

Notizen zu C++

Peter Thömmes

Version: 14.03.2002 (2. Ausgabe)

Copyright © Peter Thömmes

p.thoemmes@surf25.de

www.notizen-zu-cpp.de

Notizen zu C++

Peter Thömmes

Inhaltsverzeichnis

1. Einordnung von C++	1
2. Grundsätzlicher Aufbau eines Projektes	2
2.1 Pro Klasse eine *.h und eine *.cpp-Datei.....	2
2.2 Benennung von Verzeichnissen, Dateien und Klassen	4
2.3 Zentrale Header-Datei (MySettings.h).....	4
2.4 Mehrere Schichten (Layer) verwenden (horizontale Teilung)	5
2.5 Client/Server-Modell verwenden (vertikale Teilung)	7
2.6 UML (Unified Modeling Language).....	9
2.6.1 Allgemeines.....	9
2.6.2 Kardinalitäten nach UML	10
2.6.3 Frage nach den Klassen/Objekten	10
3. Wichtige Begriffe und Sprachelemente	13
3.1 namespace und using	13
3.2 Default-Konstruktor	14
3.3 Copy-Konstruktor	14
3.4 explicit-Konstruktor	15
3.5 Zuweisungs-Operator.....	16
3.6 Abstrakte Klasse (= abstrakte Basisklasse).....	17
3.7 Default-Argumente	18
3.8 Unspezifizierte Anzahl von Argumenten	18
3.9 l-value und r-value	19
3.10 Funktionszeiger	19
3.11 union	20
3.11.1 Allgemeines.....	20
3.11.2 Objekte unterschiedlichen Typs in eine Sequenz packen (Bsp.: STL-Container list)	20
3.11.3 Mehrere Datenstrukturen für dieselben Daten verwenden (hardwareabhängig)	23
3.11.4 Bitfelder zum Abtasten von Byte-Streams (hardwareabhängig)	25
3.11.5 Test-Funktion zum Testen der Maschine auf little- bzw. big-endian	27
3.12 extern "C" zum Abschalten der Namenszerstückelung (name mangling).....	28
4. Grundsätzliche Regeln beim Programmieren.....	29
4.1 Include-Wächter verwenden	29
4.2 Kommentar // dem Kommentar /* */ vorziehen.....	29
4.3 Optimierte die Laufzeit immer gleich mit.....	29
4.3.1 Objekte erst dort definieren, wo sie gebraucht werden.....	29
4.3.2 Zuweisung an ein Objekt mit der Konstruktion verbinden.....	30
4.3.3 return, break und continue mit Geschick einsetzen	30
4.4 Laufvariable im Schleifenkopf definieren	35
4.5 Der Stack ist immer dem Heap (new/delete) vorzuziehen	35
4.6 protected nur bei Basisklassen	36
4.7 Keine Fehler beim Mischen von C und C++-Code machen	36
4.8 Ungarische Notation verwenden	37
4.9 Eingebaute (native) Datentypen nie hinter typedef verstecken	38
4.10 Implizite Typumwandlung ggf. abschalten.....	39
4.11 inline nur bei sehr sehr einfachen nicht-virtuellen Funktionen	41
4.11.1 Allgemeines.....	41
4.11.2 Widerspruch 'virtual und inline': virtual dominiert inline	42

4.11.3	Basisklasse: Virtueller Destruktor als leere inline-Funktion	42
4.12	Falsche Benutzung einer Klasse ausschliessen	43
4.12.1	Kopie eines Objektes verbieten	43
4.12.2	Konstruktion eines Objektes verbieten.....	43
4.13	Laufzeitschalter immer Compiler-Schaltern vorziehen	44
4.14	short statt bool als return-Wert bei Interface-Methoden	45
5.	Strings.....	46
5.1	ASCII-Tabelle.....	46
5.2	string in der STL	48
5.2.1	Allgemeines.....	48
5.2.2	Teil-Strings ersetzen mit string::replace() und string::find()	50
5.2.3	Zeichen löschen mit string::erase() und einfügen mit string::insert()	50
5.2.4	Umwandlung in Zahlen mit strtol() und der Methode c_str():.....	51
5.2.5	Teil eines anderen Strings anhängen mit string::append()	52
6.	Konstantes.....	53
6.1	const-Zeiger (C-Funktionen).....	53
6.2	const-Referenzen (C++-Funktionen)	54
6.2.1	Allgemeines.....	54
6.2.2	STL-Container als const-Referenzen verlangen const_iterator	54
6.3	Read-Only-Member-Funktionen.....	56
6.3.1	Allgemeines.....	56
6.3.2	mutable-Member als interne Merker (Cache-Index) verwenden	56
6.3.3	Zeiger bei Read-Only-Member-Funktion besonders beachten	58
6.4	const-return-Wert.....	59
6.5	const statt #define verwenden	60
6.5.1	Globale Konstanten	60
6.5.2	Lokale Konstanten einer Klasse	61
6.6	const-inline-Template statt MAKRO (#define) verwenden	62
7.	Globales (static-Member).....	64
7.1	static-Member	64
7.1.1	Allgemeines.....	64
7.1.2	Zugriff ohne ein Objekt zu instanziiieren	64
7.2	static-Variable in static-Methode statt globaler Variable.....	65
7.3	In globalen Funktionen: Virtuelle Methoden der Argumente nutzen	68
8.	Referenz statt Zeiger benutzen (Zeiger nur für C-Schnittstellen).....	69
9.	Funktionen, Argumente und return-Werte.....	71
9.1	Argumente sollten immer Referenzen sein	71
9.1.1	const-Referenz statt Wert-Übergabe (Slicing-Problem).....	71
9.1.2	Referenz statt Zeiger.....	72
9.2	Argumente: Default-Parameter vs. Funktion überladen.....	73
9.3	Überladen innerhalb einer Klasse vs. über Klasse hinweg	74
9.3.1	Allgemeines.....	74
9.3.2	Nie Zeiger-Argument mit Wert-Argument überladen	75
9.4	return: Referenz auf *this vs. Wert	75
9.4.1	Lokal erzeugtes Objekt zurückliefern: Rückgabe eines Wertes	75
9.4.2	Objekt (Eigner) der Methode zurückliefern: Rückgabe einer Referenz auf *this.....	76
9.4.3	Keine Zeiger/Referenzen auf private-Daten zurückliefern	77
9.5	return-Wert nie direkt an ein referenzierendes Argument übergeben	78
10.	Smart-Pointer	79
10.1	Allgemeines	79
10.2	Smart-Pointer für die Speicher-Verwaltung.....	79

10.2.1	Eigenschaften des Smart-Pointers für die Speicherverwaltung	79
10.2.2	Was zu beachten ist	80
10.2.3	Code-Beispiel	81
10.2.4	Smart-Pointer immer per Referenz an eine Funktion übergeben	83
10.2.5	Empfehlungen	84
10.3	Smart-Pointer für andere Zwecke	85
11.	new/delete	86
11.1	Allgemeines zu new	86
11.2	Allgemeines zu delete	87
11.3	Beispiel für new/delete	88
11.4	Allgemeines zu new[]/delete[]	89
11.4.1	new[]	89
11.4.2	delete[]	89
11.5	Mit Heap-Speicher arbeiten	90
11.6	Heap-Speicher als Shared Memory	90
11.7	new/delete statt malloc/free	91
11.8	Zusammenspiel von Allokierung und Freigabe	94
11.9	Eigener new-Handler statt Out-Of-Memory-Exception	96
11.10	Heap-Speicherung erzwingen/verbieten	98
11.10.1	Allgemeines zur Speicherung von Objekten	98
11.10.2	Heap-Speicherung erzwingen (protected-Destruktor)	99
11.10.3	Heap-Speicherung verbieten (private operator new)	100
12.	Statische, Heap- und Stack-Objekte	102
12.1	Die 3 Speicher-Arten	102
12.2	Statische Objekte (MyClass::Method())	103
12.3	Heap-Objekte (pObj->Method())	103
12.4	Stack-Objekte (Obj.Method())	104
13.	Programmierung einer Klasse	105
13.1	Allgemeines	105
13.1.1	Folgende Fragen sollten beim Entwurf einer Klasse beantwortet werden	105
13.1.2	Die wesentlichen Methoden einer Klasse sind zu implementieren	106
13.1.3	Durch den Compiler automatisch generierte Methoden sind zu beachten	106
13.1.4	inline-Funktionen ggf. hinter die Deklaration schreiben	107
13.1.5	Nie public-Daten verwenden	108
13.1.6	Mehrdeutigkeiten (ambiguous) erkennen	109
13.2	Der Konstruktor	111
13.2.1	Kein new im Konstruktor / Initialisierungslisten für Member	111
13.2.2	Keine virtuellen Methoden im Konstruktor aufrufen	114
13.2.3	Arrays mit memset() initialisieren	114
13.3	Der Destruktor	114
13.3.1	Generalisierung ("is-a"): Basisklassen sollten einen virtuellen Destruktor haben	114
13.4	Zuweisung per operator=()	116
13.4.1	Keine Zuweisung an sich selbst	116
13.4.2	Referenz auf *this zurückliefern	116
13.4.3	Alle Member-Variablen belegen	117
13.5	Indizierter Zugriff per operator[]()	118
13.6	Virtuelle Clone()-Funktion: Heap-Kopie über Basisklassen-Zeiger	119
13.7	Objektanzahl über private-Konstruktor kontrollieren	121
13.7.1	Objekte über eine friend-Klasse (Objekt-Manager) erzeugen	121
13.7.2	Objekte über eine statische Create()-Funktion erzeugen	123
13.7.3	Genau 1 Objekt erzeugen (Code und/oder Tabelle)	124
13.8	Klassen neu verpacken mittels Wrapper-Klasse	125
14.	Richtiges Vererbungs-Konzept	126

14.1	Allgemeines	126
14.1.1	Nie von (nicht-abstrakten) Klassen ohne virtuellen Destruktor erben	126
14.1.2	Copy-Konstruktor muß auch Basisklassen-Member belegen.....	127
14.1.3	Statischen/dynamischen Typ und statische/dynamische Bindung beachten	128
14.1.4	Nie die Default-Parameter virtueller Funktionen überschreiben.....	129
14.1.5	public-, protected- und private-Vererbung gezielt verwenden	130
14.1.6	Rein virtuell / virtuell / nicht-virtuell	134
14.1.7	Rein virtuelle Methoden immer, wenn keine generalisierte Implementierung möglich.....	134
14.2	Spezialisierung durch public-Vererbung ("is a")	135
14.3	Code-Sharing durch private-Vererbung ("contains").....	137
14.4	Composition statt multiple inheritance	140
14.5	Schnittstellen (Abstract Mixin Base Classes) public dazuerben	141
14.6	Abstrakte Basisklasse vs. Template	144
14.7	Verknüpfung konkreter Klassen mittels abstrakter Basisklasse	146
14.8	Erbung aus mehreren Basisklassen (multiple inheritance) vermeiden	147
14.8.1	Expliziter Zugriff (oder using)	147
14.8.2	Virtuelle Vererbung (Diamant-Struktur).....	147
14.9	Zuweisungen nur zwischen Childs gleichen Typs zulassen.....	149
15.	Nutzer einer Klasse von Änderungen entkoppeln	151
15.1	Allgemeines	151
15.2	Header-Dateien: Forward-Deklaration ist #include vorzuziehen.....	151
15.3	Delegation bzw. Aggregation	152
15.4	Objekt-Factory-Klasse und Protokoll-Klasse.....	154
16.	Code kapseln.....	156
16.1	Beliebig viele Kopien erlaubt: Funktions-Objekte (operator()).....	156
16.2	Nur eine Kopie erlaubt: Statische Objekte (MyClass::Method()).....	156
17.	Operatoren.....	157
17.1	Definition von Operatoren	157
17.2	Binäre Operatoren effektiv implementieren.....	158
17.3	Unäre Operatoren bevorzugt verwenden	159
17.4	Kommutativität: Globale binäre Operatoren implementieren.....	159
17.5	Operator-Vorrang (Precedence).....	161
17.6	Präfix- und Postfix-Operator.....	162
17.6.1	Allgemeines.....	162
17.6.2	Wartungsfreundlichkeit erhöhen durch ??(*this) im Postfix-Operator	164
17.6.3	Präfix(??Obj) ist Postfix(Obj??) vorzuziehen	164
17.7	Der Komma-Operator , ...	165
18.	Datentypen und Casting	165
18.1	Datentypen	165
18.2	Polymorphismus: vfptr und vftable.....	166
18.3	RTTI (type_info) und typeid bei polymorphen Objekten	168
18.4	dynamic_cast: Sicherer cast von Zeigern oder Referenzen.....	170
18.4.1	Allgemeines.....	170
18.4.2	dynamic_cast zur Argument-Überprüfung bei Zeiger/Referenz auf Basisklasse.....	172
18.5	const_cast.....	174
18.6	reinterpret_cast (! nicht portabel !) und Funktions-Vektoren.....	175
18.7	STL: Min- und Max-Werte zu einem Datentyp	176
19.	In Bibliotheken Exceptions werfen	176
19.1	Allgemeines	176
19.2	Exceptions per Referenz fangen	179
19.3	Kopien beim weiterwerfen vermeiden.....	179
19.4	Beispiel für Exception-Handling.....	180

19.5	Exception-Spezifikation.....	181
19.5.1	Allgemeines.....	181
19.5.2	Spezifikationswidrige Exceptions abfangen: set_unexpected (NICHT bei Visual C++).....	181
19.5.3	Compilerunabhängiges Vorgehen	183
20.	Die STL (Standard Template Library)	184
20.1	Allgemeines	184
20.1.1	Einführung	184
20.1.2	Wichtige STL-Member-Variablen und Methoden	187
20.2	STL-Header-Dateien.....	190
20.2.1	Aufbau: Die Endung ".h" fehlt	190
20.2.2	Nutzung: "using namespace std"	190
20.3	Generierung von Sequenzen über STL-Algorithmen.....	191
20.3.1	back_inserter().....	191
20.3.2	Schnittmenge (set_intersection)	191
20.3.3	Schnittmenge ausschliessen (set_symmetric_difference).....	192
20.3.4	Andere Menge ausschliessen (set_difference)	192
20.3.5	Vereinigungsmenge (set_union)	193
20.3.6	Liste anhängen (list::insert)	193
20.4	Wichtige Regeln.....	194
20.4.1	Einbinden der STL	194
20.4.2	Für die Objekte müssen die benötigten Operatoren implementiert werden	195
20.4.3	Iterator: ++it statt it++ benutzen.....	196
20.4.4	Löschen nach find(): Immer über den Iterator (it) statt über den Wert (*it).....	198
20.4.5	map: Nie indizierten Zugriff [] nach find() durchführen	200
20.5	Beispiele für die Verwendung der Container	202
20.5.1	list: Auflistung von Objekten, wobei Objekte mehrfach vorkommen können	202
20.5.2	set: Aufsteigend sortierte Menge von Objekten, die nur einmal vorkommen	204
20.5.3	map: Zuordnung von Objekten zu einem eindeutigen Handle	205
20.5.4	map: Mehrdimensionaler Schlüssel.....	207
20.5.5	vector: Schneller indizierter Zugriff.....	209
20.5.6	pair und make_pair(): Wertepaare abspeichern	210
20.6	hash_map	211
20.6.1	hash_map für Nutzer von Visual C++	211
20.6.2	Prinzip von hash_map	212
20.6.3	Nutzung von hash_map der STL.....	214
20.7	Facets und Locales	218
20.8	Exceptions.....	219
20.9	Standardisierungs-Dokumente	220
20.9.1	DIS (Standard von ISO-ANSI).....	220
20.9.2	ARM (Buch von Margaret Ellis und Bjarne Stroustrup).....	220
21.	Arten von Templates	221
21.1	Class-Templates	221
21.2	Function-Templates	222
21.2.1	Global Function Templates	222
21.2.2	Member Function Templates.....	222
21.3	Explizite Instanziierung von Templates.....	223
22.	Datenbank-Konsistenz durch Transaktions-Objekte	224
23.	Proxy-Klassen.....	225
23.1	Allgemeines	225
23.2	Schreiben/Lesen beim indizierten Zugriff mittels Proxy unterscheiden	227
24.	Double-Dispatching (Kollisionen je nach Partner handeln)	229

25.	80/20-Regel und Performance-Optimierung.....	234
25.1	Allgemeines	234
25.2	Zeit-Optimierungen.....	235
25.2.1	return so früh wie möglich	235
25.2.2	Präfix-Operator statt Postfix-Operator	235
25.2.3	Unäre Operatoren den binären Operatoren vorziehen	236
25.2.4	Keine Konstruktion/Destruktion in Schleifen	236
25.2.5	hash_map statt map, falls keine Sortierung benötigt wird.....	237
25.2.6	Lokaler Cache (static-hash_map) um Berechnungen/Datenermittlungen zu sparen	237
25.2.7	Löschen nach find() immer direkt über den Iterator.....	239
25.2.8	map: Nie indizierten Zugriff [] nach find() durchführen	241
25.2.9	Unsichtbare temporäre Objekte vermeiden	243
25.2.10	Berechnungen erst dann, wenn das Ergebnis gebraucht wird (Lazy Evaluation).....	246
25.2.11	Datenermittlung erst dann, wenn die Daten gebraucht werden (Lazy Fetching)	246
25.2.12	Große Anzahl kleiner Objekte blockweise lesen (Prefetching).....	246
25.2.13	Kein unnötiges Datenbank-Store.....	247
25.2.14	Datenbank-UseCases vereinbaren und nutzen	247
25.3	Speicher-Optimierungen	248
25.3.1	Sharing von Code und/oder Tabellen mittels statischem Objekt.....	248
25.3.2	Sharing von Code und/oder Tabellen mittels Heap-Objekt.....	248
25.3.3	Sharing von veränderbaren Objekten: Wert auf den Heap (Copy-On-Write)	250
25.3.4	new-Header vermeiden: Speicherpool	253

Notizen zu C++

Peter Thömmes

1. Einordnung von C++

Um **1968** entwickelten Brian **Kernighan** und Dennis **Ritchie** an den Bell Labs die **prozedurale Programmiersprache C**. C enthält nur Makros, Zeiger, Strukturen, Arrays und Funktionen.

Bjarne **Stroustrup** stellte **1986** die darauf aufbauende **objektorientierte Sprache C++** in seinem Buch "The C++ Programming Language" vor, die er bei AT&T entwickelt hatte. C++ erweitert C um die Objektorientierte Programmierung (OOP), Referenzen (Referenzparameter statt Zeiger), Templates (Vorlagen, Schablonen) und Exceptions (Ausnahmebehandlung).

Alexander **Stepanov** und Meng **Lee** konnten **1994** 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. 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.

Im Voraus sollte schon mal gesagt werden, daß es ratsam ist sich im Gebiet 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.

Ich danke meiner Frau Claudia und meinen Kindern Anna-Lena und Markus für die Geduld, die sie aufgebracht haben während der langen Zeit der Entwicklung dieses Buches.

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):

Mann 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 Konstanten statt **#define** 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 //include-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 MyString& szText) const
{
    if(szText.IsEmpty())
        return false;
    ...
    return true;
}
```

→ Hierdurch vermeidet man unnötige Konstruktionen, Speicherallokierungen und Destruktionen und erhöht die Performance
- Objekte immer erst dort instanziiieren, wo sie gebraucht werden

→ Weniger Konstruktionen/Speicherallokierungen/Destruktionen (Performance)
- Instanziierungen in 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 vorn herein** optimiert gestalten, denn wenn man es später tut macht man sehr viel 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 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. Natürlich kann man die eigenen Klassen **auch ganz ohne Präfix** benamen, was aber spätestens dann zu Problemen führt, wenn man seinen Code anderen zur Verfügung stellt.

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/MyRobotGUI.cpp
 Robot/MyRobotGUI.h

Richtig: Robot/MyGUI.cpp
 Robot/MyGUI.h

Weiterhin kann es sehr hilfreich sein, wenn man bei einer Spezialisierung durch public-Vererbung die abgeleiteten Klassen mit dem Basisklassen-Namen als Präfix versieht. Dieses verschafft Übersicht und hilft beim suchen. Bsp.:

Parent: MyView
 Child1: MyFormView
 Child2: MyData1FormView, MyData2FormView, MyData3FormView

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. Bsp.:

```
MyData1FormView formviewData1;
MyData2FormView formviewData2;
```

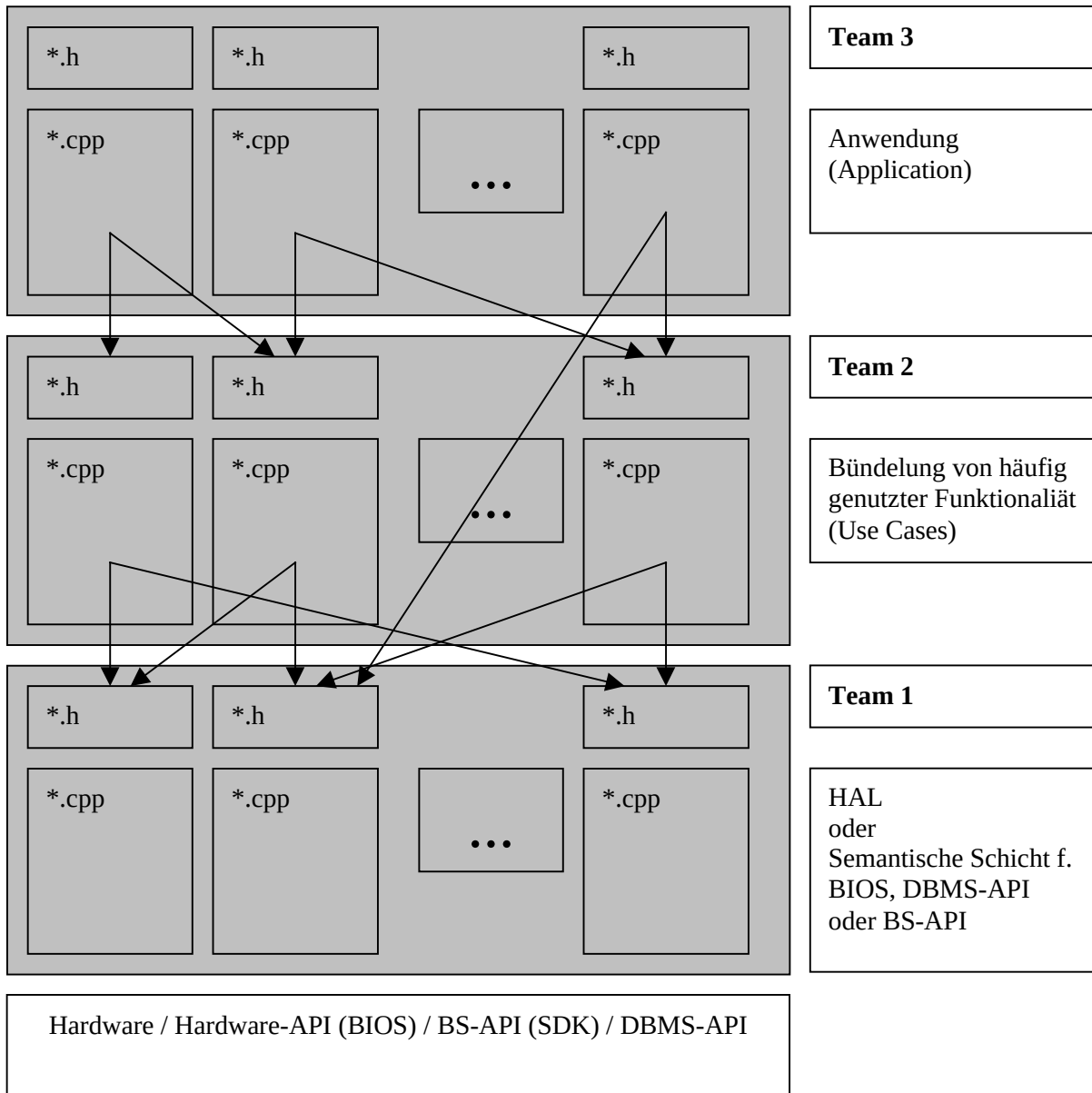
2.3 Zentrale Header-Datei (MySettings.h)

Um für das Projekt spezifische Präprozessor-, Compiler-, Linker- und Bibliotheks-Optionen zu setzen sollte man eine zentrale Header-Datei (**MySettings.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.

Dieses Datei muß dann in jedes Modul (*.h und *.cpp) eingebunden werden.

2.4 Mehrere Schichten (Layer) verwenden (horizontale Teilung)

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



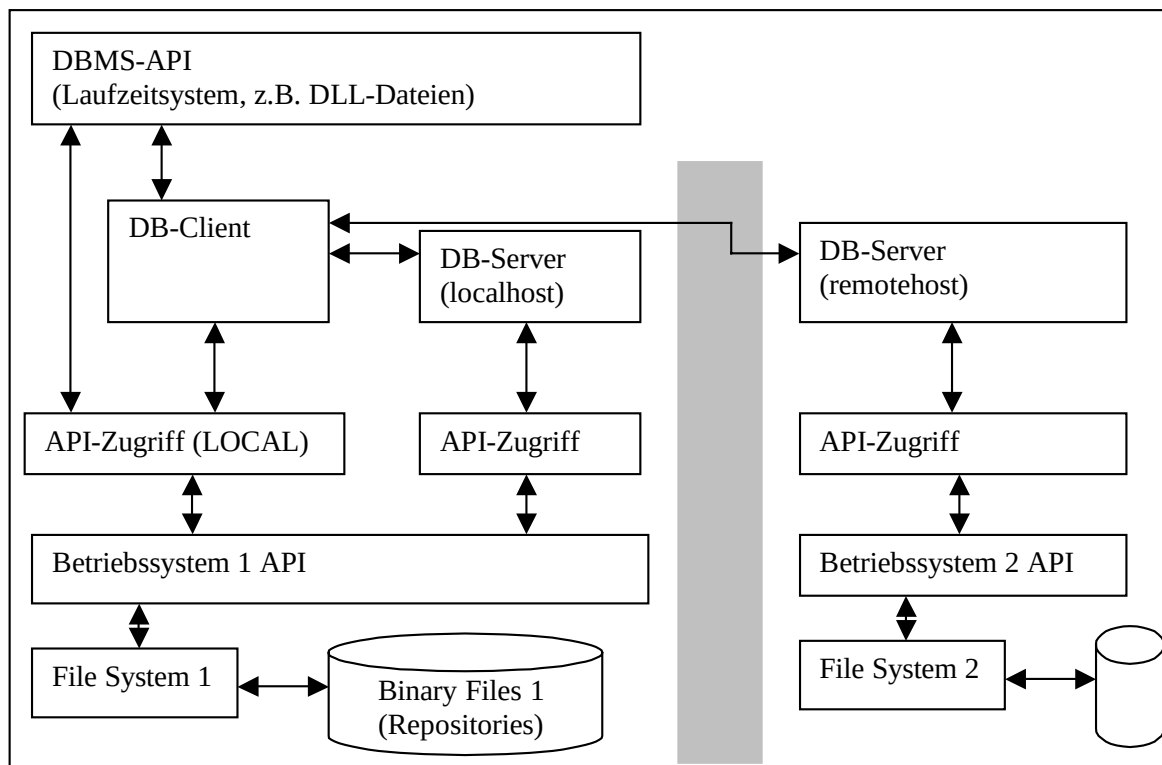
Eine Schicht ist als Sammlung von Klassen zu verstehen, die nur von Klassen der darüberliegenden Schichten verwendet werden darf, d. h. es gibt nur in Klassen der darüberliegenden Schichten eine entsprechende `#include`-Anweisung.

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 darunter liegenden 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 Gesamt-Problem vor Augen haben. So ist es möglich, das 3 Software-Teams mit unterschiedlich ausgerichteten Software-Experten relativ unabhängig voneinander an einem Gesamt-Projekt entwickeln (bis auf die Schnittstellenkonventionen, die nicht zu unterschätzen sind).

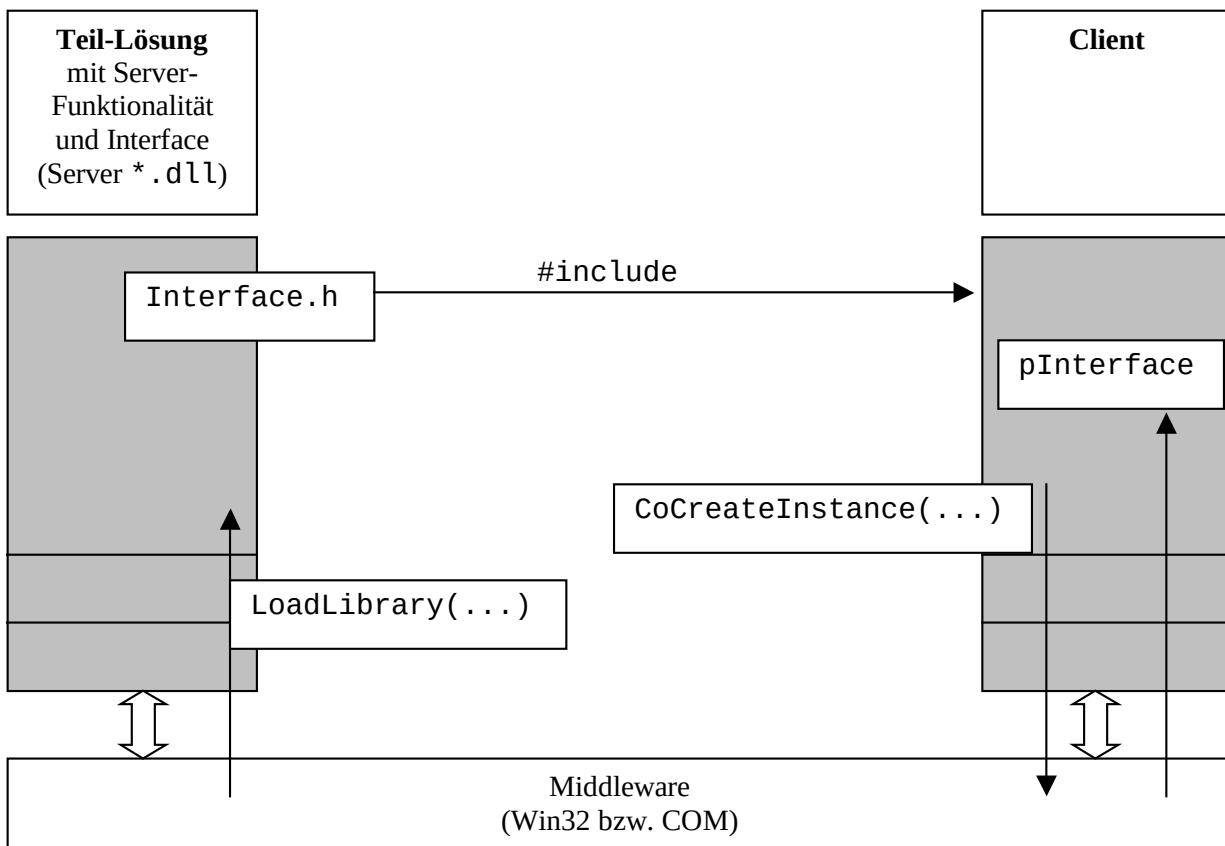
Man kann ein Software-Konzept auf unterschiedlichen Detailstufen erarbeiten (Requirements, Specification, High Level Design und Detailed Level Design). Die Grobstruktur (High-Level-Design) eines **Database Management Systems** könnte z.B. wie folgt aussehen:



2.5 Client/Server-Modell verwenden (vertikale Teilung)

Man kann eine Anwendung nicht nur horizontal (in Schichten) sondern auch vertikal teilen. Man schreibt also für abgeschlossene Teil-Aufgaben 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.

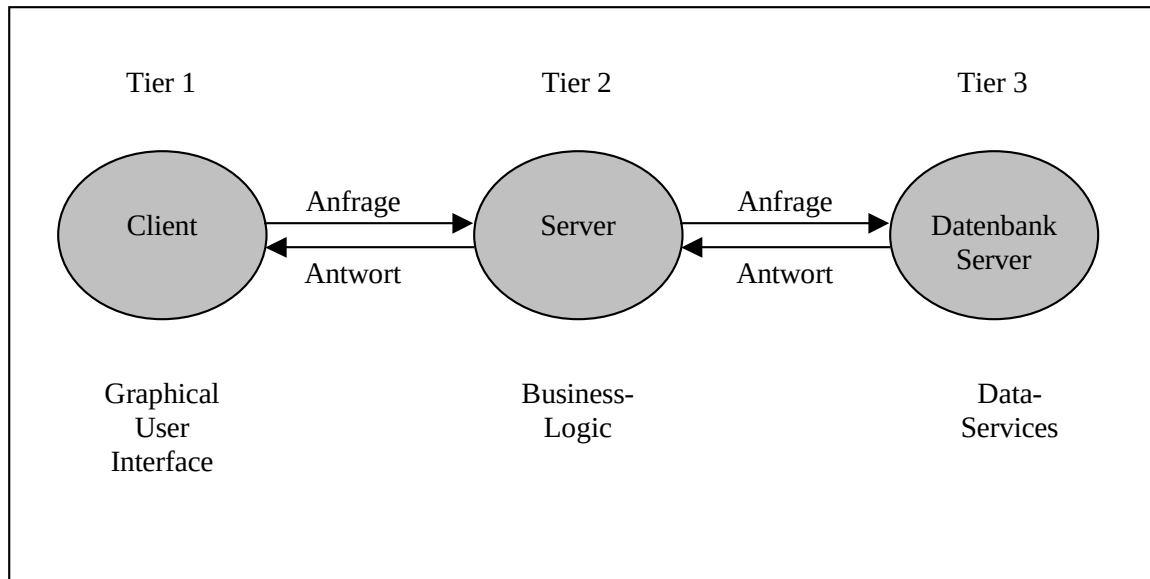
Beispiel: Microsoft Inprocess-COM-Server unter 32-Bit-Windows (nur schematisch):



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) den Server starten (mittels eindeutiger ID (GUID) und der SDK-Funktion `CoCreateInstance()`) und dessen Dienste nutzen. 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.

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. So kapselt man oft den projektspezifischen Code (Business-Logic) in den Server, während der Client nur Code für das Benutzer-Interface implementiert.

Als Beispiel hierfür 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 also 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 Kern-Aufgabe, nämlich die Implementierung der Business-Logic in Tier 2 konzentrieren.

2.6 UML (Unified Modeling Language)

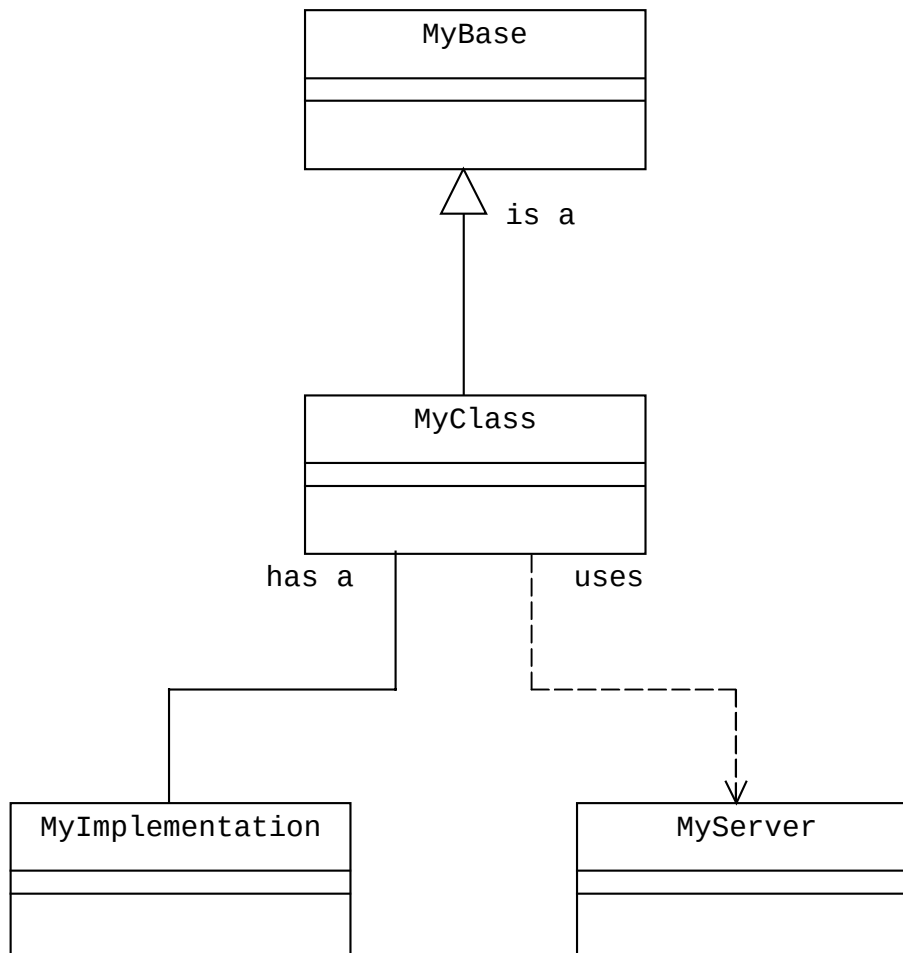
2.6.1 Allgemeines

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 Hauptgerüsts 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:

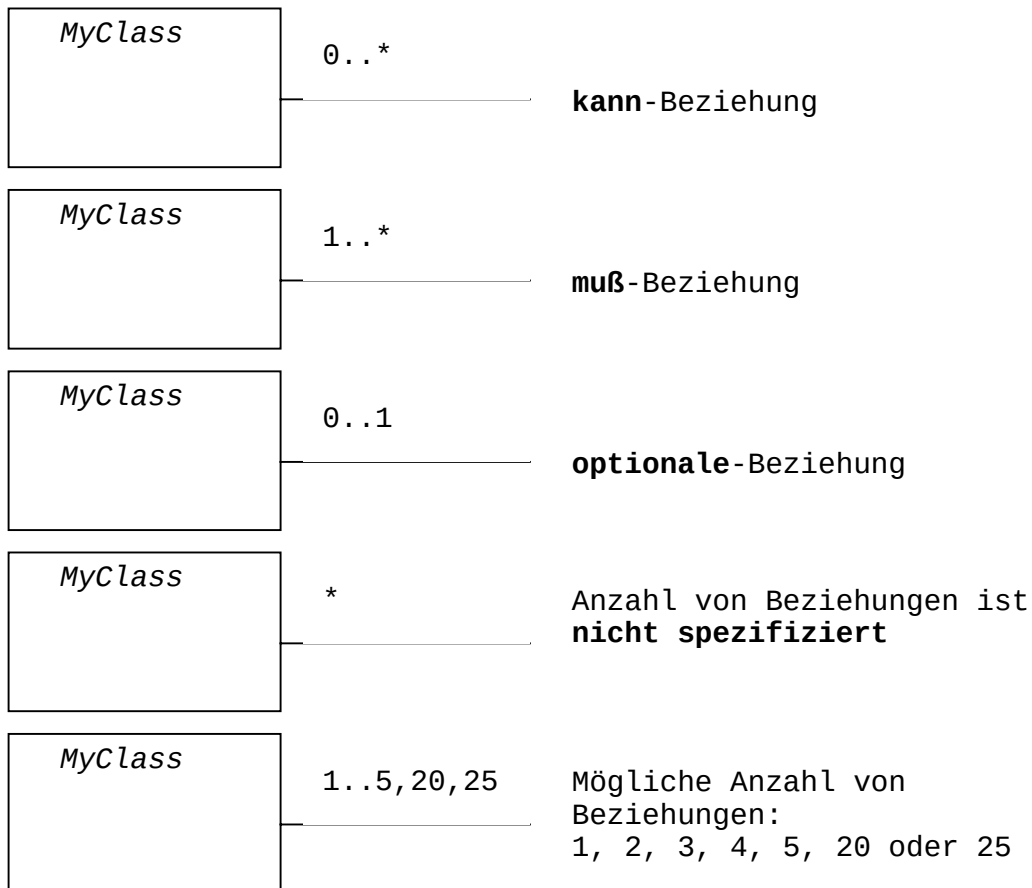
- Use Case Diagram: Wer tut was ?
- Colaboration Diagram: Welche Aktionen gibt es ? Wer kommuniziert wie mit wem ?
- Sequenz Diagram: Sequentielle Aufzeichnung der Aktionen
- Class Diagram: Klassen und ihre Beziehungen und Kardinalitäten
- State Transition Diagram: Zustandsübergänge eines Objektes

Beispiel: Class Diagram



2.6.2 Kardinalitäten nach UML

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



2.6.3 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()`
 - o Instanziierung und Initialisierung des Haupt-Objektes (`theApp`)
 - o 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);
}
```

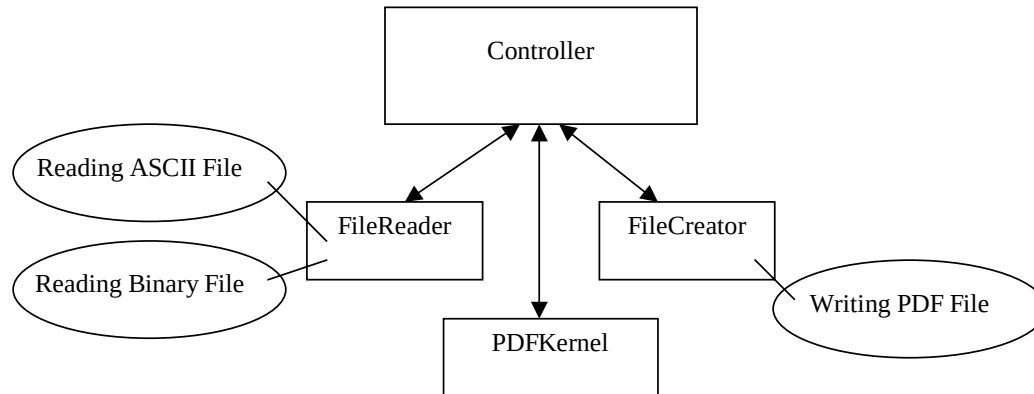
- Mausklick wird behandelt durch die Callback-Funktion `OnLeftMouseKlick()`
 - o Dem Betriebssystem wird ein Zeiger auf die Funktion `OnLeftMouseKlick()` übergeben und per Handle bzw. eindeutiger ID ein Fensterbereich mit dieser Funktion verknüpft. Jeder Mausklick in diesen Fensterbereich führt dann 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 und 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 wieder findet, 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;
}
```

Bsp.:

```
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 in der Implementierung nicht noch mal vorangestellt werden.

Beispiel:

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

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 = 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. In Folge dessen kann man kein Objekt von dieser Klasse erzeugen (instanciieren) 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:

<pre>class MyBase { public: virtual ~MyBase() {} };</pre>	=	<pre>class MyBase { public: virtual ~MyBase(); }; MyBase::~~MyBase() {}</pre>
--	---	--

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 Quellcode-Datei (* .cpp).

Beispiel:

```
class MyAbstractBase
{
public:
    virtual ~MyAbstractBase() {} //Hack: nicht wirkkl.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 kan man dies mit ... anzeigen:

Beispiel:

```
int PrintError(const char* szStr...)
{
}
```

Nun ist der Programmierer selbst dafür verantwortlich die folgenden Elemente in ihrer Art und Anzahl zu unterscheiden. Benutzt er zum Beispiel einen String als erstes Argument (siehe `printf()`), dann kann er ein bestimmtes Zeichen (Bsp.: `%`) als **Token** definieren und festlegen wie dieser Token Teil-Strings einbettet und weitere Argumente an die Reihe von Argumenten anhängt.

Beispiel:

<code>%d</code>	→	String-Darstellung eines Integers wird eingebettet Der Wert des Integers folgt als weiteres Argument
<code>%s</code>	→	String wird eingebettet Der String folgt als weiteres Argument
<code>%%</code>	→	Das Zeichen <code>%</code> 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 wieviel Argumente folgen.

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 definiert wurde (Bsp.: `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 (Bsp.: STL-Container 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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);
                break;
        }
    }

    return 0;
}

```

3.11.3 Mehrere Datenstrukturen für dieselben Daten verwenden (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 diegleiche Signifikanz, aber insgesamt nicht. Dieses Problem ist 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** (Freeware) erlaubt je nach compilierung 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:

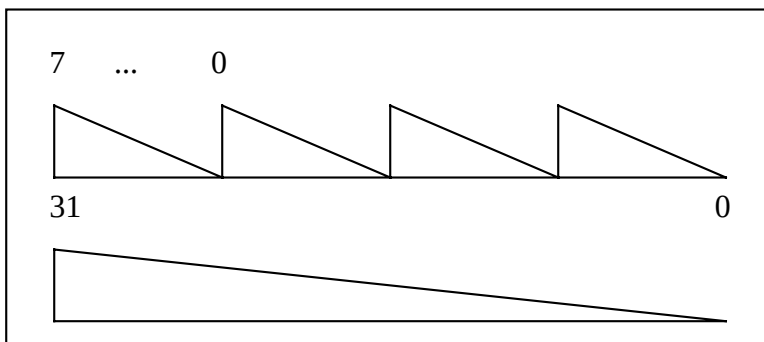
```
unsigned long lMyValue = 0x04030201L;
```

Speicherung:

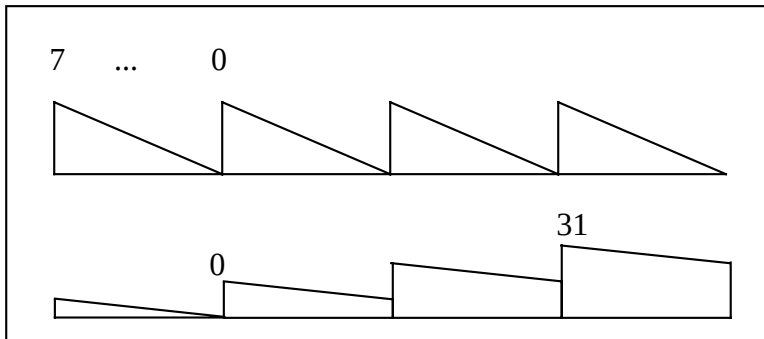
little-endian:	01 02 03 04	PC (WinNT), VAX (VMS)
big-endian:	04 03 02 01	Mac (MacOS), Sparc (Solaris), Java-JVM, TCP/IP
NUXI:	03 04 01 02	PDP-11 (NUXI)

Wenn man sich fragt, warum sich in der Network-Welt die "big-endian"-Technologie durchgesetzt hat, dann ist mit Sicherheit ein Grund, das 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;    //XXXXX XX??
};
```

Big-Endian:

```
struct MyStreamByte
{
    unsigned char Code      :    6;    //XXXXX 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.abysStreamBuffer[0] = 0x01; //ID_LO = 0000 0001
    bs.abysStreamBuffer[1] = 0x03; //Valid = 0 / ID_HI = 000 0011
    bs.abysStreamBuffer[2] = 0xFE; //TextLen = 0000 001 / Flag = 0

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

    size_t size = sizeof(bs.abysStreamBuffer);

    printf("BitStream bs.abysStreamBuffer [ %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 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 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 (name mangling)

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.

Bsp.:

```
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 `//` zu Kommentierung benutzt, dann kann man zum testen ganz einfach Blöcke mit `/* */` auskommentieren.

4.3 Optimierte 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 nochmal wesentliche Teile des Codes 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 des "gewachsenen Codes".

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 die Zuweisung zu machen. Denn im zweiten Fall werden zunächst alle Member mit Default-Werten belegt und anschließend nochmal mit anderen Werten.

4.3.3 *return, break und continue mit Geschick einsetzen*

Es gibt einige typische "Anti-Effektivitäts-Regeln", denen man leider allzuoft 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
}
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 mit `return` raus gehen.

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 keine neues Resultat mehr bringt. Somit ist auch hier bevorzugt (vor allem anderen) zu überprüfen ob ein `break` oder `continue` angebracht ist.

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 sicher gestellt, 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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 bietet jedoch die STL in der Regel immer irgend einen 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 das 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:

b Boolean	bool , BOOL → true, TRUE oder false, FALSE
c Name	char → 8 Bit mit Vorz., -128...+127, 0..127 = ASCII-Standard
n Name	short → 16 Bit mit Vorz., -32768...+32767
l Name	long → 32 Bit mit Vorz., -2147483647...+2147483648
i Name	int → auf 16 Bit Betriebssystem short auf 32 Bit Betriebssystem long
by Name	unsigned char → 1 Byte, 0...255, 0x0...0xFF
w Name	unsigned short → 2 Byte, 0...65535, 0x0...0xFFFF
dw Name	unsigned long → 4 Byte, 0...4294967295, 0x0...0xFFFFFFFF
ui Name	unsigned int → auf 16 Bit Betriebssystem unsigned short auf 32 Bit Betriebssystem unsigned long
s Name	char[] → Zeichenkette, Vektor von char -Elementen
sz Name	char[] → Zeichenkette, die mit 0x00 ended (zeroterminated)
str Name	string → string-Object (STL)
f Name	float → 32 Bit Fließkomma, 7 Stellen
d Name	double → 64 Bit Fließkomma, 15 Stellen
e Name	enum → Aufzählung von int
v Name	void -> kann alles sein

Stufe 2: Präfix vor dem Präfix von Stufe 1:

a ...	Array (Vektor, Matrix), Bsp.: <code>int anNumbers[256];</code>
p ...	Pointer, Bsp.: <code>MyClass* pObj;</code>

Stufe 3: Präfix vor dem Präfix von Stufe 2:

g ...	Globale Variable
m ...	Member-Variable

Beispiele: `m_szText`, `dValue`, `g_ThreadStatus`, `byFlags`

Man sollte aber auch sonst auf einleuchtende Präfixe zurückgreifen:

<code>list<MyClass></code>	list Objs;
<code>set<MyClass></code>	set Objs

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 Bibliothek, dann sollte der Hersteller dieser ebenfalls diese Regel beherzigen (Bsp.: Die Bibliothek 'MFC' von Microsoft benutzt das 'C' - `class CString`). Ein Präfix sollte jedoch wegen der Wiederverwendbarkeit in anderen Projekten eigentlich ganz entfallen aber zumindest **nie projektabhängig** sein bzw. den Projektnamen beinhalten.

4.9 Eingebaute (native) Datentypen nie hinter typedef verstecken

Es ist ein Irrglaube das 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 anderem 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 sehr einfachen nicht-virtuellen Funktionen

4.11.1 Allgemeines

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

Bsp.:

```
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 Eliminieren 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 Compilieren 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 vtable 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 **virtuellen** 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 Compilerschalter umdefinieren (dabei überprüfen welche Kombinationen von Compilerschaltern 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("Optimized for Speed\n");
    #else
        printf("Optimized 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("Optimized for Speed\n");
            return 0;
        }
    }
    printf("Optimized 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 allzuoft vorkommt, das 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 RCR
          ...
          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 ("\"") 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 ("\"") 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.

Bsp: 0x1A (eof)

	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	{		}	~	Δ

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	€	☐	,	f	"	...	†	‡	^	%	Š	<	œ	☐	Ž	☐
9	☐	\	/	"	"	•	—	—	~	™	š	>	œ	☐	ž	ÿ
A		ı	¢	£	¤	¥		§	"	©	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

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	€	☐	,	f	"	...	†	‡	^	%	Š	<	œ	☐	Ž	☐
9	☐	\	/	"	"	•	—	—	~	™	š	>	œ	☐	ž	ÿ
A		ı	¢	£	¤	¥		§	"	©	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

Beispiel: Sprach- und plattformunabhängige Übertragung von Text:

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 in 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<T> >
class basic_string;

```

Beispiel:

```

#include <stdio.h>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
```

5.2.3 Zeichen löschen mit `string::erase()` und einfügen mit `string::insert()`

`string::erase()` und `string::insert()` sind komfortable Methoden um Zeichen aus einem String rauszulöschen 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"
```

5.2.4 Umwandlung in Zahlen mit `strtol()` und der Methode `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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
```

5.2.5 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
```

6. Konstantes

6.1 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 dereferenzieren
}
```

Aufruf:

```
int i;
f(&i); //referenzieren der Variablen
```

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

Funktion:

```
void f(CDC** ppDC)
{
    **ppDC = GetDC(); //Inhalt veränd. nach dereferenzieren
}
```

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:

<code>const char* p</code>	↔	<code>char const* p</code>
identisch		

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

6.2 const-Referenzen (C++-Funktionen)

6.2.1 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

6.2.2 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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; //not 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
```

6.3 Read-Only-Member-Funktionen

6.3.1 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
{
    ...
}
```

6.3.2 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}

```

6.3.3 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"
}
```

6.4 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;
}
```

6.5 const statt #define verwenden

6.5.1 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 is not 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;
}
```

6.5.2 Lokale Konstanten einer Klasse

Wenn 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 };
        ...
};
```

6.6 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;
}
```

7. Globales (static-Member)

7.1 static-Member

7.1.1 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.

7.1.2 Zugriff ohne ein Objekt zu instanziiieren

Auf **static**-Variablen kann **immer**, auch ohne das 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.

7.2 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 Programmablauf zum ersten Mal an der Definition der Variablen vorbei läuft.

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;
}
```

7.3 In globalen Funktionen: Virtuelle Methoden der Argumente nutzen

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.

Bsp.:

```
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;
}
```

8. Referenz statt Zeiger benutzen (Zeiger nur für C-Schnittstellen)

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, denn 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 gelöscht) 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, das 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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).

9. Funktionen, Argumente und return-Werte

9.1 Argumente sollten immer Referenzen sein

9.1.1 *const-Referenz statt Wert-Übergabe (Slicing-Problem)*

Man sollte die const-Referenz immer der Wert-Übergabe vorziehen!
Ausnahme: einfache Datentypen wie int.

```

Statt:
void Foo(MyClass Obj)
{
    ...
}

Immer:
void Foo(const MyClass& Obj)
{
    ...
}

```

Gründe:

- Keine zeitintensiven Speicherallokierungen/-freigaben, Konstruktor-/Destruktoraufrufe und Kopien
- Kein unnötiger Speicherverbrauch durch das Anlegen einer lokalen Kopie
- 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.

9.1.2 Referenz statt Zeiger

Man sollte die Referenz immer dem Zeiger vorziehen!

Statt: <pre> void Foo(MyClass* pObj) { ... } Immer: void Foo(MyClass& Obj) { ... } </pre>

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 angewendet 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.

9.2 Argumente: Default-Parameter vs. Funktion überladen

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

Bsp.: Default-Werte ab dem 2-ten 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

Bsp.:

```
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.

9.3 Überladen innerhalb einer Klasse vs. über Klasse hinweg

9.3.1 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.

9.3.2 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.

Bsp.:

```
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.

9.4 return: Referenz auf `*this` vs. Wert

Wie man das Ergebnis einer Methode zurückgibt, hängt davon ab, ob ein lokal in der Methode erzeugtes Objekt oder das Objekt, zu der die Methode gehört, zurückgeliefert werden soll.

9.4.1 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;
}
```

9.4.2 Objekt (Eigner) der Methode zurückliefern: Rückgabe einer Referenz auf **this*

Wenn eine Methode das Objekt verändert, zu welcher 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;
}
```

9.4.3 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 ($\rightarrow$ 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, nur wenn der Referenzzähler Null ist. Andernfalls wird zunächst nur ein Lösch-Flag gesetzt. Wenn nun der letzte Nutzer des internen Strings sich abmeldet (Referenzzähler geht auf Null), dann wird der Speicher sofort freigegeben, da das Lösch-Flag gesetzt ist. Hierbei ist also sehr genau darauf zu achten, das nie ein **ReleaseBuffer()** vergessen wird, da man sonst ein klassisches Speicherloch produziert.

9.5 return-Wert nie direkt an ein 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 Argument
{
    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.

10. Smart-Pointer

10.1 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.

10.2 Smart-Pointer für die Speicher-Verwaltung

10.2.1 *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** zertört. 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).

10.2.2 Was zu beachten ist

- Falls der Typ eine Klasse ist muß der Zugriffoperator 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)... (impl.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, was zum Beispiel 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;
}
```

implementiert werden.

10.2.3 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;
}

```

10.2.4 *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;
}
```

10.2.5 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 pStr2;
    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.

10.3 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.

11. new/delete

11.1 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 (pObj) genau dort konstruieren:

→ **Placement-new**

```
pObj = new (pBuf) MyClass(...);
```

Beachte: Der Programmierer kann einen Konstruktor nicht explizit aufrufen!

11.2 Allgemeines zu delete

Das eingebaute Sprachelement **delete** bewirkt durch

```
delete pObj;
```

folgende, **durch den Compiler generierte Dinge**:

a) Destruktor aufrufen (kann der Programmierer nicht explizit):

```
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);
```

11.3 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!

11.4 Allgemeines zu new[]/delete[]

11.4.1 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;
}
```

11.4.2 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.

11.5 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.

11.6 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;
}

```

11.7 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];
```

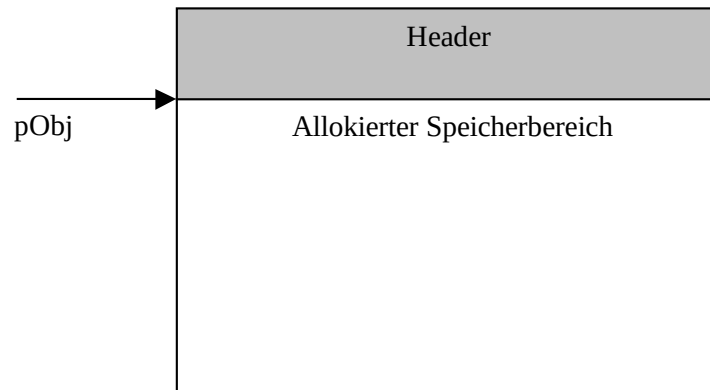
delete

→ 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;
}

```

11.8 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 handle 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;
}
```

11.9 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 frei zu geben.

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;
}

```

11.10 Heap-Speicherung erzwingen/verbieten

11.10.1 Allgemeines zur Speicherung von Objekten

Es gibt 3 Lokalitäten, wo Objekte gespeichert werden können:

a) Statischer Speicher

→ **Statische** Objekte

Hierzu gehören auch **globale** Objekte, da sie immer sichtbar sind und somit ihr Inhalt sich nicht durch Unsichtbarkeit ändern kann

```
static MyClass Obj(...);
static const MyClass Obj(...);
int Func(...);
```

b) Heap

→ Mit **new** erzeugte Objekte

```
MyClass* pObj = new MyClass;
MyClass* pObj = new MyClass(...);
MyClass* pObj = new MyClass[256];
```

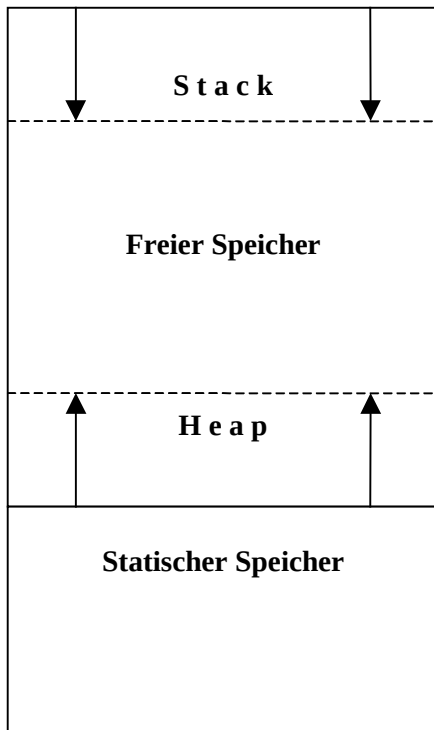
c) Stack

→ Direkt konstruierte Objekte

Hierzu gehören auch die Member-Variablen des Objektes.

```
MyClass Obj;
MyClass Obj(...);
```

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:



11.10.2 Heap-Speicherung erzwingen (protected-Destruktor)

Um die Heap-Speicherung 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 bei dem 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.

Bsp.:

```
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;
}

```

11.10.3 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:

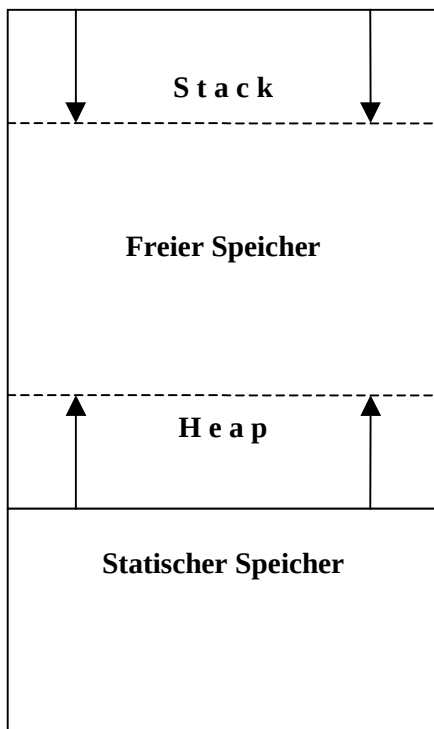
```
MyClass* p = new MyClass(8);
```

Allerdings wird es behandelt wie ein Stack-Objekt (Stack im Heap).

12. Statische, Heap- und Stack-Objekte

12.1 Die 3 Speicher-Arten

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:



Im folgenden werden Tatsachen, die eigentlich bis hierher schon klar sein sollten, nochmal aus anderer Sicht dargestellt. Es wird gezeigt, wie man Klassen schreibt um gezielt eine der 3 Speicherarten zu **erzwingen**.

12.2 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;
}
```

12.3 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;
}
```

12.4 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;
}
```

13. Programmierung einer Klasse

13.1 Allgemeines

13.1.1 Folgende Fragen sollten beim Entwurf einer Klasse beantwortet werden

- Wo (Statischer Speicher, Stack, Heap) sollen die Objekte erzeugt werden?
 - Statisch: Verstecken von Konstruktor und Destruktor; nur `static`-Methoden
 - Heap: Verstecken des Destruktors und implementieren von `Destroy()`-Funktion
 - Stack: Verstecken von `operator new()` und `operator new[]()`
- 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 und 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 nicht implementieren
- Soll die Klasse für verschiedene Daten-Typen der enthaltenen Member definiert werden?
 - ggf. als Template definieren (Bsp.: Liste)

13.1.2 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 ($\rightarrow$ vtable $\rightarrow$ RTTI $\rightarrow$ 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[]()

13.1.3 Durch den Compiler automatisch generierte Methoden sind zu 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 |
| • operator=() (bitweise Kopie): | Falls Zuweisung im Code vorkommt |
| • operator&(): | Falls entsprechende Referenzierung im Code vorkommt |

Bsp.:

```
class Empty
{
};

int main()
{
    Empty Obj1; //-> Compiler gener. Default-Konstruktor u. 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 soweit in die Tiefe geht, bis er auf einen Copy-Konstruktor oder einen `operator=()` oder einen eingebauten elementaren Datentyp ($\rightarrow$ 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 !!!
};
```

13.1.4 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() {}
        ...
}
```

13.1.5 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)

13.1.6 Mehrdeutigkeiten (ambiguous) erkennen

Problem:

Wenn der Compiler durch automatische implizite Typumwandlung mehr als eine passende überladene Funktion für ein Argument findet, dann streikt er: "Error - ambiguous".

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: ambiguous:
                  //   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 Mehrfachvererbung 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 - ambiguous

    return 0;
}
```

Abhilfe: Expliziter Zugriff auf die Methode einer Basisklasse

Zum obigen Beispiel:

```
class MyClass : public MyBase1, public MyBase2
{
    public:
        void Do() { MyBase1::Do(); }
};
```

13.2 Der Konstruktor

13.2.1 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 **Initialisierungsliste** 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
- c) Die Initialisierungsliste ist **effizienter** als Konstruktion + Zuweisung per `operator=()`

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 MyClass
{
    public:
        MyClass(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 MyClass
{
    public:
        MyClass(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 sie 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.

13.2.2 Keine virtuellen Methoden im Konstruktor aufrufen

Innerhalb eines Konstruktors sollte man keine virtuellen Methoden aufrufen, denn erst nach kompletten Durchführung der Konstruktion des Objektes steht eindeutig fest, wohin der Methodenaufruf geleitet werden muß (vtable).

13.2.3 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));
```

13.3 Der Destruktor

13.3.1 Generalisierung ("is-a"): Basisklassen sollten einen 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 **vtable**. Da das Vorhandensein einer vtable Bedingung dafür ist, das 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"); }
};
```

13.4 Zuweisung per operator=()

13.4.1 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;
}
```

13.4.2 Referenz auf *this zurückliefern

Der Zuweisungs-Operator muß eine Referenz auf `*this` zurückliefern!

Gründe:

- Verkettung muß möglich sein:

Bsp: `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.

13.4.3 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;
}
```

13.5 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;
}
```

13.6 Virtuelle Clone()-Funktion: Heap-Kopie über Basisklassen-Zeiger

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)
    {
        printf("Obj-ID: %d\n", (*it2)->GetID()); //Kopie untersuchen
        delete (*it2); //Kopie löschen
        (*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;
}
```

13.7 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.

13.7.1 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;
}

```

13.7.2 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;
    }

```

13.7.3 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;
}

```

13.8 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 das ein Hardware-Hersteller für den Betrieb seiner Hardware nur ein C-Interface bereitstellt und neben der Header-Datei nur eine vor-compilierte Binärdatei mitliefert. Damit schützt er seinen Code vor fremden Augen und verhindert das 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.

14. Richtiges Vererbungs-Konzept

14.1 Allgemeines

14.1.1 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 Basisdestruitor virtuell ist
    return 0;
}
```

14.1.2 Copy-Konstruktor muß auch Basisklassen-Member belegen

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 MyClass : public MyBase
{
    public:
        MyClass(int nID,int nValue):MyBase(nID),m_nValue(nValue) {}
        ~MyClass() {}
        void SetValue(int nValue) { m_nValue = nValue; }
        int GetValue() const { return m_nValue; }
    private:
        int m_nValue;
};

int main()
{
    MyClass Obj(1,2);
    MyClass ObjClone(Obj);
    printf(    "Obj-Value = %d ObjClone-Value = %d",
              Obj.GetValue(),
              ObjClone.GetValue()
    );

    return 0;
}
```

14.1.3 Statischen/dynamischen Typ und statische/dynamische Bindung beachten

Der Compiler bindet **nicht-virtuelle Funktionen** statisch, d.h. ihr Code steht bereits nach der Compilierung fest. **Virtuelle Methoden** werden dynamisch gebunden. Das bedeutet, daß erst zur Laufzeit feststeht, welcher Code zu verwenden ist (vtable).

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 **vtable** 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;
}
```

14.1.4 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;
}
```

14.1.5 *public-, protected- und private-Vererbung gezielt verwenden*

➤ Default-Zugriff auf Klassen bzw. Strukturen

Klassen: Default-Zugriff ist **private**

<pre>class MyClass { ... };</pre>	=	<pre>class MyClass { private: ... };</pre>
---------------------------------------	---	--

Strukturen: Default-Zugriff ist **public**

<pre>struct MyClass { ... };</pre>	=	<pre>class MyClass { public: ... };</pre>
--	---	---

➤ 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;
};
```

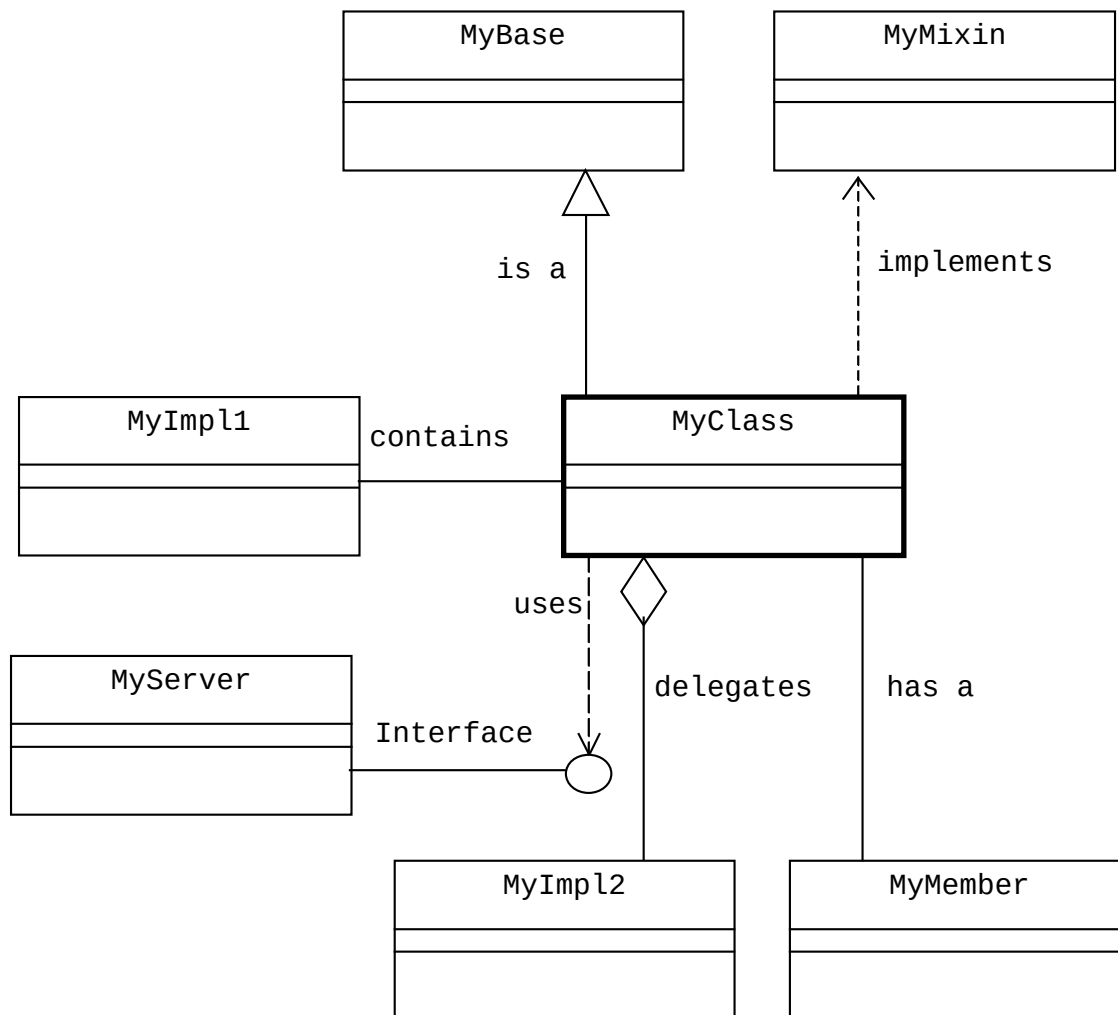
→ Ein MyClass-Objekt **benutzt ein** MyInterface-Objekt, welches in der Regel in ein anderes Objekt eingebettet ist (MyServer::XInterface::Func())

- 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:



14.1.6 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.

14.1.7 Rein virtuelle Methoden immer, wenn keine generalisierte Implementierung 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;
}

```

14.2 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 Compilieren 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();
}

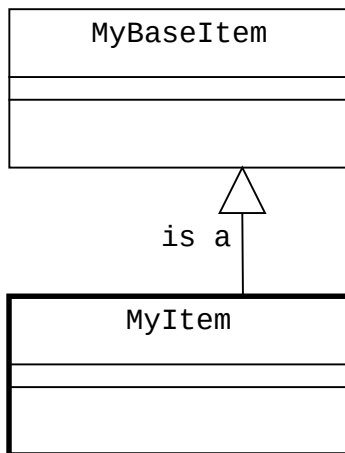
void PrintInfo(MyBaseItem* pObj)
{
    pObj->PrintObjInfo();
}

int main()
{
    MyBaseItem Obj1;
    PrintInfo(&Obj1);
    PrintInfo(Obj1);

    MyItem Obj2("Green");
    PrintInfo(&Obj2);
    PrintInfo(Obj2);

    return 0;
}

```



14.3 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.

Ein 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:

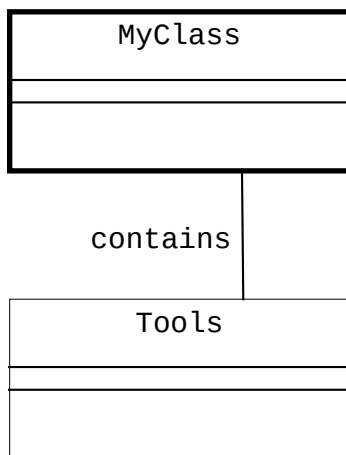
```

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;
}
```



14.4 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;

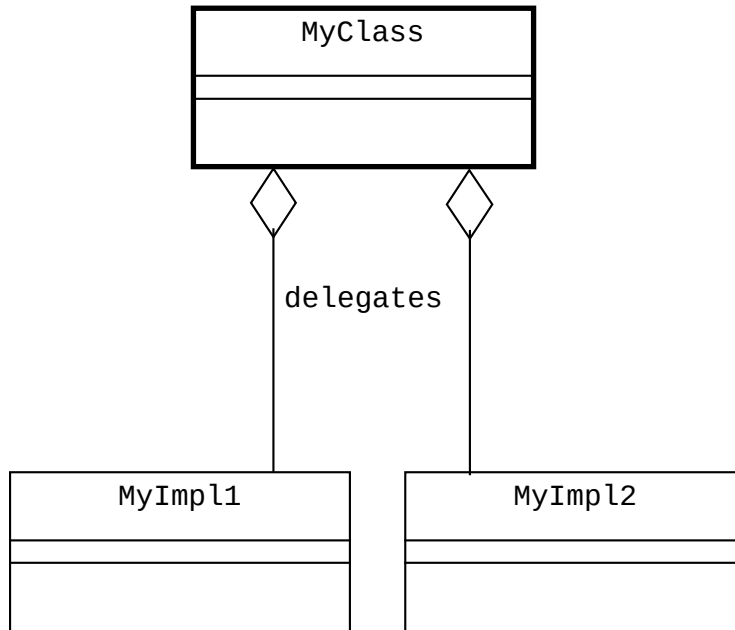
class MyClass
{
    public:
        MyClass() : m_pImpl1(NULL),m_pImpl2(NULL) {}
        void Do();
        void Verify();
    private:
        MyImpl1* m_pImpl1;
        MyImpl2* m_pImpl2;
};

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;
}
```



14.5 Schnittstellen (Abstract Mixin Base Classes) 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:

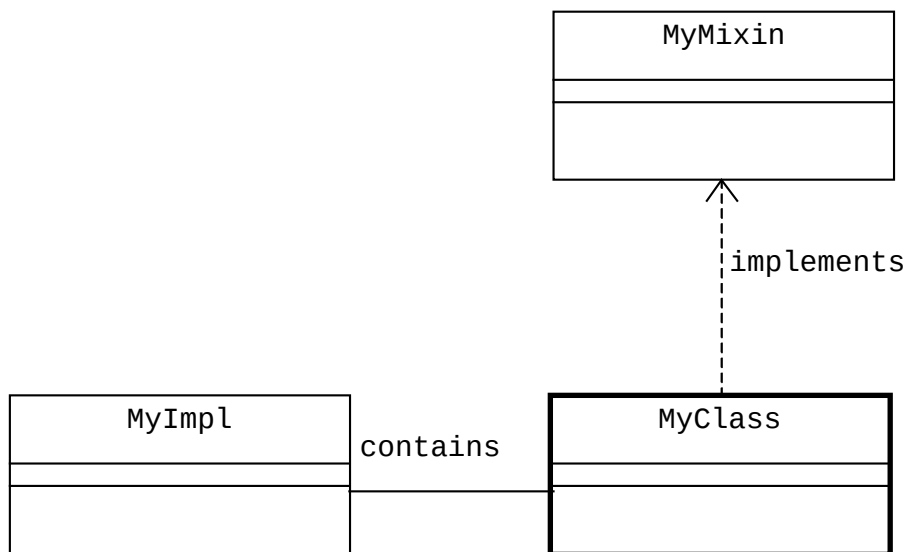
```
class MyMixin
{
    public:
        virtual int Meth1() = 0;
        virtual int Meth2() = 0;
        virtual int Meth3() = 0;
};

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;
}
```

Das Ganze in UML:



14.6 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:** Es ist 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**: Der Empfang über verschiedene Schnittstellen ist technisch unterschiedlich realisiert

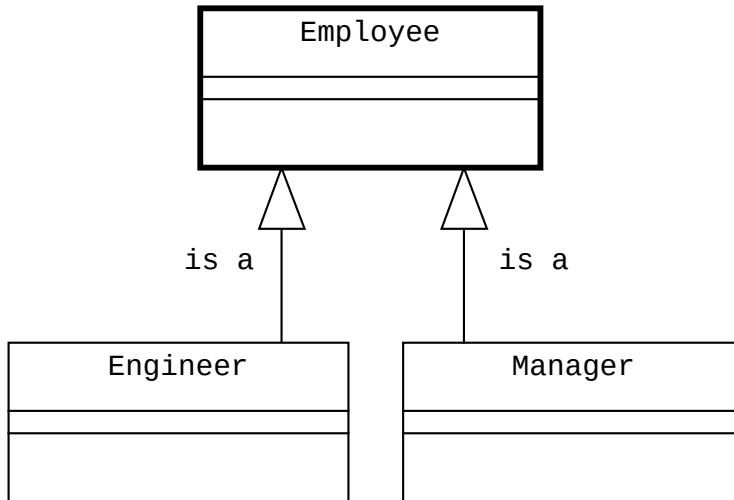
```
class DataInterface
{
    public:
        virtual ~DataInterface() {} //Hack: nicht wirklich inline
        virtual void Init() = 0;
        virtual char GetChar() = 0;
        virtual void SendChar(char c) = 0;
};

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);
};
```

14.7 Verknüpfung konkreter Klassen mittels abstrakter 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.

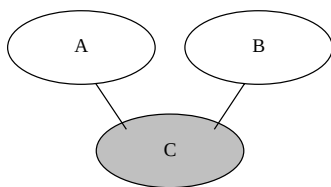
14.8 Erbung aus mehreren Basisklassen (multiple inheritance) vermeiden

Die folgenden Konzepte finden bei multiple inheritance Anwendung und sprechen dagegen das Erben aus mehreren Basisklassen vorzusehen:

14.8.1 Expliziter Zugriff (oder using)

Wenn Methoden den gleichen Namen haben, dann muss explizit (oder mittels using) darauf zugegriffen werden.

Bsp.:



Wenn A und B die Methode Func() deklarieren/implementieren, dann kann C nur folgendermaßen eindeutig auf eine davon zugreifen:

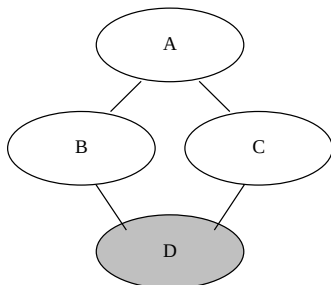
```

A::Func()
B::Func()
  
```

Dies ist **unhandlich** und das **Konzept der virtuellen Funktionen**, die von der Basisklasse zur abgeleiteten Klasse weitergeleitet werden, **wird lahmgelegt**.

14.8.2 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 (ambiguous):



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;
}
```

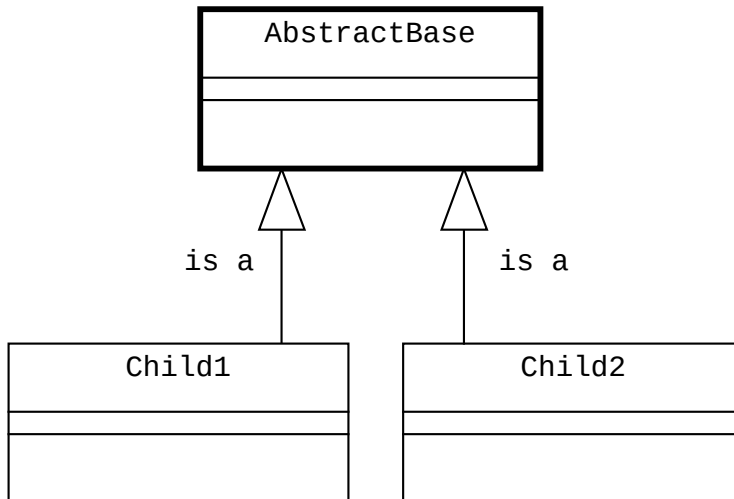
Probleme, die hierbei auftreten:

- Beim Entwurf von A, B und C kann man noch nicht wissen, daß später mal eine Diamant-Struktur entstehen wird, d.h. das 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 (ambiguous).

14.9 Zuweisungen nur zwischen Childs gleichen Typs 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:
        AbstractBase& operator=(const AbstractBase& Obj);//versteckt
        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;
}

```

15. Nutzer einer Klasse von Änderungen entkoppeln

15.1 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.

15.2 Header-Dateien: Forward-Deklaration ist `#include` vorzuziehen

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
};
```

15.3 Delegation bzw. Aggregation

Ein Konzept, welches zwar den Aufbau der Software maßgeblich ändert, was 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-Klasse) hält also lediglich einen **Zeiger auf die Implementierung** und beherrscht damit nicht die internen Implementierungsmethoden 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.

Beispiel:

```
//*****
// 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.

Beispiel:

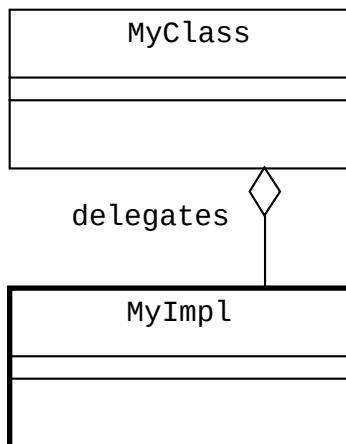
```
//*****
// MyImpl.h:
//*****

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); }
```



15.4 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:
//*****

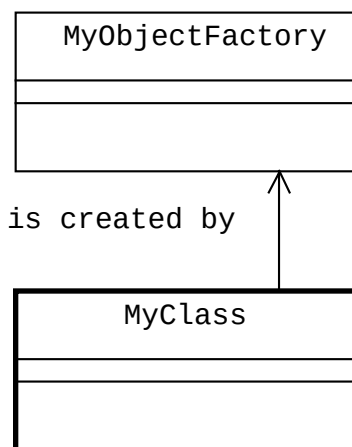
#include "MyObjFactory.h"

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;
}

```



16. Code kapseln

16.1 Beliebig viele Kopien erlaubt: Funktions-Objekte (operator())

Man kann Algorithmen innerhalb 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, instanziert 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;
}
```

16.2 Nur eine Kopie erlaubt: Statische Objekte (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;
}
```

17. Operatoren

17.1 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' und 'Zuweisung'

    MyClass Obj2(50);

    MyClass Obj3(Obj2-Obj1); //'Operator -' u. 'Typumwandlung von int'
    int i = Obj3;           //'Typumwandlung nach int'

    MyClass Obj4(123);
    if(Obj3 == Obj4)        //'Vergleich'
        return 1;

    return 0;
}

```

17.2 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, das **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:

```

//Unäre 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;
};

```

```
//Binäre 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;
}
```

17.3 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;
```

17.4 Kommutativität: Globale binäre 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 (Anmerkung: in Java und C# ist alles ein Objekt, auch int) 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 (ambiguous).

```

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!

17.5 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 rechts 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 links stehende Operator zuerst** ausgewertet:

```

51.  x , y

```

17.6 Präfix- und Postfix-Operator

17.6.1 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.

17.6.2 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 sieht man deutlich am Beispiel des Operators **++**:

```
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
}
```

17.6.3 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
}
```

17.7 Der Komma-Operator ,

Der Komma-Operator , wird in folgenden Fällen angewendet:

➤ **Definition von Variablen**

Beispiel:

```
int a,b,c;
```

➤ **Im Kopf der for-Schleife**

Beispiel:

```
int x = 0, y = 0;
for(x = 0, y = 10; x < y; ++x, --y)
{
    ...
}
```

Die Reihenfolge der Bearbeitung ist so, daß zuerst der Ausdruck links vom Komma und dann der Ausdruck rechts vom Komma ausgewertet wird.

18. Datentypen und Casting

18.1 Datentypen

➤ **Arithmetische Datentypen (built-in)**

- 1.) Abzählbare Werte (Integrale Datentypen)
bool, char, int, short, long
- 2.) Gleitkommazahlen
float, double

Zum Typ **bool**:

$$\text{bool } b = \begin{cases} \text{false falls } 0 \\ \text{true sonst} \end{cases}$$

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{M0,DI,MI,D0,FR,SA,S0};
enumDay eToday;
enumDay eYesterday;

if(eYesterday == M0)
    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 ein kann:

static_cast:	Statischer cast
dynamic_cast:	Sicheres cast von Zeigern oder Referenzen
const_cast:	Konstantheit weg-casten
reinterpret_cast:	Wilder cast zwischen Zeigern auf verschiedenste Objekte

18.2 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;
}

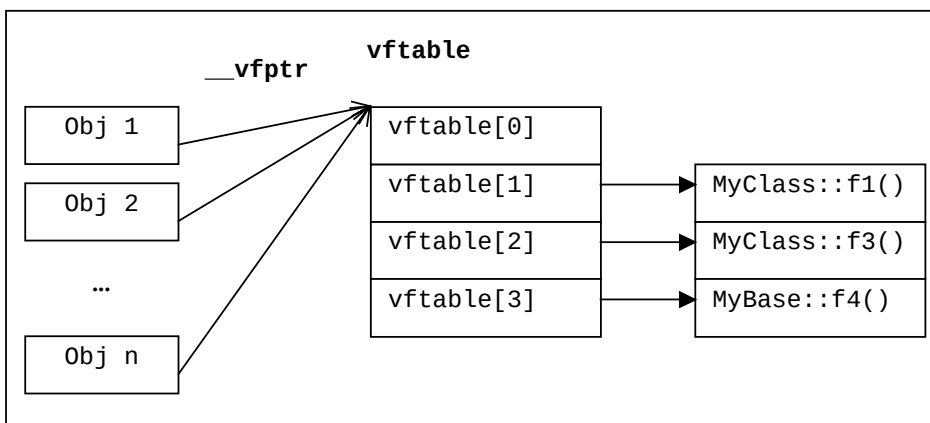
```

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.

18.3 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;
}
```

18.4 dynamic_cast: Sicherer cast von Zeigern oder Referenzen

18.4.1 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*>(pObjekt)
if (!pNewObjekt)
{
    ... //Fehler
}
```

dynamic_cast von Referenzen:

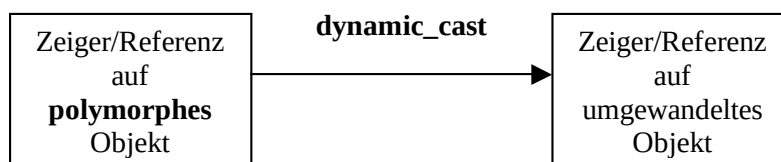
Hier kann keine Überprüfung auf NULL erfolgen → try-catch-Block:

```
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 Upcast
    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;
}

```

18.4.2 *dynamic_cast zur Argument-Überprüfung bei Zeiger/Referenz auf Basisklasse*

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;
}

```

18.5 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;
}

```

18.6 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;
}

```

18.7 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()
```

19. In Bibliotheken Exceptions werfen

19.1 Allgemeines

Wenn man eine Software-Bibliothek schreibt, dann kann es vorkommen, daß die geschriebenen Funktionen, die der Anwender nutzen kann auf Grenzen stoßen (Bsp.: Festplatte voll, Datei gesperrt, ...) und ihre Aufgabe nicht erfüllen können. In dem Fall können sie jedoch nicht reagieren, indem sie z.B. eine Fehler-Meldung an die Benutzeroberfläche bringen. Ein Grund dafür ist, daß sie nicht wissen was der globale Zusammenhang für den Nutzer ist (Bsp.: Konfiguration konnte nicht gespeichert werden) und somit keinen sinnvollen Text finden können. Der andere Grund ist, daß sie keinen Zugriff auf das Interface der Benutzeroberfläche haben. Die Bibliotheks-Funktionen können weiterhin keine Entscheidung über die weitere Vorgehensweise treffen, da sie den Kontext des Gesamtprozesses nicht kennen und von der Hierarchie her überhaupt keine Entscheidungsbefugnis dazu haben. Genau deshalb hat man das Konzept der Exceptions erfunden und **nicht etwa um return-Werte zu ersetzen**.

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 der 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 werden 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()";
}

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.

Beispiel:

```
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.**

19.2 Exceptions per Referenz fangen

Für das Fangen eines Objekts 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.

19.3 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:

- ExType fängt auch alle abgeleiteten Typen
- ExType kann auch ein Zeiger-Typ sein (void* fängt 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.

19.4 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;
}
```

19.5 Exception-Spezifikation

19.5.1 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.

Bsp.:

```
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

19.5.2 Spezifikationswidrige Exceptions abfangen: *set_unexpected* (NICHT bei Visual C++)

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 compiliert wird.
- **Man benötigt 3 catch-Blöcke** um sicher alles zu fangen

19.5.3 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** zurück liefert. 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;
}
```

20. Die STL (Standard Template Library)

20.1 Allgemeines

20.1.1 Einführung

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 einen solchen, der auf einer Listenstruktur basiert (Bsp.: `list`). **Assoziative Container** bestehen aus 2 Listen: Die Liste der Schlüsselfelder (key) und die Liste der Wertefelder (value). Wenn man einen Wert (value) reinschreibt, 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 **verschiedenste Implementierungen** der STL:

- STL von ISO/ANSI-C++ (ISO98)
- STL von HP
- STL von SGI (Silicon Graphics)
- STL von Roguewave
- STL von Microsoft (keine hash_map)
- STL von Stingray
- ...

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 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** eingefügen.

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.

20.1.2 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
size_type	→	Einheit der Längen- bzw. Größenangaben
difference_type	→	Einheit des iterierens zum nächsten Element
iterator	→	Zeiger value_type* der bei ++ <u>nach vorne</u> dreht
const_iterator	→	Zeiger const value_type* der bei ++ <u>nach vorne</u> dreht
reverse_iterator	→	Zeiger value_type* der bei ++ <u>zurück</u> dreht
reverse_const_iterator	→	Zeiger const value_type* der bei ++ <u>zurück</u> dreht
reference	→	Referenz value_type&
const_reference	→	Referenz const value_type&

Wichtige STL-Methoden:

Allgemeiner Zugriff:

empty()	→	Prüft ob 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
end()	→	Zeiger <u>hinter</u> das letzte Container-Element
rbegin()	→	Zeiger auf das letzte Container-Element
rend()	→	Zeiger <u>vor</u> das erste Container-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 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 an den Anfang einer unsortierten Sequenz anhängen
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 bestimmten 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
replace_copy()	→	Elemente kopieren und dabei Inhalt ersetzen
replace_copy_if()	→	Elemente kopieren und dabei Inhalt ersetzen
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 plazieren
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 kleinerem Inhalt bestimmen
max()	→	Das Element von Zweien mit dem größeren Inhalt bestimmen
find()	→	Bestimmtes Element finden
find_if()	→	Bestimmtes Element finden
find_first_of()	→	Bestimmtes Element finden
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 Seq. finden
upper_bound()	→	Letztes Element mit bestimmtem Inhalt in sortierter Seq. finden
search()	→	Nach einer Teilsequenz mit bestimmtem Inhalten suchen
search_n()	→	Nach einer Teilsequenz mit n übereinstimmenden Inhalten suchen
find_end()	→	Letztes Auftreten einer Teilsequenz suchen
equal_range()	→	Teilsequenz mit bestimmten Inhalt in sortierter Sequenz finden
mismatch()	→	Erstes unterschiedliches Element in 2 Sequenzen finden
next_permutation()	→	nächste Permutation in lexikographischer Reihenfolge finden
prev_permutation()	→	vorherige 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

20.2 STL-Header-Dateien

20.2.1 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>
}
```

20.2.2 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;
}
```

20.3 Generierung von Sequenzen über STL-Algorithmen

20.3.1 *back_inserter()*

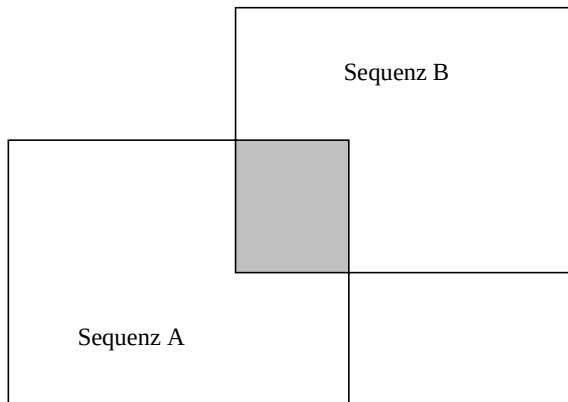
Die Funktion `back_inserter()` dient dazu, die von einem STL-Algorithmus generierte lokale Sequenz in einen vom Aufrufer bereitgestellten Container zu kopieren. Die Anwendung von `back_inserter()` entspricht also in etwa der Anwendung eines Referenz-Parameters beim Aufruf einer Funktion.

Bsp.:

```
list<MyClass> listDiff;
set_difference(listA.begin(), listA.end(),
              listB.begin(), listB.end(),
              back_inserter(listDiff) );
```

20.3.2 *Schnittmenge (set_intersection)*

Die STL kann eine Schnittmenge zweier Sequenzen ermitteln:

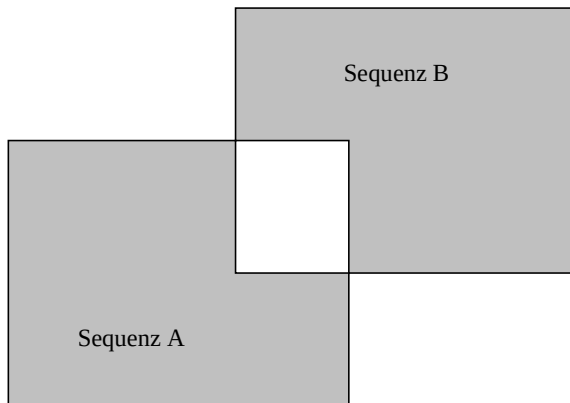


```
listA.sort();
listB.sort();

list<MyClass> listIntersection;
set_intersection(listA.begin(), listA.end(),
                 listB.begin(), listB.end(),
                 back_inserter(listIntersection) );
```

20.3.3 Schnittmenge ausschliessen (*set_symmetric_difference*)

Die STL kann die Schnittmenge zweier Sequenzen von der Gesamtmenge ausschliessen:

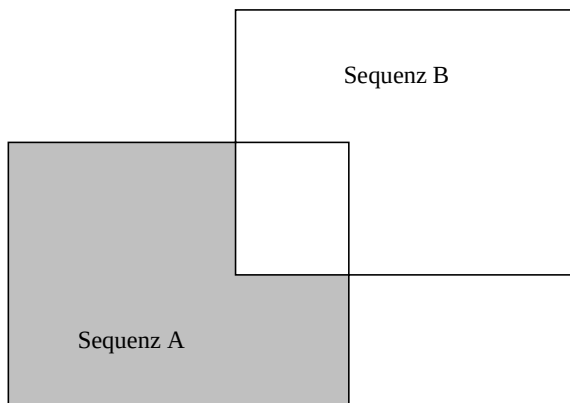


```
listA.sort();
listB.sort();

list<MyClass> listSymDiff;
set_symmetric_difference( listA.begin(), listA.end(),
                          listB.begin(), listB.end(),
                          back_inserter(listSymDiff) );
```

20.3.4 Andere Menge ausschliessen (*set_difference*)

Die STL kann die eine zweite Sequenzen von der Gesamtmenge ausschliessen:

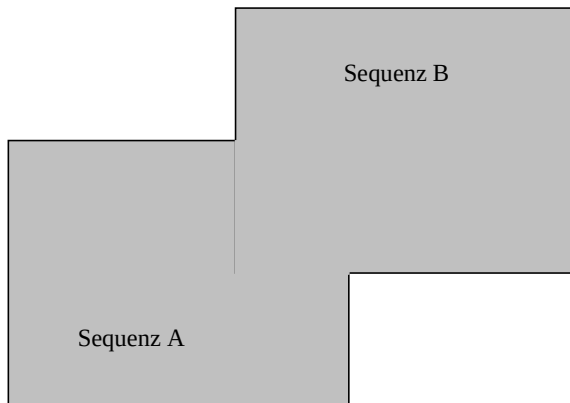


```
listA.sort();
listB.sort();

list<MyClass> listDiff;
set_difference( listA.begin(), listA.end(),
               listB.begin(), listB.end(),
               back_inserter(listDiff) );
```

20.3.5 Vereinigungsmenge (*set_union*)

Die STL kann eine Vereinigungsmenge zweier Sequenzen ermitteln:

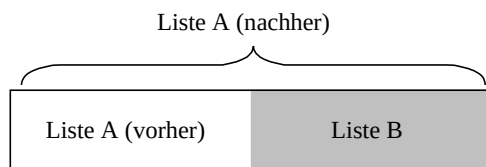


```
listA.sort();
listB.sort();

list<MyClass> listUnion;
set_difference( listA.begin(), listA.end(),
               listB.begin(), listB.end(),
               back_inserter(listUnion) );
```

20.3.6 Liste anhängen (*list::insert*)

Die STL kann eine Liste an eine andere Liste anhängen:



```
listA.insert( listA.begin(), listA.end(),
             listB.begin(), listB.end() );
```

20.4 Wichtige Regeln

20.4.1 Einbinden der STL

Es kann sein, dass der Compiler eine Warnung bringt, weil er Klassen-Namen der STL kürzt (C4786). Diese Warnung kann man mit '**#pragma warning(disable:4786)**' generell unterbinden. Das gleiche gilt für C4715 (unreferenced inline function has been removed). **Danach** sind die Algorithmen über **#include <algorithm>** und die benötigten Container einzubinden. **Danach** nimmt man **std** in den global namespace auf, d.h. alle Namen im namespace std werden zu globalen Namen.

Beispiel:

```
//-----
// STL einbinden:
//-----

//Keine Warnung wg. verkürzten Namen oder beseitigten inline-Funktionen:

#pragma warning(disable:4786)
#pragma warning(disable:4514)

//Algorithmen:

#include <algorithm>

//Container:

#include <list>
#include <set>

//std in global space aufnehmen:

using namespace std;
```

20.4.2 Für die Objekte müssen die benötigten Operatoren implementiert werden

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) : m_nID(nID) {} //Default
        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;
}
```

20.4.3 *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
}
```

20.4.4 Löschen nach find(): Immer über den 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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!

20.4.5 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}

```

20.5 Beispiele für die Verwendung der Container

20.5.1 *list*: Auflistung von Objekten, wobei Objekte mehrfach vorkommen können

```

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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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()`**

→ Der allgemeiner Algorithmus `find()` ist anzuwenden, wobei zu beachten ist, das hierbei nur das erste Vorkommnis in der `list` gefunden wird. Man kann jedoch die `list` vorher sortieren und `unique` machen. In dem Fall ist es jedoch effektiver von vorn herein eine `set` zu benutzen, da dort bereits beim einfügen sortiert wird, was schneller geht.

```

it = find(listObjs.begin(), listObjs.end(), Obj1);

```

- **Nur `remove()` löscht sicher alle Vorkommnisse eines Wertes**

→ Wenn man alle Vorkommnisse eines Wertes löschen will ist **nicht** `erase()` zu verwenden

- **Die Reihenfolge ist fix**

→ Nur Algorithmen wie `sort()` können etwas an der Reihenfolge der Objekte in der `list` ändern

20.5.2 set: Aufsteigend sortierte Menge von Objekten, die nur einmal vorkommen

```

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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}

```

20.5.3 map: Zuordnung von Objekten zu einem eindeutigen Handle

```

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

int main()
{
    MyClass    Obj1(1);
    MyClass    Obj2(2);
    MyClass    Obj3(3);

    map<long,MyClass> mapHandleToObjs;
    mapHandleToObjs[11L] = Obj3;    //Element einfügen
    mapHandleToObjs[102L] = Obj1;    //Element einfügen
    mapHandleToObjs[1026L] = Obj2;    //Element einfügen

    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]!!!

```

20.5.4 *map: Mehrdimensionaler Schlüssel*

Um einen mehrdimensionalen Schlüssel zu verwenden definiert 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}

```

20.5.5 *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 zu 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
```

Besonderheiten bei `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').

20.5.6 *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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
```

20.6 hash_map

20.6.1 hash_map für Nutzer von Visual C++

Da nicht alle Compiler (Bsp.: Visual C++ 6.0) eine STL mit **hash_map** anbieten, kann man sich die STL von SGI (Silicon Graphics) unter <http://www.stlport.org> herunterladen (Freeware). Diese STL liegt zunächst nur als Quell-Code vor. Für Visual C++ 6 sei hier das Einbinden in ein Projekt gezeigt:

- In der Datei "**stlport\stl_site_config.h**" sind folgende Einstellungen **vor dem Bau** der STL vorzunehmen:

- o **Multithreading** aktivieren:

```
//#define _NOTHREADS //aktiviert
```

- o 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

```
#pragma warning(disable:4514) //no warning 'unrefer. inline
                                //function has been removed'
#pragma warning(disable:4786) //no warning 'truncated identifier'
```

20.6.2 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**.

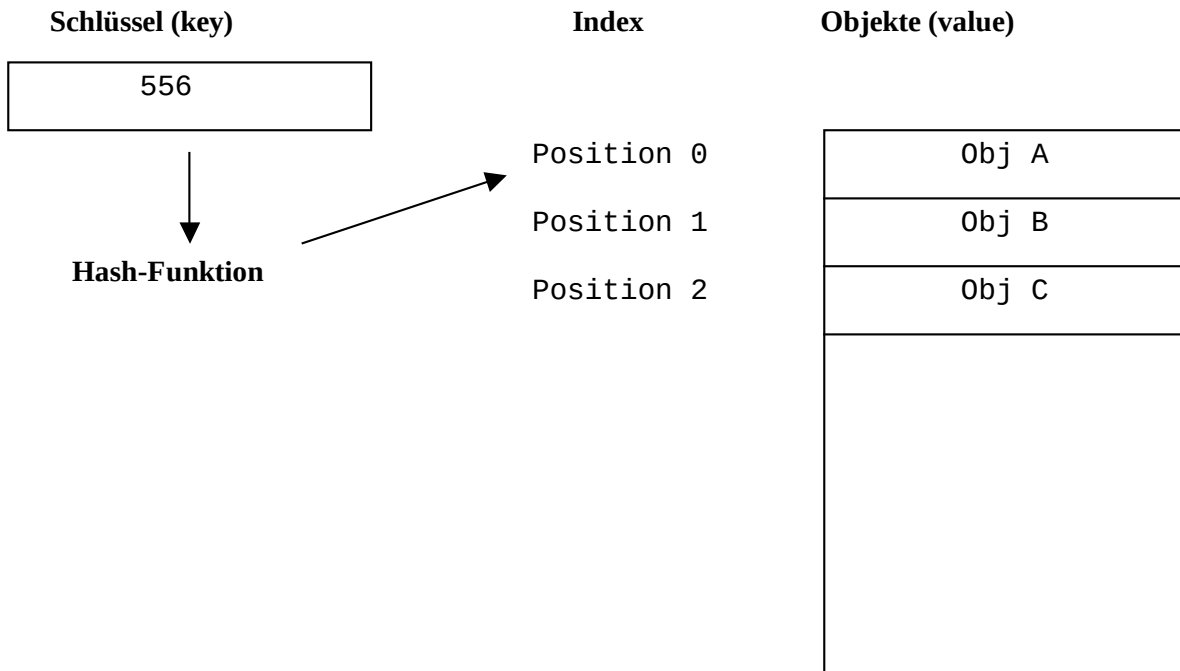
Prinzip der map:

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** sollte einen **eindeutigen Index liefern**, d.h. zwei unterschiedliche keys sollten nicht zum gleichem Index führen. Ist dies nicht der Fall, dann 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
- **Keine Sortierung** möglich
- **Bereitstellung der hash-Funktion**
Benutzt man eine selbst geschriebene Klasse als key, dann muß man auch die hash-Funktion selbst programmieren.

20.6.3 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 dürfte es schwierig sein, die STL zu überbieten.

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 des Strings ermittelt. Man verwendet die 8-Bit-ASCII-Codes der Zeichen zum errechnen eines Indexes:

"ABCD" → 0xD₁D₀C₁C₀B₁B₀A₁A₀

```
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:

Der key besteht aus Strings, die 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:

```
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;
}
```

20.7 Facets und Locales

Facets beschreiben länderspezifische Merkmale (z.B. wie das Datum oder die Währung angezeigt wird). **Locales** fassen die Facets eines Landes als eine Gruppe zusammen.

```
class locale
{
    public:
        typedef int category;

        locale();
        explicit locale(const char *s);
        locale(const locale& x, const locale& y, category cat);
        locale(const locale& x, const char *s, category cat);
        bool operator()(const string& lhs, const string& rhs) const;
        string name() const;
        bool operator==(const locale& x) const;
        bool operator!=(const locale& x) const;
        static locale global(const locale& x);
        static const locale& classic();

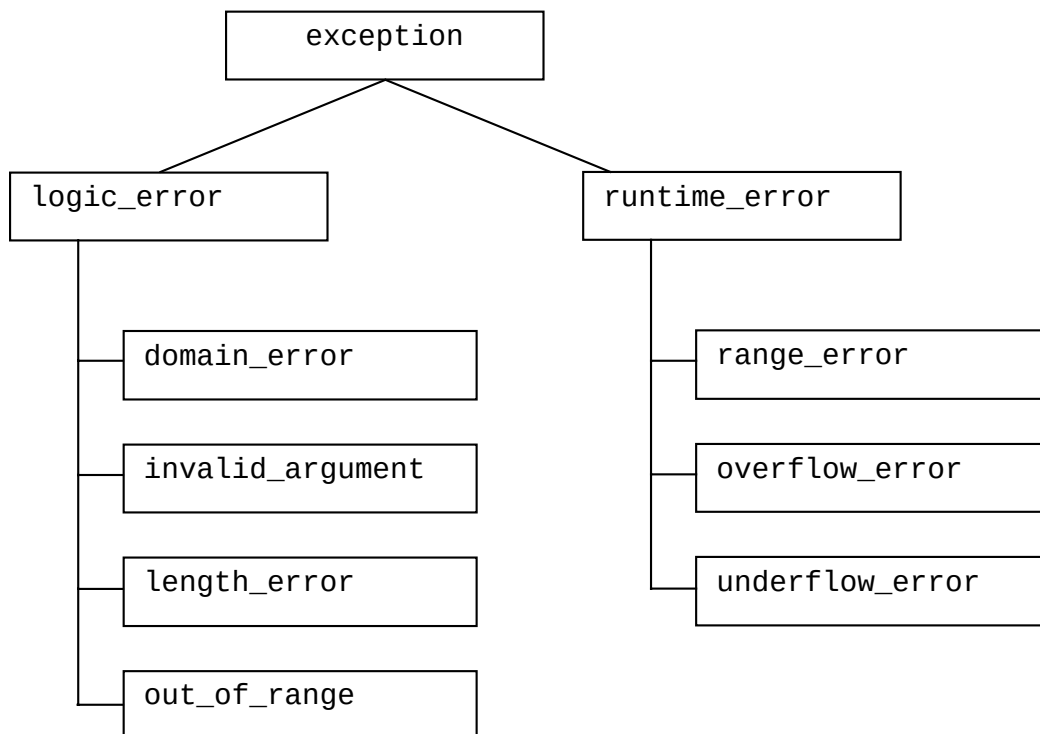
        class facet;
        class id;

        static const category none;
        static const category collate;
        static const category ctype;
        static const category monetary;
        static const category numeric;
        static const category time;
        static const category messages;
        static const category all;
};
```

20.8 Exceptions

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:

```
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();
};
```

Mit **what ()** kann man die Fehlermeldung (String) erfragen.

20.9 Standardisierungs-Dokumente

20.9.1 *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

20.9.2 *ARM (Buch von Margaret Ellis und Bjarne Stroustrup)*

Annotated C++ Reference Manual

→ Ursprüngliche C++-Referenz von 1990

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

21. Arten von Templates

21.1 Class-Templates

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 Template wird folgendermaßen instanziiert, wenn man den Typ `T = MyClass` benutzt:

```
SmartPtr<MyClass> pMyClass(new MyClass);
```

21.2 Function-Templates

21.2.1 Global Function Templates

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.

21.2.2 Member Function Templates

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;
}

```

21.3 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 inkludiert. Dann fügt man eine explizite Instanziierung ein → beim Bau wird eine *.obj-Datei mit dem entsprechenden Binär-Code erzeugt.

➤ Class-Templates explizit instanziiieren

```

#include "MyClass.h"
#include "SmartPtr.h"

template class SmartPtr<MyClass>;

```

➤ Global Function Templates explizit instanziiieren

```

#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.

22. Datenbank-Konsistenz durch Transaktions-Objekte

Der Zugriff auf Datenbanken geschieht in der Regel so, daß ein Datenbank-Server für jeden Nutzer der Datenbank eine eigene **Session** generiert. Das Session-Objekt beinhaltet die Verweise auf die genutzten Datenbank-Objekte und ermöglicht lesen, schreiben, locken, prüfen... . Jeder Nutzer der Datenbank kann zu jeder Zeit Daten (auf die er Leserecht besitzt) lesen, aber nur einer kann die Daten zur selben Zeit locken und damit beschreiben.

Innerhalb einer Datenbank-Session ist es möglich mit **Store()**-Befehlen Daten zu schreiben, falls man den Datenspeicher vorher locken konnte. Statt nun alle Store-Operationen gleich vom Datenbank-Server durchführen zu lassen, kann es manchmal sinnvoll sein (bspw. wenn ein Abbruch durch den Anwender erlaubt ist) die Operationen zunächst in ein **Transaktions-Objekt** zu schreiben, wo sie aufgezeichnet werden. Die Operationen werden erst dann tatsächlich durchgeführt, wenn sicher ist, das der gesamte Transaktionsvorgang bestätigt wird. Hierzu stellt das Transaktions-Objekt die Methode **Commit()** zur Verfügung. Im anderen Fall wird **Abort()** aufgerufen. Das Transaktions-Objekt implementiert **Lock()**, **Save()** und **Unlock()** innerhalb von **Store()**.

Prinzip:

```
MyDataBase Database("localhost","TestDatabase");
MyTransaction Transaction(Database);
try
{
    MyDatabaseObject DBObj(Transaction);
    DBObj.SetFieldValues(1,2,3,"Test");
    DBObj.Store(); //Lock(), Save(), Unlock()
    Transaction.Commit();
}
catch(const MyDBException& DBEx)
{
    Transaction.Abort();
    throw;
}
catch(...)
{
    Transaction.Abort();
    throw;
}
```

Da die Datenbank-Transaktionen eine Technologie für sich darstellen, wurden im Laufe der Jahre sogenannte Transaktions-Server entwickelt, z.B. MTS (= Microsoft Transaction Server = `mtx.exe`) und JTS (= Java Transaction Server). Im Prinzip funktionieren sie ähnlich: Es gibt einen Container (MTS → `mtxex.dll` / JTS → EJB-Container) der zunächst eine Objekt-Fabrik (MTS → Class-Factory / JTS → Home-Object) beherrscht. Diese Fabrik generiert auf Anfrage des Clients das Session-Objekt (MTS → Server-Object / JTS → EJB) und eine Wrapper-Klasse (MTS → Object-Wrapper / JTS → Remote-Object) dafür. Weiterhin gibt es zu dem Session-Objekt ein Context-Objekt. Die Middleware (MTS → COM / JTS → Application-Server) überträgt Requests des Clients zu dem Server-Objekt und Response des Server-Objekts zum Client. Das Session-Objekt kommuniziert mit dem Datenbank-System (DBMS = Database Management System) über ein bestimmtes Protokoll (MTS → ODBC / JTS → JDBC).

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).

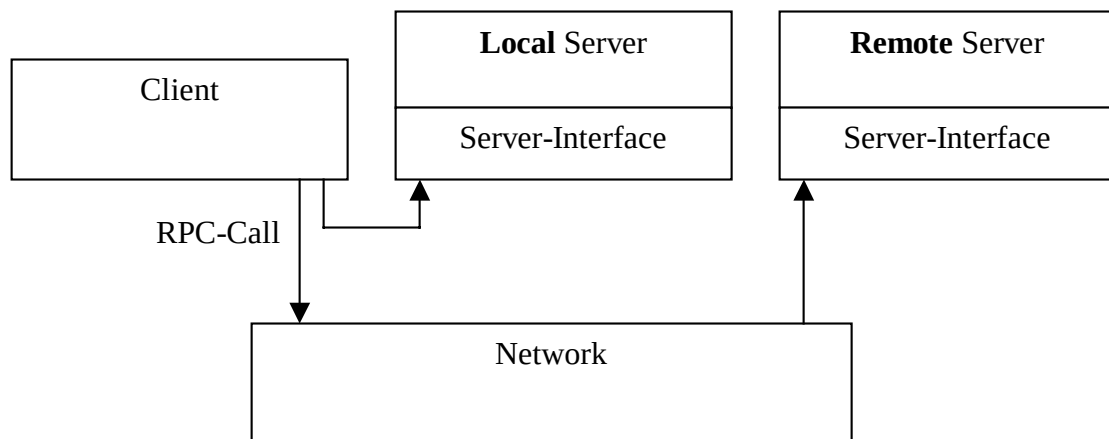
In der Regel möchte man dem Nutzer die Möglichkeit geben zusätzliche Funktionalität zu nutzen ohne daß dieser seinen Code dafür umschreiben muß. Ruft der Nutzer bspw. einen Server auf seinem System (localhost) an und man möchte diese Funktionalität auf ein Netzwerk erweitern (remotehost), dann schaltet 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 weitergibt (localhost) oder in RPC-Calls verpackt (remotehost).

Beispiel:

RPC (Remote-Procedure Call):

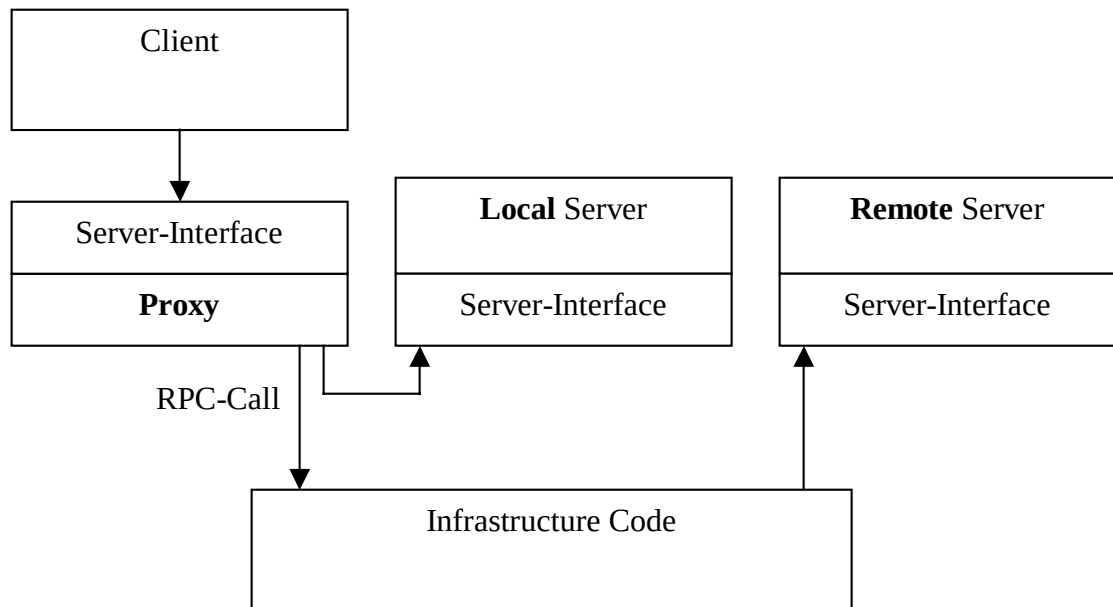
➤ Ohne Proxy:

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 mittels Proxy unterscheiden

Statt direkt mit dem []-Operator auf das Datum zuzugreifen, wird ein **Proxy-Objekt dazwischengeschaltet**. 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:

- **Keine Übergabe an char&-Referenzparameter möglich**

```
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 handle sich um eine Lesezugriff, denn anschliessend hat Besitzer des Zeigers volle Schreibzugriffsmöglichkeit über den Zeiger.

- **Benötigte Operatoren (+=, -=, usw.) müssen in struct CharProxy definiert werden**

24. Double-Dispatching (Kollisionen je nach Partner handeln)

Problem:

Eine Funktion (Handler) soll das Aufeinandertreffen von 2 Objekten handeln. Dabei hängt das, was zu tun ist davon ab, 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 abstrakten 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 passiert also ein **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 und den Partner an den Handler weiter.

Und hier nun das komplette Beispiel 'am Stück':

```
//----- STL: -----

#include <stdio.h>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
```

25. 80/20-Regel und Performance-Optimierung

25.1 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 **PROFILER** suchen

Man benötigt also einen PROFILER. Und zwar solch einen, der genau die Resource (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.

25.2 Zeit-Optimierungen

25.2.1 *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;
}
```

25.2.2 *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
}
```

25.2.3 Unäre Operatoren den binären Operatoren vorziehen

Man sollte die unären Operatoren immer den binären vorziehen, da sie entweder genau so 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));`

25.2.4 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); //-> Konstruktion + Zuweisung (Copy-Konstruktor)
    ...
} //-> 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

25.2.5 *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 was zu finden, aber zugleich **keine** Sortierung der keys erforderlich ist.

25.2.6 *Lokaler Cache (static-hash_map) um Berechnungen/Datenermittlungen zu sparen*

Wenn eine große Menge an Berechnungen bzw. Datenermittlungen (bspw. aus Tabellen) mehrmals durchgeführt werden müssen, dann rentiert sich oft die Implementierung eines lokalen Caches mit Hilfe einer **static-hash_map**:

Beispiel:

```
#include <stdio.h>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}

```

25.2.7 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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
```

25.2.8 *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>
#pragma warning(disable:4514)
#pragma warning(disable:4786)
#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;
}
```

25.2.9 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 per Wert**

→ 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());
}
```

25.2.10 Berechnungen erst dann, wenn das Ergebnis gebraucht wird (Lazy Evaluation)

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.

→ **Keine Berechnungen auf Vorrat!**

25.2.11 Datenermittlung erst dann, wenn die Daten gebraucht werden (Lazy Fetching)

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. Dies gilt vor allem bei zeitintensiven **Tabellen- oder Datenbank-Zugriffen** vor einer graphischen oder textuellen Ausgabe.

→ **Keine Datenermittlung auf Vorrat!**

25.2.12 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);
```

25.2.13 *Kein unnötiges Datenbank-Store*

Das Speichern (Store) auf Datenbankseite 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 ein Datenbank-Store nur so oft einsetzen, wie es die Sicherheit des Datenbank-Systems erfordert, d.h. nur so oft das die Recovery-Mechanismen (Rollback) noch sicher funktionieren.

25.2.14 *Datenbank-UseCases vereinbaren und nutzen*

Zwischen Datenbank-Seite und Applikations-Seite können sogenannte UseCases vereinbart werden, die der Datenbank-Client des DBMS (Database Management Systems) dann implementiert. Hierbei stellt der Datenbank-Client einen großen **Daten-Cache** zur Verfügung, der nach der **Initialisierung** des UseCases alle für den UseCase erforderlichen Daten auf einmal von der Datenbank liest und im **Arbeitsspeicher** hält. Die gesamte weitere Arbeit dieses UseCases geschieht jetzt im Arbeitsspeicher des Clients, wodurch **Datenbank-Server-Calls und Festplattenzugriffe eingespart** werden. Zudem sind die Schreib-/Lese-Zugriffe zwischen Client und Datenbank (LOCAL, localhost, remotehost) speziell auf den UseCase hin optimiert. Nach **Deinitialisierung** des UseCases schreibt der Client die Daten in die Datenbank zurück.

Ein UseCase sollte so gestaltet sein, daß der Code der Applikation nicht anders aussieht, als er ohne UseCase aussehen würde (abgesehen von Initialisierung/Deinitialisierung).

25.3 Speicher-Optimierungen

25.3.1 *Sharing von Code und/oder Tabellen mittels statischem Objekt*

Statt viele Kopien eines **konstanten** 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;
}
```

25.3.2 *Sharing von Code und/oder Tabellen mittels Heap-Objekt*

Statt viele Kopien eines **konstanten** 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.

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 lRefCount = 0L;
    static MyCode* pObj = NULL;

    if(bAddRef) //Objekt referenzieren
    {
        if(!(lRefCount + 1L)) //Überlaufstest
            return NULL;
        if(!(lRefCount++)) //falls erste Referenz
            pObj = new MyCode;
    }
    else //Objekt freigeben
    {
        if(!lRefCount) //Unterlaufstest
            return NULL;
        if(!(--lRefCount)) //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;
}

```

25.3.3 *Sharing von veränderbaren Objekten: Wert auf den Heap (Copy-On-Write)*

Ganz so einfach wie bei **konstanten** Objekten ist es bei **veränderbaren** Objekten nicht. Hier muß ein Nutzer in dem Moment eine eigene Kopie bekommen, sobald er schreibend auf das Objekt zugreift (**Copy-On-Write**). Genau dann wird der Objekt-Wert auch mit einem **Lock** versehen.

Man implementiert dieses Verfahren, indem man einem Objekt eine **verschachtelte Klasse** einverleibt, die den Wert 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 eine solche Klasse 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 letztes 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);
    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()
{
    MyString szHello("Hello"); //Neu: Heap(1)
    MyString szHi(szHello);    //Referenz auf Heap(1)
    MyString szWorld;          //Neu: Heap(2)
    szWorld = szHello;         //Gelöscht: Heap(2), Ref. auf Heap(1)
    //--- An dieser Stelle: Nur noch Heap(1) mit 3 Referenzen ---

    szWorld[0] = 'w';          //Neu: Heap(3), da Copy-On-Write
    szWorld[1] = 'o';
    szWorld[2] = 'r';
    szWorld[3] = 'l';
    szWorld[4] = 'd';
    MyString szText(szWorld);  //Neu: Heap(4), da Lock auf Heap(3)
    szText = "Wie ?";          //Lock auf Heap(4)
    //--- An dieser Stelle: Heap(1), Heap(3) und Heap(4) ---

    return 0;
}

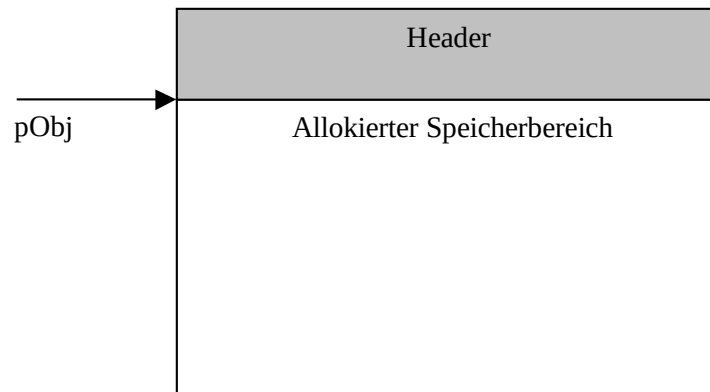
```

25.3.4 *new-Header vermeiden: Speicherpool*

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:

Der erste `new`-Aufruf reserviert einen großen Speicherblock (Pool) und bei weiteren `new`-Aufrufen wird lediglich ein Zeiger auf den noch freien Speicher geliefert. `delete` gibt den Speicher wieder frei und löscht nach der letzten Freigabe den Pool.

Beispiel:

```

#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 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() == (unsigned int)
                                   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;
}

long& MyClass::NumConstructedItems()
{
    static long lNumConstructedItems = 0L;
    return lNumConstructedItems;
}

```

```
int main()
{
    MyClass* p1 = new MyClass; //Neuer Pool und Heap(1)
    MyClass* p2 = new MyClass; //Neuer Heap(2)
    p1->Destroy();             //Frei: Heap(1)
    MyClass* p3 = new MyClass; //Referenziert: Heap(1)
    p2->Destroy();             //Frei: Heap(2)
    p3->Destroy();             //Frei: Heap(1) und Pool
    MyClass* p4 = new MyClass; //Neuer Pool und Heap(1)
    p4->Destroy();             //Frei: Heap(1) und Pool
    return 0;
}
```