func2()\n");
                  return 1;
            }
};

class MyClass : public MyMixin, private MyImpl
{
      public:
            ~MyClass() {}
            int Meth1() { return Func1(); }
            int Meth2() { return Func2(); }
            int Meth3() { return Func3(); }
      private:
            int Func3()
            {
                  printf("Func3()\n");
                  return 1;
            }
};

int main()
{
      MyClass Obj;
      Obj.Meth1();
      Obj.Meth2();
      Obj.Meth3();
      return 0;
}
```



15.6 Abstrakte Basisklasse vs. Template

Es gibt folgende klare Regeln, wann eine abstrakte Basisklasse und wann ein Template als Wurzel in der Vererbungshierarchie einzusetzen ist:

Abstrakte Basisklasse,

wenn unterschiedliche Objekt-Typen unterschiedliches Verhalten aufweisen!

Template,

wenn unterschiedliche Objekt-Typen dasselbe Verhalten aufweisen!

Beispiel für eine Template-Anwendung: Stack

Beim Stack ist es egal, von welchem Typ die Objekte sind, die als Stack organisiert werden.

```
template<class T>
class Stack
{
	public:
		Stack();
		~Stack();
		void Push(const T& Obj);
		T Pop();
		void Clear();
	private:
		struct Node
		{
			T		m_Data;
			Node*		m_pPrev;
			Node(const T& Data,Node* pPrev) :
				m_Data(Data),m_pPrev(pPrev) {}
		};
		Node* m_pTop;
		Node* m_pNode;
};
```

Beispiel für eine abstrakte Basisklasse: DataInterface

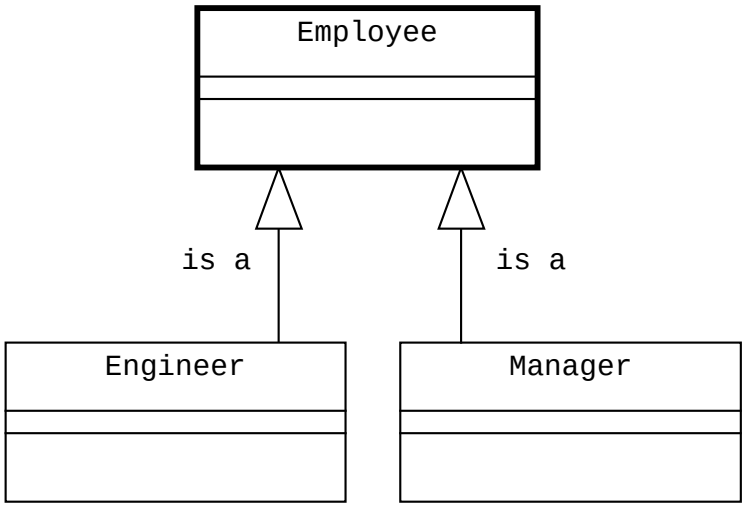
Beim Dateninterface ist der Empfang über verschiedene Schnittstellen technisch unterschiedlich realisiert.

```
class DataInterface
{
	public:
		virtual ~DataInterface() {} //Hack: nicht wirklich inline
		virtual void Init() = 0;
		virtual char GetChar() = 0;
		virtual void SendChar(char c) = 0;
};
```

```
class PROFIBUS : public DataInterface
{
      public:
            PROFIBUS();
            ~PROFIBUS();
            void Init();
            char GetChar();
            void SendChar(char c);
};

class RS232 : public DataInterface
{
      public:
            RS232();
            ~RS232();
            void Init();
            char GetChar();
            void SendChar(char c);
};
```

15.7 Verknüpfung konkreter Klassen: abstrakte Basisklasse

Wenn man in der Klassen-Hierarchie mehrere konkrete Klassen miteinander verknüpfen will, dann tut man dies am besten über eine **abstrakte** Basisklasse. So kann man problemlos auch mehrere Verknüpfungen herstellen, denn man kann im Prinzip von so vielen abstrakten Basisklassen erben, wie man will.



Um die abstrakte Basisklasse spezifizieren zu können, muss man eine Generalisierung der konkreten Klassen vornehmen. Die Basis enthält dann die gemeinsame verallgemeinerte Funktionalität.

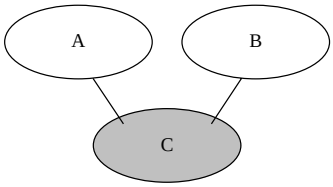
15.8 Erben aus mehreren Basisklassen vermeiden

Die folgenden Konzepte finden bei multiple inheritance Anwendung und sprechen dagegen, das Erben aus mehreren Basisklassen vorzusehen:

15.8.1 Expliziter Zugriff (oder using)

Wenn Methoden den gleichen Namen haben, dann muss explizit (oder mittels `using`) darauf zugegriffen werden.

Beispiel:



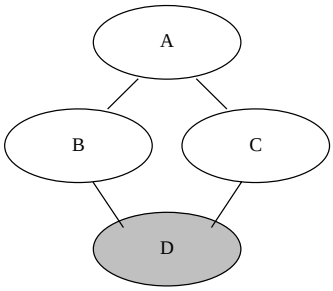
Wenn A und B die Methode Func() deklarieren/implementieren, dann kann C nur folgendermaßen eindeutig auf eine davon zugreifen:

```
A::Func()
B::Func()
```

Dies ist **unhandlich** und das **Konzept der virtuellen Funktionen**, die von der Basisklasse zur abgeleiteten Klasse weitergeleitet werden, **wird lahmgelegt**.

15.8.2 Virtuelle Vererbung (Diamant-Struktur)

In einer **Diamant-Struktur** gibt es bei normaler Vererbung mehrere Wege für den Funktionsaufruf, seinen Code zu finden (`vftable`). Deshalb meldet der Compiler einen Fehler (`ambigious`):



Abhilfe schafft die virtuelle Vererbung:

Beispiel:

```
class A
{
	public:
		explicit A(int nID = 0) : m_nID(nID) {}
		~A() {}
		void Calculate();
		void SetID(int nID) { m_nID = nID; }
		int GetID() { return m_nID; }
	private:
		int m_nID;
};
class B : virtual public A
{
	public:
		explicit B(int nID = 0) { SetID(nID); }
		~B() {}
		void Calculate() { printf("B::Calculate()\n"); }
		void AlgorithmB() { printf("AlgorithmB()\n"); }
};
class C : virtual public A
{
	public:
		explicit C(int nID = 0) { SetID(nID); }
		virtual ~C() {}
		void AlgorithmC() { printf("AlgorithmC()\n"); }
};
class D : public B, public C
{
	public:
		explicit D(int nID = 0) { SetID(nID); }
};
int main()
{
	D Obj(8);
	int i = Obj.GetID();
	Obj.AlgorithmB();
	Obj.AlgorithmC();
	Obj.Calculate();
	return 0;
}
```

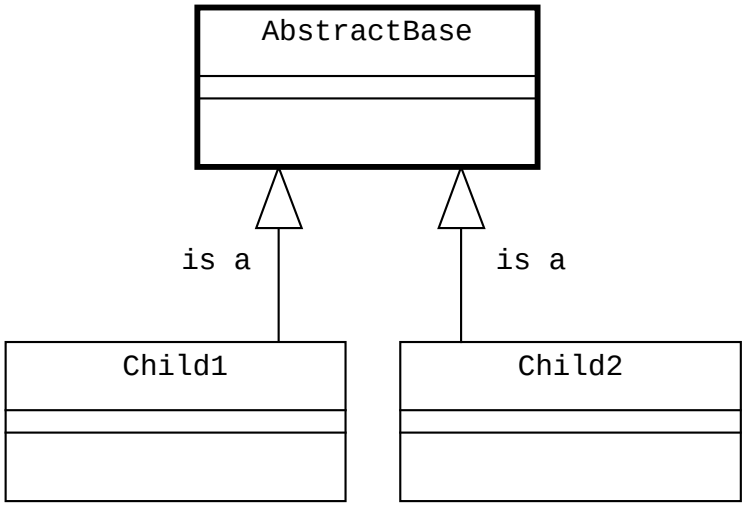
Probleme, die hierbei auftreten:

- Beim Entwurf von A, B und C kann man noch nicht wissen, dass später mal eine Diamant-Struktur entstehen wird, d.h. dass für das Erben von A später mal **virtuelle** Vererbung erforderlich wird. Deshalb wird in der Regel beim Entwurf von **A** nicht berücksichtigt, dass A **keine virtuellen Funktionen** enthalten darf. Außerdem wird B und C in der Regel nicht virtuell von A abgeleitet vorliegen.
- **Eindeutigkeit** der Methoden der virtuellen Basisklasse A ist nur gegeben, wenn B nur Methoden überschreibt, die C nicht überschreibt und umgekehrt. Wenn also Calculate() durch B und durch C überschrieben wird, dann meldet der Compiler einen Fehler (ambigious).

15.9 Zuweisungen nur zwischen gleichen Child-Typen zulassen

Damit der Inhalt eines Childs (Spezialisierung einer Basisklasse über "is-a"-Vererbung) nur dann einem anderen Child zugewiesen wird, wenn beide vom gleichen Typ sind, muss jede Child-Klasse einen eigenen Zuweisungsoperator definieren und die gemeinsame Basisklasse muss diesen Operator verstecken.

Beispiel:




```
class AbstractBase
{
      public:
            virtual ~AbstractBase() {} //Hack: nicht wirklich inline
            virtual void Do() = 0;
            void SetID(int nID) { m_nID = nID; }
            int GetID() const { return m_nID; }
      private:
            //Versteckter Zuweisungsoperator:
            AbstractBase& operator=(const AbstractBase& Obj);

            int m_nID;
};

class Child1 : public AbstractBase
{
      public:
            Child1(int nID = 1) { SetID(nID); }
            ~Child1() {}
            void Do() { printf("Child1::Do() / ID = %d\n",GetID()); }
            Child1& operator=(const Child1& Obj)
            {
                  if(this == &Obj)
                        return *this;
                  SetID(Obj.GetID());
                  return *this;
            }
};
```

```
class Child2 : public AbstractBase
{
     public:
          Child2(int nID = 2) { SetID(nID); }
          ~Child2() {}
          void Do() { printf("Child2::Do() / ID = %d\n",GetID()); }
          Child2& operator=(const Child2& Obj)
          {
               if(this == &Obj)
                    return *this;
               SetID(Obj.GetID());
               return *this;
          }
};

int main()
{
     Child1 Obj1_1(11);
     Child1 Obj1_2(12);
     Obj1_1.Do();
     Obj1_2.Do();

     Obj1_2 = Obj1_1;
     Obj1_2.Do();

     Child2 Obj2_1(21);
     Obj2_1 = Obj1_1; //-> Compiler-Fehler
     Obj2_1.Do();
     return 0;
}
```

16. Nutzer einer Klasse von Änderungen entkoppeln

16.1 Allgemeines

Wenn sich an der **Anzahl** oder der **Aufruf-Konvention** der **`private`-Methoden** (Funktionen und Sub-Funktionen der internen Implementierung) etwas ändert, dann bekommt der Benutzer der **`public`**-Methoden das gut zu spüren, obwohl er die `private`-Methoden gar nicht aufrufen kann. Der Grund hierfür ist, dass die durch `#include` eingebundene Header-Datei `*.h` sich ändert. Dies zieht also nach sich, dass **alle `*.cpp`-Module, die diese Header-Datei einbinden, neu kompiliert werden müssen**. Da dies ein sehr unschöner Effekt ist, hat man verschiedene Verfahren und Regeln entwickelt, die so etwas vermeiden.

16.2 Header-Dateien: Forward-Deklaration statt #include

In Header-Dateien (Klassendeklarationen) hat man oftmals die Möglichkeit `#include` zu vermeiden. Der Hintergrund ist, dass man eine fremde Klassendeklaration (also `#include`) nur dann benötigt, wenn man Speicher für ein fremdes Objekt allokieren muß, d.h. wenn man ein Objekt einer fremden Klasse als Member-Variable benutzt oder Methoden der Klasse über Zeiger oder Referenzen aufruft. Ist dies in der Klassendeklaration nicht der Fall, hat man also bestenfalls **Zeiger** oder **Referenzen** auf Objekte anderer Klassen definiert (ohne damit auf deren Methoden zuzugreifen), dann muss man lediglich den Namen der Klasse bekannt machen (Forward-Deklaration).

In Header-Dateien sollte man also versuchen `#include` durch die Forward-Deklaration zu ersetzen:

- Nicht auf Referenz-Parameter zugreifen (Code in die Implementierung verlegen)
- Keine Allokierung von Heap-Speicher (new) für Zeiger
  (Code in die Implementierung verlegen)
- Komplexe Implementierungen per Zeiger aufnehmen

Beispiel:

```
class MyMember; //Forward-Deklaration
class MyImpl;   //Forward-Deklaration

class MyClass
{
     public:
          MyClass() : m_pImpl(NULL) {}
          void Do(const MyMember& Obj); //hier kein Zugriff auf Obj
     private:
          MyImpl* m_pImpl; //Zugriff auf Implementierung per Zeiger
};
```

16.3 Delegation bzw. Aggregation

Ein Konzept, welches zwar den Aufbau der Software maßgeblich ändert, aber die `#include`-Anweisung vermeidet und damit den Nutzer einer Klasse von Änderungen der `private`-Methoden (also der internen Implementierung) entkoppelt, ist die **Implementierung per Letter-Klasse, auf die es dann einen Zeiger in der eigentlichen Klasse gibt**. Die eigentliche Klasse (Envelope, Umschlag) hält also lediglich einen **Zeiger auf die Implementierung** (Letter, Brief) und beherbergt damit nicht die internen Implementierungs-Methoden hinter `private`.

Beispiel:

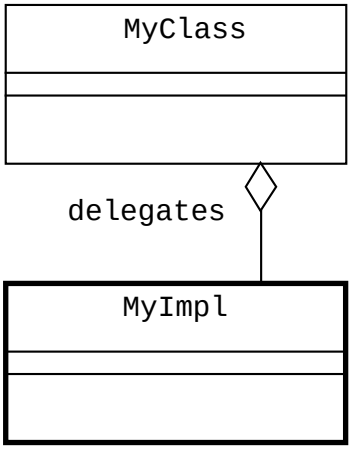
```
//*****************************************************************
// MyClass.h:
//*****************************************************************

class MyImpl;   //Forward-Deklaration der Letter-Klasse

class MyClass //Envelope-Klasse für MyImpl
{
     public:
          MyClass() : m_pImpl(NULL) {}
          void Meth1();
          void Meth2();
     private:
          MyImpl* m_pImpl; //Zugriff auf Implementierung per Zeiger
};
```

Die **Envelope-Klasse** implementiert eine **Delegation** an die Implementierung, also die Letter-Klasse. Man kann auch sagen, sie **aggregiert** Letter-Klassen-Aufrufe:

```
//*****************************************************************
// MyClass.cpp:
//*****************************************************************

#include "MyClass.h"       //Envelope-Klasse
#include "MyImpl.h"        //Letter-Klasse

void MyClass::Meth1() { pImpl->Meth1(); } //Delegation (Aggregation)
void MyClass::Meth2() { pImpl->Meth2(); } //Delegation (Aggregation)
```

Die **Letter-Klasse** hingegen implementiert die Funktionalität und beinhaltet die gesamten Methoden (interne Funktionen und Sub-Funktionen), die dazu notwendig sind:

```
//*******************************************************************
// MyImpl.h:
//*******************************************************************

class MyImpl
{
      public:
            void Meth1();
            void Meth2();
      private: //interne Implementierung
            int Calculate1(int n);
            int Calculate2(int n);
            void Print(int n);
};

//*******************************************************************
// MyImpl.cpp:
//*******************************************************************

#include "MyImpl.h"
void MyImpl::Meth1()
{
      int i = Calculate1();
      Print(i);
}
void MyImpl::Meth2()
{
      int i = Calculate2();
      Print(i);
}
int MyImpl::Calculate1(int n)
{
      return n + 1;
}
int MyImpl::Calculate2(int n)
{
      return n + 2;
}
void MyImpl::Print(int n)
{
      printf("The caculated number is: %d\n",n);
}
```



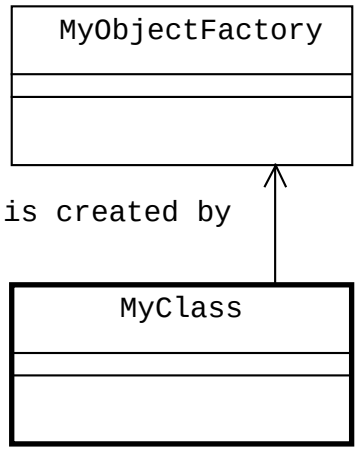
16.4 Objekt-Factory-Klasse und Protokoll-Klasse

Object-Factory-Klassen sind ein weiteres Konzept, um den Benutzer einer Klasse unabhängig von Änderungen der privaten Implementierung zu machen. Der Benutzer kennt nur die **virtuelle Schnittstelle** der **Object-Factory-Klasse** und kann sich zur Laufzeit sozusagen eine Implementierung besorgen, indem er sich über die Schnittstelle ein **Objekt der Protokoll-Klasse erzeugen** lässt. Hierzu ist eine **static**-Methode in der Object-Factory implementiert.

Beispiel:

```
//*******************************************************************
// MyObjFactory.h:
//*******************************************************************

class MyClass; //Forward-Deklaration

class MyObjFactory
{
     public:    //Interface
          MyObjFactory() {}
          static MyClass* CreateObj(int nID = 0);
          virtual void Meth1() = 0; //virtuelle Schnittstelle
          virtual void Meth2() = 0;
};

//*******************************************************************
// MyObjFactory.cpp:
//*******************************************************************

#include "MyObjFactory.h"  //Objekt-Factory-Klasse
#include "MyClass.h"       //Protokoll-Klasse

MyClass* MyObjFactory::CreateObj(int nID)
{
     return new MyClass(nID);
}
```

```
//*******************************************************************
// MyClass.h:
//*******************************************************************

#include "MyObjFactory.h"

class MyClass : public MyObjFactory   //Protokoll-Klasse ("is-a")
{
     public:
          MyClass(int nID = 0) : m_nID(nID) {}
          void Meth1() { Print("MyClass::Meth1()"); }
          void Meth2() { Print("MyClass::Meth2()"); }
     private:
          void Print(const char* szStr)
          {
               printf("*** %s ***\n",szStr);
          }
          int m_nID;
};

//*******************************************************************
// Main.cpp:
//*******************************************************************

int main()
{
     MyObjFactory* pObj = MyClass::CreateObj(1);
     pObj->Meth1();
     pObj->Meth2();
     delete pObj;
     return 0;
}
```



17. Code kapseln

17.1 Beliebig viele Kopien erlaubt: Funktions-Obj. (operator())

Man kann Algorithmen innerhalb von sogenannten Funktions-Objekten kapseln. Der **Function-Call-Operator** `()` wird dann überschrieben, so dass sich der Algorithmus wie eine normale Funktion aufrufen lässt. Immer wenn der Code gebraucht wird, instanziiert man ein Objekt der Klasse und ruft es dann wie eine Funktion auf.

Beispiel:

```
struct Multiply //struct = class in der alles public ist
{
     int operator() (int x,int y) const { return (x*y); }
};

int main()
{
     Multiply mul;
     int z = mul(2,3);
     return 0;
}
```

17.2 Nur 1 Kopie erlaubt: Statische Obj. (MyClass::Method())

Man versteckt **Konstruktor**, **Copy-Konstruktor** und **Destruktor** hinter `protected` und verwendet ausschließlich `static`-**Methoden**.

Beispiel:

```
class MyClass
{
     public:
          static void Method() { printf("Method()\n"); }
     protected:
          MyClass();
          MyClass(const MyClass& Obj);
          ~MyClass();
};

int main()
{
     MyClass::Method();
     return 0;
}
```

18. Operatoren

18.1 Definition von Operatoren

Es kann sinnvoll sein, die Operanden mit Präfixen zu kennzeichnen:

lhs für links (left hand side)
rhs für rechts (right hand side)

Beispiel:

```
//Innerhalb einer Klasse:
class MyClass
{
	public:
		MyClass(int nID = 0) : m_nID(nID) //Typumwandlung von int
		{
		}
		MyClass(const MyClass& Obj) : m_nID(Obj.GetID()) {}
		int GetID() const { return m_nID; }
		MyClass& operator=(const MyClass& rhsObj) //Zuweisung
		{
			if(this == &rhsObj)
				return *this;
			m_nID = rhsObj.GetID();
			return *this;
		}
		bool operator==(const MyClass& rhsObj) //Vergleich
		{
			return (rhsObj.GetID() == m_nID);
		}
		operator int() const //Typumwandlung nach int
		{
			return m_nID;
		}
	private:
		int m_nID;
};


//Global:
const MyClass operator-(const MyClass& lhsObj,const MyClass& rhsObj)
{
	return MyClass(lhsObj.GetID() - rhsObj.GetID());
}
```

```
int main()
{
     MyClass Obj1;         //Default: m_nID = 0
     Obj1 = 4;             //'Typumwandlung von int' & 'Zuweisung'

     MyClass Obj2(50);

     MyClass Obj3(Obj2-Obj1); //'Operator -' & 'Typumwandl. von int'
     int i = Obj3;            //'Typumwandlung nach int'

     MyClass Obj4(123);
     if(Obj3 == Obj4)         //'Vergleich'
          return 1;

     return 0;
}
```

18.2 Binäre Operatoren effektiv implementieren

- **Innerhalb des binären Operators** sollte man (falls möglich) **unäre Operatoren verwenden**, denn dann reduziert sich die Software-Pflege auf die unären Operatoren.
- Am besten benutzt man ein **Function-Template** für einen **binären** Operator, denn dies hat den Vorteil, dass **automatisch für alle Typen eine implizite Instanziierung** stattfindet, sobald der Operator im Code benutzt wird.
- Als **return-Wert** benutzt man am besten einen **Copy-Konstruktor** (statt eines Objektes), da dann nicht implizit und versteckt ein temporäres Objekt für die Rückgabe erzeugt wird.

Beispiel:

```
     //Unärer Operator in der Klasse implementiert:
     class MyClass
     {
          public:
               MyClass(int nID = 0) : m_nID(nID) {}
               MyClass(const MyClass& Obj) : m_nID(Obj.GetID()) {}
               int GetID() const { return m_nID; }
               MyClass& operator+=(const MyClass& Obj)
               {
                    m_nID += Obj.GetID();
                    return *this;
               }
               MyClass& operator=(const MyClass& rhsObj)
               {
                    if(this == &rhsObj)
                         return *this;
                    m_nID = rhsObj.GetID();
                    return *this;
               }
          private:
               int m_nID;
     };
```

```
//Binärer Operator als globales Function-Template:
template<class T>
const T operator+(const T& lhsObj,const T& rhsObj)
{
	return T(lhsObj) += rhsObj; //return von Konstruktor
}

int main()
{
	MyClass Obj1(8);
	MyClass Obj2(5);
	MyClass Obj3 = Obj1 + Obj2;
	return 0;
}
```

18.3 Unäre Operatoren bevorzugt verwenden

Man sollte die unären Operatoren immer den binären vorziehen, da sie effektiver als die binären sind.

Beispiel:

Statt:

```
	Obj = Obj + 4;
```

Besser:

```
	Obj += 4;
```

18.4 Kommutativität: Globale bin. Operatoren implementieren

Problem mit unärem Operator:

```
class Multiply
{
	public:
		Multiply(int nNumerator = 0,int nDenominator = 1)
		: m_nNumerator(nNumerator),m_nDenominator(nDenominator)
		{
		}
		const Multiply operator*(const Multiply& Obj) const
		{
			return Multiply(
				m_nNumerator * Obj.GetNumerator(),
				m_nDenominator * Obj.GetDenominator());
		}
		int GetNumerator() const { return m_nNumerator; }
		int GetDenominator() const { return m_nDenominator; }
	private:
		int m_nNumerator;
		int m_nDenominator;
};
```

```
int main()
{
      Multiply OneEigth(1,8);
      Multiply OneHalf(1,2);
      Multiply Result(0);
      Result = OneHalf * OneEigth;     //-> ok
      Result = Result * OneEigth;      //-> ok
      Result = OneHalf * 2; //-> ok, Typumwandlung über Multiply(2)
      Result = 2 * OneHalf; //-> Compiler-Fehler!!!
      return 0;
}
```

Ursache:
Da in C++ `int` kein Objekt einer Klasse ist, sondern ein System-Datentyp kann **kein unärer Operator zu der 2** gefunden werden. Weiterhin gibt es **keinen globalen binären Operator, der Multiply als Argument akzeptiert**.

Abhilfe:
Möchte man nun **trotzdem Kommutativität** erreichen, dann muss man den Operator als **binären globalen Operator** definieren und dann noch die implizite Typumwandlung des Compilers nutzen (`2` → `Multiply(2)`). Der **unäre Operator muss allerdings weg**, da es sonst das Problem gibt, dass der Compiler mehrere Möglichkeiten zur Auswertung des Ausdrucks

```
Result = OneHalf * 2;
```

findet (`ambigious`).

```
class Multiply
{
      public:
            Multiply(int nNumerator = 0,int nDenominator = 1)
            : m_nNumerator(nNumerator),m_nDenominator(nDenominator)
            {
            }
            int GetNumerator() const { return m_nNumerator; }
            int GetDenominator() const { return m_nDenominator; }
      private:
            int m_nNumerator;
            int m_nDenominator;
};
const Multiply operator*(
                  const Multiply& lhsObj,const Multiply& rhsObj)
{
      return Multiply(
                  lhsObj.GetNumerator()*rhsObj.GetNumerator(),
                  lhsObj.GetDenominator()*rhsObj.GetDenominator());
}
int main()
{
      Multiply OneEigth(1,8);
      Multiply OneHalf(1,2);
      Multiply Result(0);
      Result = OneHalf * 2; //-> ok
      Result = 2 * OneHalf; //-> ok
      return 0;
}
```

Man sollte immer einen `const`-Wert zurückliefern, damit Folgendes **nicht** möglich ist:

```
d = (a * b) = c;
```

d.h. das Ergebnis kann nicht verändert werden, ohne es zuerst in eine nicht-`const`-Variable zu speichern!

18.5 Operator-Vorrang (Precedence)

Die sicherste Methode, die Reihenfolge der Auswertung eines Ausdrucks zu definieren, ist die Klammerung. Benutzt man keine Klammerung, dann gilt folgende Vorrangigkeit:

Bei folgenden Operatoren wird **der links stehende Operator zuerst** ausgewertet:

```
1.    x++
2.    x--
3.    function( )
4.    array[ ]
5.    x->y
6.    x.y
```

Bei folgenden Operatoren wird **der rechts stehende Operator zuerst** ausgewertet:

```
7.    ++x
8.    --x
9.    !x
10.   ~x
11.   -x
12.   +x
13.   &x
14.   *x
15.   sizeof x
16.   new x
17.   delete x
18.   (type) x
```

Bei folgenden Operatoren wird **der links stehende Operator zuerst** ausgewertet:

```
19.   x.y*
20.   x->y*
21.   x * y
22.   x / y
23.   x % y
24.   x + y
25.   x - y
26.   x << y
27.   x >> y
28.   x < y
29.   x <= y
30.   x > y
31.   x >= y
32.   x == y
33.   x != y
34.   x & y
```

```
35.   x ^ y
36.   x | y
37.   x && y
38.   x || y
39.   x ? y : z
```

Bei folgenden Operatoren wird **der <u>rechts</u> stehende Operator zuerst** ausgewertet:

```
40.   x = y
41.   x *= y
42.   x /= y
43.   x %= y
44.   x += y
45.   x -= y
46.   x <<= y
47.   x >>= y
48.   x &= y
49.   x ^= y
50.   x |= y
```

Bei folgenden Operatoren wird **der <u>links</u> stehende Operator zuerst** ausgewertet:

```
51.   x , y
```

18.6 Präfix- und Postfix-Operator

18.6.1 Allgemeines

Problem:

Präfix oder Postfix kann man nicht wie bei überladenen Funktionen (innerhalb einer Klasse) anhand ihrer Argumente unterscheiden.

Lösung:

C++ definiert Folgendes:

Präfix → kein Argument
Postfix→ `int`-Argument (der Compiler übergibt immer eine 0)

Beispiel mit dem Operator ++:

```
class MyClass
{
	public:
		MyClass(int nID = 0) : m_nID(nID) {}
		MyClass& operator++()		//-> Präfix (++Obj)
		{
			++m_nID;
			return *this; //Rückgabe einer Referenz auf *this
		}
		const MyClass operator++(int)	//-> Postfix (Obj++)
		{
			MyClass Obj(m_nID); //temporäres Objekt
			++m_nID;
			return Obj; //Rückgabe eines Objektes per Wert
		}
	private:
		int m_nID;
};

int main()
{
	MyClass Obj;
	Obj++;	//-> Obj.operator++();
	++Obj;	//-> Obj.operator++(0);
	return 0;
}
```

Der **Postfix**-Operator muss einen **Wert** zurückgeben (keine Referenz auf `*this`), da er (wie man sieht) das Objekt für die Rückgabe nur lokal und temporär erzeugen kann. Er sollte aber immer einen **`const`-Wert zurückgeben**, damit folgendes **falsche Verhalten** verhindert wird:

```
/* const */ MyClass MyClass::operator++(int)	//-> Postfix (Obj++)
{
	MyClass Obj(m_nID); //temporäres Objekt
	++m_nID;
	return Obj; //Rückgabe eines Objektes per Wert
}

	→	Obj++++;	würde zu	Obj.operator++(0).operator++(0);
```

Dies würde bedeuten, dass **auf dem temporären Objekt der ersten Operation** nochmal ++ aufgerufen würde. Dies veränderte aber nicht den Wert des eigentlichen Objektes und bliebe daher ohne Auswirkung auf die beteiligten Objekte, und es wäre:

```
	Obj2 = Obj1++++;	identisch mit	Obj2 = Obj1++;
```

was nur schwer einleuchtend ist.

*18.6.2 Wartungsfreundlichkeit erhöhen durch ++(*this) im Postfix-Operator*

Wenn man den Postfix-Operator `++` mit der Anweisung **`++(*this)`** implementiert, also den Präfix-Operator darin aufruft, dann bleibt der **Postfix-Operator für alle Zeiten wartungsfrei**. Dies gilt mutatis mutandis auch für den Operator `--`.

```
MyClass& MyClass::operator++()              //-> Präfix (++Obj)
{
      ++m_nID;
      return *this; //Rückgabe einer Referenz auf *this
}
const MyClass MyClass::operator++(int)      //-> Postfix (Obj++)
{
      MyClass Obj(m_nID); //temporäres Objekt
      ++(*this);
      return Obj; //Rückgabe eines Objektes per Wert
}
```

18.6.3 Präfix(++Obj) ist Postfix(Obj++) vorzuziehen

Der Präfix-Operator ist effizienter, da er kein temporäres Objekt erzeugen muss, wie man leicht am Beispiel des Operators `++` sieht:

```
MyClass& MyClass::operator++()              //-> Präfix (++Obj)
{
      ++m_nID;
      return *this; //Rückgabe einer Referenz auf *this
}
const MyClass MyClass::operator++(int)      //-> Postfix (Obj++)
{
      MyClass Obj(m_nID); //temporäres Objekt
      ++(*this);
      return Obj; //Rückgabe eines Objektes per Wert
}
```

18.7 Der Komma-Operator ,

Der Komma-Operator `,` wird in folgenden Fällen angewendet:

- **Definition von Variablen**

  Beispiel:

  ```
  long a = 3,b = 5,c = 8;
  ```

- **Im Kopf der `for`-Schleife**

  Beispiel:

  ```
  for(long x = 0,y = 10;x < y;++x,--y)
  {
        ...
  }
  ```

  ACHTUNG!
  Die Verwendung des Komma-Operators im **Schleifentest** wird leicht mit einer &&-Operation (AND) verwechselt, ist aber keine:

  Beispiel:

  ```
  for(long x = 0,y = 10;x < xMax,y < yMax;++x,++y)
  {
        ...
  }
  ```

  $\neq$

  ```
  for(long x = 0,y = 10;(x < xMax) && (y < yMax);++x,++y)
  {
        ...
  }
  ```

  In Wirklichkeit is das Verhalten nämlich so: Der Gesamtausdruck (Schleifentest) wird **von rechts nach links** ausgewertet. Sobald ein Ausdruck "TRUE" liefert, läuft die Schleife weiter, wie das Beispiel unten zeigt.

Beispiel:

```
#include <stdio.h>
int main()
{
      static const long xMax = 2;
      static const long yMax = 14;
      for(long x = 0,y = 10;x < 2,y < 14;++x,++y)
      {
      }
      printf("x = %ld,y = %ld\r\n",x,y);
      if(x == xMax)
            printf("x == xMax\r\n");
      else if(x > xMax)
            printf("x > xMax!!!\r\n");
      if(y == yMax)
            printf("y == yMax\r\n");
      else if(y > yMax)
            printf("y > yMax!!!\r\n");
      return 0;
}
```

Man wird sehen, dass die Schleife abbricht mit:

```
x = 4,y = 14
x > xMax!!!
y == yMax
```

19. Datentypen und Casting

19.1 Datentypen

- Arithmetische Datentypen (built-in)

  1.) Abzählbare Werte (Integrale Datentypen)
  `bool, char, int, short, long`
  2.) Gleitkommazahlen
  `float, double`

  Zum Typ **bool**:

  $$\texttt{bool b =} \begin{cases} \textit{false} \text{ falls } 0 \\ \textit{true} \text{ sonst} \end{cases}$$

  `bool` ist wegen seiner Typsicherheit unbedingt `BOOL` vorzuziehen!

  `BOOL` ist nur ein `typedef` für `unsigned short` und kein eingebauter Datentyp:

  ```
  typedef unsigned short BOOL;
  #define FALSE   0
  #define TRUE    1
  ```

- Benutzerdefinierte Datentypen

  1.) Aufzählungen
  `enum...`
  2.) Strukturen
  `struct...`
  3.) Klassen
  `class...`

  Beispiel zum Typ **`enum`**:

  ```
  enum enumDay{MO,DI,MI,DO,FR,SA,SO};
  enumDay eToday;
  enumDay eYesterday;

  if(eYesterday == MO)
        eToday = DI;
  ```

Zum Thema Datentypen gehört natürlich auch das Thema **casts** (also Umformen):

Alt: `x = (MyClass) y;` `//statischer cast`

Neu: `x = ..._cast<MyClass>(y);`

wobei **`..._cast`** Folgendes sein kann:

`static_cast`:	Statischer cast
`dynamic_cast`:	Sicherer cast von **Zeigern oder Referenzen**
`const_cast`:	Konstantheit weg-casten
`reinterpret_cast`:	Wilder cast zwischen **Zeigern** auf verschiedenste Objekte

19.2 Polymorphismus: vfptr und vftable

Jedes Objekt einer Klasse, welche **überladene oder überladbare Funktionen** besitzt, bekommt vom Compiler einen Zeiger (**`vfptr`** = virtual function pointer) verpasst, der auf die **`vftable`** der Klasse zeigt. Die `vftable` ist nur einmal pro Klasse vorhanden. Der `vfptr` ist pro Objekt einmal vorhanden. Objekte, die einen `vfptr` haben, sind **polymorph**.

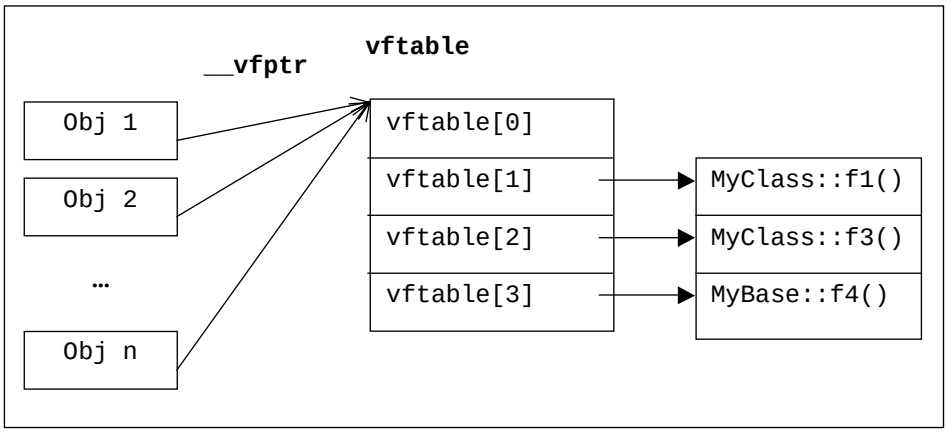
Beispiel:

```
class MyBase
{
     public:
          virtual ~MyBase() {} //Hack: nicht wirklich inline
          virtual void f1();
          void f2();
          virtual void f3() = 0;
          virtual void f4();
};

void MyBase::f1() { printf("MyBase::f1()\n"); }
void MyBase::f2() { printf("MyBase::f2()\n"); }
void MyBase::f4() { printf("MyBase::f4()\n"); }

class MyClass : public MyBase
{
     public:
          void f1() { printf("MyClass::f1()\n"); }
          void f3() { printf("MyClass::f3()\n"); }
          void f5() { printf("MyClass::f5()\n"); }
};
```

```
int main()
{
      MyClass Obj; //-> vfptr zeigt auf MyClass::'vftable'
      Obj.f1(); //MyClass::f1()
      Obj.f2(); //MyBase::f2()
      Obj.f3(); //MyClass::f3()
      Obj.f4(); //MyBase::f4()
      Obj.f5(); //MyClass::f5()
      return 0;
}
```

Hierbei sieht die **`vftable`** folgendermaßen aus (`__vfptr = &vftable[0]`):

```
vftable[0]:     MyClass::'vector deleting destructor'(unsigned int)
vftable[1]:     MyClass::f1(void)
vftable[2]:     MyClass::f3(void)
vftable[3]:     MyBase::f4(void)
```



Ein Funktionsaufruf einer virtuellen Methode unterscheidet sich also von einem Funktionsaufruf einer normalen Methode:

```
      MyClass Obj;

      Obj.f1();  → wird zu '(Obj.vfptr[1])();'
      Obj.f2();  → bleibt unverändert
```

Insgesamt ist das Aufrufen virtueller Funktionen **relativ effizient** und kostet kaum mehr Zeit als ein normaler Funktionsaufruf.

19.3 RTTI (type_info) und typeid bei polymorphen Objekten

Hat eine Klasse **überladene oder überladbare Funktionen** (polymorph), dann implementiert der Compiler eine **`vftable`**. In dieser wird dann **zur Laufzeit** ein Laufzeit-Typ-Info-Objekt (**RTTI-Objekt**, RunTime-Type-Info-Objekt) angehängt. Dieses Objekt ist vom Typ **`type_info`**.

Mit dem eingebauten Sprachelement **`typeid`** kann man den **Quell-Code-Namen** (name) und den **Linker-Namen** (raw_name) **der Klasse** eines polymorphen Objektes aus dem RTTI-Objekt (`type_info`-Objekt) ermitteln. Es wird eine **Referenz auf die Klasse `type_info`** zurückgeliefert:

```
class type_info
{
  public:
         virtual ~type_info();
         int operator==(const type_info& rhs) const;
         int operator!=(const type_info& rhs) const;
         int before(const type_info& rhs) const;
         const char* name() const;
         const char* raw_name() const;
  private:
         void *_m_data;
         char _m_d_name[1];
         type_info(const type_info& rhs);
         type_info& operator=(const type_info& rhs);
};
```

Beispiel:

```
#include <typeinfo.h>
#include <string.h>

int main()
{
      MyClass Obj;

      char szTypeName[256];
      strcpy(szTypeName,typeid(Obj).name());
      printf("Typ-Name: %s\n",
             szTypeName); //-> 'class MyClass'

      char szRawName[256];
      strcpy(szRawName,typeid(Obj).raw_name());
      printf("Name fuer Linker: %s\n",
             szRawName); //-> '.?AVMyClass@@'

      return 0;
}
```

Hinweis:

Man sollte (wenn möglich) **virtuelle Methoden** vorziehen, um Informationen über ein Objekt zugänglich zu machen. Dann kann jede Klasse dort ihre Information eintragen, wie das Beispiel zeigt:

```
class MyBase
{
      public:
            virtual const char* GetTypeName();
};

const char* MyBase::GetTypeName()
{
      return "class MyBase";
}

class MyClass : public MyBase
{
      public:
            const char* GetTypeName();
};

const char* MyClass::GetTypeName()
{
      return "class MyClass";
}

int main()
{
      MyBase BaseObj;
      printf("Typ-Name von BaseObj: %s\n",BaseObj.GetTypeName());

      MyClass Obj;
      printf("Typ-Name von Obj: %s\n",Obj.GetTypeName());

      return 0;
}
```

19.4 dynamic_cast: Sicherer cast von Zeigern oder Referenzen

19.4.1 Allgemeines

Mit Hilfe von **`dynamic_cast`** kann eine sichere Typumwandlung (cast = formen) von **Zeigern oder Referenzen** durchgeführt werden. Sicher ist die Umwandlung deshalb, weil eine fehlgeschlagene Umwandlung einen Fehler zurückmeldet (anders als bei anderen casts):

`dynamic_cast` von Zeigern:

Falls die Umwandlung möglich ist, wird ein cast durchgeführt, ansonsten wird `NULL` zurückgeliefert:

```
NewType* pNewObjekt = dynamic_cast<NewType*>(pObject)
if(!pNewObjekt)
{
      ... //Fehler
}
```

`dynamic_cast` von Referenzen:

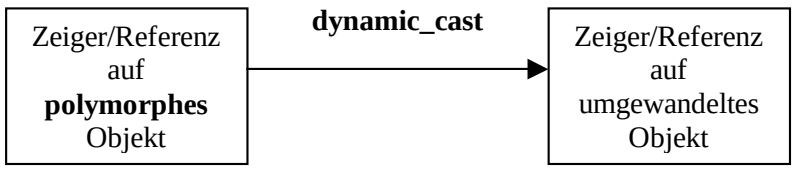
Hier kann keine Überprüfung auf `NULL` erfolgen → `try-catch`-Block:

```
try
{
      NewType& NewObjekt = dynamic_cast<NewType&>(Object)
}
catch(...)
{
      ... //Fehler
}
```

Besonderheiten:

- `dynamic_cast` benötigt **ein Objekt mit überladenen oder überladbaren Funktionen** (polymorphes Objekt bzw. Objekt mit einem `vfptr`), da nur bei solchen Objekten die benötigte **`vftable`** mit dem **RTTI-Objekt (`type_info`-Objekt)** angehängt ist. Eine entsprechende Compiler-Option (Enable RTTI) muss ggf. vor dem Übersetzen aktiviert werden.

  → **`dynamic_cast` kann nur angewendet werden, wenn das Objekt einen `vfptr` hat.**



→ **`static_cast` kann stattdessen benutzt werden, wenn das Objekt keinen `vfptr` hat.**

- Man sollte immer wenn ein cast schief gehen kann `dynamic_cast` verwenden und **nicht `static_cast`**. `dynamic_cast` ist zwar langsamer, dafür aber sicherer.

Beispiel für fehlschlagenden cast:

```
class MyBase
{
	public:
		MyBase(int nID = 0) : m_nID(nID) {}
		virtual ~MyBase() {}
		void SetID(int nID) { m_nID = nID; }
		int GetID() const { return m_nID; }
	private:
		int m_nID;
};

class MyClass : public MyBase
{
	public:
		MyClass() {}
		~MyClass() {}
};

int main()
{

	//Upcast eines Zeiger und einer Referenz
	//-> Bei richtiger "is-a"-Vererbung geht das "automatisch"

	MyClass Obj;

	MyBase* pBaseObj = NULL;
	pBaseObj = &Obj;                //impliziter statischer Upcast
	pBaseObj = (MyBase*) &Obj;      //herkömmlicher statischer Up-
	cast
	pBaseObj = static_cast<MyBase*>(&Obj);  //statischer Upcast
	pBaseObj = dynamic_cast<MyBase*>(&Obj); //dynamischer Upcast
	if(pBaseObj == 0)
		printf("FEHLER: *Upcast* hat nicht funktioniert!\n");

	try
	{
		MyBase& rBaseObj
				= dynamic_cast<MyBase&>(Obj); //dyn.Upcast
	}
	catch(...)
	{
		printf("FEHLER: *Upcast* hat nicht funktioniert!\n");
	}
```

```
	//Downcast eines Zeiger und einer Referenz
	//-> funktioniert implizit gar nicht, statisch nur scheinbar,
	//   bei dynamic_cast definitiv nicht

	MyBase BaseObj;

	MyClass* pObj = NULL;
	pObj = (MyClass*) &BaseObj;
		//-> gültiger Zeiger trotz unmöglichem Cast!
	pObj = static_cast<MyClass*>(&BaseObj);
		//-> gültiger Zeiger trotz unmöglichem Cast!
	pObj = dynamic_cast<MyClass*>(&BaseObj);
		//dynamischer Downcast -> hier NULL-Zeiger
	if(pObj == 0)
		printf("FEHLER: *Downcast* hat nicht funktioniert!\n");
	try
	{
		MyClass& rObj = dynamic_cast<MyClass&>(BaseObj);
		//-> hier Exception
	}
	catch(...)
	{
		printf("FEHLER: *Downcast* hat nicht funktioniert!\n");
	}

	return 0;
}
```

19.4.2 dynamic_cast zur Argumentprüfung bei Basisklassen-Zeiger/Referenz

Wenn man eine Funktion schreibt, die als Argument einen Zeiger auf die Basisklasse erwartet oder ein Objekt der Basisklasse referenziert, dann kann man mit `dynamic_cast` **überprüfen, ob der Nutzer ein gültiges Objekt übergeben hat**:

Beispiel:

```
class MyBase
{
	public:
		virtual ~MyBase() {}
		virtual void SetID(int nID) {}
		virtual int GetID() const { return 0; }
};

class MyClass1 : public MyBase
{
	public:
		MyClass1(int nID = 0) : m_nID(nID) {}
		~MyClass1() {}
		void SetID(int nID) { m_nID = nID; }
		int GetID() const { return m_nID; }
	private:
		int m_nID;
};
```

```
class MyClass2 : public MyBase
{
      public:
            MyClass2(int nID = 0) : m_nID(nID) {}
            virtual ~MyClass2() {}
            void SetID(int nID) { m_nID = nID; }
            int GetID() const { return m_nID; }
      private:
            int m_nID;
};

void f(MyBase* p)
{
      MyBase* pBaseObj = dynamic_cast<MyBase*>(p);
      if(!pBaseObj)
            printf("FEHLER in f(): Falscher Objekt-Typ!\n");
}
void g(MyBase& r)
{
      try
      {
            MyBase& BaseObj = dynamic_cast<MyBase&>(r);
      }
      catch(...)
      {
            printf("FEHLER in g(): Falscher Objekt-Typ!\n");
      }
}
void h(MyBase* p)
{
      MyClass1* pObj1 = dynamic_cast<MyClass1*>(p);
      if(!pObj1)
            printf("FEHLER in h(): Falscher Objekt-Typ!\n");
}
int main()
{
      MyBase BaseObj;
      MyClass1 Obj1;
      MyClass2 Obj2;

      //Kein Cast erforderlich:
      f(&BaseObj);    //-> ok
      g(BaseObj);     //-> ok

      //Upcast erforderlich:
      f(&Obj1);       //-> ok
      g(Obj1);        //-> ok
      h(&Obj1);       //-> ok

      //Downcast erforderlich:
      h(&BaseObj);    //-> Laufzeit-FEHLER

      //Crosscast erforderlich:
      h(&Obj2);       //-> Laufzeit-FEHLER

      return 0;
}
```

19.5 const_cast

Mit const_cast kann man const beseitigen.

- **Bei Konstanten:**

```
int main()
{
      const int nID = 2;
      const_cast<int&>(nID) = 3;
      return 0;
}
```

- **Bei `const`-Argumenten:**

```
class MyClass
{
      public:
            MyClass(int nID = 0) : m_nID(nID) {}
            void SetID(int nID) { m_nID = nID; }
            int GetID() const { return m_nID; }
      private:
            int m_nID;
};

void f(const MyClass& Obj)
{
      int i = Obj.GetID(); //Read-Only-Methoden aufrufbar
      const_cast<MyClass&>(Obj).SetID(8);
};

int main()
{
      MyClass Obj(5);
      f(Obj);
      return 0;
}
```

19.6 reinterpret_cast (!nicht portabel!) und Funktions-Vektoren

Mit `reinterpret_cast` kann man wild zwischen Zeigern auf beliebige Dinge wie Objekte, Funktionen oder Strukturen casten.

Beispiel:

Es sollen Zeiger auf `int`-Funktionen in einen Vektor mit Zeigern auf `void`-Funktionen aufgenommen und wieder ausgelesen werden:

```
typedef void (*pvoidFoo)();

int f1() { return 1; }
int f2() { return 2; }
int f3() { return 3; }

int main()
{
      pvoidFoo arrFoos[10];
      arrFoos[0] = reinterpret_cast<pvoidFoo>(f1());
      arrFoos[1] = reinterpret_cast<pvoidFoo>(f2());
      arrFoos[2] = reinterpret_cast<pvoidFoo>(f3());

      int i = 0;
      i = reinterpret_cast<int>(arrFoos[0]);
      i = reinterpret_cast<int>(arrFoos[1]);
      i = reinterpret_cast<int>(arrFoos[2]);

      return 0;
}
```

19.7 STL: Min- und Max-Werte zu einem Datentyp

Die STL bietet Templates, die einem den minimalen bzw. den maximalen Wert zu einem Datentyp liefern:

```
numeric_limits<Typ>::min()
numeric_limits<Typ>::max()
```

Beispiel:

```
numeric_limits<int>::min()
```

20. In Bibliotheken Exceptions werfen

20.1 Allgemeines

Wenn man eine Software-Bibliothek schreibt, dann kann es vorkommen, dass die geschriebenen Funktionen auf Grenzen stoßen (Bsp.: Festplatte voll, Datei gelocked, ...) und ihre Aufgabe nicht erfüllen können. In dem Fall können die Funktionen jedoch nicht reagieren, indem sie z.B. eine Fehler-Meldung an die Benutzeroberfläche bringen. Gründe dafür sind:

- Globaler Kontext unbekannt:

  Eine Bibliotheks-Funktion kann nicht wissen, in welchem globalen Kontext sie aufgerufen wurde, und kann somit keinen sinnvollen Text für die Fehlerbeschreibung finden (wie z.B. 'Konfiguration konnte nicht gespeichert werden').

- Kein Zugriff auf die (höher angesiedelte) Benutzeroberfläche:

  Eine Bibliotheks-Funktion hat keinen Zugriff auf das Interface der Benutzeroberfläche, da dieses höher in der Hierarchie liegt.

- Keine Entscheidungsbefugnis:

  Eine Bibliotheks-Funktion kann keine Entscheidung über die weitere Vorgehensweise nach dem Auftreten eines Fehlers treffen, da sie von der Hierarchie her überhaupt keine Entscheidungsbefugnis dazu hat.

- `return`-Werte/Referenz-Parameter können durch höheren Layer weggekapselt sein:

  Wenn eine Bibliotheks-Funktion als Helfer (Helper) für eine Interface-Funktion benutzt wird, und diese Interface-Funktion keine `return`-Werte oder Referenz-Parameter hat, dann bekommt der Nutzer der Interface-Funktion die zurückgelieferten Werte der Bibliotheks-Funktion nicht durchgereicht.

Aus diesen Gründen hat man das Konzept der Exceptions erfunden und **nicht etwa, um `return`-Werte zu ersetzen** (was schon wegen der schlechten Performance des Exception-Handlings Unsinn wäre).

Konzept:

Eine Funktion entdeckt einen Fehler, den sie selbst nicht behandeln kann, da die Funktion nur eine Aufgabe für eine höhere (unbekannte) Instanz ausführt. Die Funktion wirft (**`throw`**) deshalb eine **Exception** (Ausnahme) und unterbricht die Abarbeitung ihrer eigentlichen Aufgabe damit sofort.
Der Aufrufer kann die Funktion innerhalb eines **`try`**-Blocks aufrufen, so dass er in der Lage ist, eine von der Funktion geworfene Exception zu fangen (**`catch`**-Block) und den Fehler auf höherer Ebene zu behandeln.

Mechanismus → Stack-Unwinding:

Durch **`throw`** wird der **lokale Stack** der aufrufenden Funktion (also derjenigen mit dem `try`-`catch`-Block) zurückgewickelt, d.h. **alles, was konstruiert wurde, wird auch wieder destruiert**. Dann wird der Code in dieser Funktion hinter dem `try`-Block nach `catch`-Blöcken abgesucht. Passt der Argument-Typ von `catch()`, dann wird der **im `catch`-Block geschriebene Code ausgeführt**. Andernfalls wird der nächste `catch`-Block in der Funktion gesucht. Passt kein `catch`, dann wird das Gesamtprogramm beendet.
Um das Stack-Unwinding ordnungsgemäß durchführen zu können, muss das Programm sich eine Liste mit allen zu diesem Zeitpunkt voll konstruierten Objekten halten. Es werden dann zunächst die entsprechenden Destruktoren aufgerufen. Danach wird dann geprüft, ob es eine Exception-Spezifikation gibt → Falls ja: Wenn die Exception dieser nicht genügt, muss **`unexpected()`** aufgerufen werden.

Demonstration:

```
class MyClass
{
	public:
		MyClass(int nID = 0) : m_nID(nID) {}
		void Print() const { printf("Err No.: [ %d ]\n",m_nID); }
	private:
		int m_nID;
};

void f()
{
	throw "Error in f()";
}
```

Um die **`main`**-Funtion besser überblicken zu können, findet man sie auf der nächsten Seite:

```
int main()
{
   try
   {
      try
      {
         try
         {
            try
            {
               f();
            }
            catch(const char* szEx) //fängt geworfene Strings
            {
               printf("Exception: %s\n",szEx);
               throw 1; //wirft Integer weiter
            }
         }
         catch(const int nEx) //fängt geworfene Integer
         {
            printf("Exception: ** %d **\n",nEx);
            MyClass Obj(1);
            throw Obj; //wirft MyClass-Objekt weiter
         }
      }
      catch(const MyClass& ExObj) //fängt geworf. MyClass-Objekte
      {
         ExObj.Print();
         throw; //wirft die Exception unverändert weiter
      }
   }
   catch(...) //fängt alle geworfenen Objekte
   {
      printf("Exception: UNBEKANNT");
   }
   return 0;
}
```

Bei mehreren `catch`-Blöcken, die hintereinander angeordnet sind, wird nur in den ersten passenden eingetreten. Ausnahme: Er liegt in der Schachtelung in einer höheren Ebene. Dies ist aber innerhalb einer Funktion normalerweise wohl kaum der Fall (die obige Demonstration ist eine Ausnahme).

Beispiel für eine normale `try`-`catch`-Implementierung:

```
void f() throw (const char*)
{
      throw "Error in f()";
}
```

```
void g()
{
      try
      {
            f();
      }
      catch(const char* szEx) //fängt geworfene Strings
      {
            printf("Exception: %s\n",szEx);
      }
      catch(...)
      {
            printf("Exception: [unbekannt]\n");
      }
}

int main()
{
      g();
      return 0;
}
```

Da `try-catch`-Blöcke den Compiler dazu verdammen, sich eine Menge Laufzeitinformationen zu merken (großer Overhead), sollte man sparend mit diesen umgehen, z.B. ist eine **Schleife in den `try-catch`-Block zu legen und nicht umgekehrt**.

20.2 Exceptions per Referenz fangen

Für das Fangen eines Objektes hat man 2 Möglichkeiten:

- Man fängt **per Referenz** (zu empfehlen):

```
catch(ExType& ex)
{
      ...
}
```

- Man fängt per Wert (nicht zu empfehlen):

```
catch(ExType ex)
{
      ...
}
```

Da beim Fangen per Wert eine zusätzliche Kopie angelegt wird, kann das Slicing-Problem auftauchen (abgeleitete Objekte werden unter Umständen in Basisklassen-Objekte umgewandelt). Somit ist das Fangen per Referenz zu empfehlen.

20.3 Kopien beim Weiterwerfen vermeiden

Von geworfenen Objekten wird **immer** eine **Kopie** angelegt, um diese an die catch-Anweisung weiterzureichen, da der Gültigkeitsbereich bestehender Objekte verlassen wird und somit keine Referenzen weitergegeben werden können. Hierbei wird üblicherweise eine Kopie des **statischen Typs des Objektes** (Typ laut der Objekt-Definition) gemacht.

Wird nun im catch-Block eine Exception weitergeworfen, dann kann man dies auf 2 Arten tun:

- Die gleiche Kopie **nochmal werfen** (zu empfehlen):

```
catch(ExType& ex)
{
      ...
      throw;
}
```

- Eine weitere Kopie erzeugen und werfen (nicht zu empfehlen):

```
catch(ExType& ex)
{
      ...
      throw ex;
}
```

Zu beachten:

- Ist `ExType` eine Klasse, dann fängt `catch` auch alle davon abgeleiteten Klassen.
- `ExType` kann auch ein Zeiger-Typ sein. Mit `void*` fängt man alle geworfenen Zeiger.
- Man sollte das Werfen von Zeigern vermeiden, da die Gefahr besteht, dass sie auf ein nicht mehr gültiges Objekt zeigen und man sich daher immer genau überlegen muss, worauf ein zu werfender Zeiger überhaupt zeigen darf.

20.4 Beispiel für Exception-Handling

```
class MyString
{
	public:
		MyString(const char* const szStr)
		{
			if(szStr[0] == 0)
				throw "string is empty!";
			strcpy(m_szStr,szStr);
		}
		char& operator[](int pos) { return m_szStr[pos]; }
	protected:
		char m_szStr[256];
};

int main()
{
	MyString* pStr1 = NULL;
	try
	{
		pStr1 = new MyString("Hello");
	}
	catch(const char* const szError)
	{
		printf("Error: %s\n",szError);
		delete pStr1; //nicht vergessen!!! (vor dem weiterwerfen)
		throw;
	}
	delete pStr1;
	return 0;
}
```

20.5 Exception-Spezifikation

20.5.1 Allgemeines

Man kann (statt es zu kommentieren) genau spezifizieren, welche Exceptions eine Funktion werfen kann. Damit wird das Programm in die Funktion **`unexpected()`** geleitet, falls eine Ausnahme geworfen wird, die nicht der Spezifikation entspricht. Dadurch geschieht in der Regel ein sofortiger Programmabbruch.

Beispiel:

```
void Func1() throw(int); //-> darf nur Exceptions vom Typ int werfen
void Func2() throw(); //-> darf keine Exceptions werfen
```

Man sollte innerhalb einer Funktion mit Exception-Spezifikation **nie eine Funktion aufrufen, die nicht der Exception-Spezifikation genügt** → im Zweifelsfall die Spezifikation weglassen

20.5.2 Spezifikationswidrige Exceptions abfangen: set_unexpected

Idee:

Statt einen Programmabbruch (`terminate()`) hinzunehmen, wenn unerwartete Exceptions (solche, die einer vorhandenen Spezifikation nicht genügen) in **`unexpected()`** münden, stellt man selbst einen Handler zur Verfügung:

Statt:

```
void f() throw(int);
```

Jetzt:

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

void f() throw(int,bad_exception)
{
     throw "Bad";
}

void UnexpectedHandler()
{
     bad_exception Ex;
     throw Ex;
}

int main()
{
     set_unexpected(UnexpectedHandler);
     try
     {
          f();
     }
     catch(const int& Ex)
     {
          printf("Exception: [ #%d ]\n",Ex);
     }
     catch(const bad_exception& Ex)
     {
          printf("Exception: [ %s ]\n",Ex.what());
     }
     catch(...)
     {
          printf("Exception: [ unbekannt ]\n");
     }
     return 0;
}
```

Die Sache hat jedoch Nachteile:

- **Visual C++ kann es nicht**, obwohl es ordnungsgemäß kompiliert wird.
- **Man benötigt 3 `catch`-Blöcke**, um sicher alles zu fangen.

20.5.3 Compilerunabhängiges Vorgehen

Am einfachsten ist es sicher, wenn man sich eine Klasse schreibt, die sowohl den Fehler in einem **String** beschreibt, als auch eine **eindeutige ID** zurückliefert. Diese Klasse spezifiziert man bei allen Funktionen, die Exceptions werfen können. Der Aufrufer hingegen fängt einmal genau diese Exception und zum anderen alle anderen Exceptions (`catch(...)`):

```
#include <string.h>
class MyEx
{
	public:
		MyEx(const char* const szError,int nID = 0)
		: m_nID(nID) { strncpy(m_szError,szError,ERR_STR_LEN); }
		int GetID() const { return m_nID; }
		const char* GetError() const { return m_szError; }
	private:
		enum{ ERR_STR_LEN = 1023 };

		int	m_nID;
		char	m_szError[ERR_STR_LEN + 1]; //0-terminated
};

#include <exception>
using namespace std; //bad_exception

void f() throw(MyEx)
{
	throw MyEx("Test",1);
}

int main()
{
	try
	{
		f();
	}
	catch(const MyEx& Ex)
	{
		printf("Exception: [ %s ]\n",Ex.GetError());
	}
	catch(...)
	{
		printf("Exception: [ unbekannt/unspezifiziert ]\n");
	}
	return 0;
}
```

21. Die STL (Standard Template Library)

21.1 Allgemeines

Um nicht Standard-Algorithmen für **Sequenzen** (Listen, Vektoren, ...) von Objekten immer wieder neu implementieren zu müssen (und dies dann noch pro Datentyp einmal) hat man sich etwas einfallen lassen:

a) Man schreibt einen **Container**, d.h. eine Klasse, die die gesamte Funktionalität für eine Sequenz beinhaltet. Als **sequentiellen Container** bezeichnet man einen solchen, der auf einer Listenstruktur basiert (Bsp.: `list`). **Assoziative Container** bestehen aus 2 Listen: Die Liste der Schlüsselfelder (keys) und die Liste der Wertefelder (values). Wenn man einen Wert (value) hineinschreibt, dann muß man angeben, unter welchem Schlüssel (key) dieser gespeichert werden soll (Bsp.: `map`).

b) In einem weiteren Schritt sorgt man dafür, dass dieser Container **beliebige Elemente** aufnehmen kann, wozu man das **Template**-Konzept (Vorlagen-Konzept) einführt:

Beispiel für die Definition eines Templates:

```
template<class T>
class SmartPtr
{
      public:
            SmartPtr(T* pT = NULL) : m_pT(pT) {}
            SmartPtr(SmartPtr<T>& Obj) : m_pT(Obj.GetPtr())
            { Obj.Release(); }
            ~SmartPtr() { delete m_pT; }
            void Release() { m_pT = NULL; }
            T* GetPtr() { return m_pT; }
            template<class C>
            bool Transform(SmartPtr<C>& Obj)
            {
                  T* pT = dynamic_cast<T*>(Obj.GetPtr());
                  if(!pT)
                        return false;
                  m_pT = pT;
                  return true;
            }
            T* operator->() { return m_pT; }
            T& operator*() { return *m_pT; }
            ...
      private:
            T* m_pT;
};
```

Instanziierung dieser Vorlage für den Typ `MyClass`:

```
SmartPtr<MyClass> pMyClass(new MyClass);
```

c) Als letztes schafft man sich dann noch ein **Zugriffsobjekt**, über welches man auf jedes Element der Sequenz zugreifen kann, einen sogenannten **Iterator**:

Beispiel:

```
list<int> listInteger;
list<int>::iterator it;

for(it = listInteger.begin();it != listInteger.end();++it)
     (*it).push_back(5); //Zugriff auf die Methode push_back()
for(it = listInteger.begin();it != listInteger.end();++it)
     printf("%d\n",(*it)); //Ausgabe des gespeicherten Wertes
```

Diese Dinge haben Informatiker schon seit langem erledigt und optimiert, so dass **1994** Alexander **Stepanov** und Meng **Lee** das Ergebnis ihrer langjährigen Forschungsarbeiten im Bereich der Standard-Algorithmen (bei der Firma HP) erfolgreich als **STL** (**Standard-Template-Library**) in das ISO/ANSI-C++-Werk einbringen konnten.

Es gibt **verschiedene Implementierungen** der STL. Hier ein paar Beispiele für portable Implementierungen:

- STL von STLport **http://www.stlport.org/**
- STL von Rogue Wave **http://www.ccd.bnl.gov/bcf/cluster/pgi/pgC++_lib/stdlib.htm**
- STL von SGI (Silicon Graphics Inc.) **http://www.sgi.com/tech/stl/**
- STL von HP (Hewlett-Packard) bzw. D.R. Musser **ftp://ftp.cs.rpi.edu/pub/stl/**

Beispiele für Container-Klassen der STL:

`vector`	→	Eindimensionales Feld
`list`	→	Doppelt verkettete Liste
`queue`	→	Schlange
`deque`	→	Schlange mit 2 Enden
`stack`	→	Stack
`set`	→	Sortierte Menge
`bitset`	→	Sortierte Menge von booleschen Werten
`map`	→	Sortierte Menge (Schlüsselfelder), die mit anderer Menge (Wertefelder) assoziiert ist

Besonderheiten:

- **Token:**

  Die **Token** (Steuerzeichen) < und > sind etwas problematisch: Bei **Verschachtelungen von Templates** kann eine **Verwechslung der doppelten Klammerung mit** << **oder** >> passieren. Deshalb muss man bei Verschachtelungen ggf. ein **SPACE** einfügen.

Beispiel:

```
Falsch:     set<long,less<long>> setMyObjects;

Richtig:    set<long,less<long> > setMyObjects;
                                ↑
```

- **Iteratoren:**

  Iterator-Objekte arbeiten wie Zeiger und entkoppeln die Algorithmen von den Daten, so dass diese typunabhängig werden. Sie sind praktisch Schnittstellen-Objekte.

  Innerhalb von **Read-Only**-Member-Funktionen (`const`) muss man mit

  `const_iterator`

  statt `iterator` arbeiten.

- **Effektivität:**

  Die STL bietet ein **optimiertes dynamisches Heap-Speicher-Management** mit **generischem Code**, was kaum zu überbieten sein dürfte. So sollte man sich intensiv der STL bedienen, wenn man etwas **"Dynamisches" auf dem Heap** benötigt. Kann man die zu lösende Aufgabe jedoch mittels nicht- dynamischem Array lösen (keine dynamische Länge des Arrays; kein Code, sondern nur Daten im Array) dann sollte man prüfen, ob die Nutzung eines Arrays auf dem Stack nicht doch effektiver ist.

  Beispiel:

  `vector<int> vectValues;`
  → Container auf dem Stack, der über die Methode `push_back()` intern Heap allokiert, um Daten zu speichern

  `int aValues[100];`
  → Array auf dem Stack ("purer" Stack-Speicher)

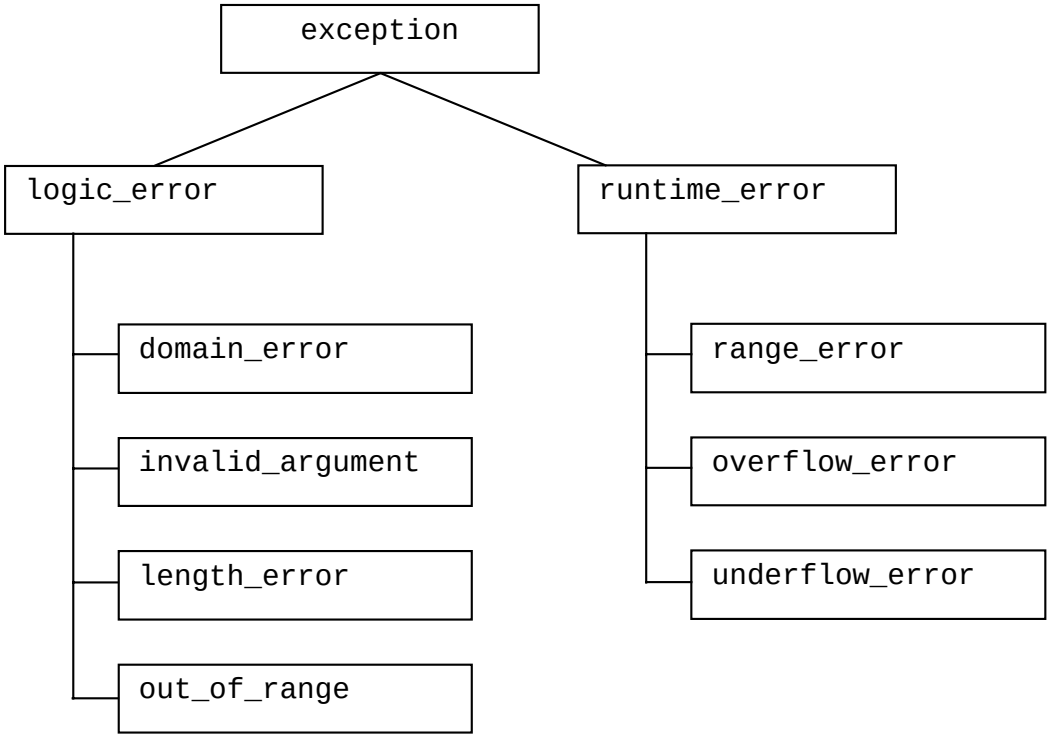
- **Güte der Implementierung und `hash_map`:**

  Möchte man eine Implementierung der STL testen, dann sollte man erst mal einen Blick auf die `hash_map` werfen. Besonders schwach ist die Implementierung, wenn es gar keine `hash_map` gibt. Die Untersuchung der Schnelligkeit der `hash_map` sagt viel aus → wenn so etwas Kompliziertes wie eine `hash_map` gut funktioniert, dann ist das schon mal ein sehr gutes Zeichen. Zum Testen kann man zunächst als Schlüssel (key) den Typ `string` verwenden.

Exceptions der STL:

Die STL definiert in der Header-Datei **`stdexcept`** folgende Exceptions:

```
namespace std
{
      class logic_error;
            class domain_error;
            class invalid_argument;
            class length_error;
            class out_of_range;
      class runtime_error;
            class range_error;
            class overflow_error;
            class underflow_error;
};
```



Die Exceptions sind alle von `exception` abgeleitet (**`what()`** → Erfragen der Fehlermeldung):

```
class exception
{
      public:
            exception() throw();
            exception(const exception& rhs) throw();
            exception& operator=(const exception& rhs) throw();
            virtual ~exception() throw();
            virtual const char *what() const throw();
};
```

21.2 Nutzung der STL von STLport

21.2.1 Allgemeines

Um die STL-Implementierung 'STLport' von der Firma 'STLport' (Nomen est Omen) nutzen zu können, muss man sich zunächst die erforderlichen Dateien von **http://www.stlport.org** herunterladen (Freeware). Diese STL liegt zunächst nur als Quell-Code vor.

21.2.2 STLport mit GNU unter Linux

Die Nutzung von STLport mit GNU unter Linux wird durch folgende Maßnahmen vorbereitet:

- Die Dateien in das private Entwicklungs-Verzeichnis (hier: `/Peter/Dev/`) kopieren:

  ```
  mkdir /home/Peter/Dev/STLport_STL
  cp * /home/Peter/Dev/STLport_STL -r
  ```

- Ins `src`-Verzeichnis wechseln:

  ```
  cd /home/Peter/Dev/STLport_STL/src
  ```

- Vorbereiten (nicht bauen):

  ```
  make -f gcc-linux.mak prepare
  ```

- Ins `stlport`-Verzeichnis wechseln:

  ```
  cd /home/Peter/Dev/STLport_STL/stlport
  ```

- Die Datei `stl_user_config.h` editieren:

  ```
  vi stl_user_config.h
  ```

  Kommentarzeichen `//` vor folgende Anweisung:

  ```
  // #define _STLP_NO_OWN_IOSTREAMS 1
  ```

  Dadurch wird die Nutzung der `STLport` iostreams abgeschaltet und es werden stattdessen die bereits vorhandenen iostreams gewrapped.

Nun kann STLport bei der Programmierung genutzt werden:

In einem neuen Projekt

`/home/Peter/Dev/NewProject`

nutzt man als **`include`** Pfad

`../STLport_STL/stlport`

und als **`lib`** Pfad

`../STLport_STL/lib`

Man würde also die Datei `test.cpp` wie folgt bauen:

```
g++ -I../STLport_STL/stlport test.cpp -L..STLport_STL/lib/
```

Beispiel:

`test.cpp`:

```
#include <stdio.h>
#include <string>
using namespace std;
int main()
{
	string strTest("Hello World");
	printf("%s\n",strTest.c_str());
	return 0;
}
```

`makefile`:

```
#
# Linker:
#

test: test.o
	g++ test.o -L..STLport_STL/lib/ -otest.exe

#
# Compiler:
#

test.o: test.cpp #dependencies
	g++ -c test.cpp -I../STLport_STL/stlport
```

21.2.3 STLport mit Visual C++ unter Windows

Die Nutzung von STLport unter Visual C++ wird durch folgende Maßnahmen vorbereitet:

- In der Datei **"\stlport\stl_site_config.h"** sind folgende Einstellungen **vor dem Bau** der STL vorzunehmen:
  - **Multithreading** aktivieren:

    ```
    //#define _NOTHREADS //aktiviert
    ```

  - STL zwingen, einfaches (nicht optimiertes) **new** zum Allokieren zu benutzen:

    ```
    #define _STLP_USE_NEWALLOC 1
    ```

- Make-Datei (makefile) im Unterverzeichnis **/src** für den Compiler **umbenennen**:

  "vc6.mak" → "makefile"

- Auf der **Kommandozeile** (Console) im Verzeichnis **/src bauen**:

  "nmake all"

- Dem **Compiler/Linker die Pfade der Dateien mitteilen**. Sie sollten **vor den Pfaden der Compiler/Linker-Bibliotheken** in der Liste auftauchen → Menu **"Tools.Options.Directories"**:

  Include: L:\...**stlport**
  Library Files: L:\...**lib**
  Source Files: L:\...**src**

  wobei **"L:\..."** für den Hauptpfad steht, z.B. **"D:\STLport-4.0"**

- In den Projekt-Settings sind unter **C++.Code Generation** die zu verwendenden **Runtime-Libraries** anzugeben:

  DEBUG: Debug Multithreaded DLL
  RELEASE: Multithreaded DLL

- Im Code kann man nun folgende warnings abschalten

  ```
  //Keine Warn. 'unrefer.inline function has been removed':
  #pragma warning(disable:4514)
  //Keine Warnung 'truncated identifier ...':
  #pragma warning(disable:4786)
  ```

21.3 STL-Header-Dateien

21.3.1 Aufbau: Die Endung ".h" fehlt

Bei den **Header-Dateien** der STL gibt es einige **Besonderheiten**:

- Sie haben keine Endung, also auch **nicht** **`".h"`**
- Der Code befindet sich im **`namespace std`**

  Beispiel:

```
//*******************************************
// file: 'list'
//*******************************************

namespace std
{
      ...
}
```

- Die **normalen C-Header** werden auch im **`namespace std`** angeboten, wobei jedoch der **Name um das Präfix "`c`" erweitert** wird und die **Endung "`.h`" weggelassen** wird

  Beispiel:

```
//*******************************************
// file: 'cmath'
//*******************************************

namespace std
{
      #include <math.h>
}
```

21.3.2 Nutzung: "using namespace std"

Wenn man also **Header der STL nutzt**, dann muss man grundsätzlich den **`namespace std`** mit in den global scope aufnehmen oder alle Zugriffe über **`std::`** durchführen.

Beispiel:

```
#include <list>
using namespace std;
int main()
{
      list<int> listValues;
      listValues.push_back(3);
      listValues.push_back(4);
      int i = listValues.front();
      return 0;
}
```

21.4 Wichtige STL-Member-Variablen und Methoden

Für alle Container der STL gibt es einen **gemeinsamen** Satz von Member-Variablen und Methoden:

Wichtige STL-Member-Variablen:

value_type
→ Datentyp der Elemente (values)
key_type
→ Datentyp der Schlüssel (keys) bei assoziativen Containern (wie map)
size_type
→ Einheit der Längen- bzw. Größenangaben
difference_type
→ Einheit des Iterierens zum nächsten Element

iterator
→ Zeiger **`value_type*`**, der bei ++ nach vorne dreht
const_iterator
→ Zeiger **`const value_type*`**, der bei ++ nach vorne dreht

reverse_iterator
→ Zeiger **`value_type*`**, der bei ++ zurück dreht
reverse_const_iterator
→ Zeiger **`const value_type*`**, der bei ++ zurück dreht

reference
→ Referenz **`value_type&`**
const_reference
→ Referenz **`const value_type&`**

Wichtige STL-Methoden:

Allgemeiner Zugriff:

empty()
→ Prüft, ob der Container leer ist
size()
→ Anzahl der Elemente
max_size()
→ Maximale mögliche Anzahl von Elementen

for_each()
→ for-Schleife über alle Elemente
begin()
→ Zeiger auf das erste Container-Element (für iterator)
end()
→ Zeiger hinter das letzte Container-Element (kein gültiges Element)
rbegin()
→ Zeiger auf das letzte Container-Element (für reverse_iterator)
rend()
→ Zeiger vor das erste Container-Element (kein gültiges Element)

front()
→ Erstes Element
last()
→ Letztes Element
[index]
→ Element an der Stelle **index** (ungeprüfter Zugriff)
at(index)
→ Element an der Stelle **index** (geprüfter Zugriff)

key_compare()
→ Vergleich der Schlüsselfelder (keys) assoziativer Container

Manipulationen:

clear()
→ Alle Elemente eines Containers löschen

insert()
→ Element richtig sortiert einfügen (`set`, `map`)
erase()
→ Element in einer sortierten Sequenz (`set`, `map`) löschen

push_back()
→ Element an den Schluss einer unsortierten Sequenz anhängen
pop_back()
→ Element vom Schluss einer unsortierten Sequenz entfernen
push_front()
→ Element vor den Anfang einer unsortierten Sequenz stellen
pop_front()
→ Element vom Anfang einer unsortierten Sequenz entfernen

sort()
→ Elemente in unsortierter Sequenz sortieren
stable_sort()
→ Elemente sortieren, wobei Reihenfolge gleicher Elemente bleibt
partial_sort()
→ Den ersten Teil einer Sequenz sortieren
partial_sort_copy()
→ Elemente kopieren und den ersten Teil der Sequenz sortieren
nth_element()
→ Das n-te Element an die richtige Stelle sortieren
merge()
→ 2 sortierte Sequenzen verschmelzen
inplace_merge()
→ 2 sortierte Teilsequenzen einer Sequenz verschmelzen
unique()
→ Aufeinanderfolgende Duplikate entfernen (vorher: **`sort()`**)
unique_copy()
→ Elemente kopieren und aufeinanderfolgende Duplikate entfernen
remove()
→ Element (alle Vorkommnisse) in unsortierter Sequenz löschen
remove_if()
→ Elemente mit bestimmtem Inhalt entfernen
remove_copy()
→ Elemente kopieren und die mit bestimmten Inhalt entfernen
remove_copy_if()
→ Elemente kopieren und die mit bestimmten Inhalt entfernen
replace()
→ Inhalt der Elemente durch anderen Inhalt ersetzen
replace_if()
→ Inhalt der Elemente durch anderen Inhalt ersetzen (mit Bedingung)
replace_copy()
→ Elemente kopieren und dabei Inhalt ersetzen
replace_copy_if()
→ Elemente kopieren und dabei Inhalt ersetzen (mit Bedingung)

copy()
→ Alle Elemente in gegebener Reihenfolge kopieren
copy_backwards()
→ Alle Elemente in umgekehrter Reihenfolge kopieren
reverse()
→ Umkehrung der Reihenfolge der Elemente
reverse_copy()
→ Elemente kopieren und ihre Reihenfolge umkehren

swap()
→ 2 Elemente vertauschen
iter_swap()
→ 2 Elemente, auf die die Iteratoren zeigen, vertauschen
swap_ranges()
→ 2 Bereiche von Elementen vertauschen
rotate()
→ Elemente rotieren
rotate_copy()
→ Elemente kopieren und rotieren
random_shuffle()
→ Elemente zufällig mischen
partition()
→ Bestimmte Elemente nach vorne platzieren
stable_partition()
→ Bestimmte Elemente unter Beibehaltung ihrer Reihenfolge . . .

fill()
→ Alle Container-Elemente füllen
fill_n()
→ n Container-Elemente füllen
generate()
→ Alle Elemente mit dem Ergebnis einer Operation füllen
generate_n()
→ n Elemente mit dem Ergebnis einer Operation füllen

set_union()
→ Sortierte **Vereinigungsmenge** zweier Sequenzen erzeugen
set_intersection()
→ Sortierte **Schnittmenge** zweier Sequenzen erzeugen
set_difference()
→ Andere Menge ausschließen
set_symmetric_difference()
→ Schnittmenge ausschließen

make_heap()
→ Sequenz als Heap einrichten
push_heap()
→ Element auf die als Heap eingerichtete Sequenz drauflegen
pop_heap()
→ Element von der als Heap eingerichteten Sequenz wegnehmen
sort_heap()
→ Als Heap eingerichtete Sequenz sortieren

Analysen:

min()
→ Das Element von zweien mit dem kleineren Inhalt bestimmen
max()
→ Das Element von zweien mit dem größeren Inhalt bestimmen

find()
→ Bestimmtes Element finden
find_if()
→ Bestimmtes Element finden (Bedingung)
find_first_of()
→ Bestimmtes Element finden (erstes Vorkommnis)
adjacent_find()
→ Benachbartes Elemente-Paar finden
min_element()
→ Das Element einer Sequenz mit dem kleinsten Inhalt finden
max_element()
→ Das Element einer Sequenz mit dem größten Inhalt finden
lower_bound()
→ Erstes Element mit bestimmtem Inhalt in sortierter Sequenz finden
upper_bound()
→ Letztes Element m.bestimmtem Inhalt in sortierter Sequenz finden
search()
→ Nach einer Teilsequenz mit bestimmtem Inhalt suchen
search_n()
→ Nach einer Teilsequenz mit n übereinstimmenden Inhalten suchen
find_end()
→ Letztes Auftreten einer Teilsequenz suchen
equal_range()
→ Teilsequenz mit bestimmtem Inhalt in sortierter Sequenz finden
mismatch()
→ Erstes unterschiedliches Element in 2 Sequenzen finden
next_permutation()
→ nächste Permutation in lexikographischer Reihenfolge finden
prev_permutation()
→ vorherig.Permutation in lexikographischer Reihenfolge finden

count()
→ Anzahl der Elemente mit bestimmtem Inhalt finden
count_if()
→ Anzahl der Elemente mit bestimmtem Inhalt finden
binary_search()
→ Prüfen, ob Element mit best. Inhalt in sortierter Sequenz enthalten

equal()
→ Prüfen, ob 2 Sequenzen gleich sind
includes()
→ Prüfen, ob Sequenz eine Teilsequenz einer anderen Sequenz ist
lexicographical_compare()
→ Prüfen, ob 2 Sequenzen lexikographisch gleich sind

21.5 Generierung von Sequenzen über STL-Algorithmen

21.5.1 back_inserter()

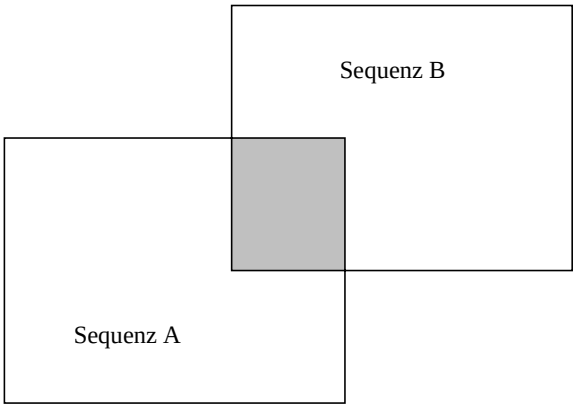
Die Funktion `back_inserter()` dient dazu, die von einem STL-Algorithmus generierte lokale Sequenz in einen vom Aufrufer bereitgestellten Container zu kopieren. Die Anwendung von back_inserter() entspricht also in etwa der Anwendung eines Referenz-Parameters beim Aufruf einer Funktion.

Beispiel:

```
list<MyClass> listDiff;
set_difference(       listA.begin(),listA.end(),
                      listB.begin(),listB.end(),
                      back_inserter(listDiff));
```

21.5.2 Schnittmenge (set_intersection)

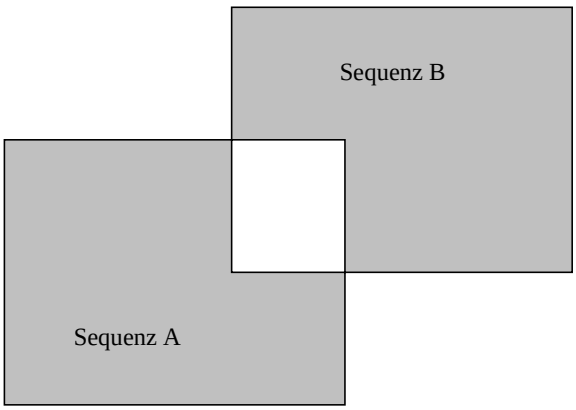
Die STL kann eine Schnittmenge zweier Sequenzen ermitteln:




```
listA.sort();
listB.sort();

list<MyClass> listIntersection;
set_intersection(     listA.begin(),listA.end(),
                      listB.begin(),listB.end(),
                      back_inserter(listIntersection));
```

21.5.3 Schnittmenge ausschließen (set_symmetric_difference)

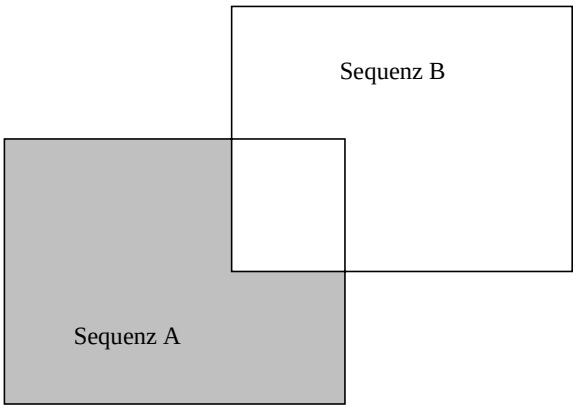
Die STL kann die Schnittmenge zweier Sequenzen von der Gesamtmenge ausschließen:




```
listA.sort();
listB.sort();

list<MyClass> listSymDiff;
set_symmetric_difference(  listA.begin(),listA.end(),
                           listB.begin(),listB.end(),
                           back_inserter(listSymDiff));
```

21.5.4 Sequenz ausschließen (set_difference)

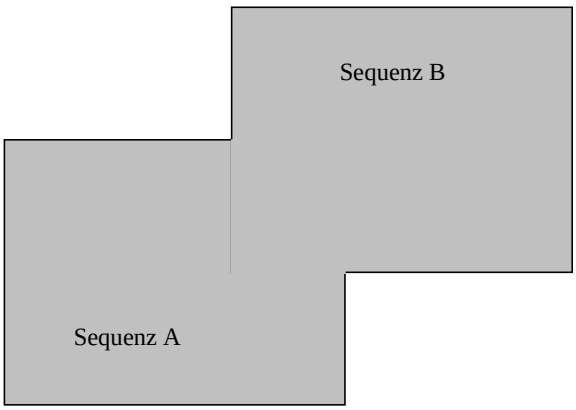
Die STL kann eine Sequenz von der Gesamtmenge ausschließen:




```
listA.sort();
listB.sort();

list<MyClass> listDiff;
set_difference( listA.begin(),listA.end(),
                listB.begin(),listB.end(),
                back_inserter(listDiff));
```

21.5.5 Vereinigungsmenge bilden (set_union)

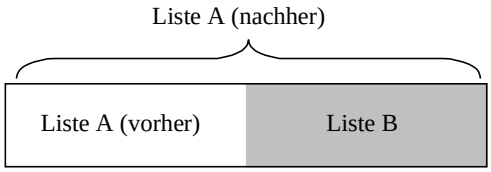
Die STL kann eine Vereinigungsmenge zweier Sequenzen bilden:




```
listA.sort();
listB.sort();

list<MyClass> listUnion;
set_difference( listA.begin(),listA.end(),
                listB.begin(),listB.end(),
                back_inserter(listUnion));
```

21.5.6 Liste an eine andere Liste anhängen (list::insert)

Die STL kann eine Liste an eine andere Liste anhängen:




```
listA.insert(   listA.begin(),listA.end(),
                listB.begin(),listB.end());
```

21.6 Wichtige Regeln

21.6.1 Einbinden der STL

Es gibt 2 Besonderheiten beim Bau einer Software mit der STL:

- Lange Namen der STL

Die STL bildet durch Verschachtelungen sehr lange Namen. Microsoft's Visual C++ hat dann Probleme diese Namen in die Debug-Informations-Datei einzutragen, da dort eine maximale Länge von 255 Zeichen erlaubt ist. Deshalb kürzt der Compiler diese und bringt eine Warnung:

→ warning: identifier was truncated to '255' characters in the debug information.

- Nichtreferenzierte `inline`-Funktionen der STL

Normalerweise werden nicht alle `inline`-Funktionen der STL durch den Code des Nutzers referenziert. Visual C++ wirft diese Funktionen beim Bau weg und warnt den Nutzer:

→ warning: unreferenced inline function has been removed.

Um diese lästigen Warnungen bei der Kompilierung mit Visual C++ abzuschalten, benutzt man folgende `#pragma`-Anweisungen:

```
#pragma warning(disable:4786)
#pragma warning(disable:4514)
```

Die Aufnahme der STL in den Quell-Code geschieht in einer zentralen Header-Datei (`Settings.h`), wobei nach dem Abschalten der Warnungen die Algorithmen der STL über **`#include <algorithm>`** und **danach** die **benötigten STL-Container** einzubinden sind. **Zum Schluss** nimmt man **`std` in den global namespace** auf, d.h. alle Namen im namespace `std` werden zu globalen Namen.

Beispiel:

```
//Keine Warnung wg. verkürzten Namen oder beseitigten
//inline-Funktionen bei MS VC++:
#if  defined(_MSC_VER) &&
     !defined(__MWERKS__) &&
     !defined(__ICL) &&
     !defined(__COMO__)
          #pragma warning(disable:4786)
          #pragma warning(disable:4514)
#endif

//STL-Algorithmen:
#include <algorithm>
//STL-Container:
#include <list>
#include <set>
//std in global space aufnehmen:
using namespace std;
```

21.6.2 Die benötigten Operatoren implementieren

Die Container der STL benötigen Objekte, für die ein Vergleich möglich ist, d.h. es muss auf Gleichheit und auf Kleiner geprüft werden können. Wenn man eine eigene Klasse für die Objekte eines Containers schreibt, dann müssen `operator==()` und `operator<()` implementiert werden. Außerdem wird in manchen Fällen der Default-Konstruktor gebraucht.

Beispiele:

Minimal-Implementierung:

```
class MyClass
{
      public:
            explicit MyClass(int nID = 0) //Default
                  : m_nID(nID)
            {
            }
            int GetID() const { return m_nID; }
            bool operator==(const MyClass& Obj) const
            {
                  if(Obj.GetID() == m_nID)
                        return true;
                  return false;
            }
            bool operator<(const MyClass& Obj) const
            {
                  if(m_nID < Obj.GetID())
                        return true;
                  return false;
            }
      private:
            int m_nID;
};
```

Implementierung mit globalen Operatoren und einem Zuweisungsoperator:

```
class MyClass
{
      public:
            MyClass(int nID = 0) : m_nID(nID) {}
            int GetID() const { return m_nID; }
            void SetID(int nID) { m_nID = nID; }
            MyClass& operator=(const MyClass& Obj)
            {
                  if(this == &Obj)
                        return *this;
                  SetID(Obj.GetID());
                  return *this;
            }
      private:
            int m_nID;
};
```

```
//Globale Operatoren:

inline bool operator==(const MyClass& lhs,const MyClass& rhs)
{
	if(lhs.GetID() != rhs.GetID())
		return false;
	return true;
}
inline bool operator!=(const MyClass& lhs,const MyClass& rhs)
{
	if(lhs == rhs)
		return false;
	return true;
}
inline bool operator<(const MyClass& lhs,const MyClass& rhs)
{
	if(lhs.GetID() < rhs.GetID())
		return true;
	return false;
}
```

21.6.3 Iterator: ++it statt it++ benutzen

Man sollte den **Präfix-Operator `++it`** immer dem Postfix-Operator `it++` vorziehen (Bsp.: im Kopf einer `for`-Schleife), da dessen **Performance besser** ist.

Beispiel:

```
int main()
{
	list<MyClass> listObjs;
	listObjs.push_back(2);
	listObjs.push_back(1);
	listObjs.push_back(1);
	listObjs.push_back(3);
	listObjs.sort();
	listObjs.unique();
	list<MyClass>::iterator it;
	for(it = listObjs.begin();it != listObjs.end();++it)
	{
		printf("%d\n",(*it));
	}
	return 0;
}
```

Grund:

Der Postfix-Operator benötigt ein **zusätzliches temporäres Objekt**, welches **konstruiert**, für die Rückgabe **kopiert** (Wert-Rückgabe) und dann noch **destruiert** werden muss:

```
template<class T>
class iterator
{
        public:
                ...
                iterator& operator++();      //Präfix (++it)
                iterator operator++(int);    //Postfix (it++)
        protected:
                T& container;
                T::iterator iter;
                ...
};

iterator& iterator::operator++()    //Präfix (++it)
{
        ++iter;
        return *this;
}

iterator iterator::operator++(int)  //Postfix (it++)
{
        iterator temp = *this;
        ++(*this);
        return temp; //der Wert vor der Operation wird zurückgegeben
}
```

*21.6.4 Löschen nach find(): Immer über Iterator (it) statt über den Wert (*it)*

Problem:

Das **Löschen über einen Wert** (**`*it`**) erfordert zunächst immer ein **internes `find()`** zum Suchen der Speicherstelle, die diesen Wert beinhaltet. Wenn man gerade erst `find()` ausgeführt hat, um herauszufinden, ob sich das Element überhaupt in der Sequenz befindet, dann verschwendet man beim Löschen über den Wert nochmal die Zeit für ein `find()`.

Abhilfe:

Man merkt sich den **Iterator** (**`*it`**), welchen `find()` zurückliefert, und operiert direkt über diesen auf der Speicherstelle.

Beispiel:

```
class MyClass
{
	public:
		explicit MyClass(int nID = 0) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <list>
#include <set>
using namespace std;

int main()
{
	MyClass Obj1(1);
	MyClass Obj2(2);
	MyClass Obj3(3);
	MyClass Obj3(4);

	set<MyClass> setObjs;
	setObjs.insert(Obj1);
	setObjs.insert(Obj2);
	setObjs.insert(Obj3);
	setObjs.insert(Obj4);
	int i = 0;
	set<MyClass>::iterator it = setObjs.find(Obj3);
	if(it != setObjs.end())
	{
		i = (*it).GetID();
		setObjs.erase(it); //nicht setObjs.erase(*it)!!!
	}

	return 0;
}
```

Anmerkung:

`erase()` kann natürlich ohne vorhergehendes `find()` angewendet werden!

21.6.5 map: Nie indizierten Zugriff [ ] nach find() durchführen

Problem:

Der **indizierte Zugriff** auf eine map mit dem []-Operator erfordert zunächst immer ein **internes `find()`** zum Suchen der Speicherstelle, die diesen Wert beinhaltet. Wenn man gerade erst `find()` ausgeführt hat, um zu wissen, ob sich das Element überhaupt in der Sequenz befindet, dann verschwendet man nochmal die Zeit für ein `find()`.

Abhilfe:

Man merkt sich den Iterator, welchen find() zurückliefert, und operiert direkt über diesen auf der Speicherstelle.

Beispiel:

```
class MyClass
{
	public:
		explicit MyClass(int nID = 0) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <map>
using namespace std;

int main()
{
	MyClass Obj1(1);
	MyClass Obj2(2);
	MyClass Obj3(3);

	map<long,MyClass> mapHandleToObj;

	mapHandleToObj[1L] = Obj1;
	mapHandleToObj[10L] = Obj1;
	mapHandleToObj[100L] = Obj1;
	mapHandleToObj[200L] = Obj2;
	mapHandleToObj[300L] = Obj3;
```

```
	long lDeletedKey = 0L;
	MyClass DeletedObj(0);

	//key suchen:
	map<long,MyClass>::iterator it = mapHandleToObj.find(10L);
	if(it != mapHandleToObj.end())
	{
		lDeletedKey = (*it).first;
		DeletedObj = (*it).second;
				//nicht DeletedObj = mapHandleToObj[10L]!!!
		mapHandleToObj.erase(it);
				//nicht mapHandleToObj.erase(*it)!!!
	}

	return 0;
}
```

21.7 Beispiele für die Verwendung der Container

21.7.1 list: Auflistung von Objekten mit möglichen Mehrfachvorkommnissen

```
class MyClass
{
	public:
		explicit MyClass(int nID = 0) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <list>
using namespace std;

int main()
{
	MyClass	Obj1(1);
	MyClass	Obj2(2);
	MyClass	Obj3(3);

	list<MyClass> listObjs;
	listObjs.push_front(Obj3); //Element an den Anfang einfügen
	listObjs.push_front(Obj2); //Element an den Anfang einfügen
	listObjs.push_back(Obj1);  //Element ans Ende anhängen
	listObjs.push_back(Obj2);  //Element ans Ende anhängen

	listObjs.sort();           //Liste sortieren
	listObjs.unique();         //Benachbarte Mehrfach-
                                   //vorkommnisse eliminieren
	MyClass Obj(0);
	Obj = listObjs.front();    //erstes Element lesen
	Obj = listObjs.back();     //letztes Element lesen

	listObjs.pop_front();      //erstes Element löschen
	listObjs.pop_back();       //letztes Element löschen

	listObjs.push_back(Obj1);  //-> Mehrfachvorkommnis von Obj1
	listObjs.push_back(Obj1);  //-> Mehrfachvorkommnis von Obj1
	listObjs.push_back(Obj1);  //-> Mehrfachvorkommnis von Obj1
```

```
    int nFirstDeletedID = 0;
    list<MyClass>::iterator it;
    it = find(listObjs.begin(),listObjs.end(),Obj1);
    if(it != listObjs.end())   //erstes Vorkommnis von Obj1
    {
        nFirstDeletedID = (*it).GetID();
        listObjs.erase(it);   //dieses Vorkommnis von Obj1 löschen
    }

    listObjs.remove(Obj1);     //alle Vorkommnisse von Obj1 löschen

    for(it = listObjs.begin();it != listObjs.end();++it)
    {
        if((*it).GetID() == 1)
            break;
    }

    return 0;
}
```

Besonderheiten bei `list`:

- **Sortierung (`sort()`) und Eindeutigkeit (`unique()`)**
  Eine `list` kann nur von Mehrfachvorkommnissen von Objekten befreit werden (`unique()`), wenn sie sortiert (`sort()`) vorliegt:
  ```
  listObjs.sort();
  listObjs.unique();
  ```

- **Keine Member-Funktion `find()`**
  Zum Suchen ist der allgemeine Algorithmus `find()` anzuwenden, wobei zu beachten ist, dass hierbei nur das erste Vorkommnis in der `list` gefunden wird:
  ```
  it = find(listObjs.begin(),listObjs.end(),Obj1);
  ```
  Man kann natürlich die `list` vorher sortieren und unique machen. In dem Fall ist es jedoch effektiver, von vornherein eine `set` zu benutzen, da dort bereits beim Einfügen sortiert wird, was schneller geht. Außerdem ist die `set` a priori unique.

- **Nur `remove()` löscht sicher alle Vorkommnisse eines Wertes**
  Wenn man alle Vorkommnisse eines Wertes löschen will, ist **nicht** `erase()` zu verwenden, sondern **`remove()`**.

- **Die Reihenfolge der Objekte in der `list` ist fix**
  Nur Algorithmen wie `sort()` können etwas an der Reihenfolge der Objekte in der `list` ändern.

21.7.2 set: Aufsteigend sortierte Menge von Objekten (unique)

```
class MyClass
{
	public:
		explicit MyClass(int nID = 0) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
	MyClass	Obj1(1);
	MyClass	Obj2(2);
	MyClass	Obj3(3);

	set<MyClass> setObjs;
	setObjs.insert(Obj3); //Element einfügen
	setObjs.insert(Obj1); //Element einfügen
	setObjs.insert(Obj2); //Element einfügen

	set<MyClass>::iterator it;

	it = setObjs.find(Obj1);
	if(it != setObjs.end())
		setObjs.erase(it);	//Element löschen

	for(it = setObjs.begin();it != setObjs.end();++it)
	{
		if((*it).GetID() == 1)
			break;
	}
	return 0;
}
```

21.7.3 map: Zuordnung von Objekten zu eindeutigen Handles

```
class MyClass
{
	public:
		explicit MyClass(int nID = 0) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <map>
using namespace std;

int main()
{
	MyClass	Obj1(1);
	MyClass	Obj2(2);
	MyClass	Obj3(3);

	map<long,MyClass> mapHandleToObjs;

	//Elemente einfügen:
	mapHandleToObjs[11L] = Obj3;	//Handle 11 -> Obj3
	mapHandleToObjs[102L] = Obj1;	//Handle 102 -> Obj1
	mapHandleToObjs[1026L] = Obj2;	//Handle 1026 -> Obj2

	map<long,MyClass>::iterator it;

	it = mapHandleToObjs.find(102L);
	if(it != mapHandleToObjs.end())
	{
		long lkey = (*it).first;		//key
		MyClass Obj = (*it).second;	//value
		mapHandleToObjs.erase(it);	//Element löschen
	}
```

```
    for(it = mapHandleToObjs.begin();
          it != mapHandleToObjs.end();
          ++it)
    {
          if((*it).second.GetID() == 1)
                break;
    }

    return 0;
}
```

Besonderheiten bei map:

- **Indizierter Zugriff führt internes `find()` aus**

  → langsam → keine Alternative zu `vector`!

- **Vor dem Lesen muss man nach dem key-Eintrag suchen**

  Wenn man vor dem Lesen nicht sucht, wird **bei nichtvorhandenem key ein neuer Eintrag generiert** (Default-Konstruktor) und dessen Wert zurückgeliefert, was katastrophale Folgen haben kann.

  Nach dem Suchen sollte man keinen indizierten Zugriff ([]-Operator) mehr durchführen, da dieser wieder ein internes `find()` durchführt:

```
MyClass Obj;
it = mapHandleToObjs.find(102L);
if(it != mapHandleToObjs.end())
      Obj = (*it).second; //nicht Obj = mapIntAndObjs[102L]!!!
```

21.7.4 map: Mehrdimensionaler Schlüssel

Um einen mehrdimensionalen Schlüssel zu verwenden, definiert man am besten eine eigene Klasse, die die notwendigen Operatoren für die `map` enthält:

Beispiel:

```
struct MyTriple
{
      unsigned short x;
      unsigned short y;
      unsigned short z;

      //Initializing (c'tor):
      MyTriple() : x(0),y(0),z(0) {}
      //Assignment:
      MyTriple& operator=(const MyTriple& Obj)
      {
            if(this == &Obj)
                  return *this;
            x = Obj.x;
            y = Obj.y;
            z = Obj.z;
            return *this;
      }
};

//Global operators:

inline bool operator==(const MyTriple& lhs,const MyTriple& rhs)
{
      if(lhs.x != rhs.x)
            return false;
      if(lhs.y != rhs.y)
            return false;
      if(lhs.z != rhs.z)
            return false;
      return true;
}
inline bool operator!=(const MyTriple& lhs,const MyTriple& rhs)
{
      if(lhs == rhs)
            return false;
      return true;
}
```

```
inline bool operator<(const MyTriple& lhs,const MyTriple& rhs)
{
      if(lhs.x < rhs.x)
            return true;
      if(lhs.x > rhs.x)
            return false;
      if(lhs.y < rhs.y)
            return true;
      if(lhs.y > rhs.y)
            return false;
      if(lhs.z < rhs.z)
            return true;
      return false;
}

#include <stdio.h>
#include <map>
using namespace std;
int main()
{
      map<MyTriple,unsigned long>     mapTriple2Long;

      //----- Store: ------

      MyTriple t;
      t.x = 5;
      t.y = 2;
      t.z = 4;

      unsigned long lNumber = 0x44d533a1L;

      mapTriple2Long[t] = lNumber;

      //----- Search (Read): ------

      t.x = 5;
      t.y = 2;
      t.z = 4;

      lNumber = 0L;

      map<MyTriple,unsigned long>::const_iterator it
                                              = mapTriple2Long.find(t);
      if(it != mapTriple2Long.end())
            lNumber = (*it).second;

      return 0;
}
```

21.7.5 vector: Schneller indizierter Zugriff

Der indizierte Zugriff bei `vector` ist effektiver als bei einer `map`, da es sich beim **Index** nicht um ein beliebiges key-Feld handelt, sondern um einen **Integer**, der direkt zur Berechnung der Adresse des Wertes benutzt wird. `vector` ist einem normalen Array vorzuziehen, wenn man die Anzahl der Elemente erst zur Laufzeit kennt (vollkommen dynamische Länge, Heap-Management).

Beispiel:

```
#include <stdio.h>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

void Fill(vector& vectFill)
{
      vectFill.clear();
      vectFill.push_back("1");
      vectFill.push_back("2");
      vectFill.push_back("3");
}
int main()
{
      //Anzahl der Elemente bekannt:
      vector<string> vectEntries(2);
      vectEntries[0] = "Hello";
      vectEntries[1] = " World!";
      string szTest("");
      szTest = vectEntries[0];
      szTest = vectEntries[1];

      //Anzahl der Elemente unbekannt:
      Fill(vectEntries);
      for(int i = 0;i < vectEntries.size();++i)
            szTest = vectEntries[i];

      return 0;
}
```

Besonderheit beim `vector`:

`push_back()` ist immer dem indizierten Schreibzugriff `[]` vorzuziehen

Der indizierte Schreibzugriff erfordert zuvor das Allokieren von Speicher. Wenn also nicht dem Konstruktor als Parameter die Anzahl der Elemente mitgeteilt wurde oder nicht schon durch `push_back()` oder `push_front()` der `vector` auf eine entsprechende Größe gebracht wurde, führt der indizierte Schreibzugriff in einen nicht definierten Speicherbereich (→ '`access violation`').

21.7.6 pair und make_pair(): Wertepaare abspeichern

Möchte man Wertepaare abspeichern, dann kann **`pair`** die richtige Unterstützung bieten. **`make_pair()`** erzeugt effektiv die passenden Wertepaare zum Einfügen.

Beispiel:

```
#include <stdio.h>
#include <algorithm>
#include <list>
using namespace std;

int main()
{
      list<pair<int,int> > listPoints;

      int x = 11;
      int y = 345;

      listPoints.push_back(make_pair(x,y));

      list<pair<int,int> >::iterator it;
      for(it = listPoints.begin();it != listPoints.end();++it)
      {
            int xTest = (*it).first;
            int yTest = (*it).second;
      }
      return 0;
}
```

21.8 hash_map

21.8.1 hash_map für Nutzer von Visual C++

Da nicht alle Compiler, wie z.B. Microsoft Visual C++, eine STL mit **`hash_map`** anbieten, muss man gegebenenfalls eine STL-Implementierung eines Drittanbieters benutzen (z.B. STLport).

21.8.2 Prinzip von hash_map

`hash_map` ist eine Tabelle, in der Objekte abhängig von ihrem **key** abgespeichert werden. Der key bestimmt die Position eines Objektes. Die Funktionsweise einer `hash_map` lässt sich am besten durch einen Vergleich mit einer (normalen) `map` erklären:

`map`:

Liest man über einen key ein Objekt aus, dann wird zunächst in einer Schlüsselfeld-Liste das Schlüsselfeld gesucht (**internes `find()`**), das diesen key enthält. Hat man das Feld gefunden, dann entspricht seine Position (Index) der Position des Objektes in einer weiteren Liste, wo das Objekt dann ausgelesen wird. Das **interne `find()`** macht die Sache insgesamt **langsam**.

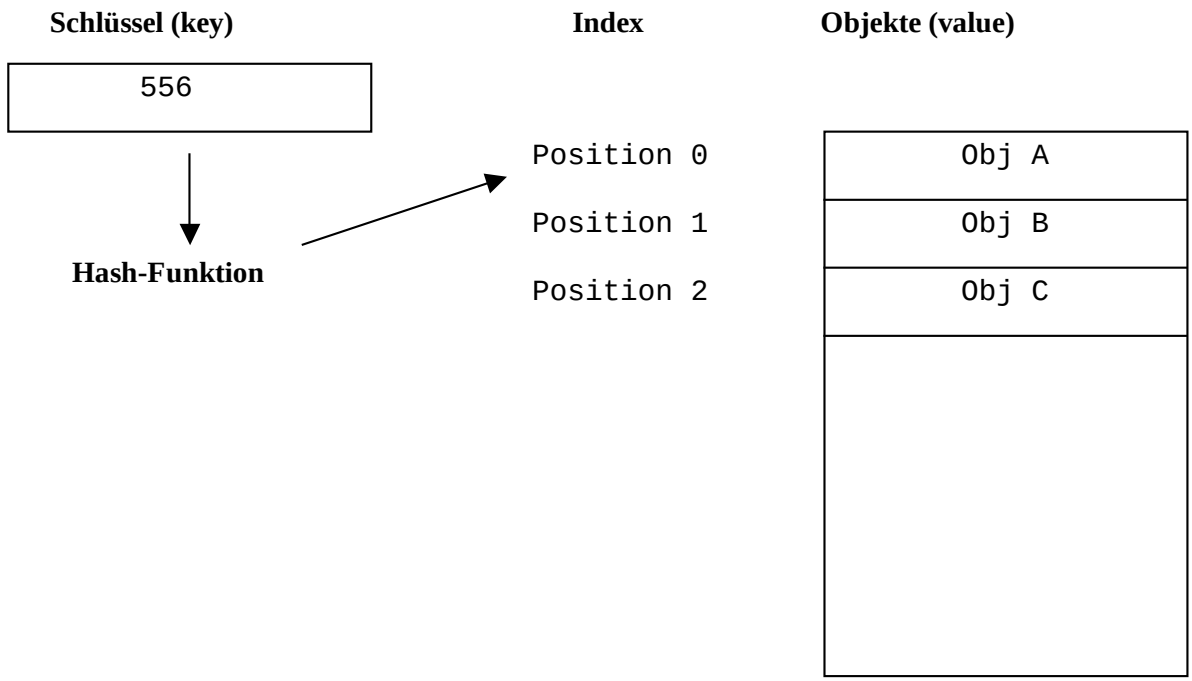
Prinzip der `map`:

Schlüsselfelder (keys)	**Indizes**	**Objekte (values)**
556	Position 0	Obj A
12	Position 1	Obj B
76538	Position 2	Obj C
	...	

hash_map:

Im Unterschied zu einer (normalen) `map` muss man **nicht erst ein Schlüsselfeld suchen**, dessen Position dann den Index für den Objekt-Zugriff liefert, sondern man **errechnet sich den Index direkt über eine sogenannte Hash-Funktion aus dem key** (hash = zerhacken).

Prinzip der `hash_map`:



Die **Hash-Funktion** (`hash<key>`) **sollte einen eindeutigen Index liefern**, d.h. zwei unterschiedliche keys sollten nicht zum gleichen Index (`equal_to<key>`) führen. Ist dies nicht der Fall, wird die Berechnung noch mit einer lokalen Suche gekoppelt. Man hat **keine lückenlos aufgefüllte Liste** wie bei der `map`. Wenn die `hash_map` zu voll wird (ab ca. 75% Füllung), dann wird das Verfahren langsam.

Die Syntax sieht wie folgt aus:

```
hash_map<key,value,hash<key>,equal_to<key> > hmapKeyToValue;
```

Das **`hash<key>`**-Template ist die Hash-Funktion. Das **`equal_to<key>`**-Template sorgt für die Eindeutigkeit der `hash_map`. Im Gegensatz zur `map` kann die `hash_map` **nicht sortieren**, da sie keinen "ist-kleiner"-Vergleich, wie z. B. die `map` mit `less<key>`, durchführt. Man findet lediglich schnell den Eintrag (`value`) zu einem Schlüssel (`key`).

Vorteil:

- Eine `hash_map` ermöglicht einen **schnellen** Objekt-Zugriff, wenn die Hash-Funktion effektiv ist.

Nachteile:

- **Performance-Verlust**, wenn die `hash_map` **zu voll** ist (ab ca. 75%)
  → Abhilfe durch automatische Vergrößerung

- Im Gegensatz zur `map` ist **keine Sortierung** der Schlüssel möglich

- **hash-Funktion** muss **für eigene Schlüssel-Klassen** bereitgestellt werden
  → Benutzt man eine selbst geschriebene Klasse als Schlüssel, dann muss man auch die hash-Funktion selbst programmieren, was nicht zu unterschätzen ist.

21.8.3 Nutzung von hash_map der STL

Es gibt **4 Arten**, die `hash_map` zu nutzen:

- **Keine eigene `hash`- oder `equal_to`-Funktion zur Verfügung stellen**:

  ```
  hash_map<key,value> hmapKeyToValue
  ```

  Diese Methode ist auf jeden Fall zu empfehlen, wenn **`key`** ein **Standard-Typ** ist, denn dann bietet die STL sowohl die `hash`- als auch die `equal_to`-Funktion an.

  Beispiel:

  ```
  int main()
  {
        hash_map<const char*,int> hmapMonthToDays;

        hmapMonthToDays["January"] = 31;
        hmapMonthToDays["February"] = 28;
        hmapMonthToDays["March"] = 31;
        hmapMonthToDays["April"] = 30;
        hmapMonthToDays["May"] = 31;
        hmapMonthToDays["June"] = 30;
        hmapMonthToDays["July"] = 31;
        hmapMonthToDays["August"] = 31;
        hmapMonthToDays["September"] = 30;
        hmapMonthToDays["October"] = 31;
        hmapMonthToDays["November"] = 30;
        hmapMonthToDays["December"] = 31;

        printf("September:                                    %d
        days\n",hmapMonthToDays["September"]);

        return 0;
  }
  ```

- **Eigene `hash`-Funktion zur Verfügung stellen**:

  `hash_map<key,value,MyHash> hmapKeyToValue`

  Diese Methode ist auf jeden Fall erforderlich, wenn man eine selbst geschriebene Klasse als `key` benutzt!

  Beispiel:

  Der `key` wird **aus den ersten 4 Zeichen eines Strings** ermittelt. Man verwendet die 8-Bit-ASCII-Codes der Zeichen zum Errechnen eines Indexes:

  "ABCD…" → $0xD_1D_0C_1C_0B_1B_0A_1A_0$

```
struct MyHash
{
      size_t operator()(const char* key) const
      {
            size_t ret = 0;
            for(int i = 0;i < 4;++i)
            {
                  if(*key == 0)
                        break;
                  ret += (1 << (i * 8)) * (*key++);
            }
            return ret;
      }
};

int main()
{
      hash_map<const char*,int> hmapMonthToDays;

      hmapMonthToDays["January"] = 31;
      hmapMonthToDays["February"] = 28;
      hmapMonthToDays["March"] = 31;
      hmapMonthToDays["April"] = 30;
      hmapMonthToDays["May"] = 31;
      hmapMonthToDays["June"] = 30;
      hmapMonthToDays["July"] = 31;
      hmapMonthToDays["August"] = 31;
      hmapMonthToDays["September"] = 30;
      hmapMonthToDays["October"] = 31;
      hmapMonthToDays["November"] = 30;
      hmapMonthToDays["December"] = 31;

      printf("September:                                    %d
      days\n",hmapMonthToDays["September"]);

      return 0;
}
```

- **Eigene `equal_to`-Funktion zur Verfügung stellen**:

  ```
  hash_map<key,value,hash<key>,MyEqualTo> hmapKeyToValue
  ```

  Beispiel:

  Die Schlüssel sind Strings (`char`-Arrays), die zwar von der STL gehashed, aber über eine eigene Funktion auf Gleichheit überprüft werden.

  ```
  struct MyEqualTo
  {
        bool operator()(const char* s1,const char* s2) const
        {
              return strcmp(s1,s2) == 0;
        }
  };

  int main()
  {
        hash_map<const char*,int,hash<const char*>,MyEqualTo>
                                                  hmapMonthToDays;
        hmapMonthToDays["January"] = 31;
        hmapMonthToDays["February"] = 28;
        hmapMonthToDays["March"] = 31;
        hmapMonthToDays["April"] = 30;
        hmapMonthToDays["May"] = 31;
        hmapMonthToDays["June"] = 30;
        hmapMonthToDays["July"] = 31;
        hmapMonthToDays["August"] = 31;
        hmapMonthToDays["September"] = 30;
        hmapMonthToDays["October"] = 31;
        hmapMonthToDays["November"] = 30;
        hmapMonthToDays["December"] = 31;

        printf(
          "September: %d days\n",hmapMonthToDays["September"]);

        return 0;
  }
  ```

- **Eigene `hash`- und `equal_to`-Funktion zur Verfügung stellen**:

  ```
  hash_map<key,value,MyHash,MyEqualTo> hmapKeyToValue
  ```

  Beispiel:

  Der `key` wird sowohl selbst gehashed (**aus den ersten 4 Zeichen**), als auch auf Gleichheit überprüft.

```
struct MyEqualTo
{
      bool operator()(const char* s1,const char* s2) const
      {
            return strcmp(s1,s2) == 0;
      }
};
struct MyHash
{
      size_t operator()(const char* key) const
      {
            size_t ret = 0;
            for(int i = 0;i < 4;++i)
            {
                  if(*key == 0)
                        break;
                  ret += (1 << (i * 8)) * (*key++);
            }
            return ret;
      }
};
int main()
{
      hash_map<const char*,int,MyHash,MyEqualTo>
                                          hmapMonthToDays;

      hmapMonthToDays["January"] = 31;
      hmapMonthToDays["February"] = 28;
      hmapMonthToDays["March"] = 31;
      hmapMonthToDays["April"] = 30;
      hmapMonthToDays["May"] = 31;
      hmapMonthToDays["June"] = 30;
      hmapMonthToDays["July"] = 31;
      hmapMonthToDays["August"] = 31;
      hmapMonthToDays["September"] = 30;
      hmapMonthToDays["October"] = 31;
      hmapMonthToDays["November"] = 30;
      hmapMonthToDays["December"] = 31;

      printf(
        "September: %d days\n",hmapMonthToDays["September"]);

      return 0;
}
```

21.9 Lokalisierung mit der STL (streams und locales)

Die STL bietet eine C++-Schnittstelle zur Lokalisierung an. Das bedeutet, dass sprach- und länderspezifische Merkmale (z.B. Datum und Preise/Währungen) automatisch für verschiedene Sprach/Land-Kombinationen (locale) richtig formatiert werden.

Hierbei gibt es 3 wesentliche Begriffe, die man kennen sollte:

facet:

Eine bestimmte sprach- und landesspezifische Sache.
Beispiele:

Eingeben der Uhrzeit (`time_get`)
Auslesen und Darstellen der Uhrzeit (`time_put`)

category:

Zusammenfassung von facets, die thematisch zusammengehören.
Beispiele:

`all`	`(LC_ALL)`	→	Alle Kategorien
`collate`	`(LC_COLLATE)`	→	Stringvergleich
`ctype`	`(LC_CTYPE)`	→	Zeichen-Handling
`monetary`	`(LC_MONETARY)`	→	Preisformatierung
`numeric`	`(LC_NUMERIC)`	→	Dezimalpunktformatierung
`time`	`(LC_TIME)`	→	Zeitformatierung
`messages`	`(LC_MESSAGES)`	→	Nachrichtenformatierung

locale:

Zusammenfassung aller facets einer Sprach/Land-Kombination.
Beispiele:

`de_DE@euro`	→	Deutsch/Deutschland + Euro
`en_US`	→	Englisch/USA

Das Prinzip der STL-Lokalisierung sieht so aus, dass die Operationen, die auf einen **Stream** angewendet werden, durch ein **`locale`**-Objekt beeinflusst werden (**imbuing** = durchdringen). Das bedeutet also, dass man auf jeden Fall immer einen **stream** und ein **`locale`-Objekt** benötigt, um diese Lokalisierung auszuführen. Da man bei der Programmierung von Anwendungen mit graphischer Benutzeroberfläche nicht die **iostreams** `cout` und `cin` benutzen wird, kommt bei solchen Anwendungen hauptsächlich der **stringstream** zum Einsatz.

Für einen STL-Hersteller ist es möglich, die STL so zu implementieren, dass sie lediglich die C-Funktion `localeconv()` benutzt, um sich beim Betriebssystem über die Parameter der verschiedenen locales zu informieren. In dem Fall beinhaltet die STL keine eigene locale-Implementierung.

Vorteile der Nutzung von `locale`-Objekten:

- Sprach- und landesspezifische Formatierungen geschehen automatisch.
- Man muss die Implementierung für die Lokalisierung nicht selbst pflegen.
- Wenn die STL-Implementierung `localeconv()` benutzt, um sich beim Betriebssystem über die Parameter der verschiedenen locales zu informieren, dann bleibt die gebaute Software zeitlos.

Nachteile:

- Wenn die STL-Implementierung lediglich `localeconv()` benutzt, also keine eigene locale-Implementierung hat, dann...
  - ...ist in der Regel auch der String-Parameter des Konstruktors von `locale()` betriebssystemabhängig.
  - ...sind die Formatierungen systemabhängig und können je nach System unterschiedlich aussehen.

- Wenn die STL-Implementierung **nicht** `localeconv()` benutzt, sondern eine eigene Implementierung der kompletten Lokalisierung beinhaltet, dann...
  - ...ist die gebaute Software nicht zeitlos, wohl aber der eigene Quell-Code, der lediglich mit einer neuen STL-Implementierung neu gebaut werden muss (bspw. nach der Umstellung einer Währung, wie z.B. "DM" → "EURO")

Beispiel für die locales "**`de_DE`**" und "**`en_US`**":

```
#include <stdio.h>
#include <time.h>

#include <locale> //STL-Lokalisierung
#include <sstream> //string streams
using namespace std;

int main()
{
     printf("Locales:\r\n");

     //Default locale:

     locale localeDefault;
     printf("   Default locale: '%s'\r\n",localeDefault.name());


     //Die category "time" durch diejenige in "de_DE" ersetzen:

     locale localeCustomized(   localeDefault,
                          locale("de_DE"),
                          locale::time);
     printf("   Customized locale: '%s'\r\n",
            localeCustomized.name());
```

```
//Facet 'time_put' der category 'time'
//('struct tm' buffer wird gebraucht):

printf("time_put:\r\n");

timeSystem = time(NULL);
ostringstream ossTime;
ossTime.imbue(locale("de_DE"));
const time_put<char>& tp
          = use_facet<time_put<char> >(ossTime.getloc());

tp.put(ossTime.rdbuf(),
       ossTime,
       ossTime.fill(),
       localtime(&timeSystem),
       'x');
ossTime << " ";

tp.put(ossTime.rdbuf(),
       ossTime,
       ossTime.fill(),
       localtime(&timeSystem),
       'A');
ossTime << " ";

tp.put(ossTime.rdbuf(),
       ossTime,
       ossTime.fill(),
       localtime(&timeSystem),
       'B');
string strTime(ossTime.str());
printf("   Time in '%s': [%s]\r\n",
       ossTime.getloc().name().c_str(),
       strTime.c_str());

//Facet 'ctype' der category 'ctype' (Zeichen-Handling):

printf("ctype:\r\n");
locale locCType("de_DE");
const ctype<char>& ct = use_facet<ctype<char> >(locCType);
ct.tolower(const_cast<char*>(
            strTime.c_str()),strTime.c_str()+strTime.length());
printf("   Time in '%s' - 'lowercase': [%s]\r\n",
       locCType.name().c_str(),
       strTime.c_str());

//Facet 'money_put' der category 'monetary' (Preise/Währungen):

printf("money_put:\r\n");

const string strPriceText("720000000000");
ostringstream ossPriceText;
ossPriceText.imbue(locale("en_US"));
bool bInternational = false;
ossPriceText.setf(ios_base::showbase); //basefield (Währung)
const money_put<char>& mp
     = use_facet<money_put<char> >(ossPriceText.getloc());
```

```
mp.put(ossPriceText.rdbuf(),
       bInternational,
       ossPriceText,
       ' ',
       strPriceText);
string strPrice(ossPriceText.str());
printf("   Price in '%s': [%s]\r\n",
       ossPriceText.getloc().name().c_str(),
       strPrice.c_str());

//Facet 'num_put' der category 'numeric':

printf("num_put:\r\n");

locale locNum("de_DE");
const num_put<char>& np = use_facet<num_put<char> >(locNum);

double dNum = 3827298.77;
ostringstream ossFloatNum;
ossFloatNum.imbue(locNum);
ossFloatNum.setf(ios_base::fixed | ios_base::internal);
ossFloatNum.precision(2); //Anzahl der Stellen nach dem Komma
np.put(ossFloatNum.rdbuf(),ossFloatNum,' ',dNum);
string strNum(ossFloatNum.str());
printf("   Floatingpoint in '%s': [%s]\r\n",
       ossFloatNum.getloc().name().c_str(),
       strNum.c_str());

ostringstream ossScientificNum;
ossScientificNum.imbue(locNum);
ossScientificNum.setf(
                ios_base::scientific | ios_base::floatfield);
ossScientificNum.precision(6); //Anzahl d.Stellen nach d.Komma
np.put(ossScientificNum.rdbuf(),ossScientificNum,' ',dNum);
strNum = ossScientificNum.str();
printf("   Scientific in '%s': [%s]\r\n",
       ossScientificNum.getloc().name().c_str(),
       strNum.c_str());

bool bFlag = false;
ostringstream ossBooleanNum;
ossBooleanNum.imbue(locNum);
ossBooleanNum.setf(ios_base::boolalpha); //bool: "true"/"false"
np.put(ossBooleanNum.rdbuf(),ossBooleanNum,' ',bFlag);
strNum = ossBooleanNum.str();
printf("   Boolean in '%s': [%s]\r\n",
       ossBooleanNum.getloc().name().c_str(),
       strNum.c_str());
```

```
      unsigned long dwHexNum = 0x12F5418E;
      ostringstream ossHexNum;
      ossHexNum.imbue(locNum);
      ossHexNum.setf(ios_base::uppercase); //uppercase, 'f' -> 'F'
      ossHexNum.setf(
            ios_base::hex,ios_base::basefield); //basefield in HEX
      ossHexNum.width(15); //Länge -> 15 Stellen
      np.put(ossHexNum.rdbuf(),ossHexNum,'-',dwHexNum); //Füllz.='-'
      strNum = ossHexNum.str();
      printf("   HEX number in '%s': [%s]\r\n",
             ossHexNum.getloc().name().c_str(),
             strNum.c_str());

      return 0;
}
```

Beispiel für die Implementierung einer "besser lesbaren" Zahlendarstellung:

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

string GetLocalizedStyleNumber(
      const double& dNum,unsigned long dwDigitsAfterDecPoint=0)
{
      locale locNum("en_US");
      const num_put<char>& np = use_facet<num_put<char> >(locNum);
      ostringstream ossFloatNum;
      ossFloatNum.imbue(locNum);
      ossFloatNum.setf(ios_base::fixed | ios_base::internal);
      ossFloatNum.precision(dwDigitsAfterDecPoint);

      np.put(ossFloatNum.rdbuf(),ossFloatNum,' ',dNum);
      return ossFloatNum.str();
}

int main()
{
      string strNum = GetLocalizedStyleNumber(10000000,2);
      printf("Num = %s\r\n",strNum.c_str());
      return 0;
}
```

Hierdurch wird die Zahl 10000000 in der Form

```
            10,000,000.00
```

dargestellt, was wesentlich besser lesbar ist.

22. Arten von Templates

22.1 Class-Template

Vorlage für eine Klasse. Hier gezeigt am Beispiel einer Smart-Pointer-Klasse:

```
template<class T>
class SmartPtr
{
	public:
		SmartPtr(T* pT = NULL) : m_pT(pT) {}
		SmartPtr(SmartPtr<T>& Obj) : m_pT(Obj.GetPtr())
		{ Obj.Release(); }
		~SmartPtr() { delete m_pT; }
		void Release() { m_pT = NULL; }
		T* GetPtr() { return m_pT; }
		template<class C>
		bool Transform(SmartPtr<C>& Obj)
		{
			T* pT = dynamic_cast<T*>(Obj.GetPtr());
			if(!pT)
				return false;
			m_pT = pT;
			return true;
		}
		bool IsValid()
		{
			if(m_pT != NULL)
				return true;
			return false;
		}
		operator bool() { return IsValid(); }
		SmartPtr<T>& operator=(SmartPtr<T>& Obj)
		{
			if(this == &Obj)
				return *this;
			if(m_pT)
				delete m_pT;
			m_pT = Obj.GetPtr();
			Obj.Release();
			return *this;
		}
		T* operator->() { return m_pT; }
		T& operator*() { return *m_pT; }
	private:
		T* m_pT;
};
```

Ein Objekt dieses Templates wird folgendermaßen instanziiert, wenn man den Typ `T = MyClass` benutzt:

```
SmartPtr<MyClass> spMyClass(new MyClass);
```

22.2 Function-Template

22.2.1 Global Function Template

Vorlage für eine globale Funktion. Hier gezeigt am Beispiel der Maximum-Ermittlung:

```
template<class T>
inline const T& MyMax(const T& a,const T& b)
{
	return ((a) > (b) ? (a) : (b));
}
```

Eine solche Funktion wird ganz normal aufgerufen:

```
int i = MyMax(3,4);        //→ i == 4
char c = MyMax('k','l');   //→ c == 'l'
double d = MyMax(2.5,4.6); //→ d == 4.6
```

Der Compiler erkennt den Typ `T` und setzt ihn richtig ein.

22.2.2 Member Function Template

Vorlage für eine Member-Funktion. Ein typisches Beispiel ist die Transform-Funktion des Smart-Pointers:

```
template<class T>
class SmartPtr
{
	public:
		...
		template<class C>
		bool Transform(SmartPtr<C>& Obj);
		{
			T* pT = dynamic_cast<T*>(Obj.GetPtr());
			if(!pT)
				return false;
			m_pT = pT;
			return true;
		}
		...
	private:
		T* m_pT;
};
```

```
template<class T,class C>
inline bool SmartPtr<T>::Transform(SmartPtr<C>& Obj)
{
      T* pT = dynamic_cast<T*>(Obj.GetPtr());
      if(!pT)
            return false;
      m_pT = pT;
      return true;
}
```

22.3 Explizite Instanziierung von Templates

Problem:

Solange ein Template nicht genutzt wird, wird auch kein Code dafür erzeugt. Möchte man Binär-Code in einer `*.obj`-Datei für bestimmte Template-Instanziierungen erzeugen, dann muss man zunächst etwas Pseudo-Code dafür in die entsprechende *.cpp-Datei schreiben.

Abhilfe:

Explizite Instanziierung: Man fügt dem Projekt eine `*.cpp`-Datei hinzu, die die Template-Deklaration und die benötigten Typ-Deklarationen includiert. Dann fügt man eine explizite Instanziierung ein → beim Bau wird eine `*.obj`-Datei mit dem entsprechenden Binär-Code erzeugt.

- **Class-Template explizit instanziieren**

```
#include "MyClass.h"
#include "SmartPtr.h"

template class SmartPtr<MyClass>;
```

- **Global Function Template explizit instanziieren**

```
#include "MyMax.h"

template const int& MyMax<int>(const int&,const int&);
```

Grundsätzlich ist es so, dass **jede Template-Instanziierung nur 1 mal im endgültigen Bau** (also nach dem Linken der `*.obj`-Dateien) vorkommt.

23. Proxy-Klassen

23.1 Allgemeines

Ein Proxy ist ein zwischengeschaltetes Objekt, welches vom Nutzer statt dem eigentlichen Objekt genutzt wird. Der Nutzer merkt jedoch nichts davon, dass er nicht das eigentliche Objekt verwendet. (Proxy = nahe am eigentlichen Objekt).

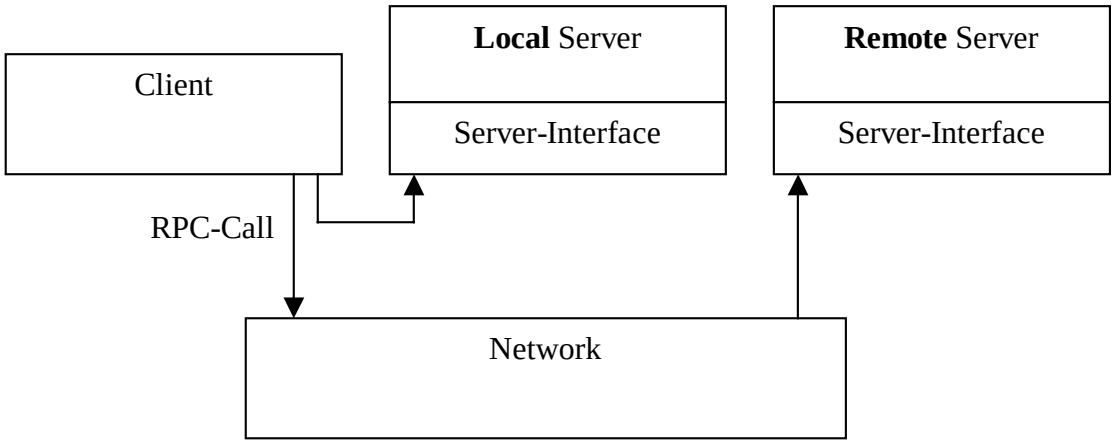
In der Regel möchte man dem Nutzer die Möglichkeit geben, zusätzliche Funktionalität zu nutzen, ohne dass dieser seinen Code dafür umschreiben muss. Ruft der Nutzer bspw. einen Server auf seinem System (localhost) an und man möchte diese Funktionalität auf ein Netzwerk erweiteren (remotehost), dann schaltet man einfach ein Objekt dazwischen (Proxy), welches das gleiche Interface wie der eigentliche Server hat und je nach Adressierung (localhost oder remotehost) die Server-Calls direkt lokal umsetzt (localhost) oder in RPC-Calls verpackt und remote ausführt (remotehost).

Beispiel:

RPC (Remote-Procedure Call):

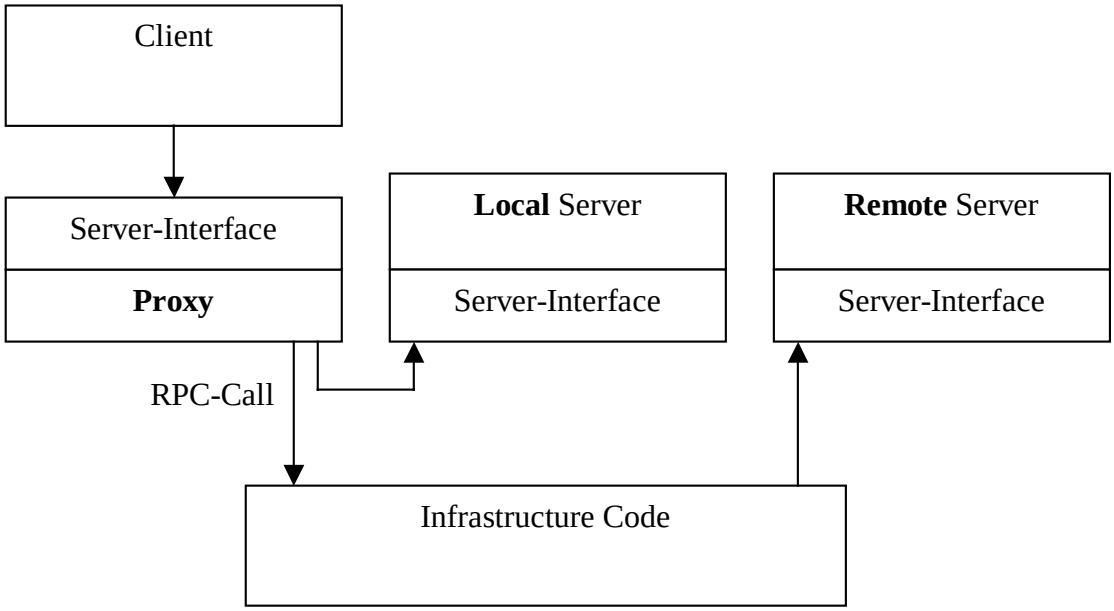
- **Ohne Proxy:**

  Der Client muss genau wissen, ob der Server lokal oder über das Netzwerk (remote) aufgerufen wird, und muss für beides Code bereitstellen.



- **Mit Proxy:**

  Der Client behandelt jeden Server wie einen lokalen Server.



In diesem Beispiel ist der Proxy schon etwas komplex. Noch komplexere Proxies sind die, die eine Firma zwischen ihr Intranet und das Internet schaltet. Dort werden Web-Seiten gecached, gefiltert und vieles mehr. Es gibt aber auch winzig kleine Proxies für prozessinterne Zwecke, wie zum Beispiel für die Zeichen eines String-Objektes zum Unterscheiden von indiziertem Schreib- und Lesezugriff.

23.2 Schreiben/Lesen beim indizierten Zugriff unterscheiden

Statt direkt mit dem `[]`-Operator auf das Datum zuzugreifen, wird ein **Proxy-Objekt dazwischen-geschaltet**. Der `[]`-Operator liefert das Proxy-Objekt und der Compiler versucht über eine **implizite Typumwandlung** dieses Objekt in den Typ umzuwandeln, den die Anweisung im Code verlangt. Da hier zwischen **l-value** und **r-value** unterschieden wird, hat man 2 verschiedene Funktionen für **Schreibzugriff** (**Proxy ist l-value**) und **Lesezugriff** (**Proxy ist r-value**). Bewusst stellt man dem Compiler für die implizite Typumwandlung diese beiden Funktionen zur Verfügung und weiß somit genau, ob ein Schreibzugriff oder ein Lesezugriff stattfindet.

Beispiel:

```
#include <string.h>

class MyString
{
	public:
		//Proxy-Klasse (Stellvertreter für ein Zeichen):
		struct CharProxy
		{
			CharProxy(char* szStr,int nPos,MyString* pParent)
				:	m_szParentStr(szStr),
					m_nPos(nPos),
					m_pParent(pParent)
			{}
			~CharProxy() {}
			char& operator=(char c) //l-value (Schreibzugriff)
			{
				m_pParent->PreWriteAccess();
				m_szParentStr[m_nPos] = c;
				return m_szParentStr[m_nPos];
			}
			operator char() //r-value (Lesezugriff)
			{
				m_pParent->PreReadAccess();
				return m_szParentStr[m_nPos];
			}
			int		m_nPos;
			char*		m_szParentStr;
			MyString*	m_pParent;
		};

		MyString(const char* const szStr)
		{
			strcpy(m_szStr,szStr);
		}

		//Statt dem Zeichen wird ein Proxy-Objekt zurückgegeben:
		CharProxy operator[](int pos)
		{
			return CharProxy(m_szStr,pos,this);
		}

		void PreWriteAccess()
		{
			printf(
				"Indizierter Schreibzugriff folgt jetzt..\n");
		}
		void PreReadAccess()
		{
			printf("Indizierter Lesezugriff folgt jetzt...\n");
		}
	private:
		char m_szStr[1024];
};
```

```
int main()
{
     MyString szHello("Hello");
     char c = szHello[0];  //impl.Typumw.: 'CharProxy' -> 'char'
     szHello[0] = 'W';     //impl.Typumw.: 'CharProxy' -> 'char&'
     return 0;
}
```

Beachte:

- Es ist **keine Übergabe an `char&`-Referenzparameter möglich** (in der Praxis kaum hinderlich)

  ```
  char& r = szHello[1]; //wird nicht übersetzt!
  ```

- **Zeiger auf einzelne Zeichen können möglich gemacht werden**

  ```
  char* p = &szHello[1];
  ```

  → Überladen des &-Operators in `struct CharProxy`:

  ```
  char* operator&()
  {
        return &m_szParentStr[m_nPos];
  }
  ```

  Hier tritt jedoch das Problem auf, dass man nicht mehr sagen kann, es handele sich um einen puren Lesezugriff, denn anschließend hat der Besitzer des Zeigers volle Schreibzugriffmöglichkeit über den Zeiger.

- Benötigte **Operatoren (`+=`, `-=`, usw.) müssen in `struct CharProxy` definiert werden**

  Die String-Klasse ist so natürlich nicht vollständig
  → Operatoren wie `+=`, `-=`, usw. müssen noch implementiert werden.

24. Datenbank-Zugriff

24.1 Zugriff auf objektorientierte Datenbanken

Der Mechanismus des Zugriffs auf objektorientierte Datenbanken sieht in der Regel so aus, dass ein Datenbank-Server für jeden Nutzer der Datenbank nach Aufbau der Verbindung eine eigene **Session** generiert. Alles was innerhalb einer Session geschieht, ist zunächst nicht sichtbar für andere. **Transaktionen** von Daten in die Datenbank werden erst gültig und damit sichbar für andere, nachdem ein "**Commit**" des Nutzers der Session eingeht. Nach dem "Commit" wird anhand von einem **Abhängigkeitsgraphen** jedes in die Transaktion verwickelte Objekt mit einem "**Schreib-Lock**" versehen und danach werden die Änderungen in die Objekte geschrieben und der "Lock" wieder aufgehoben. Ein "Schreib-Lock" verhindert jeglichen Zugriff von anderen Datenbank-Nutzern auf ein Objekt (ausgenommen "Dirty Reads"), während ein "**Lese-Lock**" nur das Schreiben blockiert, aber Lese-Zugriffe anderer Nutzer erlaubt. Der Programmierer kann in der Regel ein Finetuning des Lock-Verhaltens durchführen. So gibt es meherere Isolierungsstufen eines Locks bis hin zur totalen Isolierung eines Objektes, bei dem absolut kein Zugriff von anderen (auch nicht lesend) auf das Objekt möglich ist.

Für den Programmierer heißt das: Innerhalb einer Datenbank-Session ist es möglich, mit **`Store()`**-Befehlen Daten zu schreiben, falls man den Datenspeicher vorher locken konnte (der Datenbank-Server prüft dies entsprechend dem für das Objekt gültigen Abhängigkeitsgraphen). Statt nun alle Store-Operationen **automatisch** im Datenbank-Server als "gültig" zu erklären (AUTOCOMMIT), kann es manchmal sinnvoll sein (bspw. wenn ein Abbruch durch den Anwender erlaubt ist), die **Operationen in Transaktionen einzubetten**, wodurch sie zwar in die Datenbank übertragen werden, jedoch erst als "gültig" erklärt werden, wenn **`Commit()`** aufgerufen wurde. Das Datenbank-Objekt wird dazu in ein **Transaktions-Objekt** eingebettet. Das Transaktions-Objekt stellt die Methode **`Commit()`** zur Verfügung. Im Falle eines Abbruchs ist statt `Commit()` die Methode **`Abort()`** aufzurufen.

Beispiel:

```
DBConn Database("localhost","TestDatabase"); //Verbindung
Transaction Transaction(Database); //Einbettung in Transakt.
try
{
      DatabaseObject DBObj(Transaction);
      DBObj.SetFieldValues(1,2,3,"Test");
      DBObj.Store(); //Lock(), Save(), Unlock()
      Transaction.Commit();
}
catch(...)
{
      Transaction.Abort();
      throw;
}
```

24.2 Zugriff auf relationale Datenbanken

Da relationale Datenbanken sehr populär geworden sind, wurden viele Verfahren zur Ankopplung an diese entwickelt. Unter anderem auch Wrapper, die die relationale Datenbank als eine objektorientierte Datenbank erscheinen lassen und den Zugriff, wie oben erwähnt, ermöglichen. Tatsächlich werden jedoch solche Datenbanken über sogenannte **SQL-Statements** (ASCII-Strings) angesteuert, d.h. es wird ein String zum Datenbank-Server geschickt, der Dinge wie Schreiben ('`INSERT...`', '`UPDATE...`', ...) oder Lesen ('`SELECT...`') bewirkt. Im Falle des Lesens ('`SELECT...`') kann das Ergebnis dann über entsprechende API-Funktionen abgefragt werden.

Der Datenbank-Hersteller stellt **Laufzeitbibliotheken** zur Verfügung, die die SQL-Statements je nach Verbindungsparameter an die richtige Stelle lenken (Local-DB-Server, Remote-DB-Server). Im Falle eines entfernten Servers sind diese Laufzeitbibliotheken meist sehr umfangreich und das dynamische Laden und der Verbindungsaufbau kann eine Weile dauern.

Transaktionen über SQL werden durch das Senden von SQL-Statements (also ASCII-Strings) eingeleitet und abgeschlossen bzw. abgebrochen. Dazu werden die Statements nacheinander an die Datenbank gesendet.

Beispiel:

```
SET AUTOCOMMIT=OFF      (statt OFF ist je nach Datenbank auch schon mal 0 einzusetzen)
INSERT ...
UPDATE ...
...
COMMIT
```

Das **Abschalten von `AUTOCOMMIT`** bewirkt, dass nicht jedes Statement automatisch als gültig erklärt wird, sondern erst nach Eingang von **`COMMIT`**. Statt `COMMIT` ist **`ROLLBACK`** im Falle eines Abbruchs zu benutzen, was in der Tat wiederspiegelt, was in Wirklichkeit in der Datenbank vor sich geht, nämlich das Zurückrollen zum Zustand vor der Transaktion.

Beachte:

Nicht jede relationale Datenbank kann Transaktionen ausführen. So ist z.B. die Datenbank **MySQL** normalerweise nicht transaktionssicher, was aber durch Aktivierung einer daruntergehängten Datenbank-Engine wie **BDB** (Berkeley Database) oder **InnoDB** ermöglicht wird. Um z.B. eine MySQL mit InnoDB unter Linux auf Intel mit GNU zu bauen, muss man die "**max**"-Ausgabe der MySQL

```
mysql-max-3.23.52-pc-linux-gnu-i686.tar.gz
```

haben und vor dem Bau mit

```
./configure --with-innodb
```

konfigurieren. Nach dem Bau ist dann noch die Konfigurationsdatei my.cnf anzupassen (z.B. `innodb_data_file_path=ibdata:30M`). Das Erzeugen einer transaktionssicheren Tabelle ("`CREATE TABLE...`") durch einen Client hat dann mit Angabe des Typs "`InnoDB`" zu geschehen.

Als **Beispiel** sei die Ansteuerung einer **MySQL Datenbank** unter **LINUX** aufgeführt. Um das Beispiel nachvollziehen zu können, benötigt man die Client-Bibliothek des Herstellers (in diesem Fall ist das Code-Archiv dasselbe wie das für den Bau der Datenbank selbst), hier gezeigt für die Version 3.23.52. Der Name "`mysql-3.23.52`" steht für den tatsächlichen Namen des Archivs, wie zum Beispiel "`mysql-3.23.55-pc-linux-gnu-i686.tar.gz`". Man kopiert also die Datei

`mysql-3.23.52.tar.gz`

in ein beliebiges Verzeichnis und geht wie folgt vor:

- Entpacken:

  ```
  tar -xvzf mysql-3.23.52.tar.gz
  ```

- Verzeichnis wechseln:

  ```
  cd mysql-3.23.52
  ```

- Den nachfolgenden Bau als "Client" konfigurieren:

  ```
  ./configure --without-server
  ```

- Bauen und Installieren:

  ```
  make && make install
  ```

- Nun kann man das API in sein Projekt mit aufnehmen:

  `include` Pfad:

  ```
  /usr/local/include/mysql
  ```

  `lib` Pfad

  ```
  /usr/local/lib/mysql
  ```

  mit den Linker-Optionen

  ```
  -lmysqlclient -lz
  ```

  Man kann also bspw. `test.cpp` mit einem GNU C++ Compiler folgendermaßen bauen:

  ```
  g++ -I/usr/local/include/mysql test.cpp \
       -L/usr/local/lib/mysql -lmysqlclient -lz
  ```

Der Code von '**test.cpp**' könnte wie folgt aussehen:

```
#include <stdlib.h>
#include <stdio.h>
#include <mysql.h>

void PrintMySQLError(MYSQL* pMySQLParams)
{
      unsigned long dwErrNo = mysql_errno(pMySQLParams);
      if(dwErrNo)
      {
            printf("Error #%lu_______\n%s\n\n",
                   dwErrNo,
                   mysql_error(pMySQLParams));
      }
}

int main()
{
      //Initialisierung:
      MYSQL MySQLParams;
      if(!mysql_init(&MySQLParams))
      {
            printf("ERROR\n");
            PrintMySQLError(&MySQLParams);
            return 0;
      }

      //Verbindung zum Datenbank-Server:
      if(!mysql_real_connect(    &MySQLParams,
                                 "localhost",    //host
                                 "Peter",        //user
                                 "",             //passwd
                                 "test",         //database
                                 0,         //port, 0 -> default
                                 NULL,      //pointer to unix socket
                                 0))        //clientflag
      {
            printf("ERROR\n");
            PrintMySQLError(&MySQLParams);
            return 0;
      }

      //Ausführen des SQL Statem. (hier: Tabelle 'mytbl' erzeugen):
      mysql_query(
        &MySQLParams,
        "CREATE TABLE mytbl (Name varchar(128),eMail varchar(255))");

      return 0;
}
```

Die zugehörige Make-Datei '**makefile**' würde etwa so aussehen:

```
#
# Linker:
#

test: test.o
      g++ test.o -L/usr/local/lib/mysql -lmysqlclient -lz -otest.exe

#
# Compiler:
#

test.o: test.cpp #dependencies
      g++ -c test.cpp -I/usr/local/include/mysql
```

Datenbank-Ansteuerung mit einem herstellerspezifischen API (C++ Bibliothek) kann für jede Datenbank ein eigenes Buch füllen, das gezeigte Prinzip bleibt jedoch immer gleich. Wie man leicht sehen kann, ist die Kenntnis von **SQL** für den Programmierer relationaler Datenbanken essentiell wichtig. Aus diesem Grund hier ein paar Beispiele (jedes SQL-Statement wird hier mit '**;**' abgeschlossen):

1.) Tabelle erzeugen:

```
create table <TableName>
(
      <ColumnName1> <ColumnDataType1>,
      <ColumnName2> <ColumnDataType2>,
      ...
      <ColumnNameN> <ColumnDataTypeN>
);
```

Beispiel:

```
create table contacts_pt
(
      FirstName varchar(128),
      LastName varchar(128),
      eMail varchar(255)
);
```

2.) Zeile in Tabelle einfügen:

```
insert into <TableName>
      ( <ColumnName1>,<ColumnName2>,...,<ColumnNameN> )
      values
      ( <Value1>,<Value2>,...,<ValueN> );
```

Achtung! String-Werte: `'<string>'`

Beispiel:

```
insert into contacts_pt
      ( FirstName,LastName,eMail )
      values
      ( 'Peter','Thoemmes','p.thoemmes@surf25.de');
```

3.) Inhalt einer Tabelle anschauen:

```
select <ColumnName1>,<ColumnName2>,...,<ColumnNameN>
      from <TableName>;
```

Beispiele:

```
      select FirstName,LastName,eMail from contacts_pt;
      select * from contacts_pt;
```

4.) Einträge suchen und ersetzen:

```
update <TableName>
      set <ColumnName> = <NewValue>
      where <ColumnName> = <NewValue>;
```

Beispiel:

```
update contacts_pt
      set LastName = 'Thoemmes'
      where LastName = 'Thoemmes' and FirstName = 'Peter';
```

5.) Tabelle löschen:

```
drop table <TableName>
```

Auf Windows-Plattformen setzte sich der **Standard ODBC** durch, wobei der Datenbank-Hersteller einen ODBC-Treiber bereitstellt und der Programmierer die SDK-Funktionen des Betriebssystems (`SQL...()`) als **einheitliche Programmierschnittstelle für alle Datenbanken** benutzt. Der **ODBC-Treiber** des Herstellers generiert dann den eigentlichen Zugriff auf die Datenbank. Der große Vorteil dabei ist, dass man Datenbankzugriffe völlig unabhängig von der später angekoppelten Datenbank programmieren kann (abgesehen vom Datenbank-Schema und der Fähigkeit einer Datenbank, Transaktionen auszuführen). Die Datenbank bekommt nur einen symbolischen Namen, den sogenannten **Data Source Name** (**DSN**). Die dahinter stehende Datenbank kann man nach Belieben auswechseln (zumindest theoretisch, denn in der Praxis zeigt sich, dass manche Dinge nicht mit jedem ODBC-Treiber funktionieren).

Ein Problem stellt der **interne Zustand des ODBC-Treibers** dar, der es nicht erlaubt, eine **Transaktion** einfach mit "**`SET AUTOCOMMIT=OFF`**" einzuleiten. Dies hat bei ODBC implizit zu geschehen und wird mit dem Aufruf

```
SQLRETURN sr = SQLSetConnectAttr(
                                hConn,
                                SQL_ATTR_AUTOCOMMIT,
                                (SQLPOINTER) SQL_AUTOCOMMIT_OFF,
                                0);
```

getan. **COMMIT** bzw. **ROLLBACK** werden ebenfalls über spezielle Funktionsaufrufe ausgeführt.

```
SQLRETURN sr = SQLEndTran(SQL_HANDLE_DBC,hConn,SQL_COMMIT);
SQLRETURN sr = SQLEndTran(SQL_HANDLE_DBC,hConn,SQL_ROLLBACK);
```

Da die COM-Technologie (Component Object Model) in alle Bereiche der Windows-Programmierung Einzug hielt, wurde dann auch schließlich das ODBC-API mittels COM-Klassen/Schnittstellen gewrapped, was unter der Bezeichnung **OLE DB** auf den Markt gebracht wurde, im Wesentlichen aber nichts Neues mit sich bringt.

Was jedoch etwas wesentlich Neues mit sich brachte, war der sogenannte Distributed Transaction Coordinator (**DTC**), welcher mittels 2-Phasen-Commit-Protokoll Folgendes fertigbrachte: Eine verteilte Transaktion, also eine Transaktion, bei der Daten an zumindest 2 verschiedene Datenbanken zu übertragen sind, wird entweder vollständig (Commit) oder gar nicht (Abort) durchgeführt. Auf keinen Fall jedoch wird sie nur teilweise ausgeführt. Dies geschieht, indem in der 1. Phase die SQL-Statements an beide Datenbanken übertragen werden, jedoch nicht committed werden. Vielmehr wird jede Datenbank danach gefragt, ob sie in der Lage ist, die Statements problemlos auszuführen. Erst ein von allen Seiten eingehendes "Ready for Commit" erlaubt es dem DTC, die 2. Phase einzuleiten und definitiv die Gültigkeit der Daten bei allen beteiligten Datenbanken zu erklären (Commit). **ODBC**-Verbindungen können in den DTC **enlistet** werden, was sich in dem ODBC-Wrapper (**OLE DB**) "**JoinTransaction**" nennt. Jede enlistete Verbindung wird implizit in den Transaktions-Modus geschaltet (`SET AUTOCOMMIT=OFF`) und der DTC ist die übergeordnete Instanz, die zwar mit `Commit()` oder `Abort()` vom Programmierer angestoßen wird, jedoch letztendlich selbst über einen erfolgreichen Verlauf der Dinge entscheidet.

Beispiel für die Transaktionssteuerung einer ODBC-Verbindung durch den DTC:

Enlist:

```
SQLRETURN sr = ::SQLSetConnectAttr(
                                hConn,
                                SQL_ATTR_ENLIST_IN_DTC,
                                pITransaction,
                                0);
```

Commit:

```
pITransaction->Commit(bRetaining,grfTC,0);
```

Die Idee von verteilten Transaktionen und verallgemeinerten Programmierschnittstellen wie ODBC und OLE DB wurde immer weiter verfeinert, was unter Windows schließlich zur Geburt von **ADO** führte. Es wurde der Dienst **Microsoft Transaction Server** (MTS) erfunden, um einen **Container** (`mtxex.dll`) für **Datenbank-Zugriffs-Objekte** (ADO, ActiveX Data Object) zu beherbergen. Die ADO-Objekte leben in einem völlig eigenständigen Prozess und sind dem Nutzer nur über ein (sprachunabhängiges) IDL-Interface zugänglich. Der große Vorteil bei der Sache ist, dass der Container über das

Ladeverhalten der Komponenten bestimmen kann, d.h. er kann ADO-Objekte im Speicher belassen, obwohl sie nicht mehr gebraucht werden und sie erst verzögert aus dem Speicher laden (**Object-Pooling**). Er kann auch Verbindungen offen halten und erst verzögert schließen oder auch ohne zu schließen durch ein Reconnect auf hohem Level für andere Applikationen wiederverwenden (**Connection-Pooling**). Da das Laden der erforderlichen Laufzeitbibliotheken der Datenbank-Hersteller sowie der Verbindungsaufbau viel Zeit kostet, bringen diese Verfahren einiges an Performance. Das Handling für den Programmierer wird dadurch vereinfacht, dass er ausschließlich über **Smart-Pointer** auf Verbindungen und Daten zugreift. So wird zum Beispiel das Ergebnis eines "SELECT" in einem sogenannten **`_Recordset`**-Objekt zwischengespeichert und für den Aufrufer bereitgehalten, wozu der Programmierer den Smart-Pointer

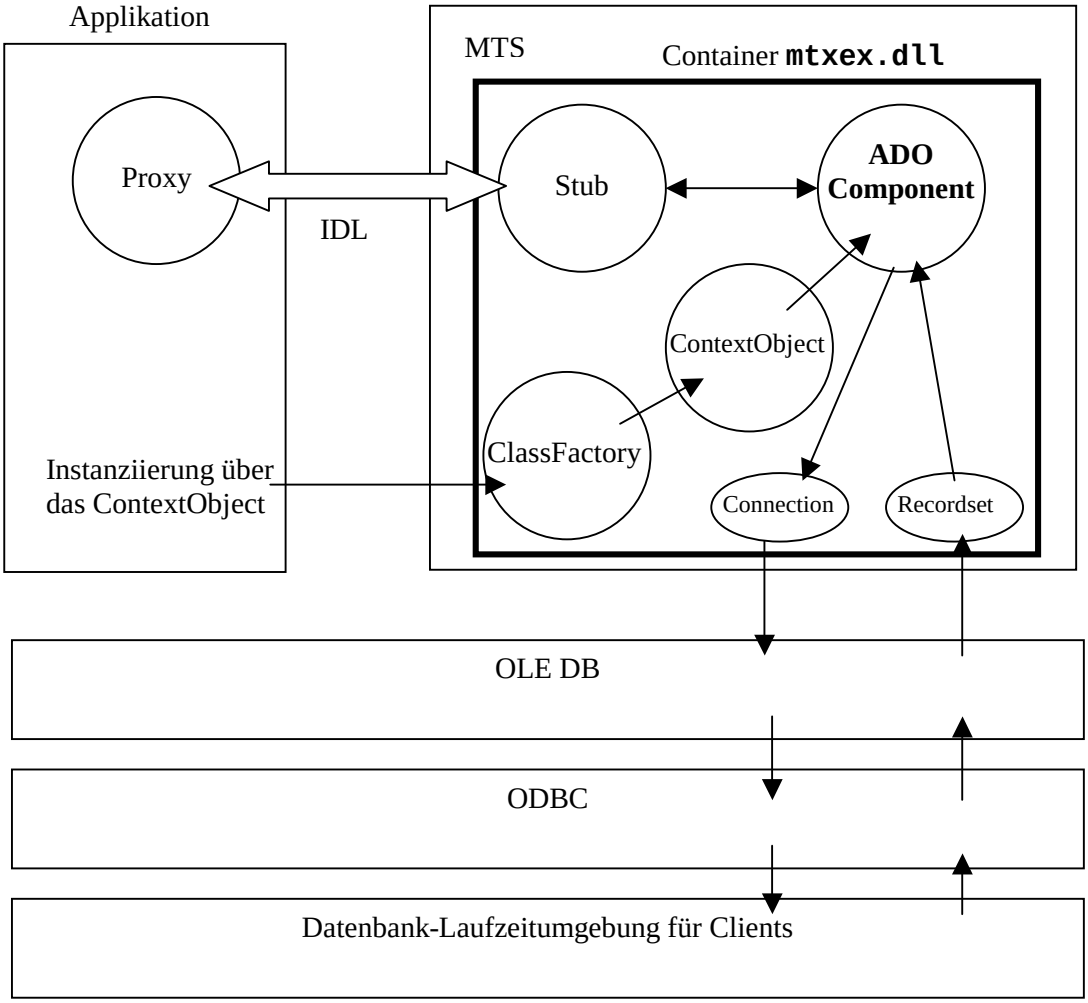
```
_RecordsetPtr spRecordset(__uuidof(adoRecordset));
```

benutzt. Analog dazu benutzt er für die Verbindung ein **`_Connection`**-Objekt, das er über folgenden Smart-Pointer ansteuert:

```
_ConnectionPtr spConnection;
```

Es soll nun in einem Bild verdeutlicht werden, wie das Zusammenspiel der einzelnen Komponenten funktioniert. Dabei ist ODBC als niedrigste Schicht, darüber OLE DB und darüber ADO zu finden. Die ADO-Komponente wird nicht direkt instanziiert. Vielmehr wird zuerst ein **ContextObjekt** über eine **ClassFactory** erzeugt. Dieses erzeugt dann ein Objekt der selbst geschriebenen Klasse, also der **ADO-Komponente**. Danach ist die Inter-Prozess-Kommunikation über das **IDL-Interface** möglich. Programmtechnisch sieht das Ganze so aus, dass auf der Nutzer-Seite ein **Proxy**-Objekt jeden Aufruf einer Methode der ADO-Komponente in einen Bytestrom umwandelt (**Marshalling**), sich dabei für den Nutzer jedoch wie die ADO-Komponente selbst verhält, d.h. der Nutzer merkt nichts davon, dass er sich mit einem Proxy und nicht mit der ADO-Komponente selbst "unterhält". Auf der anderen Seite macht das **Stub**-Objekt (auch **Skeleton** genannt) das gleiche in die andere Richtung, das heißt, es baut aus einem Bytestrom wieder einen Funktionsaufruf zusammen. Für die Rückgabewerte läuft der Vorgang umgedreht ab, d.h. der Stub macht das Marshalling und der Proxy baut die Werte wieder zusammen. Als **Middleware** für diese Umsetzung wird die **COM/DCOM**-Laufzeitumgebung eingesetzt (auch **OLE-Automation** oder **COM+** genannt).

Wird ein "SELECT"-Statement (Query) an die Datenbank geschickt, dann speichert die ADO-Komponente (bei entsprechender Programmierung bzw. Nutzung von `spRecordset`) die selektierten Daten in einem Cache, dem Recordset ab. Dort befinden sich dann alle Zeilen (Rows, Records), die die Antwort auf die Anfrage (Query) bilden. Es ist möglich, einen echten Zeiger auf das Recordset bis in die Applikation durchzureichen, was erfordert, dass die Applikation den Datentyp `Recordset` kennen muss. Dadurch wird jedoch die klare Trennung zwischen Datenbankzugriff (Tier n-1) und Applikation (Tier n) aufgehoben und eine saubere n-Tier-Architektur zerstört.



Zusammengefasst bringt ADO also folgende Vorteile:

- **Object-Pooling**
  ADO-Objekte im Speicher belassen, obwohl sie nicht mehr gebraucht werden und sie erst verzögert aus dem Speicher laden.
- **Connection-Pooling**
  Verbindungen offen halten und erst verzögert schließen oder auch ohne zu schließen durch ein Reconnect auf hohem Level für andere Applikationen wiederverwenden.
- **Distributed Transaction Coordinator (DTC)**
  Automatisches Abwickeln von verteilten Transaktionen

Anmerkung:

Es gibt auch für andere Plattformen ODBC-Umgebungen (Bsp.: Linux). Auch für die später entwickelte Java-Plattform (JVM, Java Virtual Machine) wurde etwas Analoges unter dem Namen **JDBC** entwickelt. Die Idee des MTS wurde dort als **JTS** (Java Transaction Server) nachgebaut.

C++ (WINDOWS)	JAVA (JVM)
ODBC	JDBC
MTS	JTS
Container mtxex.dll	EJB-Container
ADO Component	EJB
ContextObject	ContextObject
ClassFactory	HomeObject
COM/DCOM-Laufzeitumgebung	ApplicationServer
ORPC über DCE RPC über TCP/IP	RMI / IIOP über TCP/IP

Der Container wird dort **EJB-Container** genannt und die Komponenten **EJBs** (Enterprise Java Beans). Die ClassFactory bekam den Namen HomeObject und die Middleware heißt nicht COM/DCOM, sondern **ApplicationServer**.

24.3 Zugriff auf hierarchische Datenbanken

Der Vollständigkeit halber soll hier kurz der Zugriff auf hierarchische Datenbanken skizziert werden. Die Daten in einer hierarchischen Datenbank sind in einer Baumstruktur angeordnet. Das heißt, ein Schlüssel setzt sich zusammen aus einem Haupt-Schlüssel und ein oder mehreren Sub-Schlüsseln, wie ein Pfad bei einem Verzeichnisbaum einer Festplatte. Möchte man also einen Wert lesen oder schreiben, dann muss man zunächst seinen vollständigen Schlüssel zusammenbauen.

Beispiel: **Windows NT Registry**

Das Betriebssystem bietet threadsichere Funktionen für das Lesen und Schreiben der Werte an. Für String-Werte heißen die Funktionen `WriteProfileString()` und `GetProfileString()`.

Es gibt folgende Hauptschlüssel:

```
HKEY_CLASSES_ROOT:
      Mapping von ProgID (String) zu ClassID (GUID) von COM-Modulen
HKEY_USERS:
      Konfigurationen der konfigurierten Benutzer
      →    HKEY_CURRENT_USER enthält eine Kopie aus HKEY_USERS,
           die nach dem Hochlauf für den angemeldeten Benutzer
           angelegt wird
HKEY_LOCAL_MACHINE:
      Hardware-Konfiguration
      →    HKEY_CURRENT_CONFIG enthält eine Kopie aus
           HKEY_LOCAL_MACHINE, die für PnP (Plug & Play) gebraucht
           wird, speziell wenn mehrere Konfigurationen möglich sind.
```

Ein vollständiger Schlüssel könnte hier z.B. folgendermaßen aussehen:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSDTC\MTxOCI
```

25. Aktion nach Kollision über Objekttyp steuern

Problem:

Eine Funktion (Handler) soll das Aufeinandertreffen von 2 Objekten handeln. Die auszuführende Aktion soll dabei davon abhängen, welche Objekt-Typen aufeinandertreffen.

Lösung:

Zunächst schafft man sich eine Handler-Map, die alle möglichen Handler aufnimmt:

```
class Base; //Forward-Deklaration

typedef void (*HitFuncPtr)(Base&,Base&);

class HandlerMap
{
      public:
            static void AddEntry(
                        int nID1,int nID2,HitFuncPtr Handler);
            static HitFuncPtr Lookup(int nID1,int nID2);
      private:
            HandlerMap(); //Konstruktor verstecken
            HandlerMap(const HandlerMap&); //Copy-Konstr.
verstecken
      private:
            static map<long,HitFuncPtr>& theHandlerMap();
};
```

In diese Map kann man für jede mögliche Kombination nID1 mit nID2 einen Handler, z.B.

```
void Handler13(Base& Obj1,Base& Obj3)
{
      ...
}
```

aufnehmen:

```
HandlerMap::AddEntry(1,3,&Handler13);
```

Die Klassen aller Objekte werden von folgender abstrakter Basisklasse abgeleitet:

```
class Base
{
      public:
            virtual ~Base() {}
            virtual void Handler(Base& Partner) = 0;
            virtual int GetID() const = 0;
};
```

Hier ein Beispiel:

```
class Child1 : public Base
{
      public:
            void Handler(Base& Partner)
            {
                  HitFuncPtr hfHandler
                        = HandlerMap::Lookup(m_nID,Partner.GetID());
                  if(hfHandler != NULL)
                        hfHandler(*this,Partner);
            }
            virtual int GetID() const { return m_nID; }
      private:
            enum{ m_nID = 1 };
};
```

Eine **Kollision** sieht wie folgt aus:

```
Child1 Obj1;
Child2 Obj2;
Child1.Handler(Child2); //1 kollidiert mit 2
```

Es passieren also 2 Dispatches (**Double Dispatching**):

- Erster Dispatch:

  Der Kollisions-Partner `Obj2` wird an den Handler des kollidierenden Objektes `Obj1` weitergeleitet.

- Zweiter Dispatch:

  Der Handler des kollidierenden Objektes `Obj1` sucht in der Handler-Map den Kollisions-Handler für die Paarung `Obj1` mit `Obj2` und gibt sich selbst und den Partner an den Handler weiter.

Und hier nun das komplette Beispiel 'am Stück':

```
//--------- STL: ----------------------------------------------

#include <stdio.h>
#include <map>
using namespace std;
```

```
//--------- HandlerMap: ----------------------------------------

class Base; //Forward-Deklaration

typedef void (*HitFuncPtr)(Base&,Base&);

class HandlerMap
{
      public:
            static void AddEntry(
                              int nID1,int nID2,HitFuncPtr Handler);
            static HitFuncPtr Lookup(int nID1,int nID2);
      private:
            HandlerMap();                    //Konstruktor verstecken
            HandlerMap(const HandlerMap&);   //Copy-Konstr. verstecken
      private:
            static map<long,HitFuncPtr>& theHandlerMap();
};

map<long,HitFuncPtr>& HandlerMap::theHandlerMap()
{
      static map<long,HitFuncPtr> m_mapHandler;//eigentl. Handler-Map
      return m_mapHandler;
}

void HandlerMap::AddEntry(int nID1,int nID2,HitFuncPtr Handler)
{
      long lID = (nID1 << 16) + nID2;
      theHandlerMap()[lID] = Handler;
}

HitFuncPtr HandlerMap::Lookup(int nID1,int nID2)
{
      long lID = (nID1 << 16) + nID2;
      map<long,HitFuncPtr>::iterator it = theHandlerMap().find(lID);
      if(it != theHandlerMap().end())
            return (*it).second;
      return NULL;
}
```

```
//--------- Kollidierende Objekte: ----------------------------

class Base
{
      public:
            virtual ~Base() {}
            virtual void Handler(Base& Partner) = 0;
            virtual int GetID() const = 0;
};
class Child1 : public Base
{
      public:
            void Handler(Base& Partner)
            {
                  HitFuncPtr hfHandler
                        = HandlerMap::Lookup(m_nID,Partner.GetID());
                  if(hfHandler != NULL)
                        hfHandler(*this,Partner);
            }
            virtual int GetID() const { return m_nID; }
      private:
            enum{ m_nID = 1 };
};
class Child2 : public Base
{
      public:
            void Handler(Base& Partner)
            {
                  HitFuncPtr hfHandler
                        = HandlerMap::Lookup(m_nID,Partner.GetID());
                  if(hfHandler != NULL)
                        hfHandler(*this,Partner);
            }
            virtual int GetID() const { return m_nID; }
      private:
            enum{ m_nID = 2 };
};
class Child3 : public Base
{
      public:
            void Handler(Base& Partner)
            {
                  HitFuncPtr hfHandler
                        = HandlerMap::Lookup(m_nID,Partner.GetID());
                  if(hfHandler != NULL)
                        hfHandler(*this,Partner);
            }
            virtual int GetID() const { return m_nID; }
      private:
            enum{ m_nID = 3 };
};
```

```
//--------- Handler: ------------------------------------------

void Print(int nIDA,int nIDB)
{ printf("%d kollidiert mit %d\n",nIDA,nIDB); }

void Handler11(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler12(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler13(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler21(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler22(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler23(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler31(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler32(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler33(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }

//--------- main(): -------------------------------------------

int main()
{
     HandlerMap::AddEntry(1,1,&Handler11);
     HandlerMap::AddEntry(1,2,&Handler12);
     HandlerMap::AddEntry(1,3,&Handler13);
     HandlerMap::AddEntry(2,1,&Handler21);
     HandlerMap::AddEntry(2,2,&Handler22);
     HandlerMap::AddEntry(2,3,&Handler23);
     HandlerMap::AddEntry(3,1,&Handler31);
     HandlerMap::AddEntry(3,2,&Handler32);
     HandlerMap::AddEntry(3,3,&Handler33);

     Child1 Obj1;
     Child2 Obj2;
     Child3 Obj3;

     Obj1.Handler(Obj2);
     Obj2.Handler(Obj1);
     Obj3.Handler(Obj1);

     return 0;
}
```

26. 80/20-Regel und Performance-Optimierung

26.1 Allgemeines

Die 80/20-Regel gibt es in mehreren Bereichen der Technik. Sie ist empirisch ermittelt und gilt zumindest bezüglich der Größenordnung.

Folgende Zusammenhänge kann man somit in Zahlen ausdrücken:

- 80% der Laufzeit verbringt ein Programm in **20% des Codes**
- 80% des Speicherbedarfs eines Programms wird von **20% des Codes** genutzt
- 80% der Festplatten-Zugriffe erfolgen durch **20% des Codes**
- 80% der Wartung wird in **20% des Codes** durchgeführt

Die **schwierige Aufgabe** hierbei ist es, die **20% des Codes ausfindig zu machen**.

Falscher Weg:	Vermutungen
Richtiger Weg:	Nicht intuitiv, sondern mittels Code-Review, Einbau von Hilfs-Code (wie `printf()` oder Zählvariablen) und Unterstützung eines **PROFILER** suchen

Man benötigt also ggf. einen PROFILER. Und zwar solch einen, der genau die Ressource (Zeit oder Speicher) untersuchen kann, für die man sich interessiert:

- Programm ist **zu langsam**
  → PROFILER muss aufzeigen können, wieviel **Zeit** in den verschiedenen Code-Abschnitten verbracht wird
- Programm ist **zu speicherintensiv**
  → PROFILER muss aufzeigen können, wieviel **Speicher** von den verschiedenen Code-Abschnitten gebraucht wird. Alternativ hierzu kann man sich die **Anzahl der `new`/`delete`-Aufrufe** anzeigen lassen.

PROFILING-Beispiel (nur skizziert):
Eine Datei namens **`test.cpp`** soll entsprechend vorbereitet, ausgeführt und dann analysiert werden. Das Ergebnis soll in die Datei `report.txt` geschrieben werden:

GNU C++-Compiler:

```
g++ -pg test.cpp -otest.exe
./test.exe
gprof ./test.exe > report.txt
```

Microsoft Visual C++-Compiler:

```
cl test.cpp /link /profile
prep /OM /FT test.exe
profile test.exe
prep /m test
plist /PW 1000 /SC test /FLAT > report.txt
```

26.2 Zeit-Optimierungen

26.2.1 return so früh wie möglich

Am Anfang einer Methode sind erst alle Argumente und sonstige Parameter zu prüfen. Die Methode so früh wie möglich verlassen, wenn sie nicht bearbeitet werden muss.

Beispiel:

```
MyClass& MyClass::operator=(const MyClass& Obj)
{
      if(this == &Obj)
            return *this;
      ...
      return *this;
}
```

26.2.2 Präfix-Operator statt Postfix-Operator

Objekte (z.B. Iteratoren) sind immer bevorzugt mit Präfix-Operatoren aufzurufen, da diese schneller sind, weil sie nicht mit einem **temporären Objekt** arbeiten müssen.

Beispiel:

```
MyClass& MyClass::operator++()              //-> Präfix (++Obj)
{
      ++m_nID;
      return *this; //Rückgabe einer Referenz auf *this
}
const MyClass MyClass::operator++(int)      //-> Postfix (Obj++)
{
      MyClass Obj(m_nID); //temporäres Objekt
      ++(*this);
      return Obj; //Rückgabe eines Objektes per Wert
}
```

26.2.3 Unäre Operatoren den binären Operatoren vorziehen

Man sollte die unären Operatoren immer den binären vorziehen, da sie entweder genauso gut oder performanter als die binären sind. Bei der Verwendung von unären Operatoren bestimmt man jede Operation selbst und es verstecken sich nicht weitere Operationen dahinter.

Beispiel:

`Obj += 4;` **ist effektiver als** `Obj = Obj + 4;`

Grund:

`Obj += 4;`	wird zu:	`Obj.operator+=(4);`
`Obj = Obj + 4;`	wird zu:	`Obj.operator=(Obj.operator+(4));`

26.2.4 Keine Konstruktion/Destruktion in Schleifen

Man sollte temporäre Objekte, die in Schleifen benötigt werden, vor der Schleife konstruieren. Innerhalb der Schleife wird dann immer dieses Objekt benutzt und lediglich der Inhalt (Wertzuweisung) überschrieben.

Beispiel:

Statt:

```
for(int i = 0;i < 200; ++i)
{
      MyClass Obj(i); //-> Konstrukt.+ Zuweisung (Copy-Konstr.)
      ...
} //-> Destruktion
```

→ **200** Konstruktionen, **200** Zuweisungen und **200** Destruktionen

Besser:

```
MyClass Obj;    //-> Konstruktion (Default-Konstruktor)
for(int i = 0;i < 200; ++i)
{
      Obj = i; //-> Zuweisung
      ...
}
```

→ **1** Konstruktion und **200** Zuweisungen

26.2.5 hash_map statt map, falls keine Sortierung benötigt wird

Da die **`hash_map`** **schneller** findet als die `map`, sollte man sie immer dann anwenden, wenn es darum geht, schnell etwas zu finden, aber zugleich **keine** Sortierung der keys erforderlich ist.

26.2.6 Lokaler Cache um Berechnungen/Datenermittlungen zu sparen

Wenn eine große Menge an Berechnungen bzw. Datenermittlungen (bspw. aus Tabellen) mehrmals durchgeführt werden muss, dann rentiert sich oft die Implementierung eines lokalen Caches mit Hilfe einer **`static-hash_map`**:

Beispiel:

```
#include <stdio.h>
#include <hash_map>
#include <string>
using namespace std;
```

```
class MyTable
{
	public:
		static string GetItem(long lTypeNum)
		{
			static hash_map<long,string> hmapTable;
			static bool bNotInitialized = true;

			if(bNotInitialized)
			{
				string szEntry("Entry No.");
				char szNo[256];
				for(long l = 0;l < 100;++l)
				{
					ltoa(l,szNo,10);
					hmapTable[l] = szEntry + szNo;
				}
				bNotInitialized = false;
			}

			hash_map<long,string>::iterator it
				= hmapTable.find(lTypeNum);
			if(it != hmapTable.end())
				return (*it).second;
			return string("");
		}
	private:
		MyTable();                      //verstecken
		MyTable(const MyTable& Obj);    //verstecken
		~MyTable();                     //verstecken
};

struct MyReader
{
	string operator()(long lTypeNum) const
	{
		static hash_map<long,string> hmapTableCache;

		//zunächst im Cache suchen:
		hash_map<long,string>::iterator it
			= hmapTableCache.find(lTypeNum);
		if(it != hmapTableCache.end())
		{
			printf("Cache-Hit bei [%ld]\n",lTypeNum);
			return (*it).second;
		}

		//Wert ermitteln und in den Cache aufnehmen:
		string szEntry = MyTable::GetItem(lTypeNum);
		hmapTableCache[lTypeNum] = szEntry;

		return szEntry;
	}
};
```

```
int main()
{
	MyReader Reader;
	string szTest("");
	szTest = Reader(1);
	szTest = Reader(2);
	szTest = Reader(3);
	szTest = Reader(1); //Cache-Hit
	szTest = Reader(2); //Cache-Hit
	szTest = Reader(3); //Cache-Hit
	return 0;
}
```

26.2.7 Löschen nach find() immer direkt über den Iterator

Das **Löschen über einen Wert** erfordert zunächst immer ein **internes `find()`** zum Suchen der Speicherstelle, die diesen Wert beinhaltet. Wenn man gerade erst `find()` ausgeführt hat, um herauszufinden, ob sich das Element überhaupt in der Sequenz befindet, dann verschwendet man nochmal die Zeit für ein internes `find()`, wenn man über den Wert statt über den Iterator löscht.

Beispiel:

```
class MyClass
{
	public:
		explicit MyClass(int nID) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};


#include <stdio.h>
#include <algorithm>
#include <list>
using namespace std;
```

```
int main()
{
	MyClass Obj1(1);
	MyClass Obj2(2);
	list<MyClass> listObjs;
	listObjs.push_back(Obj1);
	listObjs.push_back(Obj2);
	listObjs.push_back(Obj1);
	int nFirstDeletedID = 0;
	list<MyClass>::iterator it;
	it = find(listObjs.begin(),listObjs.end(),Obj1);
	if(it != listObjs.end())
	{
		nFirstDeletedID = (*it).GetID()
		listObjs.erase(it); //nicht listObjs.erase(*it)!!!
	}
	return 0;
}
```

26.2.8 map: nie indizierten Zugriff [ ] nach find() durchführen

Der **indizierte Zugriff** auf eine map (`operator[]`) erfordert zunächst immer ein **internes `find()`** zum Suchen des `key`, um aus dessen Position auf die Position des `value` zu schließen. Wenn man gerade erst `find()` ausgeführt hat, um zu wissen, ob sich der `key` in der Sequenz befindet, dann verschwendet man nochmal die Zeit für ein internes `find()`.

Beispiel:

```
class MyClass
{
	public:
		explicit MyClass(int nID = 0) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};
```

```
#include <stdio.h>
#include <algorithm>
#include <map>
using namespace std;

int main()
{
	MyClass Obj1(1);
	MyClass Obj2(2);
	map<long,MyClass> mapHandleToObj;
	mapHandleToObj[10L] = Obj1;
	mapHandleToObj[300L] = Obj2;
	MyClass DeletedObj(0);
	map<long,MyClass>::iterator it = mapHandleToObj.find(10L);
	if(it != mapHandleToObj.end())
	{
		DeletedObj = (*it).second;
			//nicht DeletedObj = mapHandleToObj[10L]!!!
		mapHandleToObj.erase(it);
			//nicht mapHandleToObj.erase(*it)!!!
	}
	return 0;
}
```

26.2.9 Unsichtbare temporäre Objekte vermeiden

In folgenden Fällen werden unsichtbare temporäre Objekte erzeugt, d.h. es findet eine unsichtbare Konstruktion und eine Destruktion statt:

- **Übergabe eines Funktions-Argumentes <u>per Wert</u>**

  → Es wird mit einem temporären Objekt statt mit dem Originalparameter gearbeitet.

  Abhilfe:

Parameter per `const`-**Referenz** übergeben!

  Beispiel:

  Statt:

  ```
  Foo(list<long> listIDs);
  ```

  Besser:

  ```
  Foo(const list<long>& listIDs);
  ```

- **Objekte als `return`-Wert einer Funktion**

  → Es wird ein temporäres Objekt für die Rückgabe erzeugt.

  Abhilfe:

Rückgabe eines **Konstruktors** statt eines Objektes!

  Die "**return-value-optimization**" des Compilers kann den Konstruktoraufruf direkt an das Objekt des Aufrufers weitergeben, ohne ein temporäres Objekt zu erzeugen.

  Beispiel:

  Statt:

  ```
  MyClass Obj(14);
  return Obj;
  ```

  Besser:

  ```
  return MyClass(14);
  ```

- **Übergabe eines Funktions-Argumentes vom 'falschen' Typ**

  → Es wird über eine **implizite Typumwandlung** ein temporäres Objekt erzeugt und übergeben.

  Beispiel:

  ```
  void f(const MyClass2& Obj)
  {
       int i = Obj.GetID();
  }

  int main()
  {
       MyClass1 Obj;
       f(Obj);
       return 0;
  }
  ```

Eine implizite Typumwandlung findet in folgenden Fällen statt:

- `MyClass2` bietet einen **nicht-explicit-Konstruktor** mit **genau 1 Argument** vom Typ `MyClass1` an:

```
class MyClass2
{
      public:
            explicit MyClass2(int nID = 0) : m_nID(nID) {}
            MyClass2(const MyClass1& Obj)
                  : m_nID(Obj.GetID()) {}
            int GetID() const { return m_nID; }
      private:
            int m_nID;
};

class MyClass1
{
      public:
            explicit MyClass1(int nID = 0) : m_nID(nID) {}
            int GetID() const { return m_nID; }
      private:
            int m_nID;
};
```

- `MyClass1` bietet einen **Typumwandlungs-Operator** nach `MyClass1` an:

```
class MyClass2
{
      public:
            explicit MyClass2(int nID = 0) : m_nID(nID) {}
            int GetID() const { return m_nID; }
      private:
            int m_nID;
};

class MyClass1
{
      public:
            explicit MyClass1(int nID = 0) : m_nID(nID) {}
            int GetID() const { return m_nID; }
            operator MyClass2() { return MyClass1(m_nID); }
      private:
            int m_nID;
};
```

Abhilfe:

Funktionen für alle Argument-Typen **überladen**!

Beispiel:

```
void fKernel(int nValue)
{
      int i = nValue;
}

void f(const MyClass1& Obj)
{
      fKernel(Obj.GetID());
}

void f(const MyClass2& Obj)
{
      fKernel(Obj.GetID());
}
```

26.2.10 Berechnungen erst dann, wenn das Ergebnis gebraucht wird

Wenn eine große Menge an Berechnungen möglich ist, dann werden oft nicht alle Ergebnisse benötigt. Man kann das natürlich nicht vorher abfragen. Aber man kann die Software so gestalten, dass sie immer erst dann ein Ergebnis berechnet, wenn es wirklich benötigt wird (Lazy Evaluation).

→ **Keine Berechnungen auf Vorrat!**

26.2.11 Datenermittlung erst dann, wenn die Daten gebraucht werden

Wenn eine große Menge an Daten zur Verfügung steht, dann werden oft nicht alle Daten benötigt. Man sollte die Software so gestalten, dass Daten erst dann ermittelt werden, wenn sie gebraucht werden (Lazy Fetching). Dies gilt vor allem bei zeitintensiven **Tabellen- oder Datenbank-Zugriffen** vor einer graphischen oder textuellen Ausgabe.

→ **Keine Datenermittlung auf Vorrat!**

26.2.12 Große Anzahl kleiner Objekte blockweise lesen (Prefetching)

Wenn man vorher **weiß**, dass eine große Anzahl Objekte gelesen werden muss (was beim Lazy-Fetching ausdrücklich nicht der Fall ist), dann kann man meist effektiver lesen, wenn man dies blockweise tut. **Tabellen- oder Datenbank-Zugriffe** sollten entsprechende Schnittstellenfunktionen anbieten.

Beispiel: Stückliste einer elektronischen Schaltung

Statt:

```
long lCircuitNo = 12;
list<long> listItems;
for(long l = 0;l < GetItemCount(lCircuitNo);++l)
     listItems.push_back(GetItem(lCircuitNo,l));
```

Besser:

```
long lCircuitNo = 12;
list<long> listItems = GetAllItems(lCircuitNo);
```

26.2.13 Kein unnötiges Speichern in die Datenbank

Das Speichern (`Store()`, `ExecuteSQLStatement()`) in die Datenbank erfordert je nachdem wie es geschieht mehr oder weniger **Datenbank-Server-Calls** und mehr oder weniger **Festplattenzugriffe** auf dem eigenen System, in beiden Fällen also CPU-Zeit-Verbrauch:

> Die Datenbank-Schnittstelle leitet den Zugriff an den eigentlichen Datenbank-Client des DBMS (Database Management Systems) weiter. Dort wird dann entweder ein **direktes lokales Speichern** (LOCAL → nur Festplattenzugriffe) oder ein **lokales Speichern über einen lokalen Server** (localhost → Server-Calls und Festplattenzugriffe) oder Speichern über Netzwerkzugriff auf einen Remote-Server (remotehost → nur Server-Calls) durchgeführt.

Man sollte also nur so oft in die Datenbank schreiben, wie es die Sicherheit des Datenbank-Systems erfordert, d.h. nur so oft, dass die Recovery-Mechanismen (Rollback) noch sicher funktionieren.

26.2.14 SQL-SELECT-Statements effektiv aufbauen: DB-Server filtern lassen

Man sollte immer soviel Filterung wie möglich vom Datenbank-Server vornehmen lassen, bevor dieser die gewünschten Daten herüberreicht.

Falsch ist es auf jeden Fall

```
SELECT * FROM mytable;
```

anzufragen und dann im eigenen Program-Code nach der Zeile mit **`ID=5`** zu suchen, um dann die Spalte **`name`** zu lesen. Der Server würde dann alle Zeilen mit voller Breite (also auch alle Spalten) an den Aufrufer übertragen, was enorm viel Zeit kosten könnte.

Besser wäre also in diesem Fall die Anfrage

```
SELECT name FROM mytable WHERE ID=5;
```

Hierdurch macht der Server eine Vorselektierung aller Zeilen mit `ID=5` und liefert dann nur die Spalte `name` zurück. Wenn die ID eindeutig ist, wird also genau 1 Feld zurück an den Aufrufer gesendet.

Bei großen operationellen Datenbanken kann das Datenbank-Schema sehr komplex sein. In dem Fall kann es unter Umständen sehr hilfsreich sein, zusammen mit dem DBA (Datenbank-Administrator) SELECT-Statements für die einzelnen Anfragen auszuarbeiten. Durch die Möglichkeit der Verschachtelung und der UND-Verknüpfung (`AND`) von Bedingungen (`WHERE`) kann man sehr viel Performance-Optimierung betreiben.

26.3 Speicher-Optimierungen

26.3.1 Sharing von Code und/oder Tabellen mittels statischem Objekt

Statt viele gleiche Kopien eines Objektes (**Code** und/oder **Tabellen**) zu machen, wird **nur ein statisches Objekt** für alle Nutzer zur Verfügung gestellt. Hierzu versteckt man **Konstruktor**, **Copy-Konstruktor** und **Destruktor** hinter **`protected`** und verwendet ausschließlich **`static`-Methoden**.

Beispiel:

```
class MyCode
{
      public:
            static void Method() { printf("Method()\n"); }
      protected:
            MyMyCode();
            MyMyCode(const MyClass& Obj);
            ~MyMyCode();
};

int main()
{
      MyCode::Method();
      return 0;
}
```

26.3.2 Sharing von Code und/oder Tabellen mittels Heap-Objekt

Statt viele gleiche Kopien eines Objektes (**Code** und/oder **Tabellen**) zu machen, wird **nur ein Heap-Objekt** mit vielen Referenzen angelegt. Ein **Referenzzähler** (`static`) zählt die Nutzer des Objektes. Meldet sich der letzte Nutzer ab, dann löscht sich das Objekt selbst. Das Heap-Objekt ist dem statischen Objekt vorzuziehen, wenn es nur temporär gebraucht wird und viel Speicher benötigt.

Beispiel:

```
class MyCode
{
	public:
		static MyCode* CreateInstance() { return Obj(true); }
		static void Release() { Obj(false); }
		void Method()
		{ printf("Method() called from this = %ld\n",this); }
	protected:
		MyCode() {} //Konstruktion intern über new MyCode
		MyCode(const MyCode& Obj);
		~MyCode() {} //Destruktion nur intern über delete
	private:
		static MyCode* Obj(bool bAddRef);
};

MyCode* MyCode::Obj(bool bAddRef)
{
	static long lRefCounter = 0L;
	static MyCode* pObj = NULL;

	if(bAddRef) //Objekt referenzieren
	{
		if(!(lRefCounter + 1L)) //Überlauftest
			return NULL;
		if(!(lRefCounter++)) //falls erste Referenz
			pObj = new MyCode;
	}
	else //Objekt freigeben
	{
		if(!lRefCounter) //Unterlauftest
			return NULL;
		if(!(--lRefCounter)) //falls letzte Freigabe
		{
			delete pObj;
			pObj = NULL;
		}
	}
	return pObj;
}

int main()
{
	MyCode* pObj1 = MyCode::CreateInstance(); //Erzeugung
	MyCode* pObj2 = MyCode::CreateInstance();
	MyCode* pObj3 = MyCode::CreateInstance();

	pObj1->Method(); //alle Aufrufe rufen denselben Code auf
	pObj2->Method();
	pObj3->Method();

	pObj1->Release();
	pObj2->Release();
	pObj3->Release(); //Zerstörung

	return 0;
}
```

26.3.3 Nach Kopie die Daten bis zum Schreibzugriff sharen (Copy-On-Write)

Nach einer Kopie (Copy-Konstruktor, Zuweisungsoperator) kann man die internen Objekt-Daten bis zum Schreibzugriff sharen. Der Nutzer des kopierten Objektes bekommt eine eigene Kopie der Daten, sobald er schreibend auf das Objekt zugreift (**Copy-On-Write**). Genau dann werden die Objekt-Daten auch mit einem **Lock** versehen.

Man implementiert dieses Verfahren, indem man einem Objekt eine **verschachtelte Klasse** einverleibt, die den Wert (Objekt-Daten) repräsentiert (**`struct Value`**). Der Trick bei der Sache besteht darin, dass man nicht einfach eine Member-Variable von diesem Typ einbettet, die dann bei der Konstruktion des Gesamt-Objektes automatisch auf den Stack gelegt würde, sondern dass man lediglich einen Zeiger auf den Wert (Objekt-Daten) als Member führt. Dadurch sind **Objekt und Wert vollständig entkoppelt**. Der **Wert** liegt **irgendwo auf dem Heap** und kann von verschiedenen Objekten genutzt werden. Jeder Wert hat einen Referenz-Zähler, der von den Objekten gehandelt wird. Derjenige, der sich als letzter vom Wert loslöst, muss ihn auch löschen.

Beispiel:

```
//MyString:
//Der Wert (struct Value) hat 3 Member: Referenzzähler (m_lRefCnt),
//Locking-Flag (m_bLocked) und das eigentliche Datum (m_szStr).

#include<string.h>
class MyString
{
      public:
            MyString();
            MyString(const char* const szStr);
            MyString(const MyString& Obj);
            virtual ~MyString();
            MyString& operator=(const MyString& Obj);
            MyString& operator=(const char* const szStr);
            char& operator[](unsigned int pos);
      private:
            struct Value
            {
                  Value(const char* szStr)
                  :     m_lRefCnt(1),
                        m_bLocked(false),
                        m_szStr(new char[strlen(szStr) + 1])
                  {
                        strcpy(m_szStr,szStr);
                  }
                  ~Value() { delete[] m_szStr; }

                  long  m_lRefCnt;
                  bool  m_bLocked;
                  char* m_szStr;
            };
            Value* m_pVal; //Value liegt *irgendwo* auf dem Heap
};
```

```
MyString::MyString()
{ m_pVal = new Value(""); }

MyString::MyString(const char* const szStr)
{ m_pVal = new Value(szStr); }

MyString::~MyString() //Release On Delete
{
      //Wert freigeben:
      //(Der Wert liegt *irgendwo* auf dem Heap)
      --(m_pVal->m_lRefCnt);     //Freigabe
      if(!(m_pVal->m_lRefCnt))   //falls 0
            delete m_pVal;       //-> Wert vom Heap löschen
}

MyString::MyString(const MyString& Obj) //Link On Construction
{
      if(!(Obj.m_pVal->m_bLocked))
      {
            m_pVal = Obj.m_pVal;
            ++(m_pVal->m_lRefCnt);
      }
      else
      {
            m_pVal = new Value(Obj.m_pVal->m_szStr);
      }
}

MyString& MyString::operator=(const MyString& Obj) //Relink
{
      if(m_pVal == Obj.m_pVal)   //falls beide Objekte sich
            return *this;        //denselben Heap-Wert teilen

      //Jetzigen Wert freigeben:
      //(Der Wert liegt *irgendwo* auf dem Heap)

      --(m_pVal->m_lRefCnt);     //Freigabe
      if(!(m_pVal->m_lRefCnt))   //falls 0
            delete m_pVal;       //-> Wert vom Heap löschen

      if(!(Obj.m_pVal->m_bLocked))
      {
            m_pVal = Obj.m_pVal;
            ++(m_pVal->m_lRefCnt);
      }
      else
      {
            m_pVal = new Value(Obj.m_pVal->m_szStr);
      }
      return *this;
}
```

```
MyString& MyString::operator=(const char* const szStr) //Copy On Write
{
      //Copy-On-Write, falls mehr als 1 Nutzer:
      if(m_pVal->m_lRefCnt > 1L)
      {
            //Jetzigen Wert freigeben:
            //(Der Wert liegt *irgendwo* auf dem Heap)
            --(m_pVal->m_lRefCnt);      //Freigabe
            //Neue Kopie auf dem Heap erzeugen (Copy-On-Write):
            m_pVal = new Value(m_pVal->m_szStr);
      }
      m_pVal->m_bLocked = true;
      strcpy(m_pVal->m_szStr,szStr);
      return *this;
}

char& MyString::operator[](unsigned int pos) //Copy On Write
{
      //Copy-On-Write, falls mehr als 1 Nutzer:
      if(m_pVal->m_lRefCnt > 1L)
      {
            //Jetzigen Wert freigeben:
            --(m_pVal->m_lRefCnt);
            //Neue Kopie auf dem Heap erzeugen (Copy-On-Write):
            m_pVal = new Value(m_pVal->m_szStr);
      }
      m_pVal->m_bLocked = true;
      unsigned int len = strlen(m_pVal->m_szStr);
      if(pos >= len)
            return m_pVal->m_szStr[len - 1];
      return m_pVal->m_szStr[pos];
}

int main() //str1.m_pVal, str2.m_pVal, ... sind zu beobachten
{
      MyString str1("Hello");    //VAL1 allokiert
      MyString str2(str1);       //pVal2 = &VAL1
      MyString str3;             //VAL3 allokiert
      str3 = str1;               //VAL3 gelöscht, pVal3 = &VAL1
      //--- Soweit: Nur noch VAL1 mit 3 Referenzen auf dem Heap

      str3[0] = 'W';  //VAL3 allokiert + gelocked (Copy-On-Write)
      str3[1] = 'o';
      str3[2] = 'r';
      str3[3] = 'l';
      str3[4] = 'd';
      MyString str4(str3);  //VAL4 allokiert (da VAL3 gelocked)
      str4 = "Wie ?";       //VAL4 gelocked
      //--- Jetzt: VAL1, VAL3 und VAL4 auf dem Heap

      return 0;
}
```

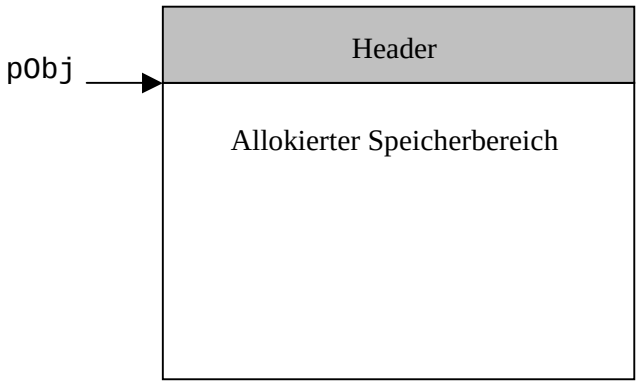
26.3.4 Object-Pooling

Motivation:

- **`new`-Header vermeiden**

  Wenn man ständig eine große Anzahl von kleinen Objekten konstruieren/destruieren muss, dann wird auch jedesmal `new` bzw. `delete` aufgerufen.

  Problem:
  `new` allokiert nicht nur Speicher für das Objekt, sondern auch einen Header mit Informationen über die Größe des allokierten Bereiches (diese Information wird von `delete` benötigt, um den Speicher wieder freizugeben).



  Abhilfe mittels Objekt-Pooling:
  Der erste `new`-Aufruf reserviert einen großen Speicherblock (Pool) und bei weiteren `new`-Aufrufen wird lediglich eine Konstruktion im noch freien Speicher durchgeführt (Placement new) oder ein gepooltes Objekt zurückgeliefert. `delete` gibt ein Objekt wieder frei (pooling) und löscht nach der letzten Freigabe den gesamten Pool.

- **Zeitverlust durch Destruktion und erneute Konstruktion vermeiden**

  Wenn man ständig große Objekte destruieren und anschließend wieder konstruieren muss, dann verliert man Zeit. Dies ist speziell dann der Fall, wenn die Objekte bei der Konstruktion eine Datenbank-Umgebung initialisieren müssen.

  Beispiel:
  Ein Server möchte jedem Client, der sich verbindet, einen Datenbank-Zugriff geben. Dies kann er über zustandslose (stateless) Zugriffs-Objekte tun, also Objekte, die kein Gedächtnis haben und nach jedem Zugriff wieder neu initialisiert werden. Dadurch ist es möglich, ein Zugriffs-Objekt nach dem Abmelden des Nutzers (Clients) in einen Pool zu geben, um es von dort aus wiederzuverwenden. Dabei spart man jedesmal eine Destruktion und eine Konstruktion des Objektes und somit Zeit.

Beispiel für die Implementierung einer Klasse mit Object-Pooling:

```
#include <new.h>
#include <list>
using namespace std;

class MyClass
{
	public:
		MyClass() {}
		void Destroy() { delete this; }
		static void* operator new(size_t size);
		static void operator delete(void* pMem);
	private:
		~MyClass() {}
		static void* operator new[](size_t size);  //versteckt
		static void operator delete[](void* pMem); //versteckt
		enum { POOL_SIZE = 1024*1024 }; //1 MB
		static void* Pool(bool bNew);
		static list<MyClass*>& ReleasedItems();
		static unsigned long& NumConstructedItems();
};

void* MyClass::operator new(size_t size)
{
	if(size != sizeof(MyClass))
		return ::operator new(size);

	void* pPool = Pool(true);

	static MyClass* pMem = NULL;
	if(pMem == NULL)
		pMem = (MyClass*) pPool;

	MyClass* pItem = NULL;
	if(ReleasedItems().empty())
	{
		pItem = ::new (pMem) MyClass; //Placement-new
		pMem += sizeof(MyClass);
		++(NumConstructedItems());
	}
	else
	{
		pItem = ReleasedItems().front();
		ReleasedItems().pop_front();
	}

	return pItem;
}
```

```
void MyClass::operator delete(void* pMem)
{
	if(pMem == NULL)
		return;

	MyClass* pItem = (MyClass*) pMem;
	ReleasedItems().push_back(pItem);
	if(ReleasedItems().size() == NumConstructedItems())
	{
		Pool(false);
	}
}

void* MyClass::Pool(bool bNew)
{
	static void* pPool = NULL;

	if(bNew)
	{
		if(pPool == NULL)
			pPool = operator new(POOL_SIZE);
	}
	else
	{
		if(pPool != NULL)
		{
			delete pPool;
			pPool = NULL;
		}
	}
	return pPool;
}

list<MyClass*>& MyClass::ReleasedItems()
{
	static list<MyClass*> listReleasedItems;
	return listReleasedItems;
}

unsigned long& MyClass::NumConstructedItems()
{
	static unsigned long dwNumConstructedItems = 0L;
	return dwNumConstructedItems;
}

int main() //pA, pB, pC und pD sind zu beobachten
{
	MyClass* pA = new MyClass; //Pool allokiert -> Obj1 platziert
	MyClass* pB = new MyClass; //Obj2 platziert
	pA->Destroy();             //Obj1 gepooled
	MyClass* pC = new MyClass; //Obj1 wieder referenziert
	pB->Destroy();             //Obj2 gepooled
	pC->Destroy();             //Obj1 gepooled -> Pool gelöscht
	MyClass* pD = new MyClass; //Pool allokiert -> Obj2 platziert
	pD->Destroy();             //Obj2 gepooled -> Pool gelöscht
	return 0;
}
```