

Inhaltsverzeichnis

1.	Einordnung von C++	1
2.	Grundsätzlicher Aufbau eines Projektes.....	3
2.1	Pro Klasse eine *.h und eine *.cpp-Datei.....	3
2.2	Benennung von Verzeichnissen, Dateien und Klassen	5
2.3	Zentrale Header-Datei (Settings.h).....	6
2.4	Der Code muss ohne Warnungen bauen	6
2.5	Mehrere Schichten verwenden (horizontale Teilung)	8
2.6	Client/Server-Modell verwenden (vertikale Teilung)	10
2.7	Das Broker-Pattern (2-Tier-Architektur), CORBA, DCOM.....	12
2.7.1	Allgemeines	12
2.7.2	IDL-Compiler	15
2.8	CORBA mit TAO (The ACE ORB) von Douglas C. Schmidt	17
2.8.1	Allgemeines	17
2.8.2	Code-Beispiel mit GNU C++-Compiler unter LINUX	19
2.8.3	Code-Beispiel mit Visual C++ 6.0-Compiler unter WINDOWS-NT ...	32
2.9	UML (Unified Modeling Language).....	35
2.9.1	Allgemeines	35
2.9.2	Kardinalitäten nach UML	36
2.9.3	Frage nach den Klassen/Objekten.....	36
3.	Wichtige Begriffe und Sprachelemente	39
3.1	namespace und using	39
3.2	Default-Konstruktor.....	40
3.3	Copy-Konstruktor.....	40
3.4	explicit-Konstruktor.....	40
3.5	Zuweisungs-Operator	41
3.6	Abstrakte Klasse (= abstrakte Basisklasse)	42
3.7	Default-Argumente	43
3.8	Unspezifizierte Anzahl von Argumenten	43
3.9	l-value und r-value.....	47
3.10	Funktionszeiger	48
3.11	union	49
3.11.1	Allgemeines	49
3.11.2	Objekte unterschiedlichen Typs in eine Sequenz packen (list)	49
3.11.3	Mehrere Datenstrukturen für dieselben Daten (hardwareabhängig)...	51
3.11.4	Bitfelder zum Abtasten von Byte-Streams (hardwareabhängig)	54
3.11.5	Maske per Referenz anwenden	56
3.11.6	Test-Funktion zum Testen der Maschine auf little- bzw. big-endian .	56
3.12	extern "C" zum Abschalten der Namenszerstückelung.....	57
4.	Grundsätzliche Regeln beim Programmieren	58
4.1	Include-Wächter verwenden.....	58
4.2	Kommentar // dem Kommentar /* */ vorziehen.....	58

4.3	Optimiere die Laufzeit immer gleich mit	58
4.3.1	Objekte erst dort definieren, wo sie gebraucht werden	58
4.3.2	Zuweisung an ein Objekt mit der Konstruktion verbinden.....	59
4.3.3	return, break und continue mit Geschick einsetzen	60
4.4	Laufvariable im Schleifenkopf definieren.....	64
4.5	Der Stack ist immer dem Heap (new/delete) vorzuziehen	64
4.6	protected nur bei Basisklassen.....	65
4.7	Keine Fehler beim Mischen von C- und C++-Code machen	65
4.8	Ungarische Notation verwenden	66
4.9	Eingebaute Datentypen nie hinter typedef verstecken	67
4.10	Implizite Typumwandlung ggf. abschalten	68
4.11	inline nur bei sehr einfachen nicht-virtuellen Funktionen.....	70
4.11.1	Allgemeines	70
4.11.2	Widerspruch "virtual und inline": virtual dominiert inline.....	71
4.11.3	Basisklasse: Virtueller Destruktor als leere inline-Funktion	72
4.12	Falsche Benutzung einer Klasse ausschließen	72
4.12.1	Kopie eines Objektes verbieten	72
4.12.2	Konstruktion eines Objektes verbieten	72
4.13	Laufzeitschalter immer Compiler-Schaltern vorziehen.....	73
4.14	short statt bool als return-Wert bei Interface-Methoden	74
5.	Strings	76
5.1	ASCII-Tabelle	76
5.2	string der STL.....	78
5.2.1	Allgemeines	78
5.2.2	String formatieren mit Hilfe von sprintf()	80
5.2.3	Teil-Strings ersetzen mit string::replace() und string::find().....	81
5.2.4	Zeichen löschen mit string::erase() und einfügen mit string::insert()	81
5.2.5	Umwandlung in Zahlen mit strtol() und der Methode string::c_str():	82
5.2.6	Teil eines anderen Strings anhängen mit string::append()	82
5.2.7	Konfigurationsdateien parsen mit string::compare() und string::copy()	83
5.2.8	Worte sortieren mit set<string>	86
5.2.9	Strings zuschneiden mit string::replace() und string::resize()	87
5.3	string streams der STL.....	87
5.3.1	Allgemeines	87
5.3.2	Text mit istringstream nach enthaltenen Worten parsen	88
6.	Zeitermittlung	89
6.1	Weltweit eindeutiger Timestamp (GMT), Jahr-2038.....	89
6.2	Automatische Lokalisierung der Zeitdarstellung (strftime)	90
7.	Konstantes	93
7.1	const-Zeiger (C-Funktionen).....	93
7.2	const-Referenzen (C++-Funktionen)	94
7.2.1	Allgemeines	94
7.2.2	STL-Container als const-Referenzen verlangen const_iterator.....	94
7.3	Read-Only-Member-Funktionen	96
7.3.1	Allgemeines	96
7.3.2	mutable-Member als interne Merker (Cache-Index) verwenden.....	96
7.3.3	Zeiger bei Read-Only-Member-Funktion besonders beachten.....	97

7.4	const-return-Wert.....	98
7.5	const statt #define verwenden.....	99
7.5.1	Globale Konstanten.....	99
7.5.2	Lokale Konstanten einer Klasse	100
7.6	const-inline-Template statt MAKRO (#define) verwenden	101
8.	Globales (static-Member).....	103
8.1	static-Member.....	103
8.1.1	Allgemeines	103
8.1.2	Zugriff, ohne ein Objekt zu instanziiieren	103
8.2	Vorsicht bei static-Variablen in nicht-statischen Methoden	104
8.3	static-Variable in static-Methode statt globaler Variable	105
8.4	Lokale statische Arrays durch Main-Thread instanziiieren.....	108
8.5	Globale Funktionen: Nutze virtuelle Argument-Methoden	113
9.	Referenz statt Zeiger (Zeiger für C-Interface)	114
10.	Funktionen, Argumente und return-Werte	116
10.1	Argumente sollten immer Referenzen sein	116
10.1.1	const-Referenz statt Wert-Übergabe (Slicing-Problem).....	116
10.1.2	Referenz statt Zeiger	117
10.2	Argumente: Default-Parameter vs. überladene Funktion	118
10.3	Überladen innerhalb einer Klasse vs. über Klasse hinweg.....	119
10.3.1	Allgemeines	119
10.3.2	Nie Zeiger-Argument mit Wert-Argument überladen	120
10.4	return: Referenz auf *this vs. Wertrückgabe	120
10.4.1	Lokal erzeugtes Objekt zurückliefern: Rückgabe eines Wertes	120
10.4.2	Objekt der Methode zurückliefern: Referenz auf *this.....	121
10.4.3	Keine Zeiger/Referenzen auf private-Daten zurückliefern.....	121
10.5	return-Wert nie an referenzierendes Argument übergeben	123
11.	Smart-Pointer	124
11.1	Allgemeines	124
11.2	Smart-Pointer für die Speicher-Verwaltung	124
11.2.1	Eigenschaften des Smart-Pointers für die Speicherverwaltung	124
11.2.2	Was zu beachten ist.....	125
11.2.3	Code-Beispiel.....	126
11.2.4	Smart-Pointer immer per Referenz an eine Funktion übergeben.....	128
11.2.5	Empfehlungen	128
11.3	Smart-Pointer für andere Zwecke.....	130
12.	new/delete.....	131
12.1	Allgemeines zu new.....	131
12.2	Allgemeines zu delete.....	132
12.3	Beispiel für new/delete	133
12.4	Allgemeines zu new[]/delete[].....	133
12.4.1	new[]	133
12.4.2	delete[]	134
12.5	Mit Heap-Speicher arbeiten.....	134

12.6	Heap-Speicher als Shared Memory	135
12.7	new/delete statt malloc/free	136
12.8	Zusammenspiel von Allokierung und Freigabe	138
12.9	Eigener new-Handler statt Out-Of-Memory-Exception	140
12.10	Heap-Speicherung erzwingen/verbieten.....	141
	12.10.1 Heap-Speicherung erzwingen (protected-Destruktor)	141
	12.10.2 Heap-Speicherung verbieten (private operator new)	142
13.	Statische, Heap- und Stack-Objekte	144
13.1	Die 3 Speicher-Arten	144
13.2	Statische Objekte (MyClass::Method()).....	146
13.3	Heap-Objekte (pObj->Method())	146
13.4	Stack-Objekte (Obj.Method()).....	147
14.	Programmierung einer Klasse.....	147
14.1	Allgemeines	147
	14.1.1 Fragen, die beim Entwurf einer Klasse beantwortet werden sollten	147
	14.1.2 Die wesentlichen Methoden einer Klasse sind zu implementieren ..	148
	14.1.3 Durch den Compiler automatisch generierte Methoden beachten	149
	14.1.4 inline-Funktionen ggf. hinter die Deklaration schreiben	150
	14.1.5 Nie public-Daten verwenden	151
	14.1.6 Mehrdeutigkeiten (ambiguous) erkennen	152
14.2	Der Konstruktor	154
	14.2.1 Kein new im Konstruktor / Initialisierungslisten für Member.....	154
	14.2.2 Keine virtuellen Methoden im Konstruktor aufrufen	157
	14.2.3 Arrays mit memset() initialisieren	157
14.3	Der Destruktor	157
	14.3.1 Generalisierung ("is-a"): Basisklasse soll virtuellen Destruktor haben.....	157
14.4	Zuweisung per operator=()	158
	14.4.1 Keine Zuweisung an sich selbst.....	158
	14.4.2 Referenz auf *this zurückliefern	159
	14.4.3 Alle Member-Variablen belegen.....	159
14.5	Indizierter Zugriff per operator[]()	160
14.6	Virtuelle Clone()-Funktion: Heap-Kopie über pBase	161
14.7	Objektanzahl über private-Konstruktor kontrollieren	163
	14.7.1 Objekte über eine friend-Klasse (Objekt-Manager) erzeugen	163
	14.7.2 Objekte über eine statische Create()-Funktion erzeugen	164
	14.7.3 Genau 1 Objekt erzeugen (Code und/oder Tabelle)	165
14.8	Klassen neu verpacken mittels Wrapper-Klasse	166
15.	Richtiges Vererbungs-Konzept.....	167
15.1	Allgemeines	167
	15.1.1 Nie von (nicht-abstrakten) Klassen ohne virtuellen Destruktor erben	167
	15.1.2 Nie den Copy-Konstruktor-Aufruf der Basisklasse vergessen	168
	15.1.3 Statischer/dynamischer Typ und statische/dynamische Bindung	169
	15.1.4 Nie die Default-Parameter virtueller Funktionen überschreiben	170

15.1.5	public-, protected- und private-Vererbung gezielt verwenden	170
15.1.6	Rein virtuell / virtuell / nicht-virtuell	174
15.1.7	Rein virtuelle Methoden, wenn keine generalisierte Implem. möglich.....	174
15.2	Spezialisierung durch public-Vererbung ("is a").....	175
15.3	Code-Sharing durch private-Vererbung ("contains")	177
15.4	Composition statt multiple inheritance.....	180
15.5	Schnittstellen (AbstractMixinBaseClass) public dazuerben	181
15.6	Abstrakte Basisklasse vs. Template	183
15.7	Verknüpfung konkreter Klassen: abstrakte Basisklasse.....	184
15.8	Erben aus mehreren Basisklassen vermeiden	185
15.8.1	Expliziter Zugriff (oder using).....	185
15.8.2	Virtuelle Vererbung (Diamant-Struktur)	185
15.9	Zuweisungen nur zwischen gleichen Child-Typen zulassen.....	187
16.	Nutzer einer Klasse von Änderungen entkoppeln	189
16.1	Allgemeines	189
16.2	Header-Dateien: Forward-Deklaration statt #include	189
16.3	Delegation bzw. Aggregation	190
16.4	Objekt-Factory-Klasse und Protokoll-Klasse.....	192
17.	Code kapseln.....	194
17.1	Beliebig viele Kopien erlaubt: Funktions-Obj. (operator()).....	194
17.2	Nur 1 Kopie erlaubt: Statische Obj. (MyClass::Method())	194
18.	Operatoren.....	195
18.1	Definition von Operatoren.....	195
18.2	Binäre Operatoren effektiv implementieren	196
18.3	Unäre Operatoren bevorzugt verwenden.....	197
18.4	Kommutativität: Globale bin. Operatoren implementieren.....	197
18.5	Operator-Vorrang (Precedence)	199
18.6	Präfix- und Postfix-Operator	200
18.6.1	Allgemeines	200
18.6.2	Wartungsfreundlichkeit erhöhen durch ++(*this) im Postfix-Operator.....	202
18.6.3	Präfix(++Obj) ist Postfix(Obj++) vorzuziehen.....	202
18.7	Der Komma-Operator ,	203
19.	Datentypen und Casting.....	205
19.1	Datentypen.....	205
19.2	Polymorphismus: vfptr und vftable	206
19.3	RTTI (type_info) und typeid bei polymorphen Objekten	208
19.4	dynamic_cast: Sicherer cast von Zeigern oder Referenzen.....	210
19.4.1	Allgemeines	210
19.4.2	dynamic_cast zur Argumentprüfung bei Basisklassen-Zeiger/Referenz	212
19.5	const_cast.....	214
19.6	reinterpret_cast (!nicht portabel!) und Funktions-Vektoren.....	214
19.7	STL: Min- und Max-Werte zu einem Datentyp	215

20.	In Bibliotheken Exceptions werfen	216
20.1	Allgemeines	216
20.2	Exceptions per Referenz fangen	219
20.3	Kopien beim Weiterwerfen vermeiden	220
20.4	Beispiel für Exception-Handling	221
20.5	Exception-Spezifikation	221
20.5.1	Allgemeines	221
20.5.2	Spezifikationswidrige Exceptions abfangen: set_unexpected	222
20.5.3	Compilerunabhängiges Vorgehen.....	223
21.	Die STL (Standard Template Library).....	224
21.1	Allgemeines	224
21.2	Nutzung der STL von STLport.....	228
21.2.1	Allgemeines	228
21.2.2	STLport mit GNU unter Linux	228
21.2.3	STLport mit Visual C++ unter Windows	230
21.3	STL-Header-Dateien	231
21.3.1	Aufbau: Die Endung ".h" fehlt.....	231
21.3.2	Nutzung: "using namespace std"	231
21.4	Wichtige STL-Member-Variablen und Methoden	232
21.5	Generierung von Sequenzen über STL-Algorithmen.....	237
21.5.1	back_inserter()	237
21.5.2	Schnittmenge (set_intersection).....	237
21.5.3	Schnittmenge ausschließen (set_symmetric_difference).....	238
21.5.4	Sequenz ausschließen (set_difference)	238
21.5.5	Vereinigungsmenge bilden (set_union)	239
21.5.6	Liste an eine andere Liste anhängen (list::insert)	239
21.6	Wichtige Regeln	240
21.6.1	Einbinden der STL	240
21.6.2	Die benötigten Operatoren implementieren.....	241
21.6.3	Iterator: ++it statt it++ benutzen	242
21.6.4	Löschen nach find(): Immer über Iterator (it) statt über den Wert (*it)	243
21.6.5	map: Nie indizierten Zugriff [] nach find() durchführen	245
21.7	Beispiele für die Verwendung der Container	247
21.7.1	list: Auflistung von Objekten mit möglichen Mehrfachvorkommnissen	247
21.7.2	set: Aufsteigend sortierte Menge von Objekten (unique).....	249
21.7.3	map: Zuordnung von Objekten zu eindeutigen Handles.....	250
21.7.4	map: Mehrdimensionaler Schlüssel	252
21.7.5	vector: Schneller indizierter Zugriff	254
21.7.6	pair und make_pair(): Wertepaare abspeichern	255
21.8	hash_map	256
21.8.1	hash_map für Nutzer von Visual C++	256
21.8.2	Prinzip von hash_map	256
21.8.3	Nutzung von hash_map der STL	258
21.9	Lokalisierung mit der STL (streams und locales)	262

22.	Arten von Templates	267
22.1	Class-Template	267
22.2	Function-Template.....	268
22.2.1	Global Function Template	268
22.2.2	Member Function Template.....	268
22.3	Explizite Instanziierung von Templates	269
23.	Proxy-Klassen.....	270
23.1	Allgemeines	270
23.2	Schreiben/Lesen beim indizierten Zugriff unterscheiden.....	271
24.	Datenbank-Zugriff.....	274
24.1	Zugriff auf objektorientierte Datenbanken	274
24.2	Zugriff auf relationale Datenbanken	275
24.3	Zugriff auf hierarchische Datenbanken	283
25.	Aktion nach Kollision über Objekttyp steuern.....	284
26.	80/20-Regel und Performance-Optimierung.....	289
26.1	Allgemeines	289
26.2	Zeit-Optimierungen	290
26.2.1	return so früh wie möglich	290
26.2.2	Präfix-Operator statt Postfix-Operator.....	290
26.2.3	Unäre Operatoren den binären Operatoren vorziehen	290
26.2.4	Keine Konstruktion/Destruktion in Schleifen.....	291
26.2.5	hash_map statt map, falls keine Sortierung benötigt wird.....	291
26.2.6	Lokaler Cache um Berechnungen/Datenermittlungen zu sparen.....	291
26.2.7	Löschen nach find() immer direkt über den Iterator	293
26.2.8	map: nie indizierten Zugriff [] nach find() durchführen	294
26.2.9	Unsichtbare temporäre Objekte vermeiden	295
26.2.10	Berechnungen erst dann, wenn das Ergebnis gebraucht wird.....	298
26.2.11	Datenermittlung erst dann, wenn die Daten gebraucht werden	298
26.2.12	Große Anzahl kleiner Objekte blockweise lesen (Prefetching)	298
26.2.13	Kein unnötiges Speichern in die Datenbank	299
26.2.14	SQL-SELECT-Statements effektiv aufbauen: DB-Server filtern lassen.....	299
26.3	Speicher-Optimierungen.....	300
26.3.1	Sharing von Code und/oder Tabellen mittels statischem Objekt.....	300
26.3.2	Sharing von Code und/oder Tabellen mittels Heap-Objekt	300
26.3.3	Nach Kopie die Daten bis zum Schreibzugriff sharen (Copy-On-Write).....	302
26.3.4	Object-Pooling	305
	Index	309