

ProgCPP_ZF FS18

N. Kaelin

24. Juli 2018

Inhaltsverzeichnis

I	Einführung	10
1	Charakteristiken von C++	10
2	Entstehung von C++	10
3	Welches C++?	10
4	C++-Unterstützung von Texas Instruments (TI)	10
5	Hello World!	11
6	C++-Compiler (noch nicht Eclipse)	11
7	Lexikalische Elemente von C++	11
8	Styleguide: Bezeichner (-Namen)	11
9	Typkonzept	12
9.1	Datentypen	12
9.2	#define	12
10	Ausdrücke und Operatoren	12
11	Anweisungen	12
12	Streams	12
12.1	Streamkonzept	12
12.2	Einsatz von Streams	12
12.3	Ausgabe: Klasse ostream	13
12.4	Eingabe: Klasse istream	13
12.5	Formatierte Ein- und Ausgabe	13
12.5.1	Format-Flags um Überblick (unvollständig)	14
II	Funktionen	15
13	Grundlegendes	15
13.1	Synonyme für Funktionen	15
13.2	Aufgabe einer Funktion	15
13.3	Funktionen (Vergleich zu C)	15
13.4	Definition von Funktionen	15
13.5	Deklaration von Funktionen (Funktionsprototypen)	16
13.6	Kosten einer Funktion	16

14 C-Makro	16
14.1 C-Makro mit #define	16
14.2 Beispiel mit C-Makro: Maximum zweier int-Werte	16
14.2.1 Was passiert wirklich?	17
15 inline-Funktionen	17
15.1 Grundlegendes	17
15.2 Beispiel mit inline-Code: Maximum zweier int-Werte	17
16 Grundsätze für Optimierungen	17
17 default-Argumente	17
17.1 Vorbelegte Parameter (default-Argumente)	17
17.1.1 Beispiel: default-Argumente	18
17.2 Nutzen von default-Argumenten	18
18 Overloading	18
18.1 Überladen von Funktionen (overloading)	18
18.2 Overloading in C++	18
18.2.1 Deklaration von überladenen Funktionen: Regeln	19
18.2.2 Funktionen sollen nur dann überladen werden, wenn ...	19
19 default-Parameter vs. Overloading	19
 III Pointer und Referenzen	 20
20 Höhere und strukturierte Datentypen	20
20.1 Höhere Datentypen	20
20.2 Strukturierte Datentypen	20
21 Pointer	20
21.1 Adresse	20
21.2 Standarddarstellung von Pointern	20
21.3 Pointer und Datentyp	20
21.4 Definition einer Pointervariablen	20
21.4.1 Initialisierung mit Null-Pointer	21
21.4.2 Der Adressoperator & (Referenzierung)	21
21.4.3 Kopieren von Adressen	21
21.4.4 Der Inhaltsoperator * (Dereferenzierung)	22
21.4.5 const bei Pointern: Vorsicht	22
21.4.6 void-Pointer	22
21.4.7 Pointer auf Funktionen	23
21.4.8 Interruptvektortabelle: Tabelle von Funktionspointern	23
21.4.9 Umsetzung von Funktionspointern in C/C++	23
21.4.10 Beispiel für Funktionspointer	24
22 Referenzen	24
22.1 Syntax	24
22.2 Einsatz	25
23 Pointer und Referenzen auf lokale Variablen	25
24 Zeiger und Referenzen als Parameter und Rückgabewerte	25
24.1 Call by Value vs. Call by Reference	25
24.1.1 3 Beispiele	25
24.2 Call by reference: wann einsetzen?	25
24.3 Merke	26

IV Arrays, Dynamische Speicherverwaltung	27
25 Arrays	27
25.1 Arrays: Vektoren	27
25.2 Der Array (Feld, Vektor)	27
25.2.1 Zugriff auf ein Arrayelement	27
25.3 Arrays und Pointer	28
25.3.1 Pro Memoria: Eindimensionales Array (Vektor)	28
25.4 Äquivalenz von Array- und Pointernotation	28
25.5 Vergleichen von Arrays	28
25.6 Der Arrayname ist ein nicht modifizierbarer L-Wert	28
25.7 Automatische Initialisierung von Arrays	28
25.8 Explizite Initialisierung von Arrays	28
25.8.1 Beispiel	29
25.8.2 Goodies für die explizite Initialisierung	29
25.9 Mehrdimensionale Arrays	29
25.9.1 Initialisierung eines mehrdimensionalen Arrays	30
26 Übergabe von Arrays und Zeichenketten	30
26.1 Beispiel: Array (Vektor) als Parameter	30
26.2 Übergabe einer Matrix mittels offenem Array	30
26.3 Zeichenketten (Strings)	31
27 Dynamische Speicherverwaltung	31
27.1 Pro Memoria: Variablen	31
27.2 Dynamische Speicherverwaltung	31
27.3 Syntax	31
27.4 Vorsichtsmassnahmen	31
27.5 Memory Leak, Garbage Collection	32
27.6 Dynamische Allokierung von Arrays	32
27.7 Dynamische Allokierung von Matrizen	32
27.7.1 Dynamische Matrix mit 4 Zeilen und 3 Spalten	33
27.7.2 Zugriff auf dynamisch erzeugte Matrix	33
27.7.3 Dynamische Matrix freigeben	34
27.8 Effizienz der Matriximplementationen	34
V Scope, Deklarationen, Type Casts	35
28 Strukturen in C++	35
29 Gültigkeitsbereiche, Namensräume und Sichtbarkeit	35
29.1 Gültigkeitsbereiche von Namen (Scope)	35
29.2 Gültigkeitsbereiche in C++	35
29.3 Gültigkeit (Scope) von Variablen	35
29.4 Lebensdauer von Variablen	36
29.5 Sichtbarkeit von Variablen	36
29.6 Schlussfolgerung (naheliegend aber falsch)	36
29.7 Lebensdauer (grau) und Sichtbarkeit (weiss)	36
29.8 Codierstil	36
30 Namensräume (Namespaces)	37
30.1 Explizite Namensräume in C++	37
30.2 C++-Mechanismen für Namespaces	37
30.3 Deklaration von Namespaces	37
30.3.1 Beispiel	38
30.4 using-Deklaration	38

30.5	using-Direktive	38
30.6	using namespace kann zu Konflikten führen	38
30.7	Namenlose Namespaces	39
30.8	Zugriff auf globale Variable mit Scope-Operator	39
31	Speicherklassen	39
31.1	Speicherklassen in C++	39
31.1.1	Speicherklasse static: Variablen	40
31.1.2	Speicherklasse static: Funktionen	40
31.1.3	Speicherklasse extern: Externe Variablen	40
31.2	Typqualifikationen (Kap. 9.2.2)	40
31.3	Funktionsattribute	40
32	Typdefinitionen	41
32.1	typedef zur Vereinbarung eigener Datentypen	41
32.1.1	Beispiel	41
32.2	Gewährleistung von Portabilität	41
32.3	Wie setzt der Compiler ein typedef um?	42
33	Initialisierung	42
34	Type-Cast (Typumwandlungen)	42
34.1	Typumwandlungen im Allgemeinen	42
34.2	Implizite Typumwandlung	42
34.3	Explizite Typumwandlung	42
34.3.1	Explizite Typumwandlung #1, 2: C-Stil und Funktionsstil	42
34.3.2	Typumwandlung mit C-Stil und Funktionsstil	43
34.3.3	Explizite Typumwandlung #3: const_cast	43
34.3.4	Explizite Typumwandlung #4: static_cast	43
34.3.5	Explizite Typumwandlung #5: dynamic_cast	43
34.3.6	Explizite Typumwandlung #6: reinterpret_cast	43
VI	Module und Datenkapseln	45
35	Modul (Unit)	45
35.1	Nomenklatur: Modul vs. Unit	45
35.2	Ziele der Modularisierung	45
35.3	Eigenschaften einer Unit (eines Moduls)	45
35.4	Bestandteile eine C++-Programms	45
35.5	Unitkonzept	45
35.6	Geheimnisprinzip (Information Hiding)	45
35.6.1	Information Hiding	45
35.6.2	Konzept der Datenkapsel	46
35.6.3	Beispiel für Datenzugriff bei Datenkapsel	46
35.6.4	Beispiel für Unit Rechteck (ohne Datenkapsel)	47
35.6.5	Beispiel für Unit Rechteck: Verbesserung #1	47
35.6.6	Beispiel für Unit Rechteck: Verbesserung #2	48
35.7	Unit nutzen	48
35.8	Unit-Schnittstelle definieren (in Headerdatei)	48
35.9	Deklarationsreihenfolge in der Headerdatei (*.h)	48
35.10	Reihenfolge in der Implementierungsdatei (*.cpp)	49
35.11	#include-Konzept	49
35.12	Unit compilieren	49
35.13	Units linken	49
35.14	Buildprozess	49
35.15	Make-Tool	50

35.15.1 Abhängigkeiten zwischen Dateien	50
35.15.2 make-File	50
35.15.3 Beispiel: makefile	50
VII Eclipse IDE	51
36 Eclipse	51
36.1 Workspace	51
36.2 Ressourcen (Resources)	51
36.3 Project	51
36.4 Debugger	51
36.4.1 Testen und Debugging	51
36.4.2 Funktionen eines Debuggers	51
36.4.3 Assertions (Zusicherungen)	51
36.4.4 Zu beachten bei Assertions	52
VIII Klassen	53
37 Beispielprojekt Stack	53
37.1 Stack	53
37.1.1 Stack - Operationen	53
37.1.2 Demo: Codebeispiel für Stack (Stack_Datenkapsel)	53
37.2 Queue	56
37.2.1 Queue - Operationen	57
38 Objektorientierte Programmierung	57
38.1 Modell und Realität	57
38.2 Zu (subjektiven) Sichtweise des Betrachters	57
38.3 Prozedurale vs. Objektorientierte Sicht	57
39 Unified Modeling Language (UML)	57
39.1 Entstehung der UML	57
39.2 Was ist UML?	57
39.3 Klassenbegriff	58
39.4 UML-Notation der Klasse	58
39.5 Klasse definieren und Objekte anlegen: Syntax	58
40 Zugriffsschutz bei Klassen	59
40.1 Zugriffsschutz mit public, protected und private	59
40.1.1 Üblicher Aufbau einer Klassenschnittstelle	59
40.2 Operationen einer Klasse	59
40.3 Information Hiding	60
41 Beispiel einer Klasse: Rechteck (Rectangle)	60
41.1 Klassendeklaration	60
41.2 Klassendefinition	61
41.3 Klassendefinition direkt	61
41.4 Reprise: Projekt Stack	61
41.5 Klassenschnittstelle	62
42 Elementfunktionen	62
42.1 Klassifizierung von Elementfunktionen	62
42.2 inline-Elementfunktionen	62
42.3 Projekt Stack	62
42.4 const - Elementfunktionen	62

42.4.1	const - Elementfunktionen: Schlussfolgerung	63
42.5	Projekt Stack	63
42.6	mutable-Attribut	63
42.7	Projekt Stack	64
43	Konstruktoren/Destruktoren	64
43.1	this-Pointer	64
43.2	friend-Elemente	64
43.3	static-Klassenelemente (Kap.11.5)	65
43.3.1	Syntax	65
43.4	Konstruktor (Constructor, Ctor)	65
43.4.1	Aufruf	65
43.4.2	Beispiel: Klasse TString (nach Lippman)	66
43.4.3	Implementation von TString::TString()	67
43.4.4	Überladen von Konstruktoren	67
43.4.5	Erweiterung der Klasse TString	67
43.4.6	Konstruktoren und Function Casts	68
43.4.7	Erweiterung der Klasse TString 2	69
43.4.8	Copy-Konstruktor	69
43.4.9	Copy-Konstruktor wird automatisch aufgerufen, wenn...	69
43.4.10	Erweiterung der Klasse TString 3	70
43.4.11	Shallow Copy vs. Deep Copy	70
43.4.12	Copy-Konstruktor der Klasse TString	71
43.5	Destruktor (Destructor, Dtor)	71
43.5.1	Eigenschaften des Destruktors	71
43.5.2	Erweiterung der Klasse TString 4	71
43.5.3	Implementation des Destruktors	72
43.5.4	Schnittstelle der Klasse TString	72
43.5.5	Demo: Klasse Stack	72
44	Handhabung von Klassen und Objekten	72
44.1	Repetition	72
44.2	Welche Ctors werden bei folgenden Anweisungen aufgerufen?	73
44.3	Automatisch generierte Elementfunktionen (11.7.4)	73
44.4	Kanonische Form von Klassen (11.7.5)	73
44.5	Benutzerdefinierte Typumwandlungen: Problemstellung & Lösung	73
44.6	Typumwandlung mit Konstruktor	74
44.7	Weiterführende Themen	74
44.7.1	Zeiger auf Klassenelemente	74
44.8	Unions (Varianten)	74
44.8.1	Eigenschaften einer Union	74
44.8.2	Definition von Uniontypen und Unionvariablen	75
44.8.3	Beispiel: Definition einer Union	75
44.9	Bitfelder	75
44.9.1	Eigenschaften von Bitfeldern	75
44.9.2	Definition von von Bitfeldern	76
44.9.3	Bitfelder: Folgerungen	76
44.10	Demo: Klasse Date	76
IX	Vererbung	77
45	Motivation	77
46	Artikel als Gemeinsamkeit von Buch und CD	77
47	Grundkonzept	77

48 Einsatz der Vererbung	77
49 UML-Notation	77
49.1 ist ein Beziehung	77
50 Beispiel: Vererbungshierarchie Lebewesen	77
50.1 C++-Syntax	77
50.2 Zugriff auf Elemente der Basisklasse	78
51 Spezifikation von Basisklassen	78
52 Beispiel: ComicCharacter (Comics01)	78
53 Einsatz von protected bei Klassenelementen	78
54 Objektgrösse bei der Vererbung	78
55 Schlechter (falscher) Einsatz von Vererbung	78
56 Substitutionsprinzip	78
X Polymorphismus	79
57 Static vs. Dynamic Binding	79
57.1 Dynamic Binding	79
57.1.1 Beispiel: Zeichnen von geometrischen Figuren	79
57.2 Statischer vs. dynamischer Datentyp	79
57.3 Aufruf von virtuellen Elementfunktionen	80
57.3.1 Statischer Aufruf von virtuellen Elementfunktionen	80
57.3.2 Dynamischer Aufruf von virtuellen Elementfunktionen	80
57.4 Polymorphe Klassen (Virtuelle Klassen)	80
57.4.1 Repräsentation virtueller Objekte im Speicher	81
58 Abstrakte Klassen	81
58.1 Anwendungen von abstrakten Klassen (Beispiele)	81
59 Mehrfachvererbung (Multiple Inheritance, MI)	81
59.1 Virtuelle Basisklassen	82
60 Laufzeit-Typinformationen (Run-Time Type Information, RTTI)	82
60.1 Operator dynamic_cast	82
60.2 Operator typeid	83
XI Überladen von Operatoren (Operator overloading)	84
61 Operator overloading in C++	84
61.1 Operatorfunktion	84
61.2 Randbedingungen zu Operator overloading	84
61.3 Umsetzungsvarianten für Operator overloading	85
61.3.1 Beispiel	86
61.3.2 Zuweisungsoperator =	87
61.3.3 Indexoperator []	87
61.3.4 Beispiel Klasse TString	87
61.3.5 Eigenen Zuweisungsoperator definieren	88
61.3.6 Zur Erinnerung: Kanonische Form einer Klasse	88
61.4 Streamkonzept	88

61.4.1	Ausgabe: Klasse ostream	88
61.4.2	Operator « überschreiben	88
61.4.3	Eingabe: Klasse istream	88
61.4.4	Operator » überschreiben	89

XII Templates 90

62 Generische Programmierung mit Templates (Schablonen) 90

62.1	Motivation für Templates	90
62.2	Lösung mit bekannten Techniken	90
62.3	Generische Programmierung mit Templates	90

63 Funktions-Templates 90

63.1	Syntax für Funktions-Templates	90
63.2	Beispiel (aus Prata): Zwei Werte vertauschen	91
63.3	inline bei Templates	91
63.4	Beispiel: kleinstes Element finden	91
63.5	Ausprägung von Funktions-Templates	92
63.6	Explizite Qualifizierung von Funktions-Templates	92
63.7	Überladen von Funktions-Templates	92

64 Klassen-Templates 92

64.1	Definition: Klassen-Template	92
64.2	Syntax für Klassen-Templates	92
64.2.1	Beispiel zu Klassen-Template: Deklaration	93
64.2.2	Beispiel zu Klassen-Template: Definition	93
64.2.3	Beispiel zu Klassen-Template: Nutzung (Ausprägung)	93
64.3	Bemerkungen	93
64.4	Explizite Ausprägung von Klassen-Templates	94
64.5	Klassen-Templates und getrennte Übersetzung: export	94
64.6	Klassen-Templates und getrennte Übersetzung	94
64.6.1	File-Organisation #1 bei Klassen-Templates	94
64.6.2	Ausschliessen eines cpp-Files von Compilierung in Eclipse	94
64.6.3	File-Organisation #2 bei Klassen-Templates	94
64.7	Fazit	95

XIII Exceptions („Ausnahmen“) 96

65 Exception vs. Error 96

66 Mögliche Reaktionen auf Ausnahmen 96

67 Exceptioncodes als Rückgabewert 96

68 Exceptioncodes als Referenzparameter 96

69 Globaler Exceptioncode 97

70 Wo sollen Exceptions behandelt werden? 97

70.1	Beispiel: Übertragungssystem	97
------	--	----

71 Ziel für Exception Handling 97

72 Exception Handling in C++	97
72.1 Exception Handling in C++: Syntax	98
72.2 Auslösen (Werfen) von Ausnahmen	98
72.2.1 Beispiel für Exception Handling: unübliche Variante	99
72.3 Vordefinierte Ausnahmeklassen	99
72.4 Exception-Hierarchie in C++	99
72.5 Laufzeit- vs. Logische „Fehler“	99
72.6 Exceptions und ihre Header-Dateien	100
72.7 Exception Handler	100
72.7.1 Handler mit beliebigen Exceptions ohne Hierarchie	100
72.7.2 Handler mit System-Exceptions	100
72.7.3 Handler mit schlechter Reihenfolge	100
72.8 Exception Handler 2	100
72.9 Exception Propagation	100
72.10 Exception Specification	100
72.10.1 Exception Specification: Beispiele	101
72.10.2 Exception Handling in der Praxis	101
72.10.3 Throw exceptions by value, catch them by const reference	101
72.10.4 Handling von System Exceptions	101
72.10.5 Handling-Strategie von System Exceptions bei verschiedenen Programmiersprachen	101
72.10.6 Betreibt meine Umgebung Exception Mapping?	102
 XIV Preprocessor	 103
73 Eigenschaften des Preprocessors	103
74 Preprocessor-Direktiven und Bedingungsanweisungen	103
74.1 #define	103
74.2 #undef	103
74.3 #include	104
74.4 #line	104
74.5 #error	104
74.6 #pragma	104
74.7 Bedingungsanweisungen	104
74.7.1 Beispiele für Bedingungsanweisungen	105
74.8 Weitere Features des Preprocessors	105
74.9 Kritische Würdigung des Preprocessors	105

Teil I

Einführung

1 Charakteristiken von C++

- C++ erlaubt sowohl prozedurale, objektorientierte als auch generische Programmierung

Achtung: Nicht jedes C++-Programm ist objektorientiert!

- C++ ist sehr mächtig
- C++ ist eine Obermenge von C
- Syntaktisch ist C++ sehr ähnlich oder identisch zu C
- C++ ist sicherer als C

2 Entstehung von C++

- C Ritchie 1971, typisiert
- ANSI C seit 1983
- C++ Stroustrup 1986, Klassen und OO
- Final Standard ISO/ANSI ab 1990

ISO/IEC 14882:2003 Programming Languages - C++, aka C++03

ISO/IEC 14882:2011 Programming Languages - C++, aka C++11

ISO/IEC 14882:2014 Programming Languages - C++, aka C++14

3 Welches C++?

- **Im Modul ProgCPP setzen wir den Standard 14882:2003, d.h. C++03 ein**
- Wieso nicht C++11 oder C++14?
 - Der neue Standard hat einige interessante Erneuerungen zu bieten, die C++ noch näher an C# kommen lassen
 - Durch diese neuen Features wird die Sprache leider nicht einfacher sondern umfangreicher und komplizierter
 - Der Nutzen von einzelnen Neuerungen ist m.E. fraglich
 - Bei Embedded Systems wird heute noch mehrheitlich C verwendet. Diejenigen, die C++ einsetzen, nehmen C++03
 - Ich bin gespannt, ob und wann C++11 im Embedded-Bereich den Durchbruch schafft

4 C++-Unterstützung von Texas Instruments (TI)

- TI als Repräsentant für einen Anbieter von Embedded-Entwicklungsumgebungen
- The TI compilers for all devices support
 - C++98 (ISO/IEC 14882:1998)
 - C++03 (this is a bug fix update to C++98)
- The TI compiler does not support
 - C++ TR1
 - C++11 (ISO/IEC 14882:2011)
- Noch neuere Versionen erst recht nicht

5 Hello World!

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

int main(void)
{
    cout << "Hello World!" << endl;
    return 0;
}
```

6 C++-Compiler (noch nicht Eclipse)

Statt `gcc` (für C-Programme) muss `g++` (für C++-Programme) oder `clang++` verwendet werden

`g++ -o hello hello.cpp`

`clang++ -o hello hello.cpp`

7 Lexikalische Elemente von C++

- C++ ist hier völlig identisch zu C
 - Bezeichner
 - Schlüsselwörter (werden ergänzt durch zusätzliche)
 - Literale
 - Operatoren (werden ergänzt durch zusätzliche)
 - Kommentare
- Codierstil beachten (Element-Richtlinien)

8 Styleguide: Bezeichner (~Namen)

- Variablen, Konstanten und Objekte
 - mit Kleinbuchstaben beginnen
 - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)
 - keine Underscores

Beispiele: `counter`, `maxSpeed`

- Funktionen
 - mit Kleinbuchstaben beginnen
 - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)
 - Namen beschreiben Tätigkeiten
 - keine Underscores

Beispiele: `getCount()`, `init()`, `setMaxSpeed()`

- Klassen, Strukturen, Enums
 - mit Grossbuchstaben beginnen
 - erster Buchstaben von zusammengesetzten Wörtern ist gross (mixed case)
 - keine Underscores

Beispiele: `MotorController`, `Queue`, `Color`

9 Typkonzept

9.1 Datentypen

- In C++ gibt es die gleichen Basisdatentypen (plain old data types, POD types) wie in C
- Zusätzlich: es gibt einen Typ für boole'sche Werte: **bool**
- Der Typ **bool** hat die beiden Werte **true** und **false**

9.2 #define

- In C oft noch geduldet, in C++ verpönt
- *#define* bewirkt eine reine Textersetzung durch den Präprozessor, umgeht dadurch Syntax- und Typenprüfung
- Für die Definition von symbolischen Konstanten soll statt *#define* das Schlüsselwort **const** oder **enum** verwendet werden

schlecht:
`#define PI 3.14159`
`#define VERSUCHE_MAX 4`

gut:
`const double pi = 3.14159;`
`const int versucheMax = 4;`
`enum{versucheMax = 4}; //noch besser als const int`

10 Ausdrücke und Operatoren

- C++ ist hier völlig identisch zu C
- Es gibt noch ein paar zusätzliche Operatoren in C++ (z.B. für type casting)

11 Anweisungen

- Die Syntax der Blockanweisung ist identisch zu C
- Die Syntax der Steuerstrukturen ist identisch zu C
 - Sequenz
 - Iteration (for, while, do ... while)
 - Selektion (if, if ... else, switch)
- Sprunganweisungen sind auch identisch zu C
 - break
 - continue
 - return
 - goto → nicht verwenden!!!

12 Streams

12.1 Streamkonzept

- Ein Stream repräsentiert einen sequentiellen Datenstrom
- Die Operatoren auf dem Stream sind « und »
 Für vordefinierte Datentypen sind diese Operatoren schon definiert, für eigene selbstdefinierte Klassen können diese Operatoren überladen werden (*siehe 61.4*)
- C++ stellt 4 Standardströme zur Verfügung
 - cin: Standard-Eingabestrom, normalerweise die Tastatur
 - cout: Standard-Ausgabestrom, normalerweise der Bildschirm
 - cerr: Standard-Fehlerausgabestrom, normalerweise der Bildschirm
 - clog: mit *cerr* gekoppelt
- Alle diese Ströme können auch mit einer Datei verbunden werden

12.2 Einsatz von Streams

- In C werden printf() und scanf() mit stdin, stdout, stderr verwendet.
- printf() und scanf() könnten in C++ immer noch verwendet werden. Dies soll jedoch im Normalfall vermieden werden.
- C++ bietet analog die Streams cin, cout, cerr an.
- Der Zugriff auf cin und cout ist einfacher und komfortabler als die Verwendung von printf() und scanf().

- cin und cout müssen immer ganz links in einer Befehlszeile stehen. Die Daten kommen von cin (Tastatur, Operator ») und gehen zu cout (Konsole, Operator «).

12.3 Ausgabe: Klasse ostream

- Methoden für die Ausgabe von vordefinierten Datentypen, z.B.:
ostream& operator <<(int n);
ostream& operator <<(double d);
ostream& operator <<(char c);
- Weitere Klassen können diesen Operator überschreiben, z.B.:
ostream& operator <<(std::string str);
- Nutzung mit cout (vordefiniertes Objekt der Klasse ostream):
int i = 45;
cout <<"Hallo " <<i <<endl;

12.4 Eingabe: Klasse istream

- Methoden für die Eingabe von vordefinierten Datentypen, z.B.:
istream& operator >>(int& n);
istream& operator >>(double& d);
istream& operator >>(char& c);
- Weitere Klassen können diesen Operator überschreiben z.B.:
istream& operator >>(std::string& str);
- Nutzung mit cin (vordefiniertes Objekt der Klasse istream):
double d;
string str;
cin >>d >>str;

12.5 Formatierte Ein- und Ausgabe

ios, eine Basisklasse von iostream, stellt verschiedene Möglichkeiten (Format Flags) vor, um die Ein- und Ausgabe zu beeinflussen.

Beispiel:

```
cout << showbase << hex << 27; // Ausgabe: 0x1b
```

12.5.1 Format-Flags um Überblick (unvollständig)

Flag	Wirkung
boolalpha	bool-Werte werden textuell ausgegeben
dec	Ausgabe erfolgt dezimal
fixed	Gleitkommazahlen im Fixpunktformat
hex	Ausgabe erfolgt hexadezimal
internal	Ausgabe innerhalb Feld
left	linksbündig
oct	Ausgabe erfolgt oktal
right	rechtsbündig
scientific	Gleitkommazahl wissenschaftlich (Mantisse und Exponent)
showbase	Zahlenbasis wird angezeigt
showpoint	Dezimalpunkt wird immer ausgegeben
showpos	Vorzeichen bei positiven Zahlen anzeigen
skipws	Führende Whitespaces nicht anzeigen
unitbuf	Leert Buffer des Outputstreams nach Schreiben
uppercase	Alle Kleinbuchstaben in Grossbuchstaben wandeln

Teil II

Funktionen

13 Grundlegendes

13.1 Synonyme für Funktionen

- Unterprogramm
- Subroutine
- Prozedur (Funktion ohne Rückgabewert)
- Methode (in der Objektorientierten Programmierung)

13.2 Aufgabe einer Funktion

- Gleichartige, funktional zusammengehörende Programmteile unter einem eigenen Namen zusammenfassen. Der Programmteil kann mit diesem Namen aufgerufen werden.
- Einige Funktionen (im speziellen mathematische) sollen parametrisiert werden können, z.B. die Cosinusfunktion macht nur Sinn, wenn sie mit unterschiedlichen Argumenten aufgerufen werden kann.
- Divide et impera (divide and conquer, teile und herrsche):
Ein grosses Problem ist einfacher zu lösen, wenn es in mehrere einfachere Teilprobleme aufgeteilt wird.

13.3 Funktionen (Vergleich zu C)

- Alles was in C möglich ist, gibt es auch in C++
- Einige Punkte sind in C++ zusätzlich eingeführt worden:
 - Operatorfunktion (Spezialität von C++, folgt später)
 - inline-Funktion
 - Vorbelegung von Parametern (default-Argumente)
 - Überladen von Funktionen (overloading)

13.4 Definition von Funktionen

- Funktionskopf
 - legt die Aufrufschnittstelle (Signatur) der Funktion fest
 - besteht aus:
 - * Rückgabotyp
 - * Funktionsname (fast beliebig wählbar)
 - * Parameterliste
- Funktionsrumpf
 - Lokale Vereinbarungen und Anweisungen innerhalb eines Blocks

13.5 Deklaration von Funktionen (Funktionsprototypen)

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

void init(int* alpha); // Funktionsprototyp (Deklaration)

int main(void)
{
    int a;
    init(&a);
    cout << "Der Wert von a ist " << a << endl;
    return 0;
}

void init(int* alpha)
{
    *alpha = 10;
}
```

13.6 Kosten einer Funktion

- Der Code einer Funktion ist nur einmal im Speicher vorhanden.
 - Vorteil: spart Speicher
- Der Aufruf einer Funktion bewirkt eine zeitliche Einbusse im Vergleich zu einer direkten Befehlsausführung.
 - Nachteil: Zeitverlust, Overhead

14 C-Makro

14.1 C-Makro mit #define

- C-Makros bewirken eine reine Textersetzung ohne jegliche Typenprüfung
- Bei Nebeneffekten (welche zwar vermieden werden sollten) verhalten sich Makros oft nicht wie beabsichtigt

Achtung: C-Makros lösen zwar das Problem mit dem Overhead, sind aber sehr unsicher. Bitte nicht einsetzen!

14.2 Beispiel mit C-Makro: Maximum zweier int-Werte

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

int z1 = 4;
int z2 = 6;
int m = MAX(z1, z2);
wird expandiert zu: m = ((z1)>(z2) ? (z1) : (z2)); // m=6, z1=4, z2=6
m = MAX(++z1, ++z2);
erwartet wird: m=7, z1=5, z2=7
```


14.2.1 Was passiert wirklich?

```
m = MAX(++z1, ++z2);
wird expandiert zu:
m = ((++z1) > (++z2) ? (++z1) : (++z2)); m = ((5) > (7) ? (++z1) : (8));
// z2 wird zweimal inkrementiert!
// m=8, z1=5, z2=8
erwartet wird:
m=7, z1=5, z2=7
```

15 inline-Funktionen

15.1 Grundlegendes

- Lösen das Overhead-Problem
 - Code wird direkt eingefügt, kein Funktionsaufruf
- Typenprüfung findet statt
- Einsetzen wenn der Codeumfang der Funktion sehr klein ist und die Funktion häufig aufgerufen wird (z.B. in Schleifen)
- Achtung: Rekursive Funktionen und Funktionen, auf die mit einem Funktionspointer gezeigt wird, werden nicht inlined.

15.2 Beispiel mit inline-Code: Maximum zweier int-Werte

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}

int main()
{
    int z1 = 4;
    int z2 = 6;
    int m = max(z1, z2);
    // m=6, z1=4, z2=6
    m = max(++z1, ++z2);
    // m=7, z1=5, z2=7
}
```

16 Grundsätze für Optimierungen

1. Optimize: don't do it
2. If you have to do it: do it later

17 default-Argumente

17.1 Vorbelegte Parameter (default-Argumente)

```
void prtDate(int day=1, int month=3, int year=2009);
```

- Parametern können im Funktionsprototypen (bitte nur dort!) Defaultwerte zugewiesen werden.
- Beim Funktionsaufruf können (aber müssen nicht) die Parameter mit default-Werten weggelassen werden

Achtung: Hinter (rechts von) einem default-Argument darf kein nicht vorbelegter Parameter mehr folgen, d.h. wenn bei einem Parameter ein default definiert wird, dann müssen bei allen weiteren Parametern dieser Funktion ebenfalls defaults definiert werden.

- Grund: Die Parameterübergabe erfolgt in C++ von links nach rechts.

17.1.1 Beispiel: default-Argumente

```
void prtDate(int day=1, int month=3, int year=2009);
```

nicht erlaubt sind z.B. diese Deklarationen:

```
void prtDate2(int day=7, int month, int year=2009);
void prtDate3(int day, int month=3, int year);
```

erlaubt sind z.B. die folgenden Aufrufe:

```
prtDate();           // 1-3-2009
prtDate(23);         // 23-3-2009
prtDate(15,6);        // 15-6-2009
prtDate(24,7,2012);  // 24-7-2012
```

17.2 Nutzen von default-Argumenten

- Wenn in einer bereits existierenden Funktion neue Argumente aufgenommen werden müssen, dann:
 - Neue Argumente hinten als default-Argumente anfügen.
 - Die bereits bestehenden alten Aufrufe (mit weniger Argumenten) können unverändert beibehalten werden.
 - Die Implementation der Funktion muss angepasst werden.
- Sehr nützlich z.B. bei Konstruktoren in der objektorientierten Programmierung.

18 Overloading

18.1 Überladen von Funktionen (overloading)

- Zweck:
 - Eine Funktion sollte allenfalls mit unterschiedlichen Parametern aufgerufen werden können:

```
void print(char ch);
void print(int i);
void print(double d);
```
- Alternative (in C) wäre:


```
void printChar(char ch);
void printInt(int i);
void printDouble(double d);
```

 → Ist umständlicher und unverständlicher

18.2 Overloading in C++

- Die Identifikation einer Funktion erfolgt über die Signatur, nicht nur über den Namen
 - Die Signatur besteht aus:
Name der Funktion **plus** die Parameterliste (Reihenfolge, Anzahl, Typ)
(Der Returntyp wird nicht berücksichtigt)
- Der Name der Funktion ist identisch

- Die Implementation muss für jede überladene Funktion separat erfolgen

Hinweis: Overloading sollte zurückhaltend eingesetzt werden. Wenn möglich sind default-Argumente vorzuziehen.

18.2.1 Deklaration von überladenen Funktionen: Regeln

- Entsprechen Rückgabetyt und Parameterliste der zweiten Deklaration denen der ersten, so wird die zweite als gültige Re-Deklaration der ersten aufgefasst.
- Unterscheiden sich die beiden Deklarationen nur bezüglich ihrer Rückgabetyten, so behandelt der Compiler die zweite Deklaration als fehlerhafte Re-Deklaration der ersten.
→ Der Rückgabetyt von Funktionen kann nicht als Unterscheidungskriterium verwendet werden.
- Nur wenn beide Deklarationen sich in Anzahl oder Typ ihrer Parameter unterscheiden, werden sie als zwei verschiedene Deklarationen mit demselben Funktionsnamen betrachtet (überladene Funktionen).

18.2.2 Funktionen sollen nur dann überladen werden, wenn ...

- die Funktionen eine vergleichbare Operation bezeichnen, die jeweils mit anderen Parametertypen ausgeführt wird
- dieselbe Wirkung nicht durch default-Parameter erreicht werden kann

19 default-Parameter vs. Overloading

Variante mit Overloading:

3 unterschiedliche Funktionne belegen Speicher
3 unterschiedliche Funktionen müssen gewartet werden

```
void print(int i);  
void print(int i, int width);  
void print(int i, char fillchar, int width);
```

Variante mit default-Parametern:

Eine einzige Funktion belegt Speicher
Nur eine Funktion muss gewartet werden

```
void print(int i, int width=0, char fillchar=0);
```

Achtung: Keinesfalls default-Parameter in überladenen Funktionen verwenden!

Teil III

Pointer und Referenzen

20 Höhere und strukturierte Datentypen

20.1 Höhere Datentypen

- Pointer
- Referenzen
- Vektoren

20.2 Strukturierte Datentypen

- Strukturen
- Klassen

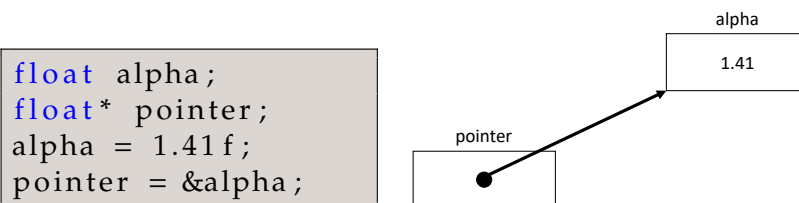
21 Pointer

- Synonym: Zeiger
- Ein Pointer ist eine Variable, welche die Adresse einer im Speicher befindlichen Variablen oder Funktion aufnehmen kann
- Man sagt, der Pointer zeige (to point) auf diese Speicherzelle
- Pointer in C++ sind zu 99.99% identisch zu Pointern in C

21.1 Adresse

- Die Nummer einer Speicherzelle wird als **Adresse** bezeichnet
- Bei einem byteweise adressierbaren Speicher (ist üblich) liegt an jeder Adresse genau 1 Byte

21.2 Standarddarstellung von Pointern



21.3 Pointer und Datentyp

- Pointer in C++ sind typisiert (wie in C), sie zeigen auf eine Variable des definierten Typs
- Oder anders ausgedrückt: Der Speicherbereich, auf den ein bestimmter Pointer zeigt, wird entsprechend des definierten Pointer-Typs interpretiert
- Der Speicherbedarf einer Pointervariablen ist **unabhängig (!)** vom Pointer-Typ. Er ist so gross, dass die maximale Adresse Platz findet (z.B. 32 Bit für 2^{32} Adressen)

21.4 Definition einer Pointervariablen

Typname Datentyp des Pointers

* Kennzeichnung des Pointers

```
Typname* pointerName;
int* ptr1; // ptr1: Pointer auf int
double* ptr2; // ptr2: Pointer auf double
```

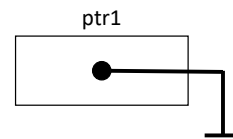


21.4.1 Initialisierung mit Null-Pointer

Mit dem Null-Pointer wird angezeigt, dass der Pointer auf **kein** Objekt zeigt. Dem Pointer wird ein definierter Nullwert zugewiesen.

Hinweis: Der Pointer zeigt nicht auf die Adresse 0!

```
int* ptr = 0; // nicht NULL verwenden!
```



21.4.2 Der Adressoperator & (Referenzierung)

Ist x eine Variable vom Typ Typname, so liefert der Ausdruck &x einen Pointer auf die Variable x, d.h. er liefert die Adresse der Variablen x.

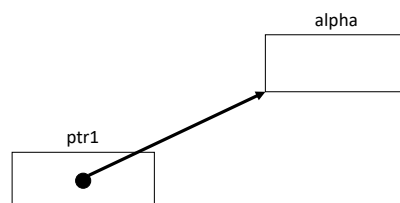
```
int wert;
int* ptr;    // Pointer ptr auf Typ int
             // zeigt auf eine nicht definierte Adresse

ptr = &wert;  // ptr zeigt nun auf Variable wert
             // -> ptr enthaelt Adresse der Variablen wert
```

21.4.3 Kopieren von Adressen

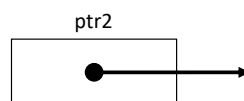
1.

```
float alpha;
float* ptr1 = &alpha;
;
```



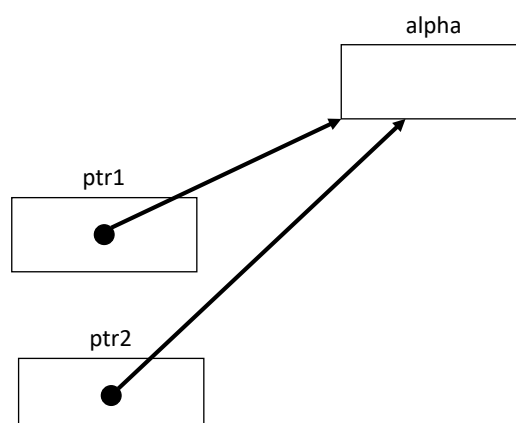
2.

```
float* ptr2;
```



3.

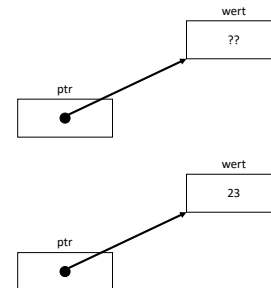
```
ptr2 = ptr1;
```



21.4.4 Der Inhaltsoperator * (Dereferenzierung)

Ist *ptr* ein Pointer vom Typ *Typname*, so liefert der Ausdruck **ptr* den Inhalt der Speicherzelle, auf welche *ptr* zeigt.

```
int wert;           // Variable vom Typ int
int* ptr;           // Pointer ptr auf Typ int
                    // zeigt auf eine nicht definierte Adresse
ptr = &wert;        // ptr zeigt nun auf Variable wert
                    // -> ptr enthaelt die Adresse der Variablen wert
*ptr = 23;          // Aequivalent: wert = 23;
```



21.4.5 const bei Pointern: Vorsicht

1. Variante: konstanter String

```
char str[] = "Ein_String";
const char* text = str;
```

Dies bedeutet nicht, dass der Pointer *text* konstant ist, sondern dass *text* auf einen konstanten String zeigt.

Von rechts nach links lesen:

"text ist ein Pointer auf eine char-Konstante"

erlaubt:

```
char ch = text[1]; // == 'i'
text = "Ein_anderer_String";
```

nicht erlaubt:

```
text[1] = 's';
```

2. Variante: konstanter Pointer

```
char str[] = "Ein_String";
char* const text = str;
```

Hier ist nun der Pointer *text* konstant. Die Position von *const* ist sehr relevant!

Von rechts nach links lesen:

"text ist ein konstanter Pointer auf ein char"

erlaubt:

```
char ch = text[1]; // == 'i'
```

nicht erlaubt:

```
text[1] = 's';
text = "Ein_anderer_String";
```

3. Variante: konstanter Pointer, konstanter String

```
char str[] = "Ein_String";
const char* const text = str;
```

Hier ist nun der Pointer *text* konstant und der Text, wohin er zeigt.

Von rechts nach links lesen:

"text ist ein konstanter Pointer auf eine char-Konstante"

erlaubt:

```
char ch = text[1]; // == 'i'
```

nicht erlaubt:

```
text[1] = 's';
text = "Ein_anderer_String";
```

const bei Pointern in Funktionsköpfen

```
void foo (const int* ptr)
{
    *ptr = 14; // nicht erlaubt
}
```

ptr ist ein Pointer auf eine int-Konstante.

21.4.6 void-Pointer

- void-Pointer sind Objekte, die eine gültige Adresse darstellen
- einem void-Pointer kann jeder Pointer zugewiesen werden

- ein void-Pointer kann ohne Typecast nur anderen void-Pointern zugewiesen werden (**anders als in C**)
- ein void-Pointer kann nicht dereferenziert werden

Hinweis: In C++ sollten void-Pointer kaum noch angewendet werden!

void-Pointer: Beispiele

erlaubt:

```
int a;
int* pi = &a;
void* pv = pi;
pd = static_cast<double*>pv;
```

nicht erlaubt:

```
double* pd = pv
```

21.4.7 Pointer auf Funktionen

- Jede Funktion befindet sich an einer definierten Adresse im Codespeicher
- Diese Adresse kann ebenfalls ermittelt werden
- Interessant wäre, dynamisch zur Laufzeit in Abhängigkeit des Programmlaufs eine unterschiedliche Funktion über einen Funktionspointer aufzurufen

Hinweis: In C++ gibt es für viele Situationen bessere Alternativen zu Funktionspointern (Polymorphismus, *siehe* 57.3)!

21.4.8 Interruptvektortabelle: Tabelle von Funktionspointern

Pointer auf ISR n
...
...
Pointer auf ISR 2
Pointer auf ISR 1

ISR = Interrupt Service Routine

21.4.9 Umsetzung von Funktionspointern in C/C++

Der Name der Funktion kann als Adresse auf den ersten Befehl der Funktion verwendet werden (analog Array).

21.4.10 Beispiel für Funktionspointer

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

int foo(char ch)
{
    int i;           // muss hier definiert werden
    for(i=1;i<=10;i++)
        cout << ch << " ";
    cout << endl;
    return i;
}

int main()
{
    int (*p) (char); // Deklaration des Funktionspointers
    char c;
    int ret;
    cout << "Buchstabe eingeben: ";
    cin >> c;
    p = foo; // ermittle Adresse von foo()
    ret = p(c); // Aufruf foo() ueber Funktionspointer
    return 0;
}
```

22 Referenzen

- Eine Referenz ist ein Alternativname (Alias) für ein Objekt.
- Referenzen ähneln Pointern, sind aber nicht dasselbe. Bei einem Pointer wird immer eine Adresse ermittelt, d.h. dieses Datenobjekt muss sich im adressierbaren Bereich befinden. Eine Referenz kann aber auch auf ein Register verweisen.
- Grundsätzlich sind Referenzen effizienter als Pointer.
- Syntaktisch sind Referenzen einfacher als Pointer, da ein expliziter Referenzierungs- und Dereferenzierungsoperator entfällt.
- Referenzen sind für den Programmierer sicherer anzuwenden als Pointer.
- In gewissen Fällen braucht es Pointer. Wenn nicht, dann sollen Referenzen bevorzugt werden.

22.1 Syntax

```
int x = 24;
int& r1 = x; // Definition der Referenz r1

x = 55; // x == 55, r1 == 55 (dasselbe Objekt)
r1 = 7; // x == 7, r1 == 7 (dasselbe Objekt)
r1++; // x == 8, r1 == 8 (dasselbe Objekt)
```

Hinweis: Referenzen können nach der Definition nicht "umgehängt" werden, d.h. eine Referenz kann und muss nur bei der Definition initialisiert werden und kann nicht später auf etwas anders "zeigen".

22.2 Einsatz

- In folgenden zwei Fällen einsetzen:
 - Bei Parameterübergabe (*call by reference*) anstatt Pointer
 - Als Returntyp (bei Referenz-Rückgabetyt anstatt Pointertyp)
- **Generell:** Objekte einer Klasse und Strukturvariablen sollen immer *by reference* übergeben werden (niemals *by value*)
- Sonst: zurückhaltend einsetzen

23 Pointer und Referenzen auf lokale Variablen

Achtung: Sie dürfen niemals einen Pointer oder eine Referenz auf eine lokale Variable oder ein lokales Objekt mittels `return` zurückgeben!

Grund: Nach Beendigung der Funktion sind die lokalen Variablen ungültig.

24 Zeiger und Referenzen als Parameter und Rückgabewerte

24.1 Call by Value vs. Call by Reference

- Parameter, die *by value* übergeben werden (Wertparameter) werden kopiert, in der Funktion wird mit Kopien gearbeitet.
- Bei Referenzparametern (*call by reference*) wird nur eine Referenz (Alias) des Originals übergeben.
- Nur Parameter, welche *by reference* übergeben werden, können in der Funktion (bleibend) verändert werden.

24.1.1 3 Beispiele

Call by value

```
void swap(int a, int b)
{
    int tmp = 1;
    a = b;
    b = tmp;
}

int main()
{
    int x = 4;
    int y = 3;
    swap(x, y);
    return 0;
}
```

Beim Aufruf von `swap()` werden nur Kopien vertauscht.

Call by reference mit Referenzen

```
void swap(int& a, int& b)
{
    int tmp = 1;
    a = b;
    b = tmp;
}

int main()
{
    int x = 4;
    int y = 3;
    swap(x, y);
    return 0;
}
```

Ok.

Call by reference mit Pointer

```
void swp(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int x = 4;
    int y = 3;
    swap(&x, &y);
    return 0;
}
```

Ok, jedoch mühsame Syntax und evtl. ineffizient.

24.2 Call by reference: wann einsetzen?

1. wenn Parameter in der Funktion verändert werden sollen
2. wenn "grosse" Parameter übergeben werden sollen (*struct, class*)
wenn verhindert werden soll, dass der Parameter verändert wird, so kann dieser mit *const* deklariert werden

```
int foo(const BigType& b);
```

Achtung: Parameterübergabe und Rückgabe von Objekten by value ist ein Hauptgrund für langsame C++-Programme!

24.3 Merke

- Variablen einer Struktur und Variablen einer Klasse (Objekte) müssen immer by reference übergeben werden, niemals by value.
- Read-only Parameter werden zusätzlich mit const spezifiziert.

Teil IV

Arrays, Dynamische Speicherverwaltung

25 Arrays

25.1 Arrays: Vektoren

Problemstellung: Sie müssen 10 Messwerte (z.B. Temperaturwerte) vom Typ `int` speichern.
Diese Darstellung ist sehr unhandlich.

Wie würden sie 1000 Messwerte abspeichern?

```
int data1;  
int data2;  
int data3;  
int data4;  
int data5;  
int data6;  
int data7;  
int data8;  
int data9;  
int data10;
```

25.2 Der Array (Feld, Vektor)

Ein Array bietet eine kompakte Zusammenfassung von mehreren Variablen des gleichen Typs.

```
int data[10]; // ein Array von 10 int-Werten  
int data[1000]; // ein Array von 1000 int-Werten  
double zahl[5]; // ein Array von 5 double-Werten
```

25.2.1 Zugriff auf ein Arrayelement

Hinweis: Der Zugriff auf ein Element eines Arrays erfolgt über den Array-Index. Ist ein Array mit n Elementen definiert, so ist darauf zu achten, dass in C++ (wie in C) der Index mit 0 beginnt und mit $n-1$ endet.

```
int alpha[5]; // Array 'alpha' mit 5 int-Elementen  
alpha[0] = 14; // 1. Element (Index 0) = 14  
alpha[4] = 3; // letztes Element (Index 4)
```

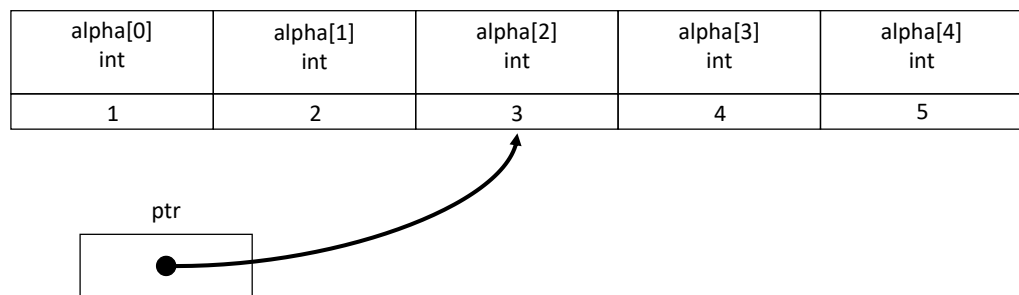
Achtung: Bereichsüberschreitung (geht in C++!)

```
alpha[5] = 4;
```

25.3 Arrays und Pointer

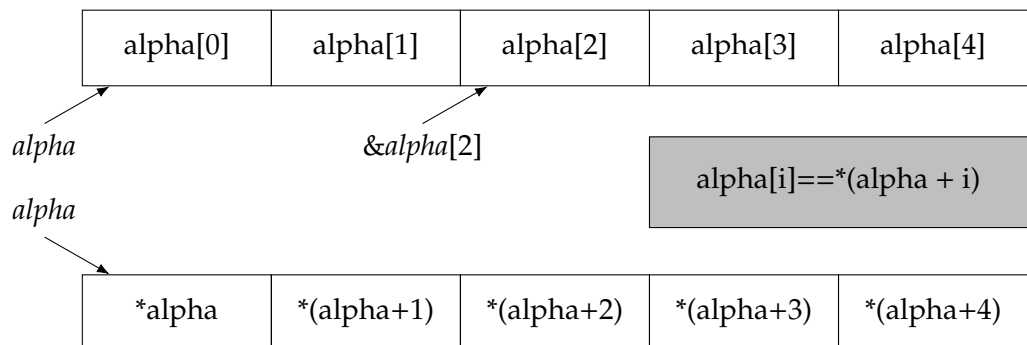
25.3.1 Pro Memoria: Eindimensionales Array (Vektor)

```
int alpha[5];
int* ptr;
ptr = &alpha[2];
*ptr = 3452;
```



25.4 Äquivalenz von Array- und Pointernotation

Der Name des Arrays kann als konstante Adresse des ersten Elementes (Index 0) des Arrays betrachtet werden.



25.5 Vergleichen von Arrays

Hinweis: In C++ gibt es keinen Operator `==`, der zwei Arrays miteinander vergleicht!

- Array-Vergleiche müssen explizit, Element um Element durchgeführt werden.
- oder: der Inhalt der beiden Speicherbereiche wird mit Hilfe der Funktion `memcmp()` byteweise verglichen.

(**Beispiel mit `==`:** Seien `arr1` und `arr2` zwei Arrays

Der Vergleich `arr1 == arr2` prüft, ob die Anfangsadressen der beiden Arrays identisch sind (wird kaum der Fall sein), nicht aber, ob deren Inhalte identisch sind.)

25.6 Der Arrayname ist ein nicht modifizierbarer L-Wert

- Der Arrayname ist die konstante Adresse des ersten Elementes des Arrays und kann nicht verändert werden.
- Auf den Arraynamen können nur die beiden Operatoren `sizeof` und `&` angewandt werden.
- Der Arrayname (z.B. `arr`), als auch der Adressoperator angewandt auf den Arraynamen (`&arr`) ergeben einen konstanten Pointer auf das erste Element des Arrays, der Typ ist jedoch verschieden.
- Einem Arraynamen kann kein Wert zugewiesen werden (einer Pointervariablen schon).

25.7 Automatische Initialisierung von Arrays

- Globale Arrays werden automatisch mit 0 initialisiert
 - globale Arrays sollten aber nur ausnahmsweise verwendet werden
- Lokale Arrays werden nicht automatisch initialisiert
 - der Inhalt eines lokalen Arrays ist bei der Definition undefiniert

25.8 Explizite Initialisierung von Arrays

- Bei der Definition eines Arrays kann ein Array explizit ("manuell") initialisiert werden.
- Der Definition folgt ein Zuweisungsoperator und eine Liste von Initialisierungswerten.

- Die Liste ist mit geschweiften Klammern begrenzt.
- Als Werte können nur Konstanten oder Ausdrücke mit Konstanten angegeben werden, **Variablen sind nicht möglich**.
- Die Werte werden mit Kommata getrennt.
- Nach der Initialisierung können die Elemente nur noch einzeln geändert werden.

25.8.1 Beispiel

```
int alpha[3] = {1, 2*5, 3};
```

ist "äquivalent" zu:

```
int alpha [3];
alpha[0] = 1;
alpha[1] = 2*5;
alpha[2] = 3;
```

25.8.2 Goodies für die explizite Initialisierung

- Werden bei der Initialisierung weniger Werte angegeben als der Array Elemente hat, so werden die restlichen Elemente mit 0 belegt.

```
int alpha[200] = {3, 105, 17};
// alpha[3] bis alpha[199] werden gleich 0 gesetzt
```

- wird bei der Definition keine Array-Grösse angegeben, so zählt der Compiler die Anzahl Elemente automatisch (offenes Array ohne Längenangabe).

```
int alpha[] = {1, 2, 3, 4};
```

25.9 Mehrdimensionale Arrays

```
int alpha[3][4];
```

alpha[0]
alpha[1]
alpha[2]

	Spaltenindex			
	↓			
Zeilenindex →	alpha[0][0]	alpha[0][1]	alpha[0][2]	alpha[0][3]
	alpha[1][0]	alpha[1][1]	alpha[1][2]	alpha[1][3]
	alpha[2][0]	alpha[2][1]	alpha[2][2]	alpha[2][3]

Matrix

Kann betrachtet werden als Vektor $alpha[0]$ bis $alpha[2]$, wobei jedes Vektorelement wiederum einen Vektor mit 4 Elementen enthält.

25.9.1 Initialisierung eines mehrdimensionalen Arrays

```
int alpha[3][4] = {
    {1, 3, 5, 7},
    {2, 4, 6, 8},
    {3, 5, 7, 9}
};
// äquivalent dazu ist die folgende Definition:
int alpha[3][4] = {1, 3, 5, 7, 2, 4, 6, 8, 3, 5, 7, 9};
```

1	3	5	7
2	4	6	8
3	5	7	9

Das erste Element kann offen sein, das zweite muss angegeben werden (z.B. `int alpha[][4] = ...`).

26 Übergabe von Arrays und Zeichenketten

- Bei der Übergabe eines Arrays an eine Funktion wird als Argument der Arrayname übergeben. (i.e. Pointer auf erstes Element des Arrays)
- Der formale Parameter für die Übergabe eines eindimensionalen Arrays kann ein offenes Array sein oder ein Pointer auf den Komponententyp des Arrays.
- Zeichenketten sind char-Arrays und werden deshalb gemäss der oben erwähnten Punkte gehandhabt.

26.1 Beispiel: Array (Vektor) als Parameter

```
enum {groesse = 3};

void init(int* alpha, int size);      // int* alpha == Pointer auf Arrayelement
void ausgabe(int alpha[], int size);  // int alpha[] == offener Array

int main()
{
    int arr[groesse];
    init(arr, sizeof(arr)/sizeof(arr[0]));    // Argument ist Name des Arrays
    ausgabe(arr, sizeof(arr)/sizeof(arr[0])); // Argument ist Name des Arrays
    return 0;
}
```

26.2 Übergabe einer Matrix mittels offenem Array

```
void printMat(const double* const mat[], // Matrix
    int m,           // Anzahl Zeilen
    int n);          // Anzahl Spalten
```

- Der Aufruf erfolgt mit

```
printMat(matA, rows, cols); // bevorzugt
```

oder

```
printMat(&matA[0], rows, cols);
```

oder

```
printMat(&(&matA[0])[0], rows, cols);
```

- `matA` ist vom Typ `double**`

```
matA[0]    ist vom Typ double*
matA[0][0] ist vom Typ double
```

26.3 Zeichenketten (Strings)

Strings sind char-Arrays, abgeschlossen mit dem Zeichen `'\0'`, bzw. `0` (alles analog C).

27 Dynamische Speicherverwaltung

27.1 Pro Memoria: Variablen

- Variablen erleichtern u.a. den Zugriff auf Speicherstellen
 - statt über eine Adresse kann auf die Speicherstelle mit Hilfe des Variablennamens zugegriffen werden
- Die Variablen müssen zur Entwicklungszeit im Code definiert werden
 - zur Laufzeit können keine neuen Variablen erzeugt werden
 - der gesamte Speicherbedarf muss zur Entwicklungszeit statisch mittels Variablen festgelegt werden
 - wie gross wähle ich z.B. einen Array, welcher Messwerte enthält?
 - weiss ich zur Entwicklungszeit bereits die Zahl der Messwerte?
- Der Speicher einer Variablen wird automatisch freigegeben, sobald die Variable nicht mehr gültig ist

27.2 Dynamische Speicherverwaltung

- Speicher kann zur Laufzeit (dynamisch) vom System angefordert werden
 - Operator: *new* (in C: Funktion `malloc()`)
- Dynamisch allozierter Speicher muss wieder explizit freigegeben werden
 - Operator: *delete* (in C: Funktion `free()`)
- Dynamischer Speicher wird nicht auf dem Stack angelegt, sondern auf dem **Heap**
- Auf dynamischen Speicher kann **nur über Pointer** zugegriffen werden

Achtung: Zugriff auf dynamischen Speicher nie verlieren!

27.3 Syntax

```
int* pInt = new int;    // Speicher fuer int alloziert
char* pCh1 = new char;  // Speicher fuer char alloziert
char* pCh2 = new char;  // Speicher fuer char alloziert

*pInt = 23;
std::cin >> *pCh1;
pCh2 = pCh1;

delete pInt; // Speicher wieder freigeben
delete pCh1;
delete pCh2;
```

pCh2 = pCh1

Weil nun auch pCh2 auf die Speicherstelle von pCh1 zeigt, geht der Zugriff auf die Speicherstelle, auf die pCh2 gezeigt hat, verloren. (Memory Leak!)

delete pCh2

Ergibt Fehler, bereits über pCh1 freigegeben.

27.4 Vorsichtsmassnahmen

- der `delete`-Operator kann auch auf den Nullpointer angewendet werden. Es passiert dadurch (definiert) nichts.
- Die Anwendung des `delete`-Operators auf einen bereits freigegebenen Speicherbereich kann Probleme verursachen.
- Oft wird deshalb ein Pointer nach der `delete`-Operation auf `0` gesetzt (defensiver Programmierstil).

```
delete pInt; // Speicher wieder freigeben
delete pInt; // Speicher ist bereits freigegeben
pInt = 0;
delete pInt; // ist problemlos
```

27.5 Memory Leak, Garbage Collection

- Dynamisch allozierter Speicher, welcher nicht freigegeben wurde oder auf welchen der Zugriff verloren ging, belegt weiterhin Platz im Speicher.
- Der faktisch nutzbare Speicher wird somit immer kleiner. Es ist, als ob der Speicher ein Leck hätte. Dieses Fehlverhalten wird deshalb als **Memory Leak** bezeichnet.
- In einigen Programmiersprachen (z.B. Java) gibt es einen **Garbage Collector** (Abfalleimer), welcher nicht mehr benötigten Speicher automatisch freigibt.

Achtung: C++ besitzt keinen Garbage Collector. Der C++-Programmierer ist verantwortlich, dass allozierter Speicher wieder freigegeben wird.

27.6 Dynamische Allokierung von Arrays

- In C++ kann Speicher für einen Array auch erst zur Laufzeit (dynamisch) vom System angefordert (alloziert) werden.
 - Operator: `new[]`
- Der Zugriff auf den Array erfolgt wie bei einem statischen Array.
- Dynamisch allozierte Arrays müssen wieder explizit freigegeben werden.
 - Operator: `delete[]`
 - **`delete[]`, nicht nur `delete`!**

```
int* pInt = new int[100]; // statt einer Konstanten kann auch Variable verwendet
                          // werden (Normalfall)
delete pInt; // Fehler: nur pInt[0] wird freigegeben
delete [] pInt; // korrekter Befehl
```

27.7 Dynamische Allokierung von Matrizen

- Oft wird eine $m \times n$ - Matrix als ein eindimensionaler Array der Grösse ($m \cdot n$) implementiert. Der Zugriff geht dann jedoch nur noch über Pointer:

```
*(matrix+2*n+3) = 23.44;
```

- Mit der im folgenden gezeigten Variante kann auf ein Matrixelement über die Arrayindizes zugegriffen werden:

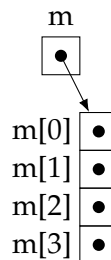
```
matrix[2][3] = 23.44;
```


27.7.1 Dynamische Matrix mit 4 Zeilen und 3 Spalten

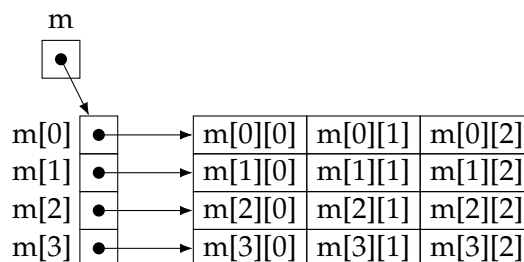
```
double** m = 0;
```



```
m = new double*[4]; // Array mit 4 Elementen vom Typ double* (Pointer auf double)
```



```
for (int i=0; i<4; ++i)
    m[i] = new double[3]; // Jedes m[i] ist Pointer auf Array mit 3 Elementen vom
                          // Typ double
                          // m[i] selbst ist vom Typ double*
```

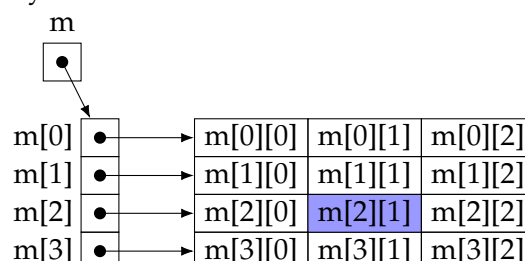


Für die Konstanten 3 und 4 könnten auch Variablen verwendet werden (im Gegensatz zu einer statisch definierten Matrix).

27.7.2 Zugriff auf dynamisch erzeugte Matrix

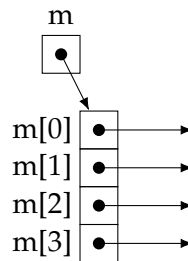
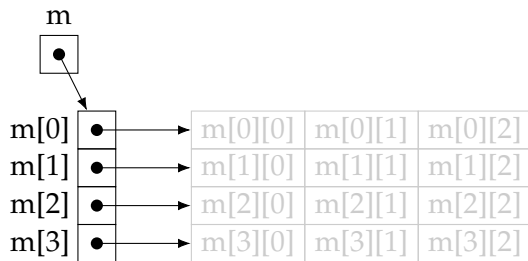
Der Zugriff erfolgt einfach ueber die Arrayindices.

```
m[2][1] = 34.675;
```

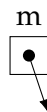
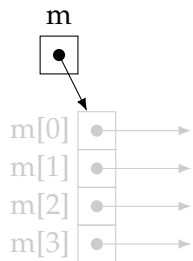


27.7.3 Dynamische Matrix freigeben

```
for(int i=0; i<4; ++i)
    delete [] m[i] // Zuerst jede Zeile freigeben
```



```
delete [] m; // Nun noch den Array mit den double* freigeben
```



Achtung: `m` zeigt immer noch auf die alte Adresse, an welcher sich aber keine gültigen Daten mehr befinden. (Allenfalls `m` wieder auf 0 setzen)

27.8 Effizienz der Matriximplementationen

- Nur mit dieser gezeigten Variante kann auf ein Matricelement über die Arrayindizes zugegriffen werden:
- Der Nachteil dieser Variante ist, dass es pro Zeile einen zusätzlichen Pointer braucht. Die einzelnen Zeilen liegen u.U. nicht auf aufeinanderfolgenden Speicherstellen.
- Wenn eine $m \times n$ - Matrix als ein eindimensionaler Array der Grösse $(m*n)$ implementiert wird, erspart man sich die Zeilenpointer, der Zugriff ist jedoch langsamer und mühsamer. Die einzelnen Elemente der Matrix liegen auf aufeinanderfolgenden Speicherstellen.

Teil V

Scope, Deklarationen, Type Casts

28 Strukturen in C++

- Grundsätzlich sind Strukturen in C++ identisch zu Strukturen in C
- In C++ haben Strukturen noch zusätzliche Möglichkeiten (folgt im Zusammenhang mit Klassen)
- Die Definition und Nutzung von Strukturen ist in C++ einfacher, typedef braucht es nicht

```
struct Point
{
    double x;
    double y;
};

Point p1;
```

29 Gültigkeitsbereiche, Namensräume und Sichtbarkeit

29.1 Gültigkeitsbereiche von Namen (Scope)

- Prinzipiell identisch wie in C
- Der Compiler arbeitet immer Dateiweise
- Namen in einer anderen Datei sind dem Compiler nicht bekannt
- (Globale) Variablen, welche in einer anderen Datei definiert werden, können mit Hilfe des extern-Statements bekannt gemacht werden
- Durch das extern-Statement wird kein Speicherplatz reserviert

```
extern int Foo_globalVariable;
```

- Funktionsprototypen und Definitionen, die von anderen Modulen genutzt werden können (Schnittstellen), werden in einer Headerdatei definiert
- Durch #include der Headerdatei wird der Header geladen und die Namen bekannt gemacht

29.2 Gültigkeitsbereiche in C++

- Lokaler Gültigkeitsbereich (local scope)
Alle in einem Block deklarierten Bezeichner gelten von ihrer Deklaration an bis zum Ende des aktuellen Blocks
- Gültigkeitsbereich Funktionsprototyp, Funktion
Alle in einem Funktionskopf deklarierten Bezeichner (Parameter) gelten in der gesamten Funktion
- Gültigkeitsbereich Namensraum (namespace)
Alle im Namensraum deklarierten Bezeichner gelten von ihrer Deklaration an bis zum Ende des Namensraums
- Gültigkeitsbereich Klasse
Alle in einer Klasse deklarierten Bezeichner gelten von ihrer Deklaration an in der gesamten Klasse

29.3 Gültigkeit (Scope) von Variablen

- Eine Variable ist an einer bestimmten Stelle gültig, wenn ihr Name an dieser Stelle dem Compiler durch eine Vereinbarung bekannt ist
- Gültige Variablen können für den Programmierer unsichtbar sein, wenn sie durch eine andere Variable desselben Namens verdeckt werden

29.4 Lebensdauer von Variablen

- Die Lebensdauer ist die Zeitspanne, in der das Laufzeitsystem des Compilers der Variablen einen Platz im Speicher zur Verfügung stellt
- Mit anderen Worten, während ihrer Lebensdauer besitzt eine Variable einen Speicherplatz
- Globale Variablen leben solange wie das Programm
- Lokale Variablen werden beim Aufruf des Blocks angelegt und beim Verlassen des Blocks wieder (automatisch!) ungültig

29.5 Sichtbarkeit von Variablen

- Variablen von inneren Blöcken sind nach aussen nicht sichtbar
- Globale Variablen und Variablen in äusseren Blöcken sind in inneren Blöcken sichtbar
- Werden lokale Variablen mit demselben Namen wie eine globale Variable oder wie eine Variable in einem umfassenden (äusseren) Block definiert, so ist innerhalb des Blockes nur die lokale Variable sichtbar. Die globale Variable bzw. die Variable in dem umfassenden Block wird durch die Namensgleichheit verdeckt.

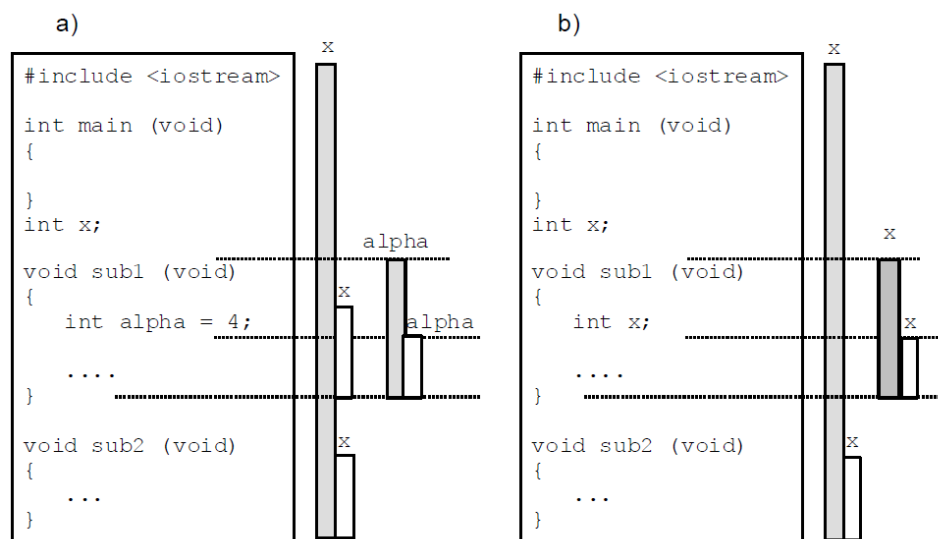
29.6 Schlussfolgerung (naheliegend aber falsch)

- Alle Variablen global definieren, dann muss ich mir keine Sorgen um die Sichtbarkeit machen

→ Stimmt schon, aber:

- Weil die Variablen in demselben Namensraum sind, müsste ich die Variablennamen im gesamten Projekt abstimmen
→ ist nicht praktikabel
- Globale Variablen haben gewichtige Nachteile: Wer hat den Variablenwert wo wie geändert?
→ schwer nachzuvollziehen

29.7 Lebensdauer (grau) und Sichtbarkeit (weiss)



29.8 Codierstil

- Variablen so lokal wie möglich definieren, d.h. im innersten möglichen Block (Tipp: nur am Anfang eines Blocks)
- Globale Variablen wenn immer möglich vermeiden. Sie müssen speziell gekennzeichnet werden. Sie sollen (in C) mit einem Prefix (Modulkürzel) gefolgt von einem underscore character (`_`) beginnen. Dadurch werden die Namen eindeutig.
Beispiel:
Die globale Variable `counter` im Modul `Foo` muss wie folgt definiert werden:

```
int Foo_counter ;
```

- Globale Variablen am Anfang der Datei definieren, d.h. auf jeden Fall vor der ersten Funktion

Hinweis: Besser (in C++): Namespace definieren

30 Namensräume (Namespaces)

Namen, die zu verschiedenen Namensräumen gehören, dürfen auch innerhalb desselben Gültigkeitsbereichs gleich sein. In C gibt es die folgenden Namensräume (gilt auch für C++):

- Marken
- Namen von Strukturen, Unions und enums
- Jede Struktur und Union für ihre Feldnamen
- Bezeichner von Variablen, Funktionen, typedef-Namen, enum-Konstanten

In C++ können zusätzlich explizit definierte Namensräume verwendet werden.

Hinweis: Dadurch können (und sollen unbedingt!) die in C üblichen Modulkürzel vermieden werden.

30.1 Explizite Namensräume in C++

- Nebst den vordefinierten (impliziten) Namensräumen des vorherigen Abschnitts können in C++ explizit eigene Namensräume (namespaces) definiert werden.
- Bezeichner müssen nur innerhalb ihres Namensraums eindeutig sein
- Für jedes Modul in C (mit Modulkürzel) soll in C++ ein Namensraum definiert werden
- Sie haben bisher in allen C++-Übungen bereits den Namensraum std verwendet

30.2 C++-Mechanismen für Namespaces

- Deklaration von Namespaces
- Namespace-Alias
Einem bestehenden Namespace einen anderen Namen zuweisen (eher selten)

```
namespace fbssLib = financial_branch_and_system_service_library ;
```

- using-Deklaration
- using-Direktive

30.3 Deklaration von Namespaces

- Alle in einem Namespace deklarierten Bezeichner werden diesem Namespace zugeordnet
- Auf die Bezeichner des Namespaces kann mit dem Scope-Operator :: zugegriffen werden
- Syntax:
Hinter dem Schlüsselwort namespace folgt der Name des Namespaces gefolgt von einem Block
- Innerhalb einer Datei kann mehr als ein Namespace deklariert werden (eher unüblich) und ein Namespace kann in mehreren Dateien deklariert sein (häufig, d.h. die Elemente eines Namespaces werden in mehreren Dateien implementiert)

30.3.1 Beispiel

```
namespace myLib1
{
    int i;
    void foo();
}

namespace myLib2
{
    int i;
    void foo();
    int go();
}

...
{
    myLib1::foo(); // vollstaendiger Name von foo()
    myLib2::i = 17; // vollstaendiger Name von i
}
```

30.4 using-Deklaration

- Eine using-Deklaration "importiert" Namen aus einem Namensraum und macht ihn ohne explizite Namensraumangabe verwendbar
- Sie kann lokal in einem Block oder global ausserhalb eines Blocks verwendet werden
- Deklariert nur einzelne Namen

```
int main()
{
    using myLib1::foo; // lokales Synonym
    foo();           // ruft myLib1::foo() auf
}
```

30.5 using-Direktive

- Eine using-Direktive macht alle Namen aus einem Namensraum ohne explizite Namensraumangabe verwendbar

```
using namespace myLib1; // ''importiert'' alle Namen aus myLib1

int main()
{
    foo();           // ruft myLib1::foo() auf
}
```

30.6 using namespace kann zu Konflikten führen

- Wenn bei mehreren using namespace-Deklarationen und/oder -Direktiven die Namen (ohne Namespace-Angabe) nicht eindeutig sind, müssen die Namen voll qualifiziert verwendet werden (mit Namespace-Angabe)

```

namespace myLib1
{
    int i;
    void foo();
}

namespace myLib2
{
    int i;
    void foo();
    int go();
}

...
{
    myLib1::foo(); //ist nicht eindeutig, Compiler reklamiert
    myLib2::i = 17; // vollstaendiger Name von i
}

```

30.7 Namenlose Namespaces

- Ein namenloser Namespace wird wie ein spezieller Namensraum mit einem systemweit eindeutigen Namen behandelt

Hinweis: Es ist guter Programmierstil, den Gültigkeitsbereich aller nur intern verwendeten Funktionen und Daten mit Hilfe von namenlosen Namespaces auf den Bereich eingrenzen, in dem die Objekte verwendet werden. (In C hat man dafür die Funktionen mit static gekennzeichnet)

```

namespace
{
}

```

30.8 Zugriff auf globale Variable mit Scope-Operator

```

int zahl = 11; // globale Variable

int main()
{
    int zahl = 22; // lokale Variable
    zahl = zahl + 4; // lokale Variable
    ::zahl = 23; // Zugriff auf globale Variable
}

```

31 Speicherklassen

31.1 Speicherklassen in C++

- auto

Ist default, wenn nichts geschrieben wird. Eine mit auto deklarierte Variable wird nach Beendigung des Scopes automatisch entfernt.

Achtung: hat ab C++11 eine andere Bedeutung!!

- register
Ist dasselbe wie auto mit zusätzlichem Hinweis an Compiler: wenn es geht in ein Register legen (sehr zurückhaltend einsetzen, besser gar nicht)
- static
- extern
- mutable
(später im Zusammenhang von Klassen)

31.1.1 Speicherklasse static: Variablen

- static-Variablen sind im Datenbereich, nicht auf dem Stack
- Sie werden automatisch auf 0 initialisiert, wenn nichts anderes steht
- Gültigkeitsbereich ist der Block, in dem die Variable definiert ist
- static-Variablen, welche ausserhalb einer Funktion definiert sind (globale Variablen). sind nur in der Datei gültig, in der sie definiert werden
- static-Variablen sind nur einmal vorhanden (auch in multi-threading-Umgebungen), d.h. ihr Wert wird erhalten, auch wenn die Funktion beendet ist. Beim nächsten Aufruf der Funktion geht es mit dem alten Wert weiter.
- Nur einsetzen, wenn man das will!

31.1.2 Speicherklasse static: Funktionen

- static-Funktionen sind nur in der Datei, in welcher sie definiert sind, sichtbar
- Alle Funktionen, welche nicht aussen (für andere Units) sichtbar sein sollen, sollten deshalb in C als static definiert werden
- In C++ können dafür namenlose Namespaces verwendet werden (bevorzugt)

31.1.3 Speicherklasse extern: Externe Variablen

- Eine externe Variable kann nur in einer einzigen Datei definiert werden (ohne Speicherklasse extern)
- In den anderen Dateien wird sie mit extern deklariert (bekannt gemacht)
- Eine manuell Initialisierung ist nur bei der Definition möglich
- Globale Variablen, welche nicht manuell initialisiert werden, werden automatisch mit 0 initialisiert
- extern-Deklarationen werden üblicherweise in einer Headerdatei deklariert und am Beginn der Datei mit #include eingefügt

31.2 Typqualifikationen (Kap. 9.2.2)

- const
const-Objekte können nicht verändert werden (read-only)
- volatile
Der Compiler wird angewiesen, keine Optimierungen (soweit sie die Variable betreffen) vorzunehmen
- volatile wird oft bei Embedded Systems angewandt, wenn z.B. "hinter" einer Variable ein Register liegt.
- const und volatile können auch kombiniert werden, z.B. bei read-only-Hardwareregistern

31.3 Funktionsattribute

- inline
bereits bekannt
- virtual
später im Zusammenhang mit Klassen
- explicit
später im Zusammenhang mit Klassen

32 Typdefinitionen

32.1 typedef zur Vereinbarung eigener Datentypen

- analog C
- In C++ kann aber z.B. bei structs das typedef weggelassen werden
- In C:

```
typedef struct {int x;
               iny;} Point;
```

- In C++:

```
struct Point {
    int x;
    int y;
};
```

- Stil: eigene Typen werden mit einem Grossbuchstaben begonnen

32.1.1 Beispiel

```
struct Point {
    int x;
    int y;
};

struct Line {
    Point p1;
    Point p2;
};

int main(void)
{
    line myLine = {12, -34,    // p1
                  783, 12}; // p2

    std::cout << "Startpunkt: (" << myLine.p1.x << " , "
               << myLine.p1.y << ")\n";
    std::cout << "Endpunkt: (" << myLine.p2.x << " , "
               << myLine.p2.y << ")\n";
    return 0;
}
```

32.2 Gewährleistung von Portabilität

- Oft muss z.B. ein Register ein 16 Bit breiter Wert geschrieben werden. Welcher Typ ist nun 16 Bit breit?
- Das ist implementationsabhängig (vielleicht unsigned short, unsigned int, ...)
- Um die Portabilität (Umschrieben auf ein anderes System) zu vereinfachen, wird ein 16 Bit breiter Datentyp (Word) definiert und dann ausschliesslich verwendet (in stddef.h sind diese Typen üblicherweise bereits definiert). Auf einem anderen System ist dann nur noch dieser typedef zu ändern.

```
typedef unsigned short uint16_t;
```

32.3 Wie setzt der Compiler ein typedef um?

- Ein typedef ist mehr oder weniger eine reine Textersetzung. Erklärung anhand des folgenden Beispiels:

```
typedef struct { int x;
               int y;} Point;
```

- Überall im Code, wo nun das Wort Point gefunden wird, ersetzt der Compiler dieses in einem ersten Durchgang mit dem Text

```
typedef struct { int x;
               int y;}
```

33 Initialisierung

- analog C

34 Type-Cast (Typumwandlungen)

34.1 Typumwandlungen im Allgemeinen

- Unsafe conversion
Wenn bei der Typumwandlung signifikante Stellen verloren gehen können (typischerweise bei einer Umwandlung von einem "grösseren" in einen "kleineren" Typ, z.B. von double nach int)
Bei int ist sowohl die Genauigkeit als auch die maximal darstellbare Zahl
- Safe conversion
Wenn bei der Typumwandlung keine signifikanten Stellen verloren gehen können (typischerweise bei einer Umwandlung von einem "kleineren" in einen "grösseren" Typ, z.B. von int nach double)

34.2 Implizite Typumwandlung

- Die implizite (automatische) Typumwandlung wird auch als Standard-Typumwandlung bezeichnet
- Sie erfolgt analog zur Programmiersprache C (siehe dort)

34.3 Explizite Typumwandlung

- Nebst den impliziten (automatischen) Typumwandlungen kann in C++ mit Hilfe von 6 verschiedenen cast-Operatoren eine explizite Typumwandlung bewirkt werden.
- Bei der expliziten Typumwandlung gibt der Programmierer explizit an, was er will.

Achtung: Bei der expliziten Typumwandlung übernimmt der Programmierer die Verantwortung, dass die Umwandlung keine Probleme ergibt.
(z.B. Umwandlung von grosser Zahl in kleineren Typ)

34.3.1 Explizite Typumwandlung #1, 2: C-Stil und Funktionsstil

- Stroustrup: "The C and C+ cast is a sledgehammer..."
- Syntax für C-Stil (einzige Variante in C):
(Zieltyp)expression

```
int a = (int) 4.6; // a == 4
```

- Syntax für Funktionsstil:
Zieltyp(expression)

```
int a = int(4.6); // a == 4
```

34.3.2 Typumwandlung mit C-Stil und Funktionsstil

Typumwandlung ist...

- einfache Reinterpretation der bitweisen Darstellung des Ausdrucks
- einfache arithmetische Grössenanpassung
- ein const- oder volatile-Attribut zu einem Ausdruck hinzufügen oder entfernen
- andere (eventuell implementierungsabhängige) Umwandlung

Achtung: Aus dem Sourcecode geht nicht hervor, welche der aufgeführten Typumwandlungen der Programmierer wollte.

Diese beiden Casts sollten in C++ nicht verwendet werden!

34.3.3 Explizite Typumwandlung #3: const_cast

Anwendung:

Ausschliesslich die (vorübergehende) Entfernung des const-Qualifikators, d.h. die Umwandlung eines Ausdrucks vom Typ T mit den optionalen Qualifikatoren const und volatile in einen Ausdruck desselben Typs ohne den Qualifikator const

Syntax:

```
const_cast<Zieltyp>expression

const char* findSubString(const char* str, const char* subStr)
{
    return strstr(const_cast<char*>(str),
                  const_cast<char*>(subStr));
}
```

Die Funktion strstr() akzeptiert nur Parameter vom Typ char* (ohne const)

34.3.4 Explizite Typumwandlung #4: static_cast

Anwendung:

Umwandlung von Objekten einer Klasse auf Objekte einer Basisklasse oder die Umwandlung mittels einer Umwandlungsfunktion.

Wenn schon Type cast, dann ist static_cast die häufigste.

34.3.5 Explizite Typumwandlung #5: dynamic_cast

Anwendung:

Umwandlung von polymorphen Objekten im Zusammenhang mit dem Typsystem von C++ eingesetzt (Stichwort RTTI = Runtime Type Information System)

Näheres folgt später im Zusammenhang mit Klassen und Polymorphismus.

34.3.6 Explizite Typumwandlung #6: reinterpret_cast

Anwendung:

reinterpret_cast ist eine neue Interpretation der zugrunde liegenden Bitkette.

Syntax:

```
reinterpret_cast<Zieltyp>expression  
  
char* p = new char[20];  
...  
int* pi = reinterpret_cast<int*>p;
```

Teil VI

Module und Datenkapseln

35 Modul (Unit)

Motivation:

- **Arbeitsteilung**
Grosse Programme werden von mehreren Personen entwickelt. Praktikabel ist, wenn nur eine Person an einer bestimmten Datei arbeitet.
- **Effizienz**
Eine Übersetzungseinheit (Datei) muss bei jeder Änderung neu übersetzt werden (je grösser die Datei desto langsamer die Übersetzung)
- **Strukturierung**
Ein grosses Programm in mehrere vernünftige Teile (Baugruppen, Units) aufteilen (Divide and conquer)

35.1 Nomenklatur: Modul vs. Unit

- Ein Programmbaustein wird traditionell mit Modul (der oder das Modul) bezeichnet
- Der Test eines Moduls heisst folglich Modultest
- Das Vorgehen, welches Module generiert, heisst Modularisierung
- Heute üblicher wird Modul mit Unit, der Test mit Unittest bezeichnet, das Vorgehen heisst weiterhin Modularisierung
- Prinzipiell spreche ich künftig meist von Unit und Unittest

35.2 Ziele der Modularisierung

- Klare, möglichst schlanke Schnittstellen definieren
- Units so bilden, dass Zusammengehörendes in einer Unit isoliert wird (Kohäsion) soll hoch sein
- Schnittstellen zwischen den Units sollen klein sein (Kopplung soll klein sein)
- Abhängigkeiten unter den Units sollen eine Hierarchie bilden, zirkuläre (gegenseitige) Abhängigkeiten müssen vermieden werden

35.3 Eigenschaften einer Unit (eines Moduls)

- realisiert eine in sich abgeschlossene Aufgabe
- kommuniziert über ihre Schnittstelle mit der Umgebung
- kann ohne Kenntnisse ihres inneren Verhaltens in ein Gesamtsystem integriert werden
- ihre Korrektheit kann ohne Kenntnis ihrer Einbettung in einem Gesamtsystem nachgewiesen werden (mittels Unittest)

35.4 Bestandteile eine C++-Programms

- Eine Hauptfunktion main()
- Eine Reihe von unabhängigen Programmbausteinen (Units)

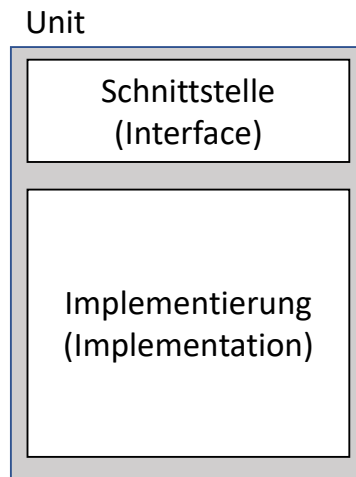
35.5 Unitkonzept

- Interface definiert die Schnittstelle, d.h. die Deklarationen wie Funktionsprototypen, etc. (Schaufenster)
- Implementation: In diesem Teil sind die Unterprogramme definiert, d.h. auscodiert (Werkstatt)
- Das Interface wird in einer Headerdatei (*.h) beschrieben, die Implementation liegt in einer *.cpp-Datei

35.6 Geheimnisprinzip (Information Hiding)

35.6.1 Information Hiding

- In der Schnittstelle (Headerdatei) wird alles beschrieben, was ein Nutzer dieser Unit wissen muss
- Der innere Aufbau der Unit (*.cpp) muss (darf) dem Nutzer der Unit nicht bekannt sein, er benötigt diese Informationen auch gar nicht
- Der Nutzer der Unit darf keine Annahmen bezüglich des inneren Aufbaus der Unit treffen



- Der Entwickler der Unit darf den inneren Aufbau der Unit ändern, solange dadurch die Schnittstelle nicht geändert werden muss

35.6.2 Konzept der Datenkapsel

- Eine Unit besteht aus Funktionen und Daten
- In der Schnittstelle wird definiert, was für den Nutzer zur Verfügung steht. Dies können Funktionen und Daten sein
- Die Datenkapsel fordert nun zusätzlich, dass auf die Daten nicht direkt zugegriffen werden darf, sondern nur über Zugriffsfunktionen

35.6.3 Beispiel für Datenzugriff bei Datenkapsel

```
// interne Daten
int counter; // gehoert zur Unit, nicht in Interface
// wie kann das in C bewerkstelligt werden?

// Schnittstelle (Interface)
void setCounter(int c)
{
    counter = c;
}

int getCounter(void)
{
    return counter;
}
```

35.6.4 Beispiel für Unit Rechteck (ohne Datenkapsel)

```
// interne Daten
double a; // 1. Seite
double b; // 2. Seite
double area; // Rechtecksflaeche

// Schnittstelle (Interface)

// direkter Zugriff auf a, b, area
// Annahme: area hat immer den aktuellen Wert, d.h. es muss
// bei jeder Aenderung von a und b (durch den Client!) berechnet werden
```

Achtung: Sehr gefährlich! (kann kaum sichergestellt werden)

35.6.5 Beispiel für Unit Rechteck: Verbesserung #1

```
// interne Daten
double a; // 1. Seite
double b; // 2. Seite
double area; // Rechtecksflaeche

// Schnittstelle (Interface)
// kein direkter Zugriff mehr auf a, b, area
// Funktionen setA(), setB(), getA(), getB(), getArea()
void setA(double newA)
{
    a = newA;
    area = a * b;
}

double getArea(void)
{
    return area;
}
```

Hinweis: Evtl. gefährlich (Berechnung von area könnte vergessen werden). Und: soll die Multiplikation wirklich bei jeder Aenderung durchgeführt werden?

35.6.6 Beispiel für Unit Rechteck: Verbesserung #2

```
// interne Daten
double a; // 1. Seite
double b; // 2. Seite
// double area; // Rechtecksflaeche , Attribut wird entfernt

// Schnittstelle (Interface)
// Attribut area wird entfernt
// Funktionen setA() , setB() , getA() , getB() , getArea()
void setA(double newA)
{
    a = newA;
}

double getArea(void)
{
    return a * b;
}
```

Hinweis: Dank Datenkapsel darf das Attribut area entfernt werden. Die Schnittstelle ändert sich dadurch nicht.

35.7 Unit nutzen

```
#include "foo.h"
// dadurch wird die Schnittstelle der Unit foo bekanntgemacht
```

35.8 Unit-Schnittstelle definieren (in Headerdatei)

```
// Datei: foo.h
#ifndef FOO_H_
#define FOO_H_

// Deklarationen

#endif /* FOO_H_ */
```

Hinweis: include-Guard: verhindert das mehrfache include derselben Datei

35.9 Deklarationsreihenfolge in der Headerdatei (*.h)

Achtung: kein using namespace ... in Headerdateien!

1. Dateikommentar
2. #include der verwendeten System-Header (iostream, etc.)

- #include <...>
- 3. #include der projektbezogenen Header (#include "...")
- 4. Konstantendefinitionen
- 5. typedefs und Definitionen von Strukturen
- 6. Allenfalls extern-Deklarationen von globalen Variablen
- 7. Funktionsprototypen, inkl. Kommentare der Schnittstelle, bzw. Klassendeklarationen

Hinweis: Punkte 2-7 sind innerhalb des include-Guards.

35.10 Reihenfolge in der Implementierungsdatei (*.cpp)

- 1. Dateikommentar
 - 2. #include der verwendeten System-Header (iostream, etc.)
 - 3. #include der projektbezogenen Header
 - 4. allenfalls globale Variablen und statische Variablen
 - 5. Präprozessor-Direktiven
 - 6. Funktionsprototypen von lokalen, internen Funktionen
 - 7. Definition von Funktionen und Klassen
- (Kommentare aus Headerdatei nicht wiederholen!)

35.11 #include-Konzept

- Mit den #includes wird oft ein Riesenchaos veranstaltet
- Der Einfachheit halber werden ab und zu einfach alle oder fast alle Headerdateien inkludiert
- Das muss unbedingt verhindert werden

Regel: In jeder Datei (*.h, *.cpp, *.c) werden genau die Headerdateien inkludiert, welche diese Datei selbst benötigt!

35.12 Unit compilieren

g++ -cfoo.cpp

- Dadurch entsteht noch kein ausführbares Programm, sondern nur die Datei foo.o, der Objectcode
- Dies muss mit allen *.cpp-Dateien gemacht werden

35.13 Units linken

g++ -o foo foo.o goo.o hoo.o

- Alle Objectdateien müssen gelinkt werden
- Dadurch werden allenfalls noch offene Verbindungen (Links) zu aufgerufenen Funktionen aufgelöst

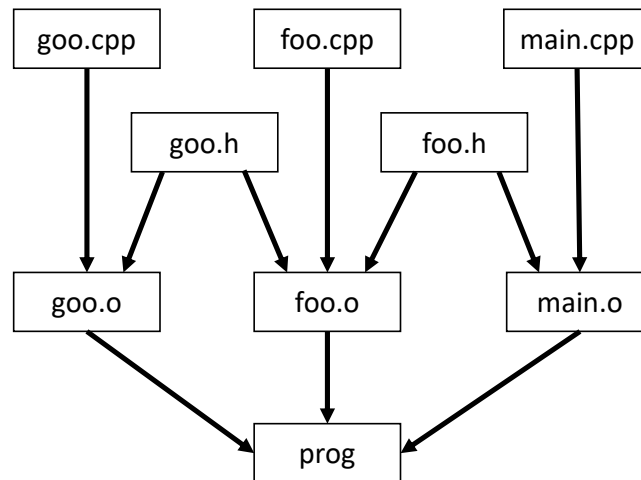
35.14 Buildprozess

- Der Buildprozess beinhaltet alle Schritte, um ein ausführbares Programm zu erhalten, bzw. aufzubauen (englisch to build)
- ```
g++ -c foo.cpp
g++ -c goo.cpp
g++ -c hoo.cpp
g++ -o foo foo.o goo.o hoo.o
```

**Hinweis:** Es wäre mühsam, wenn diese Befehle jedesmal neu eingetippt werden müssten. Deshalb wird in der Praxis oft ein Buildtool eingesetzt, z.B. make.

## 35.15 Make-Tool

### 35.15.1 Abhängigkeiten zwischen Dateien



### 35.15.2 make-File

- In einem make-File können Abhängigkeiten definiert werden
- Wenn eine Datei geändert wurde, dann werden alle Operationen ausgeführt mit den Dateien, welche von dieser geänderten Datei abhängen
- Der Befehl (g++) wird z.B. nur dann ausgeführt, wenn sich an den Dateien, zu denen eine Abhängigkeit besteht, etwas geändert hat

### 35.15.3 Beispiel: makefile

```
makefile
cc=g++
CFLAGS= -c -Wall
LFLAGS= -Wall
OBJ=foo.o goo.o main.o
EXE=prog
```

```
{EXE}: {OBJ}
 {cc} {LFLAGS} -o {@} {OBJ}

foo.o: foo.h goo.h foo.cpp
 {cc} {CFLAGS} -o {@} foo.cpp

goo.o: goo.h goo.cpp
 {cc} {CFLAGS} -o {@} goo.cpp

main.o: foo.h main.cpp
 {cc} {CFLAGS} -o {@} main.cpp

clean:
 rm -f {OBJ} {EXE}
```

**Targets:** Diese können mit make angesprungen werden, z.B. make foo.o  
Wenn make ohne Parameter aufgerufen wird, dann wird das erste Target angesprungen.

Abhängigkeitsliste, d.h. von diesen Dateien ist das Target abhängig

Befehl, der ausgeführt werden muss, falls etwas geändert hat.  
Wichtig: Erstes Zeichen muss ein TAB sein.

## Teil VII

# Eclipse IDE

## 36 Eclipse

- Integrated Development Environment (IDE, Integrierte Entwicklungsumgebung)
- Open-Source Software (OSS)
- Offene, erweiterbare Architektur basierend auf Plug-Ins
- Implementiert in Java
- Auf verschiedenen Plattformen lauffähig (multi-platform)
- Für unterschiedliche Programmiersprachen (multi-language)

### 36.1 Workspace

- Der Workspace enthält vom Benutzer definierte Daten (Projekte und Ressourcen wie Ordner und Files)
- Er enthält alle User-Metadaten (Code, Scripts, Database objects, Konfigurationsdaten)
- Ein Benutzer arbeitet zu einer bestimmten Zeit genau in einem einzigen Workspace

### 36.2 Ressourcen (Resources)

- Oberbegriff für
  - Projekte
  - Ordner
  - Files
- Üblicherweise in einer hierarchischen Struktur betrachtet
- Können editiert werden

### 36.3 Project

- Ein logisches Speicherkonzept für die Speicherung von Programmen
- Gehört einem Workspace an
- Ist implementiert als Verzeichnis in einem Workspace

### 36.4 Debugger

#### 36.4.1 Testen und Debugging

- Testen und Debugging sind zwei unterschiedliche Prozesse
- Das Ziel eines Tests ist, Fehler zu finden
- Das Ziel des Debuggings ist, diese Fehler zu lokalisieren und zu korrigieren

#### 36.4.2 Funktionen eines Debuggers

- Betrachten von Variablenwerten
- Unterbrechung des Programmablaufs mit Breakpoints
- Schrittweise Ausführung von Programmen (Step into, Step out)

#### 36.4.3 Assertions (Zusicherungen)

```
#include <cassert>
assert(i>0);

// in C:
#include <assert.h>
assert(i>0);
```

- Zweck:

Überprüfung von logischen Annahmen während der Entwicklungsphase, speziell für die Überprüfung von Anfangs- und Endbedingungen in einer Funktion

- Das Programm bricht mit einer Fehlermeldung ab, falls das Argument den bool'schen Wert false besitzt. Im Beispiel oben: Abbruch, falls  $i \leq 0$

#### 36.4.4 Zu beachten bei Assertions

- `assert()` ist wirkungslos, wenn ohne Debugschalter compiliert wird. Dies ist in der (Release-)Version der Fall, die ausgeliefert wird
- Bei den `assert()`-Anweisungen darf deshalb kein Nebeneffekt programmiert werden, da dieser in der ausgelieferten Version fehlen würde
- Beispiel:  
`assert(openFile(filename) == ok);`  
Wenn ohne Debugschalter compiliert wird, würde das File nicht mehr geöffnet

## Teil VIII

## Klassen

## 37 Beispielprojekt Stack

## 37.1 Stack



- Der Stack ist ein oft verwendetes Speicherkonstrukt für Daten
- Bei einem Stack werden neue Elemente immer oben eingefügt
- Elemente werden immer auch wieder oben weggenommen
- Synonyme:
  - Stapel
  - LIFO (Last In First Out)
  - (Kellerspeicher)

## 37.1.1 Stack - Operationen

|           |                                       |
|-----------|---------------------------------------|
| push()    | ein neues Objekt einfügen             |
| pop()     | ein Objekt entfernen                  |
| isEmpty() | liefert true falls der Stack leer ist |
| isFull()  | liefert true falls der Stack voll ist |
| init()    | initialisiert einen leeren Stack      |

## 37.1.2 Demo: Codebeispiel für Stack (Stack\_Datenkapsel)

listings/Stack\_Datenkapsel/stack.h

```
// Datei: stack.h
// Schnittstellendefinition fuer Stack
// R. Bonderer, 24.03.2010
#ifdef STACK_H_
#define STACK_H_

namespace storage
{
void init();
// initialisiert den Stack, muss als erste Methode aufgerufen werden, bevor Stack
// benutzt werden kann
```

```

void push(int e);
// legt ein Element auf den Stack, falls der Stack noch nicht voll ist
// wasError() gibt Auskunft, ob push() erfolgreich war

int pop();
// nimmt ein Element vom Stack, falls der Stack nicht leer ist
// wasError() gibt Auskunft, ob pop() erfolgreich war

int peek();
// liest das oberste Element vom Stack, falls der Stack nicht leer ist
// wasError() gibt Auskunft, ob peek() erfolgreich war

bool isEmpty();
// return: true: Stack ist leer
// false: sonst

bool isFull();
// return: true: Stack ist voll
// false: sonst

bool wasError();
// return: true: Operation war fehlerhaft
// false: sonst
} // namespace storage

#endif // STACK_H_

```

## listings/Stack\_Datenkapsel/stack.cpp

```

// Datei: stack.cpp
// implementiert Stackoperationen
// R. Bonderer, 24.03.2016

#include "stack.h"
namespace // nameless namespace fuer file-lokale Daten
{
enum {maxElems = 10}; // Anzahl Stackelemente
int elem[maxElems]; // Array fuer Speicherung des Stacks
int top; // Arrayindex des naechsten freien Elements
bool error; // true: Fehler passiert; false: sonst
}

namespace storage
{
void init()
{
top = 0;
error = false;
}

void push(int e)
{
error = isFull();
if (!error)
{
elem[top] = e;
}
}
}

```

```

 ++top;
}
}

int pop()
{
 error = isEmpty();
 if (!error)
 {
 --top;
 }
 return elem[top]; // ist auch ok im Fehlerfall
}

int peek()
{
 error = isEmpty();
 if (!error)
 return elem[top-1];
 else
 return elem[top]; // elem[top] ist immer ein gueltiges Element
}

bool isEmpty()
{
 return top == 0;
}

bool isFull()
{
 return top == maxElems;
}

bool wasError()
{
 return error;
}
} // namespace storage

```

## listings/Stack\_Datenkapsel/stacktest.cpp

```

// Datei: stacktest.cpp
// implementiert ein Testprogramm fuer Stackoperationen
// R. Bonderer, 24.03.2010
#include <iostream>
#include "stack.h"
using namespace std;
using namespace storage;

int main(void)
{
 char ch = 0;
 int i;
 init(); // storage::init()
 do
 {

```

```

cout << "\n\nOperation_ (Quit, _pUsh, _pOp, _peeK, _isEmpty, _isFull) _";
cin >> ch;
switch(ch)
{
case 'Q': // quit
case 'q':
 break;
case 'U': // push
case 'u':
 cout << "\nElement_to_push:_";
 cin >> i;
 push(i);
 if (wasError())
 cout << "\nError:_Stack_full.";
 break;
case 'O': // pop
case 'o':
 i = pop();
 if (wasError())
 cout << "\nError:_Stack_is_empty_(nothing_to_pop).";
 else
 cout << "\nPopped_element_" << i;
 break;
case 'K': // peek
case 'k':
 i = peek();
 if (wasError())
 cout << "\nError:_Stack_is_empty_(nothing_to_peek).";
 else
 cout << "\nPeeked_element_" << i;
 break;
case 'E': // isEmpty
case 'e':
 if (isEmpty())
 cout << "\nStack_is_empty.";
 else
 cout << "\nStack_contains_elements.";
 break;
case 'F': // isFull
case 'f':
 if (isFull())
 cout << "\nStack_is_full.";
 else
 cout << "\nStack_has_spare_capacity.";
 break;
default:
 cout << "\nInvalid_operation.";
 break;
}
} while (ch != 'Q' && ch != 'q');
return 0;
}

```

## 37.2 Queue





- Die Queue ist ein weiteres Speicherkonstrukt für Daten
- Bei einer Queue werden neue Elemente immer am Ende (tail) eingefügt
- Elemente werden immer am Anfang (head) weggenommen
- Synonyme:
  - Warteschlange
  - FIFO (First In First Out)
  - Pipe
  - Buffer (engl.)
  - Puffer (dt.)

### 37.2.1 Queue - Operationen

|                     |                                       |
|---------------------|---------------------------------------|
| enqueue() / write() | ein neues Objekt hinzufügen           |
| dequeue() / read()  | ein Objekt entfernen                  |
| isFull()            | liefert true falls die Queue voll ist |
| isEmpty()           | liefert true falls die Queue leer ist |
| init()              | initialisiert eine leere Queue        |

## 38 Objektorientierte Programmierung

### 38.1 Modell und Realität

- Ein Modell ist eine Abstraktion (Reduktion) der Realität auf das Wesentliche, den Modellierer Interessierende, niemals aber die ganze Realität
- Ein Modell spiegelt immer auch die Sichtweise des Betrachters

### 38.2 Zu (subjektiven) Sichtweise des Betrachters

Ein zur Hälfte mit Wasser gefülltes Glas:

- Optimist:  
Glas ist halb voll
- Pessimist:  
Glas ist halb leer
- Ingenieur:  
Glas ist überdimensioniert

### 38.3 Prozedurale vs. Objektorientierte Sicht

Die Objektorientierte Sicht ist meist die intuitivere Sicht der Realität als die prozedurale, da physisch existierende Objekte direkt als Objekte in einem objektorientierten Design modelliert werden können.

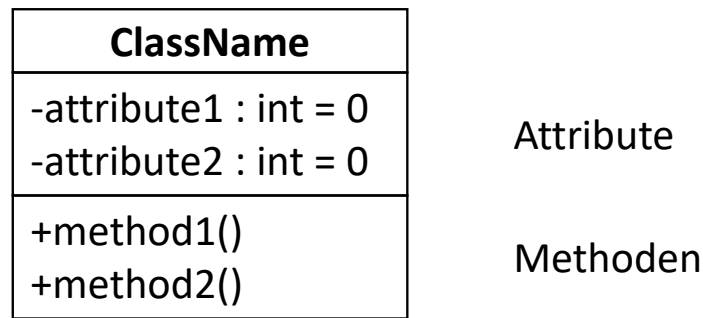
## 39 Unified Modeling Language (UML)

UML

### 39.1 Entstehung der UML

### 39.2 Was ist UML?

- UML steht für Unified Modeling Language



- UML ist eine graphische Modellierungssprache
- Ziel der UML
  - fortlaufendes (objektorientiertes) Modellierungskonzept für alle Software-Entwicklungsphasen
- UML ist heute der de facto Standard für die Softwaremodellierung
- UML ist (programmier-)sprachunabhängig
- UML unterstützt den gesamten Entwicklungsprozess
- UML integriert (fast) alle früheren Modellierungstechniken
  - Datenmodellierung
  - Prozessmodellierung
  - Zustands- und Verhaltensmodellierung
  - Steuerfluss-Modellierung
- UML ist...
  - **kein** Softwareprozess-Modell
  - **kein** Lebenszyklusmodell
  - **keine** Programmiersprache
  - nicht ohne Redundanz
    - \* es gibt oft mehrere Möglichkeiten, etwas zu modellieren
  - **kein** Softwaretool

### 39.3 Klassenbegriff

- Eine Klasse ist eine Struktur (eine Struktur besteht nur aus Daten), die mit den Funktionen, welche auf diesen Daten arbeiten, erweitert wurde.
- Eine Klasse ist also eine Struktur, welche die Daten und die Funktionen auf diesen Daten in ein syntaktisches Konstrukt packt.
- **Die Klasse ist die Umsetzung der Datenkapsel.**
- Eine Klassendeklaration ist eine Typendefinition. Die "Variablen einer Klasse werden als Objekte" bezeichnet.

### 39.4 UML-Notation der Klasse

- Eine Klasse ist der Bauplan für Objekte
- Eine Klasse besteht aus Daten (Attribute) und den Funktionen (Methoden) auf diesen Daten
- Sichtbarkeit:
  - +: public
  - : private
  - #: protected

### 39.5 Klasse definieren und Objekte anlegen: Syntax

Der Name der Klasse kann fast beliebig gewählt werden.

**Konvention:** mit Grossbuchstaben beginnen

```

class Classname // Deklaration der Klasse
{
 ...
};

Classname obj1; // Objekt definieren
Classname obj2; // Objekt definieren
Classname* objPtr; // Objekt-Pointer definieren
Classname& objRef = obj1; // Objekt-Referenz definieren

```

## 40 Zugriffsschutz bei Klassen

- Innerhalb der Klasse hat jede Methode der Klasse auf die Elemente Zugriff. (innerhalb der Klasse sind die Methoden und Attribute der Klasse "lokale Globale")
- Von ausserhalb der Klasse gibt es grundsätzlich keinen Zugriff auf Klassenelemente (default, d.h. wenn nichts steht)
- Alles, was von aussen zugreifbar sein soll, muss explizit mit `public`: gekennzeichnet werden
- Obwohl nicht unbedingt notwendig, werden die nach aussen nicht sichtbaren Elemente üblicherweise dennoch explizit mit `private`: gekennzeichnet

### 40.1 Zugriffsschutz mit `public`, `protected` und `private`

- `public`: Elemente können innerhalb und ausserhalb der Klasse angesprochen werden
  - fast alle Methoden sind `public`
  - Attribute sollen nie `public` sein
- `protected`: Elemente können von innerhalb der Klasse und von abgeleiteten Klassen angesprochen werden
  - nur sparsam einsetzen
- `private`: Elemente können nur innerhalb der Klasse angesprochen werden
  - grundsätzlich für alle Attribute und für einzelne (lokale) Methoden

#### 40.1.1 Üblicher Aufbau einer Klassenschnittstelle

```

class Classname // Deklaration der Klasse
{
 public:
 ...
 protected:
 ...
 private:
 ...
};

```

---

**Achtung:** Strichpunkt nicht vergessen!

---

### 40.2 Operationen einer Klasse

...(= Funktionen, die im Klassenrumpf definiert sind) werden als Elementfunktionen oder Methoden bezeichnet. Üblicherweise beginnen Elementfunktionen mit einem Kleinbuchstaben und werden in camelCase (mixedCase) notiert.

Beispiel:

`isEmpty()`

nicht:

`is_empty()`, `isempty()`, `IsEmpty()`, `Isempy()`, etc.

### 40.3 Information Hiding

- Klassen exportieren generell ausschliesslich Methoden.  
Alle Daten sind im Innern (private-Abschnitt) verborgen, der Zugriff erfolgt über die so genannten Elementfunktionen.
- Jede Klasse besteht damit aus zwei Dateien, der Schnittstellendatei (.h) und der Implementierungsdatei (.cpp)

## 41 Beispiel einer Klasse: Rechteck (Rectangle)

- Welche Attribute besitzt ein Rechteck?
  - Länge a
  - Breite b
- Welche Funktionen (Methoden) sollen möglich sein?
  - a und b setzen
  - a und b abfragen
  - Flächeninhalt abfragen

| Rectangle                                                                                                        |
|------------------------------------------------------------------------------------------------------------------|
| -a : double<br>-b : double                                                                                       |
| +setA(in newA : double)<br>+setB(in newA: double)<br>+getA() : double<br>+getB() : double<br>+getArea() : double |

### 41.1 Klassendeklaration

listings/rectangle.h

```
class Rectangle
{
public:
 void setA(double newA);
 void setB(double newB);
 double getA() const;
 double getB() const;
 double getArea() const;
private:
 double a;
 double b;
};
```

## 41.2 Klassendefinition

listings/rectangle.cpp

```
#include "rectangle.h"
void Rectangle::setA(double newA)
{
 a = newA;
}

void Rectangle::setB(double newB)
{
 b = newB;
}

double Rectangle::getA() const
{
 return a;
}

double Rectangle::getB() const
{
 return b;
}

double Rectangle::getArea() const
{
 return a*b;
}
```

## 41.3 Klassendefinition direkt

listings/rectangle\_direkt.h

```
class Rectangle
{
public:
 void setA(double newA) {a = newA;}
 void setB(double newB) {b = newB;}
 double getA() const {return a;}
 double getB() const {return b;}
 double getArea() const {return a*b;}
private:
 double a;
 double b;
};
```

---

**Hinweis:** Ist ok bei sehr kurzen Methoden. Verletzt Information Hiding, Methoden sind jedoch implizit inline.

---

## 41.4 Reprise: Projekt Stack

- Ausgangspunkt
  - Datenkapsel, rein prozedural
  - Daten dateiglobal, nur einfach vorhanden. Deshalb können nicht mehrere Stacks gebildet werden.
- Ziel

- ich möchte Daten und Funktionen besser zusammenfassen können
- ich will mehrere Stacks gleichzeitig instanzieren können (in der prozeduralen Variante könnte das erreicht werden, indem der jeweilige Stack immer als zusätzlicher Parameter den Funktionen übergeben wird)
- Schritt 1: Klassendeklaration erstellen (in Headerfile stack.h)

## 41.5 Klassenschnittstelle

Die Schnittstelle einer Klasse sollte minimal und vollständig sein. Vollständig in dem Sinne, dass Benutzer der Klasse alle sinnvollen Aktionen ausführen können. Minimal wiederum bedeutet, dass das Klassen-Interface so klein wie möglich sein sollte.

## 42 Elementfunktionen

- ...sind Funktionen, die in der Schnittstelle der Klasse spezifiziert sind
- Elementfunktionen haben vollen Zugriff auf alle Klassenelemente (auch auf solche, die mit `private` gekennzeichnet sind)
- Auf Elementfunktionen kann nur unter Bezugnahme auf ein Objekt der Klasse, bzw. mit dem Scope-Operator (`::`) zugegriffen werden
- Elementfunktionen sollen prinzipiell in der Implementierungsdatei (`.cpp`) implementiert werden. Dem Funktionsnamen muss dabei der Klassenname gefolgt von `::` vorgestellt werden

Beispiel:

```
int Stack::pop()
```

### 42.1 Klassifizierung von Elementfunktionen

- Konstruktoren/Destruktoren
  - Konstruktor: erzeugen eines Objekts
  - Destruktor: vernichten, freigeben eines Objekts
- Modifikatoren
  - ändern den Zustand eines Objekts (Attribute ändern)
- Selektoren
  - greifen nur lesend auf ein Objekt zu (immer `const` definieren!)
  - Beispiel:
 

```
bool Stack::isEmpty() const;
```
- Iteratoren
  - Erlauben, auf Elemente eines Objekts in einer definierten Reihenfolge zuzugreifen

### 42.2 inline-Elementfunktionen

- Elementfunktionen, die innerhalb der Deklaration der Klassenschnittstelle (im `.h`-File) implementiert sind, werden als (implizite) inline-Funktionen behandelt
- Elementfunktionen können in der Klassenimplementation explizit mit dem Schlüsselwort `inline` gekennzeichnet werden
- **Implizite inline-Funktionen verletzen das Information Hiding Prinzip und sollten deshalb puristisch betrachtet grundsätzlich vermieden werden**
- **Jedoch: die impliziten inline-Funktionen sind die Funktionen, die garantiert immer inline verwendet werden (mit einigen wenigen Ausnahmen)**

### 42.3 Projekt Stack

- Schritt 1: Klassendeklaration erstellen (in Headerfile stack.h)
- Schritt 2: Klassendefinition erstellen (in Implementationsfile stack.cpp)
- Schritt 3: Testprogramm anpassen (in stacktest.cpp)

### 42.4 const - Elementfunktionen

- Elementfunktionen, die den Zustand eines Objekts nicht ändern (Selektoren) sollen explizit mit dem Schlüsselwort `const` gekennzeichnet werden
- Das Schlüsselwort `const` muss sowohl im Prototypen als auch in der Implementierung geschrieben werden

- beispiel:

```
bool Stack::isEmpty() const;
...
bool Stack::isEmpty() const
{
 return top == 0;
}
```

Um zu verhindern, dass const-Objekte über den "Umweg" von Elementfunktionen verändert werden, dürfen "normale" Elementfunktionen nicht auf const-Objekte angewandt werden.

```
class Stack
{
public:
 int pop();
 bool isEmpty() const;
private:
 ...
};
...
void fooReadOnly(const Stack& s)
{
 bool b = s.isEmpty(); // ok. s ist const, isEmpty() ist auch const
 int i = s.pop(); // Fehler. s ist const, pop() nicht!
}
```

#### 42.4.1 const - Elementfunktionen: Schlussfolgerung

Damit mit const-Objekten überhaupt etwas gemacht werden kann, müssen die Elementfunktionen, welche die Attribute nicht verändern, konsequent mit const gekennzeichnet werden

## 42.5 Projekt Stack

- Schritt 1: Klassendeklaration erstellen (in Headerfile stack.h)
- Schritt 2: Klassendefinition erstellen (in Implementationsfile stack.cpp)
- Schritt 3: Testprogramm anpassen (in stacktest.cpp)
- Schritt 4: Selektoren mit const gekennzeichnet

## 42.6 mutable-Attribut

Ein Datenelement, das nie const werden soll (auch nicht bei const-Elementfunktionen) kann mit mutable gekennzeichnet werden.

```

class Stack
{
public:
 int pop();
 int peek() const; // peek() ist read-only, liest nur das oberste Element
 bool isEmpty() const;
private:
 int elem[maxElems]; // Array fuer Speicherung des Stacks
 int top; // Arrayindex des naechsten freien Elements
 mutable bool error; // true: Fehler passiert; false: sonst
 // mutable: auch const-Methoden koennen dieses Attribut setzen
};

int Stack::peek() const
{
 error = top == 0; // 'error' soll auch in const-Methode gesetzt werden koennen
 if (!error)
 return elem[top-1];
 else
 return elem[top];
}

```

## 42.7 Projekt Stack

- Schritt 1: Klassendeklaration erstellen (in Headerfile stack.h)
- Schritt 2: Klassendefinition erstellen (in Implementationsfile stack.cpp)
- Schritt 3: Testprogramm anpassen (in stacktest.cpp)
- Schritt 4: Selektoren mit const gekennzeichnet
- Schritt 5: mutable Attribut setzen wo nötig (zurückhaltend einsetzen)

## 43 Konstruktoren/Destruktoren

### 43.1 this-Pointer

Der this-Pointer ist ein Pointer auf das eigene aktuelle Objekt, welches eine Elementfunktion (Methode) aufgerufen hat.

```

const AnyClass& AnyClass::aMethod(const AnyClass& obj)
{
 this->anyFoo(); // Aufruf einer Methode ueber this
 // 'this' ist hier unnoetig, da Methode implizit mit aktuellem
 // Objekt ausgefuehrt wird
 if (this == &obj) // testen, ob eigene Adresse gleich der Adresse von obj ist
 ...

 return *this; // eigenes Objekt zurueckgeben
}

```

### 43.2 friend-Elemente

friend - Jede Klasse kann andere Klassen oder Funktionen zum Freunderklären. Dadurch werden die Zugriffsregeln durchbrochen.

Jeder friend darf auf **alle** Elemente der Klasse zugreifen.

friend ist eine C++-Spezialität, welche die meisten anderen Programmiersprachen (z.B. Java) nicht



anbieten.

**Achtung:** friends, insbesondere friend-Klassen, können ein Anzeichen für schlechtes Design sein. Sie durchbrechen wichtige Prinzipien der objektorientierten Programmierung. **Die Verwendung von friend sollte daher weitgehend unterbleiben.** Für ausgewählte Anwendungen kann damit jedoch sehr elegant programmiert werden (siehe später).

### 43.3 static-Klassenelemente (Kap.11.5)

- Grundsätzlich besitzt jedes Objekt einer Klasse seine eigene private Instanz aller Attribute einer Klasse.
- Wenn ein Attribut mit static gekennzeichnet wird, dann teilen sich alle Objekte dieser Klasse eine eigene Instanz dieses Attributs, d.h. ein statisches Attribut ist nur einmal für alle Objekte einer Klasse im Speicher vorhanden.
- static-Elemente befinden sich ausserhalb eines Objektkontextes
- static-Elemente können auch über den Klassennamen angesprochen werden (da sie sich im Kontext einer Klasse befinden)

#### 43.3.1 Syntax

```
class T
{
 ...
 static int nrOfObjects = 34; // Initialisierung ist ab C++03
 // in der Deklaration nicht mehr erlaubt!
 static int nrOfObjects; // Deklaration
};

static int T::nrOfObjects; // Die Initialisierung kann in der Definition (.cpp)
 // erfolgen. Das Schlüsselwort static muss hier weggelassen werden.
int T::nrOfObjects = 34; // Definition (ist notwendig)

T myT;
myT.nrOfObjects++; // Zugriff ueber Objekt (falls public)
T::nrOfObjects++; // ZUgriff ueber Klasse (falls public)
```

### 43.4 Konstruktor (Constructor, Ctor)

Aufgaben:

- die Neugründung einer Objekts einer Klasse
- das ßaubereinitialisieren des Objekts, d.h. **alle** Attribute des Objekts müssen auf einen definierten Wert gesetzt werden.
- Der Konstruktor hat in C++ denselben Namen wie die Klasse, hat keinen Rückgabetyt (auch nicht void) und kann überladen werden

```
Stack::Stack(); // (Default-) Konstruktor
```

#### 43.4.1 Aufruf

- Der Konstruktor soll nie explizit aufgerufen werden
- Der Konstruktor wird vom System automatisch (implizit) aufgerufen, wenn ein Objekt erzeugt wird

```
Stack s;
```

- Wenn durch den new-Operator Speicher angefordert **und** erhalten wird, dann wird der Konstruktor vom System ebenfalls automatisch aufgerufen

```
Stack* pS = new Stack;
```

Welcher Konstruktor wird wann aufgerufen?

- Ein Konstruktor (Ctor) wird ausschliesslich dann und nur dann aufgerufen, wenn ein neues Objekt erzeugt wird
- Wenn feststeht, dass ein Konstruktor benötigt wird, muss man sich noch überlegen, welcher der allenfalls überladenen Konstruktoren aufgerufen wird.

Default-Konstruktor

- Der Default-Konstruktor ist der Konstruktor ohne Parameter

```
Stack::Stack(); // Default-Konstruktor
```

- Er wird immer aufgerufen, wenn bei der Objekterzeugung keine Parameter mitgegeben werden.

```
Stack s;
```

- Der Default-Konstruktor kann selbst definiert werden.
  - Das ist insbesondere dann notwendig, wenn innerhalb des Objekts Speicher dynamisch alloziert werden muss (bei der Objekterzeugung).
- Der Default-Konstruktor wird vom System automatisch erzeugt, wenn für eine Klasse kein Konstruktor explizit definiert ist.

#### 43.4.2 Beispiel: Klasse TString (nach Lippman)

```
class TString
{
public:
 TString(); // Default-Konstruktor
 int getLen() const;
private:
 int len;
 char* str;
};
```

### 43.4.3 Implementation von TString::TString()

```
// mit Anweisungen
TString::TString()
{
 len = 0;
 str = 0;
}

// oder: mit Initialisierungsliste (ist besser)
TString::TString()
 : len(0), str(0)
{
}
}
```

**Hinweis:** Objektinitialisierungen werden, sofern dies möglich ist, über die Initialisierungsliste des Konstruktors und nicht im Anweisungsteil durchgeführt. (Effizienzgründe)

### 43.4.4 Überladen von Konstruktoren

- Der Default-Konstruktor wird implizit aufgerufen mit

```
TString str;
```

- Ein TString-Objekt soll auch z.B. mit folgenden Anweisungen gegründet werden können:

```
TString str1 = "Hello"; // implicit call
TString str2 = TString("Guten_Morgen"); // explicit call
```

- Dazu bedarf es anderer (überladener) Konstruktoren.

### 43.4.5 Erweiterung der Klasse TString

```
class TString
{
public:
 TString(); // Default-Konstruktor
 TString(const char* p);
 int getLen() const;
private:
 int len;
 char* str;
};
```

Implementation:

```

TString::TString(const char* p)
{
 if (p==0)
 {
 len = 0;
 str = 0;
 }
 else
 {
 len = strlen(p);
 str = new char[len+1];
 memcpy(str, p, len+1);
 }
}

```

**Hinweis:** Hier geht Initialisierungsliste nicht.

#### 43.4.6 Konstruktoren und Function Casts

- Konstruktoren mit nur einem Parameter können dazu verwendet werden, ein Objekt vom Typ T aus einem anderen Objekt zu erzeugen (Typumwandlung)
- Beispiel:  
TString soll so erweitert werden, dass dem Konstruktor eine ganze Zahl übergeben wird und dieser daraus den entsprechenden String erzeugt.

```

TString::TString(int number);

// explicit call:
TString str1 = TString(12345); // erzeugt "12345"

// implicit calls
TString str2 = 12345; // erzeugt "12345"
str2 = 789; // erzeugt temporäres Objekt "789" und weist dieses str2 zu

```

- Die implicit calls (bei Ctors mit einem Parameter)

```

TString str2 = 12345;
str2 = 789;

```

sind gelegentlich nicht erwünscht.

- Wenn der Konstruktor mit explicit gekennzeichnet wird, kann dieser Ctor nicht mehr implizit, sondern nur explizit aufgerufen werden

```

explicit TString::TString(int number);

TString str1 = TString(12345); // ok (explicit)
TString str2 = 12345; // nicht erlaubt (implicit call)
str2 = 78; // nicht erlaubt (implicit call)
str1 = 567; // nicht erlaubt (implicit call)

```

### 43.4.7 Erweiterung der Klasse TString 2

```
class TString
{
public:
 TString(); // Default-Konstruktor
 TString(const char* p);
 explicit TString(int number);
 int getLen() const;
private:
 int len;
 char* str;
};
```

### 43.4.8 Copy-Konstruktor

- Der Copy-Konstruktor wird dazu verwendet, Objekte zu kopieren
- Der Copy-Konstruktor erhält als Parameter immer eine konstante Referenz auf ein Objekt der Klasse. Für TString sieht er wie folgt aus:

```
TString(const TString& s); // Copy-Konstruktor

// weoche Konstruktoren werden aufgerufen?
TString str1("Hello_World"); // normaler Konstruktor TString(const char* p)
TString str2 = str1; // Copy-Konstruktor, ist Initialisierung, nicht
 // Zuweisung!
TString str3(str1); // Copy-Konstruktor
```

### 43.4.9 Copy-Konstruktor wird automatisch aufgerufen, wenn...

- ein Objekt erzeugt und mit einem anderen Objekt derselben Klasse initialisiert wird
- ein Objekt als Wertparameter (by value) an eine Funktion übergeben wird (nicht aber bei Referenzierungsparametern → **wichtig!**)
- ein Objekt by value als Resultat einer Funktion zurückgegeben wird (nicht bei Referenzrückgaben)

Ein Copy Ctor wird nur dann benutzt, wenn ein neues Objekt erzeugt wird, aber nicht bei Zuweisungen, also Änderungen von Objekten.

Bei Zuweisungen wird der vom System bereitgestellte Zuweisungsoperator benutzt, sofern kein eigener definiert wurde (später).

### 43.4.10 Erweiterung der Klasse TString 3

```
class TString
{
public:
 TString(); // Default-Konstruktor
 TString(const TString& s); // Copy-Konstruktor
 TString(const char* p);
 explicit TString(int number);
 int getLen() const;
private:
 int len;
 char* str;
};
```

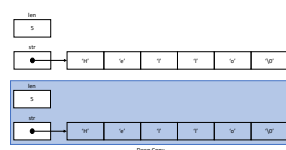
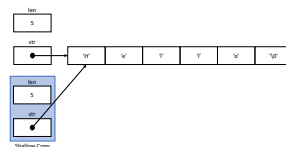
### 43.4.11 Shallow Copy vs. Deep Copy

- Wenn für eine Klasse kein Copy-Konstruktor definiert wird, erzeugt das System einen Standard-Copy-Konstruktor
- Dieser kopiert alle Datenelemente (memberwise assignment). Bei Pointern, welche auf den Heap zeigen, wird nur die Adresse kopiert, nicht aber der Speicher auf dem Heap. Man nennt das shallow copy. (shallow=flach)
- Bei einer deep copy werden auch die Speicherbereiche, auf welche Pointer zeigen, kopiert. Die deep copy muss in einem selbst definierten Copy-Konstruktor implementiert werden.

---

**Hinweis:** Wenn ein Objekt Speicher auf dem Heap alloziert, muss ein eigener Copy-Konstruktor definiert werden (in allen anderen Fällen meist nicht).

---



### 43.4.12 Copy-Konstruktor der Klasse TString

```
TString::TString(const TString& s)
: len(s.len)
{
 if (s.str == 0)
 {
 str = 0;
 }
 else
 {
 str = new char[len+1];
 memcpy(str, s.str, len+1);
 }
}
```

## 43.5 Destruktor (Destructor, Dtor)

Aufgaben:

- die vollständige Zerstörung eines nicht mehr benötigten Objekts
- das saubere Entfernen eines Objekts
- die häufigere Aufgabe ist die Freigabe von nicht mehr benötigtem Speicher auf dem Heap
- sehr häufig (wenn kein Speicher auf dem Heap vorhanden ist) wird kein Destruktor definiert, da das System dann automatisch aufräumt

### 43.5.1 Eigenschaften des Destruktors

- Destruktoren haben keine Argumente und keinen Rückgabotyp (sie können auch nicht überladen werden)
- Ihr Name besteht aus dem Klassennamen mit vorgestellter Tilde.

```
~TString(); // Destruktor
```

- Destruktoren werden automatisch aufgerufen, wenn der Gültigkeitsbereich des definierten Objekts ausläuft
- Die Reihenfolge des Aufrufs der Destruktoren ist umgekehrt wie die der Konstruktoren (das zuletzt erzeugte Objekt wird zuerst aufgeräumt)
- Nicht definierte Destruktoren werden automatisch erzeugt

### 43.5.2 Erweiterung der Klasse TString 4

```
class TString
{
public:
 TString(); // Default-Konstruktor
 TString(const TString& s); // Copy-Konstruktor
 TString(const char* p);
 explicit TString(int number);
 ~TString(); // Destruktor
 int getLen() const;
private:
 int len;
 char* str;
};
```

### 43.5.3 Implementation des Destruktors

```
TString::~TString()
{
 delete[] str; // weil str ein Array auf dem Heap ist
}
```

### 43.5.4 Schnittstelle der Klasse TString

```
class TString
{
public:
 TString(); // Default-Konstruktor
 TString(const TString& s); // Copy-Konstruktor
 TString(const char* p);
 TString(int l, char fillChar);
 explicit TString(int number);
 ~TString(); // Destruktor
 int getLen() const;
private:
 int len;
 char* str;
};
```

### 43.5.5 Demo: Klasse Stack

- Stack::init() durch ctor ersetzen
- Verhalten bei privatem Default Ctor
- Verhalten bei privatem Copy Ctor
  - Wenn der Copy Ctor privat deklariert wird (ohne ihn zu implementieren), dann verhindert der Compiler das Kopieren von Objekten dieser Klassen
  - Das kann in gewissen Fällen ein durchaus erwünschtes Verhalten sein

## 44 Handhabung von Klassen und Objekten

### 44.1 Repetition

- Ctor wird immer (und nur dann) aufgerufen, wenn ein Objekt erzeugt wird. Dabei wird der passendste Ctor genommen.
- Eine Methode (Funktion) wird immer mit einem Objekt aufgerufen, z.B. c1.print();  
Auf dieses Objekt (hier: c1) kann innerhalb der Methoden der Klasse mit dem Pointer this zugegriffen werden.
- Der Dtor wird immer dann automatisch aufgerufen, wenn ein Objekt seinen Scope verlässt, bzw. mit delete vom Heap gelöscht wird.
- Eine Klasse ist der Bauplan für Objekte dieser Klasse.



## 44.2 Welche Ctors werden bei folgenden Anweisungen aufgerufen?

```
class TString
{
public:
 TString();
 TString(const TString& s);
 TString(const char* p);
 explicit TString(int i);
 // ...
};

TString s1 = "Hoi";
TString s2;
TString s21 = s1;
s2 = s1;
TString& s3 = s21;
TString s4[2];
TString* p1 = &s21;
TString* p2 = new TString("Hallo");
TString* p3 = new TString[3];
TString s5[3] = {"Hoi", TString(), s2};
```

## 44.3 Automatisch generierte Elementfunktionen (11.7.4)

- Die folgenden Elementfunktionen werden vom Compiler automatisch erstellt, falls sie im Programm benötigt und nicht vom Programmierer explizit deklariert werden:
  - Default-Konstruktor
  - Copy-Konstruktor
  - Destruktor
  - Zuweisungsoperator
  - Adressoperator
- Automatisch generierte Elementfunktionen können als private deklariert werden (implementieren ist nicht nötig!), um die Verbindung zu unterbinden.

## 44.4 Kanonische Form von Klassen (11.7.5)

- Als kanonische Form einer Klasse bezeichnet man jene Form, die es erlaubt, eine Klasse wie einen "normalen" Datentyp zu benutzen. Dies ist für alle Klassen anzustreben.
- Dazu müssen drei Bedingungen erfüllt sein:
  - Ein korrekter Default-Konstruktor, plus evtl. weitere Konstruktoren müssen vorhanden sein
  - Wenn die Klasse dynamische Daten enthält, braucht es auch einen Zuweisungsoperator (folgt später) und einen Copy-Konstruktor
  - Ein (virtueller) Destruktor garantiert die korrekte Zerstörung von Objekten

## 44.5 Benutzerdefinierte Typumwandlungen: Problemstellung & Lösung

- Wenn zwei ganze Zahlen unterschiedlichen Typs (z.B. int und short) addiert werden, so ist der Additionsoperator *vom System* für folgende Varianten definiert:
  - int + int
  - int + short
  - short + int
  - short + short
- Dasselbe gilt auch für alle weiteren Operatoren. (In C++ können Operatoren auch selbst für eigene

Klassen definiert werden. Dies folgt später.)

- Wenn nun eine neue Klasse `VeryLargeInt` eingeführt wird, so sind die Operatoren für diese Klasse noch nicht definiert. Nur schon für den Additionsoperator zwischen `VeryLargeInt` und `int` müssten folgende Varianten definiert werden:  
`int + VeryLargeInt`  
`VeryLargeInt + int`  
`VeryLargeInt + VeryLargeInt`
- Dasselbe gilt auch für alle weiteren Operatoren. Für die Grundoperatoren `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=` müssten somit 24 Operatoren definiert werden.
- Weitere wären für `short`, `char`, `long` etc. nötig.

Lösung:

- Die einfachere Variante ist, wenn für jeden Typ eine Typumwandlung definiert wird.
- Somit braucht es pro Typ eine Umwandlungsfunktion, die Operatoren arbeiten anschliessend nur noch mit der Klasse `VeryLargeInt`.  
`VeryLargeInt + VeryLargeInt`
- Für die Grundoperatoren `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=` müssten nur noch die 8 Operatoren definiert werden.
- Zusätzlich müsste noch die Typumwandlung von jedem Typ (`short`, `int`, etc.) in `VeryLargeInt` definiert werden.

## 44.6 Typumwandlung mit Konstruktor

- Häufig werden Typumwandlungen mit Hilfe von Konstruktoren implementiert:  
`VeryLargeInt(int);`
- 

---

**Achtung:** Aufpassen bei Implicit Calls von Ctors.

---

Beispiel:

- In Embedded Systems müssen häufig Befehle als Bytestream über einen Kommunikationskanal übertragen werden. Die Befehle beinhalten meist eine Befehls-ID, eine bis mehrere Befehlsparameter, Längenangaben, etc.. Das Befehlsformat ist eindeutig definiert.
- Der Befehl könnte in einer Klasse `Command` abgebildet werden.
- Die Interpretation des Bytestreams könnte als Typumwandlung in einem Konstruktor implementiert werden:

```
Command(const uint8_t* byteStream);
```

## 44.7 Weiterführende Themen

### 44.7.1 Zeiger auf Klassenelemente

- Kann in ganz speziellen Fällen nützlich sein
- Grundsätzlich sollen Zeiger auf Klassenelemente vermieden werden (gehört nicht zum Stoff, behandeln wir nicht)
- Im Modul Embedded Software Engineering wird eine elegante Anwendung davon präsentiert

## 44.8 Unions (Varianten)

### 44.8.1 Eigenschaften einer Union

- ähnlich einer Struktur
- beinhaltet auch mehrere Felder unterschiedlichen Typs
- im Gegensatz zur Struktur ist aber nur ein einziges Feld jeweils aktiv (abhängig vom Typ)
- die Grösse einer Union ist so gross wie das grösste Feld in der Union
- Bei der Union sind dieselben Operationen wie bei einer Struktur definiert (siehe dort)

### 44.8.2 Definition von Uniontypen und Unionvariablen

Verwenden des Schlüsselworts union

Allgemeine Form:

```
union UnionName
{
 public:
 ...
 private:
 FeldTyp1 feld1;
 FeldTyp2 feld2;
 FeldTyp3 feld3;
 ...
 FeldTypN feldN;
};
```

- UnionName kann frei gewählt werden
- union UnionName ist ein hier selbst definierter Typ, der weiter verwendet werden kann
- Der Datentyp ist definiert durch den Inhalt der geschweiften Klammer
- Der Feldtyp kann wiederum eine Union oder auch eine Struktur sein

---

**Achtung:** Der Programmierer muss verfolgen, welcher Typ jeweils in der Union gespeichert ist. Der Datentyp, der entnommen wird, muss der sein, der zuletzt gespeichert wurde. Sehr zurückhaltend einsetzen!

---

### 44.8.3 Beispiel: Definition einer Union

```
union Vario
{
 private:
 int intNam;
 long longNam;
 double doubleNam;
};
```

## 44.9 Bitfelder

### 44.9.1 Eigenschaften von Bitfeldern

- innerhalb eines int können einzelne Bitgruppen definiert und angesprochen werden
- sollte nicht eingesetzt werden, um damit Speicher zu sparen
- bei Embedded Systems ist der Einsatz unter Umständen sehr nützlich, wenn auf einzelne Register zugegriffen werden soll
- **ABER:** leider definiert der C++-Standard (und auch der C-Standard) nicht, ob die Bits von Links nach Rechts oder von Rechts nach Links aufgefüllt werden. Falls der Standard dies definieren würde, wären die Bitfelder ein sehr gutes Konstrukt.

#### 44.9.2 Definition von von Bitfeldern

```
struct fieldName
{
 unsigned int a: 3; // definiert 3 Bits fuer a
 unsigned int b: 4; // definiert die naechsten 4 Bits fue b
 ...
};
```

#### 44.9.3 Bitfelder: Folgerungen

- Mit diesem Bitfeld-Mechanismus soll weder in C noch in C++ gearbeitet werden wenn der Code portabel sein soll.
- Die bessere Alternative ist die Verwendung von Bitmasken-Operationen. Diese können in C++ bspw. in inline-Funktionen verpackt werden.

#### 44.10 Demo: Klasse Date

Eine Klasse zur Speicherung und Darstellung von Kalenderdaten, wie z.B. 17.4.2013

## Teil IX

# Vererbung

### 45 Motivation

Sie müssen einen Webshop entwickeln. Sie verkaufen Bücher und CDs.

Welche Eigenschaften und Methoden benötigen Sie, um ein Buch bzw. eine CD zu charakterisieren?

### 46 Artikel als Gemeinsamkeit von Buch und CD

### 47 Grundkonzept

- Die Vererbung erlaubt, neue Klassen auf der Basis von bestehenden Klassen zu definieren. Dabei erbt (übernimmt) die neue (Unter-)Klasse alle Eigenschaften der bestehenden (Über-)Klasse.
- Man sagt auch, die Oberklasse sei eine Basisklasse, Superclass, Abstraktion oder Generalisierung (Verallgemeinerung).
- Die Unterklasse wird auch mit Subclass oder als Spezialisierung bezeichnet.

### 48 Einsatz der Vererbung

- Bestehende Klassen erweitern
  - zusätzliche Attribute erweitern
  - zusätzliche Elementfunktionen
- Bestehende Methode einer Basisklasse ändern (überschreiben)
- Das Finden von guten Basisklassen ist eine Hauptaufgabe in der Designphase.

### 49 UML-Notation

- Generalisierung/Spezialisierung
- SubClass erbt sämtliche Eigenschaften von SuperClass
- ist-ein Beziehung (SubClass **ist eine** SuperClass)
- Pfeilspitze ist ein geschlossenes Dreieck

#### 49.1 ist ein Beziehung

ist ein Beziehung (is a relationship)

Beispiel: Baum **ist eine** Pflanze, Blume **ist eine** Pflanze.

### 50 Beispiel: Vererbungshierarchie Lebewesen

#### 50.1 C++-Syntax

```
class SubClass : public SuperClass
{
 public:

 protected:

 private:

};
```

public ist Normalfall (private und protected sind auch möglich).

## 50.2 Zugriff auf Elemente der Basisklasse

## 51 Spezifikation von Basisklassen

- Grundsatz: Vererbung sollte immer public sein (zu 99,99%)
- Falls bei Vererbung protected oder private in Betracht gezogen wird, kann der Grund dafür eine falsche Verwendung der Vererbung sein
- Für ganz spezifische Anwendungen kann die Vererbung mit protected oder private sinnvoll sein

## 52 Beispiel: ComicCharacter (Comics01)

## 53 Einsatz von protected bei Klassenelementen

- Bei Datenelementen (Attributen) soll protected grundsätzlich nicht eingesetzt werden. Attribute sollen generell private sein.
- Bei Elementfunktionen kann es in Einzelfällen sinnvoll sein, diese als protected zu definieren. Dadurch wird der Zugriff gegenüber einer public-Sichtbarkeit auf die abgeleiteten Klassen beschränkt.

## 54 Objektgrösse bei der Vererbung

- Ein Objekt einer vererbten Klasse enthält alle Teile der Basisklasse(n) und zusätzlich noch die spezifischen eigenen Teile.
- Das Objekt ist somit mindestens so gross wie jenes der Basisklasse(n). (es gibt keine Vererbung "by reference")
- Wenn Vererbung schlecht eingesetzt wird (z.B. keine is-a-Beziehung), können unnötig grosse Objekte entstehen.
- Bei einer Aggregationsbeziehung kann durchaus eine Referenz (oder ein Pointer) auf ein anderes Objekt verwendet werden, d.h. Aggregation "by reference" ist möglich.

## 55 Schlechter (falscher) Einsatz von Vererbung

Zwischen Wald und Pflanze besteht nicht eine ist-ein-Beziehung. Die richtige Beziehung wäre "hat-ein", da ein Wald mehrere Pflanzen hat (folgt später).

## 56 Substitutionsprinzip

- Ein Objekt einer Oberklasse kann Objekte einer beliebigen Unterklasse aufnehmen.
- Ein Objekt einer Unterklasse kann keine Objekte der Oberklasse aufnehmen.

```
class SuperClass {};

class SubClass : public SuperClass {};

SuperClass super;
SubClass sub;
super = sub; // ok@"sub = super; *@"// geht nicht
```

## Teil X

# Polymorphismus

**Auftrag:** Stellen Sie bei allen Uhren die Zeit eine Stunde vor.

- Obwohl der Auftrag völlig klar ist, hat man Mühe ihn auszuführen, weil jede Uhr auf eine andere Art verstellt wird.
- Der Auftrag ist recht abstrakt, die Ausführung ist aber sehr konkret, sobald sie wissen, welche Uhr sie verstellen müssen

## 57 Static vs. Dynamic Binding

- Static Binding (early binding, statische Bindung)
  - bereits zur Compilezeit wird festgelegt, welcher (Elementfunktions-) Code ausgeführt wird (Normalfall)
- Dynamic Binding (late binding, dynamische Bindung)
  - erst zur Laufzeit wird in Abhängigkeit des Objekts festgelegt, welcher (Elementfunktions-) Code ausgeführt wird
  - Analogie: sobald sie wissen, welche Uhr sie verstellen müssen, können sie das konkrete Verfahren anwenden
  - das ist das Konzept des Polymorphismus

### 57.1 Dynamic Binding

- ist der mächtigste OO-Mechanismus (oft präziser mit run-time polymorphism bezeichnet)
- Elementfunktionen, die dynamisch gebunden werden, muss bei der Deklaration das Schlüsselwort **virtual** vorangestellt werden (zwingend!)
  - in der abgeleiteten Klasse soll (muss aber nicht) die Funktion auch mit virtual gekennzeichnet werden
- Polymorphismus wird häufig als ineffizient bezeichnet, oft jedoch zu unrecht (mehr darüber im Vertiefungsmodul Embedded Software Engineering)
- Regeln:
  - Eine Funktion soll dann als **virtual** deklariert werden, wenn sie in der abgeleiteten Klasse neu definiert (überschrieben) wird, sonst nicht! In diesem Fall muss auch der Destruktor **virtual** sein.
  - Der Destruktor muss auch dann **virtual** sein, wenn ein Objekt einer Unterklasse dynamisch erzeugt wird und einem Pointer auf die Basisklasse zugewiesen wird (Substitutionsprinzip).
- Achtung: nicht mit Funktionsüberladung (gleicher Name aber unterschiedliche Signatur) verwechseln!

#### 57.1.1 Beispiel: Zeichnen von geometrischen Figuren

- Eine immer gleich heissende Elementfunktion draw() hat unterschiedliche Implementationen, je nach Art des aktuellen Objekts.
- Bsp.: Eine Zeichenfunktion für geometrische Objekte hat ganz unterschiedliche Implementationen, je nach dem ob es sich um einen Kreis, ein Rechteck, Textblock, Polygonzug, etc. handelt.

### 57.2 Statischer vs. dynamischer Datentyp

```
class Article;
class Book : public Article {};
Article* pa;
pa = new Book;
```

- Der statische Datentyp bezeichnet den Datentyp bei der Deklaration → pa ist ein pointer auf Article
- Der dynamische Datentyp bezeichnet den effektiven Datentyp zur Laufzeit → pa ist ein Pointer auf Book

## 57.3 Aufruf von virtuellen Elementfunktionen

- Virtuelle Elementfunktionen können sowohl dynamisch (über Pointer oder Referenzen) als auch statisch aufgerufen werden.
- Bei statischen Aufrufen erfolgt die Auswahl der richtigen Funktion bereits bei der Übersetzung (d.h. ohne Overhead)
  - Dies ist dann der Fall, wenn das Objekt bereits zur Entwicklungszeit bekannt ist.
- Bei dynamischen Aufrufen erfolgt die Auswahl der richtigen Funktion zur Laufzeit aufgrund des tatsächlichen (dynamischen) Typs des Objekts. Dies ist mit einem Overhead verbunden.

### 57.3.1 Statischer Aufruf von virtuellen Elementfunktionen

```
Duck donald;
SuperHero luckyLuke;

donald.print(); // Duck::print()
luckyLuke.print(); // SuperHero::print()
```

### 57.3.2 Dynamischer Aufruf von virtuellen Elementfunktionen

```
void printCC1(const ComicCharacter& c)
{
 c.print(); // dynamische Auflöschung
}

void printCC2(const ComicCharacter* pc)
{
 pc->print(); // dynamische Auflöschung
}

Duck donald;
SuperHero luckyLuke;
printCC1(donald); // Duck::print()
printCC2(&luckyLuke); // SuperHero::print()
```

## 57.4 Polymorphe Klassen (Virtuelle Klassen)

- Eine Klasse, welche mindestens eine virtuelle Funktion deklariert, heisst virtuell (polymorph)
- Virtuelle Klassen bewirken einen Mehraufwand für den Compiler und sind darum langsamer in der Ausführung
- Funktionen sollten nur dann als virtuell deklariert werden, wenn sie in einer abgeleiteten Klasse überschrieben werden (sollen)
- Konstruktoren sind nie virtuell
- Destruktoren virtueller Klassen müssen immer als virtuell deklariert werden, sonst wird nur der Destruktor der Basisklasse aufgerufen
- Destruktoren müssen auch dann virtuell sein, wenn ein Heapobjekt über einen Pointer der Basisklasse freigegeben wird
- Nicht virtuelle Methoden dürfen nicht überschrieben werden



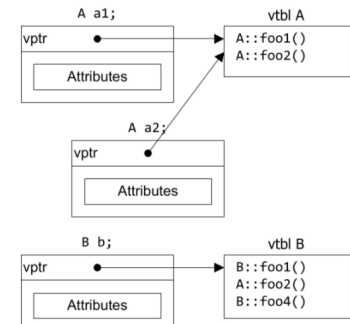
### 57.4.1 Repräsentation virtueller Objekte im Speicher

- In der Virtual Function Table (vtbl) vermerkt das System der Reihe nach die Adressen der für eine Klasse gültigen virtuellen Elementfunktionen
- Das System legt für jede polymorphe Klasse eine vtbl an
- Jedes Objekt einer polymorphen Klasse enthält einen Virtual Table Pointer (vptr), welcher auf die vtbl der entsprechenden Klasse zeigt

Beispiel für vtbl

```
class A
{
public:
 virtual void foo1();
 virtual void foo2();
 void foo3();
};

class B : public A
{
public:
 virtual void foo1();
 virtual void foo4();
};
```



## 58 Abstrakte Klassen

- Für manche Situationen sind die Vererbungsmechanismen, die wir bisher kennen gelernt haben, nicht ausreichend.
- Ein Kreis ist z.B. ein Spezialfall einer Ellipse. Es ist aber nicht sinnvoll, ihn so zu programmieren, da er sonst Eigenschaften erbt, die nicht verwendet werden.
- Es wäre möglich, Kreis und Ellipse als zwei unabhängige Klassen zu programmieren. Dann müssten aber alle Eigenschaften, die diese gemeinsam haben, doppelt programmiert werden.
- Dies versucht die objektorientierte Programmierung zu vermeiden.
- Es ist besser, die Eigenschaften, die Kreise und Ellipsen gemein haben, in einer Basisklasse zu programmieren.
- Die Kreis- und Ellipsenklassen erben dann parallel von der gemeinsamen Basisklasse.
- Die Basisklasse ist aber unvollständig, es handelt sich um eine abstrakte Klasse.
- Es können keine Objekte von abstrakten Klassen gebildet werden.
- In C++ können rein virtuelle Funktionen (pure virtual functions) deklariert werden, die in der Basisklasse nicht von einer Definition begleitet werden.
- Klassen, die mindestens eine rein virtuelle Funktion deklarieren, sind abstrakte Klassen
- Ist eine Klasse erst einmal als abstrakt definiert, kann diese nur durch Vererbung vervollständigt und dadurch nutzbar gemacht werden.

### 58.1 Anwendungen von abstrakten Klassen (Beispiele)

- Definition von Schnittstellen (Interfaces)

**Abstrakte Kommunikationsklasse** SerialCom, USB, Ethernet, etc. können davon sein

**Abstrakte Printerklasse** Jeder Printertreiber erbt davon, bzw. implementiert dieses Interface. Die Hauptapplikation (z.B. Word) muss nicht geändert werden, wenn ein neuer Printertreiber implementiert wird.

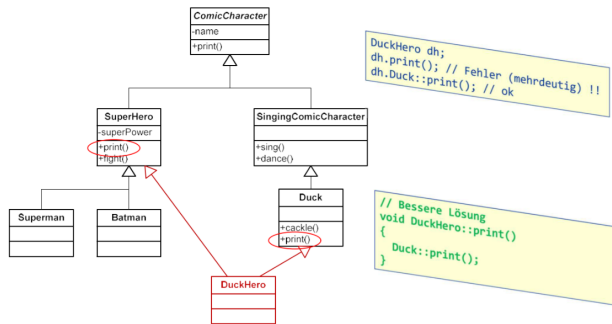
- "Real nicht existierende Abstrahierungen"
  - ...von denen es keinen Sinn macht, ein Objekt zu erzeugen.
  - Geometrische Figur als Gemeinsamkeit von Kreis, Rechteck, etc.

## 59 Mehrfachvererbung (Multiple Inheritance, MI)

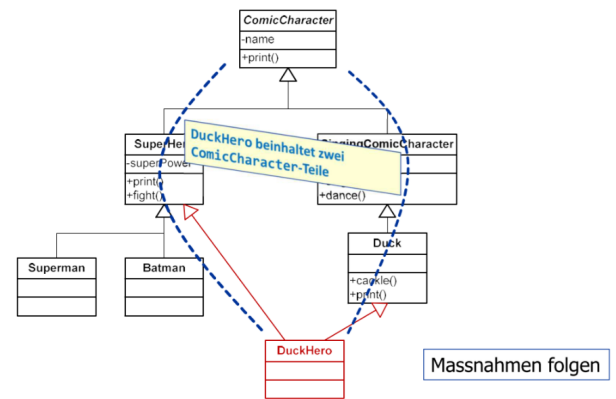
- Manchmal ist es sinnvoll, eine abgeleitete Klassen von mehreren verschiedenen Basisklassen erben zu lassen.
- Wir sprechen dann von Mehrfachvererbung.
- Bei der Mehrfachvererbung werden die Basisklassen durch Komma getrennt:
 

```
class SingingWaiter : public Waiter, public Singer
```
- Bei den Konstruktoren (Chaining) müssen die Konstruktoren der Basisklassen in der Ordnung gelistet werden, in der sie aufgerufen werden sollen.

## Problem #1: Mehrdeutigkeit



## Problem #2: Mehrfache Daten



- Mehrfachvererbung ist ein sehr mächtiges Konzept, das (richtig eingesetzt) sehr nutzbringend sein kann. Bei schlechtem Einsatz der Mehrfachvererbung können enorme Probleme eingehandelt werden.
- Ein "guter Einsatz der Mehrfachvererbung ist, wenn alle ausser höchstens einer Basisklasse ausschliesslich aus rein virtuellen Funktionen bestehen (Interfaces). Die neue Klasse implementiert dann die aufgelisteten Interfaces.
- Das ist die Art "Mehrfachvererbung", die auch in Java vorhanden ist: Vererbung gibt es nur einfach mit extends plus beliebige Implementierungen von einem bis mehreren Interfaces mit implements

### 59.1 Virtuelle Basisklassen

Um zu vermeiden, dass bei Mehrfachvererbung eine gemeinsame Basisklasse mehrfach im Speicher vorkommt, kann virtuell geerbt werden.

```
class SingingComicCharacter : virtual public ComicCharacter
{
 ...
};

class SuperHero : virtual public ComicCharacter
{
 ...
};
```

In der Klasse DuckHero muss als erstes der ComicCharacter-Konstruktor aufgerufen werden.

## 60 Laufzeit-Typinformationen (Run-Time Type Information, RTTI)

- Wie kann der (dynamische) Type eines polymorphen Objekts ermittelt werden?
- RTTI heisst der Mechanismus und **steht ausschliesslich für polymorphe Klassen zur Verfügung**
- Operator `dynamic_cast`
- Operator `typeid`
- Struktur `type_info`

**Achtung:** RTTI sollte nur äusserst zurückhaltend eingesetzt werden!

- Einsatz z.B. bei persistenten Objekten (um z.B. auch Klasseninformationen in einer relationalen Datenbank abzuspeichern)

### 60.1 Operator `dynamic_cast`

- Syntax: `dynamic_cast<SuperHero*>(p)`
- Versucht, den Zeiger `p` in eine Zeiger auf ein Objekt des Typs `SuperHero` umzuwandeln.
- Der dynamische Datentyp von `p` ist massgebend.

- Umwandlung wird dann durchgeführt, wenn p tatsächlich auf ein Objekt vom Typ SuperHero, bzw. auf eine davon abgeleitete Klasse zeigt.
  - Andernfalls ist das Resultat der Umwandlung der Nullpointer!

## 60.2 Operator typeid

- Ermitteln des dynamischen Datentyps eines polymorphen Objekts
- Ergibt eine Referenz auf ein Objekt des Typs type\_info. Diese Klasse beinhaltet u.a. eine Methode name(), welche den Namen der Klasse zurückgibt.
- Beispiel:  
`cout << "p ist ein " << typeid(*p).name() << "Objekt";`

## Teil XI

# Überladen von Operatoren (Operator overloading)

Überladen ist hier nicht im Sinne von *zu viel laden* zu verstehen, sondern im Sinne von "drüber laden", "bedecken", "neu definieren".

Mit Operator overloading kann ein bestehender Operator neu definiert (überladen) werden.

## 61 Operator overloading in C++

- Spezifität von C++ (wurde auch in C# aufgenommen, nicht aber in Java)
- Operatoren (z.B. +, ==, etc.) können wie Funktionen überladen werden, z.B.
  - Der Operator + kann bei einer String-Klasse sehr praktisch sein, um zwei Strings aneinanderzufügen.
  - Der Operator + kann definiert werden, um zwei Matrizen oder zwei komplexe Zahlen zu addieren.
  - Der Operator « kann für eine neue Klasse definiert werden, um auch Objekte dieser Klasse auf cout schreiben zu können. Damit können neue Klassen nahtlos in das bestehende C++-System integriert werden.
- Operator overloading ist zwar nicht sehr objektorientiert aber enorm praktisch.

### Überladbare (overloadable) Operatorfunktionen in C++

| new | delete | new[] | delete[] |     |
|-----|--------|-------|----------|-----|
| +   | -      | *     | /        | %   |
| ^   | &      |       |          | !   |
| =   | <      | >     | +=       | -=  |
| *=  | /=     | %=    | ^=       | &=  |
| =   | «      | »     | »=       | «=  |
| ==  | !=     | <=    | >=       | &&  |
|     | ++     | -     | ,        | ->* |
| ->  | ()     | []    |          |     |

### 61.1 Operatorfunktion

Die erwähnten Operatoren sind in C++ mit einer Operatorfunktion implementiert, d.h. im Hintergrund wird eigentlich eine Funktion aufgerufen.

Die Operatoren können sowohl in der üblichen Operatornotation als auch in der Funktionsnotation verwendet werden. **Beispiel: ein String zu einem bestehenden String addieren**

Operatornotation (üblich): `str3+= str2;`

Funktionsnotation (alternativ, unüblich): `str3.operator +=(str2);`

### 61.2 Randbedingungen zu Operator overloading

- Die Anzahl der Operanden (Argumente) muss gleich sein wie beim ursprünglichen Operator
- Die Priorität des überladenen Operators kann nicht ändern
- Neue Operatoren können nicht eingeführt werden
- Default-Argumente sind bei Operatoren nicht möglich
- nicht gefordert aber sinnvoll: die neue Operation sollte intuitiv dem bestehenden Operator zugeordnet werden können

## 61.3 Umsetzungsvarianten für Operator overloading

- Operator overloading als Elementfunktion
  - Zwingend für folgende Operatoren
    - \* Zuweisung =
    - \* Index []
    - \* Funktionsaufruf ()
    - \* Zeigeroperator ->
- Operator overloading als "normale" Funktion
  - Typischerweise für alle anderen

### Operator overloading als Elementfunktion

- Der neu definierte Operator wird als Elementfunktion implementiert. Damit ist auch der Zugriff auf private und protected Attribute der Klasse möglich.

```
class TString
{
 ...
 bool operator <(const TString& s) const;
 ...
};

bool TString::operator <(const TString& s) const
{
 ...
}
```

Nur 1 Argument, das zweite ist das aktuelle Objekt (\*this)

### Operator overloading als "normale" Funktion

Viele Operatorfunktionen werden als normale Funktion implementiert

```
class TString
{
 ...
 friend bool operator <(const TString& s1,
 const TString& s2);
 ...
};

bool operator <(const TString& s1, const TString& s2)
{
 ...
}
```

2 Argumente, da der Operator ausserhalb einer Klasse definiert ist, d.h. \*this gibt es nicht

## Unterschiedliche Implementationen

```
// Elementfunktion der Klasse TString
bool TString::operator <(const TString& s) const
{
 if (s.len == 0)
 return false;
 else
 return (strcmp(str, s.str) < 0);
}
```

```
// "normale Funktion"
bool operator <(const TString& s1, const TString& s2)
{
 if (s2.len == 0)
 return false;
 else
 return (strcmp(s1.str, s2.str) < 0);
}
```

### 61.3.1 Beispiel

#### Verwendung von überladenen Operatoren

```
TString s1 = "Hallo";
TString s2 = "Hello";
if (s1 < s2) // s1 < s2 entspricht s1.operator <(s2)
{ ... }
```

```
if (s2 < "hoo") // entspricht s2.operator <("hoo")
{ ... }
```

"hoo" wird dabei implizit in ein TString umgewandelt

```
if (TString("goo") < s2)
{ ... }
```

```
if ("goo" < s2) // entspricht "goo".operator <(s2)
{ ... }
```

Geht nicht, da der Typ von "goo" ein char\* ist. Der Typ des linken Operanden muss zwingend ein TString sein.

Problemlösung:

- Ich möchte den Operator unbedingt auch so verwenden können:

```
if ("goo" < s2)
{ ... }
```

- Da der linke Operand zwingend ein Objekt der Klasse sein muss, geht das so nicht.
- Lösung:
  - Die Operatorfunktion wird nicht als Elementfunktion der Klasse definiert sondern als "normale" Funktion ausserhalb der Klasse.
  - Dadurch besteht jedoch kein Zugriff mehr auf die private und protected Elemente der Klasse
  - Die Operatorfunktion muss deshalb als friend deklariert werden (dies ist auch praktisch die einzige "erlaubte" Verwendung von friend)

#### Operatoren und Typumwandlungen

Um zu vermeiden, dass für einen Operator unzählige Varianten mit Typen definiert werden müssen, kann auf benutzerdefinierte Typumwandlungen zurückgegriffen werden.

```
// die folgenden Deklarationen können ersetzt werden
bool TString::operator <(const TString& s) const;
bool TString::operator <(const char* p) const;
friend bool operator <(const char* p, const TString& s);
```

```
// diese Deklarationen genügen
TString(const char* p); // Typumwandlungs-Ctor
friend bool operator <(const TString& s1, const TString& s2);
```

### 61.3.2 Zuweisungsoperator =

```
TString& TString::operator=(const TString& s)
{
 if (this == &s)
 return *this; // "a1 = a1"

 delete[] str; // alte Inhalte loeschen
 str = 0;

 len = s.len; // neu allokieren und kopieren
 if (s.str != 0)
 {
 str = new char[len+1];
 memcpy(str, s.str, len+1);
 }
 return *this;
}
```

### 61.3.3 Indexoperator []

- Wir möchten auf einen einzelnen Buchstaben eines TString-Objekts mittels Indexoperator zugreifen können:

```
TString s = "Hallo";
s[2] = 'h';
char ch = s[3];
```

- Dies kann durch die explizite Implementation des Indexoperators in der Klasse TString erreicht werden
- Die Operatorfunktion soll eine Referenz auf das gewünschte Zeichen zurückgeben, d.h. ein char&  
**char& operator[](int pos);**
- Wieso braucht es zwei Varianten?  
char& operator[](int pos);  
const char& operator[](int pos) const;

```
char& operator[](int pos);
TString s = "Hallo";
s[2] = 'h';
char ch = s[3];

void foo(const TString& s)
{
 char ch = s[4];
}
```

**foo-Funktion geht mit erster Variante nicht!**

Mit einem const-Objekt können nur const-Methoden aufgerufen werden. Deshalb braucht es auch die zweite Variante.

### 61.3.4 Beispiel Klasse TString

- Indexoperator []
- Operator ()
- Zuweisungsoperator =  
(Achtung: shallow vs. deep copy)
- Typumwandlungsoperator nach char\*

- Vergleichsoperatoren == != < >
- Verkettungsoperatoren + +=

### 61.3.5 Eigenen Zuweisungsoperator definieren

Wie beim Copy Konstruktor gilt auch für den Zuweisungsoperator:

Wenn die Klasse Elemente auf dem Heap unterhält, dann muss der Zuweisungsoperator selbst definiert werden, damit eine Deep Copy erstellt wird.

### 61.3.6 Zur Erinnerung: Kanonische Form einer Klasse

- Wenn eine Klasse in der kanonischen Form vorliegt, dann kann sie verwendet werden wie ein "normaler Typ"
- Um das zu erreichen, müssen bei Klassen, welche Speicher auf dem Heap unterhalten, die folgenden Elemente definiert werden:
  - Copy Konstruktor
  - Destruktor
  - Zuweisungsoperator

## 61.4 Streamkonzept

(siehe 12.1)

### 61.4.1 Ausgabe: Klasse ostream

- (siehe 12.3)
- Nutzung mit cout (vordefiniertes Objekt der Klasse ostream):
 

```
int i = 45;
cout << "Hallo << i << endl;
TString s;
cout << s; //funktioniert nicht, wenn Operator nicht überladen
```

### 61.4.2 Operator << überschreiben

```
class TString
{
public:
 ...
 friend ostream& operator <<(ostream& os, const TString& s);
 ...
};

ostream& operator <<(ostream& os, const TString& s)
{
 if (s.str != 0)
 return os << s.str; // s.str ist char*
 else
 return os
}
```

### 61.4.3 Eingabe: Klasse istream

- (siehe 12.4)
- Nutzung mit cin (vordefiniertes Objekt der Klasse istream):
 

```
double d;
string str;
cin >> d >> str;
```



TString s;

`cint » s;` // funktioniert nicht, wenn Operator nicht überladen

#### 61.4.4 Operator » überschreiben

```
class TString
{
public:
 ...
 friend istream& operator >>(istream& is, const TString& s);
 ...
};

char buffer[stringSize]; // statisch allozierter Eingabebuffer
istream& operator >>(istream &is, TString &s)
{
 is >> buffer; // in statische Buffer lesen
 s = buffer; // dann nach s kopieren
 return is;
}
```

## Teil XII

# Templates

## 62 Generische Programmierung mit Templates (Schablonen)

### 62.1 Motivation für Templates

- Viele Algorithmen sollten für unterschiedliche Datentypen implementiert werden
- Die Algorithmen unterscheiden sich kaum oder gar nicht von Typ zu Typ
- Beispiele:
  - In einem Stack sollten verschiedene Typen abgespeichert werden können (int, double, char,...)
  - Wenn für einen bestimmten Typ eine Ordnungsrelation (<) definiert ist, dann kann die Sortierung von solchen unterschiedlichen Typen mit einem einzigen Algorithmus implementiert werden

### 62.2 Lösung mit bekannten Techniken

- Für jeden Typ müsste eine eigene Version implementiert werden
  - Dies ergibt eine grosse Anzahl von fast identischem Code
  - Alle Implementationen müssen getestet werden
  - Bei grundlegenden Korrekturen müssen alle Implementationen korrigiert werden
- Eine (durchaus mögliche) "generische"C-Variante mit void\* hat keine Typprüfung

### 62.3 Generische Programmierung mit Templates

- Single Source-Prinzip: ein bestimmtes Problem (Algorithmus) wird nur einmal unabhängig vom Typ gelöst
- Template-Definitionen erzeugen grundsätzlich noch keine einzige Zeile Code. Erst wenn das Template wirklich verwendet wird, wird Code für diesen Typ erzeugt.
- Templates sind somit für Bibliotheken ideal geeignet
- Traditionelle Bibliotheken belegen Speicher unabhängig davon, ob eine einzelne Funktion wirklich verwendet wird. Dies kann zu Dead Code führen, d.h. zu Code, der niemals ausgeführt wird.

## 63 Funktions-Templates

- Templates verwenden den Typ als "Variable"
- Die Algorithmen können unabhängig vom Typ (generisch) implementiert werden

---

**Achtung:** Templates sind keine Funktionsdefinitionen, sie beschreiben dem Compiler nur, wie er den Code definieren soll, d.h. der Compiler nimmt den konkret verwendeten Typ, setzt diesen in das Template ein und compiliert den so erhaltenen Code.

---

- Die Bindung zum konkreten Typ geschieht bereits zur Compiletime (early binding), sobald erkannt ist, mit welchem Typ das Template aufgerufen (benutzt) wird.

### 63.1 Syntax für Funktions-Templates

- Vor den Funktionsnamen wird das Schlüsselwort template, gefolgt von einer in spitzen Klammern eingeschlossenen Parameterliste gestellt.
- Die Parameterliste enthält eine (nicht leere) Liste von Typ- und Klassenparametern, die mit dem Schlüsselwort class oder typename beginnen. (typename ist zu bevorzugen). Die einzelnen Parameter werden mit Komma getrennt.

```

template<typename A, typename B>
int foo(A a, B b, int i);

template<typename Any> // or class Any
void swapIt(Any& a, Any& b);

template<typename ElemType>
ElemType minimum(ElemType elemField[], int fieldSize);

```

## 63.2 Beispiel (aus Prata): Zwei Werte vertauschen

./listings/templateSwap.cpp

```

#include <iostream>
using namespace std;

```

## 63.3 inline bei Templates

```

template<typename Any>
inline void swapIt(Any& a, Any& b)
{
 Any temp; // temp a variable of type Any
 temp = a;
 a = b;
 b = temp;
}

```

inline muss zwischen template und dem Returntyp stehen.

---

**Achtung:** Bei Verwendung von inline speziell zusammen mit Templates besteht die Gefahr von Code Bloat. Nur bei sehr kurzen Funktionen verwenden.

---

## 63.4 Beispiel: kleinstes Element finden

```

template<typename ElemType>
ElemType minimum(const ElemType elemField[], int size);

template<typename ElemType>
ElemType minimum(const ElemType elemField[], int size)
{
 int min = 0; // Index des kleinsten Elementes
 for(int i=1; i<size; ++i)
 {
 if(elemField[i] < elemField[min])
 min = i;
 }
 return elemField[min];
}

```

**Achtung:** Der Operator < muss für jeden verwendeten Typ definiert sein.

### 63.5 Ausprägung von Funktions-Templates

- Sobald ein Typ in einem Funktions-Template verwendet wird, erkennt der Compiler, dass es sich um ein Template handelt und prägt es für diesen Typ aus (implizite Ausprägung).

```
int iF[] = {1, 54, 34, 23, 67, 4};
int i = minimum(iF, sizeof(iF)/sizeof(iF[0]));
```

- Für die Auflösung werden nur die Funktionsparameter betrachtet, der Rückgabetyt wird nicht ausgewertet.

### 63.6 Explizite Qualifizierung von Funktions-Templates

- Funktions-Templates können explizit mit einem Typ qualifiziert werden

```
int iF[] = {1, 54, 34, 23, 67, 4};
int i = minimum<int>(iF, sizeof(iF)/sizeof(iF[0]));
```

- Mögliche Anwendungen: siehe Strasser s.233

### 63.7 Überladen von Funktions-Templates

- Funktions-Templates können mit anderen Funktionstemplates und auch mit "normalen" Funktionen überladen werden

---

**Hinweis:** überladen = gleicher Funktionsname, unterschiedliche Parameterliste

---

- Namensauflösung
  - Compiler geht die Liste der möglicherweise passenden Funktions-Templates durch und erzeugt die entsprechenden Template-Funktionen
  - Ergebnis ist eine Reihe von (eventuell) passenden Template-Funktionen, ergänzt durch die vorhandenen "normalen" Funktionen
  - Aus dieser ganzen Auswahl wird die am besten passende Funktion ausgewählt

## 64 Klassen-Templates

### 64.1 Definition: Klassen-Template

- Klassen-Templates sind mit Typen oder Konstanten parametrisierbare Klassen
- Im Gegensatz zu Funktions-Templates können in Klassen-Templates auch die Attribute der Klassen mit variablen Typen ausgestattet sein
- Ein Klassen-Template kann auch von Ausdrücken abhängig sein. Diese Ausdrücke müssen aber zur Compiletime aufgelöst werden können
  - Diese Möglichkeit kann gerade in Embedded Systems sehr nützlich sein

### 64.2 Syntax für Klassen-Templates

- Die Syntax ist analog zu den Funktions-Templates.
- Vor die Klassendeklaration wird das Schlüsselwort `template`, gefolgt von einer in spitzen Klammern eingeschlossenen Parameterliste gestellt.
- Die Parameterliste enthält eine (nicht leere) Liste von Typ- und Klassenparametern, die mit dem Schlüsselwort `class` oder `typename` (`typename` bevorzugen) beginnen oder auch von Ausdrücken.
- Die einzelnen Parameter werden mit Komma getrennt.

### 64.2.1 Beispiel zu Klassen-Template: Deklaration

```
template<typename ElemType, int size=100>
class Stack
{
public:
 Stack();
 ~Stack();
 void push(const ElemType& elem);
 ElemType pop();
 bool wasError() const;
 bool isEmpty() const;
private:
 ElemType elems[size];
 int top;
 bool isError;
};
```

### 64.2.2 Beispiel zu Klassen-Template: Definition

```
template<typename ElemType, int size>
void Stack<ElemType, size>::push(const ElemType& elem)
{
 // implementation goes here
}
```

Die Syntax scheint bei der Definition etwas umständlich...

### 64.2.3 Beispiel zu Klassen-Template: Nutzung (Ausprägung)

```
Stack<int, 10> s1; // s1 ist ein Stack mit 10 int
Stack<int> s2; // s2 ist ein Stack mit 100 int (Default)
Stack<double> s3; // s3 ist ein Stack mit 100 double
```

...Die Syntax ist für die Nutzung genial einfach.

## 64.3 Bemerkungen

- Mit den Templates haben sie die Möglichkeit, auf Sourcecode-Ebene Elemente mit variablem Typ und variabler Grösse zu definieren, z.B.

```
Stack<int, 10> s1;
```

- Die variable Stackgrösse 10 wird dabei nicht durch eine dynamische Allokierung auf dem Heap erreicht, sondern durch Einsetzen dieses Wertes im Template (Templateparameter size).
- Dadurch wird ein Array der Grösse 10 **vom Compiler** erzeugt. Dies ist sehr effizient.

## 64.4 Explizite Ausprägung von Klassen-Templates

Klassen-Templates können analog zu den Funktions-Templates explizit mit einem Typ qualifiziert werden

```
template<typename KeyType, typename ElemType>
class Map
{
 ...
}

class Map<int, TString>;
// Der Compiler praegt nun diese Kombination explizit aus
```

## 64.5 Klassen-Templates und getrennte Übersetzung: export

---

**Achtung:** Ab dem Standard C++11 wird export nicht mehr unterstützt!

---

- Falls ein Template auch in anderen Übersetzungseinheiten benutzt wird, muss dieses Template bei der Deklaration mit dem Schlüsselwort export gekennzeichnet werden

```
export template<class ElemType, int size>
class Stack
{
 ...
};
```

## 64.6 Klassen-Templates und getrennte Übersetzung

- Für diese Aufgabe muss eine etwas unschöne Methode gewählt werden.
- Ein Template besteht wie andere Funktionen/Klassen auch aus einer Deklaration und einer Definition. Beim Template entsteht aus der Definition aber nicht direkt Objektcode, sondern sie ist nur eine Codeschablone.
- Deshalb muss jede Clientdatei des Templates sowohl die Deklaration als auch die Definition inkludieren. Der Compiler muss die Definition des Templates zwingend kennen.
- In der nachfolgend gezeigten Variante muss weiterhin nur die Headerdatei durch die Clients inkludiert werden.

### 64.6.1 File-Organisation #1 bei Klassen-Templates

---

**Achtung:** Mögliche Probleme: stack.cpp darf nicht "normal"compiliert werden. Eine IDE wie Eclipse nimmt allenfalls alle \*.cpp-Files in ihren Compile-Prozess. stack.cpp muss davon ausgeschlossen werden.

---

### 64.6.2 Ausschliessen eines cpp-Files von Compilierung in Eclipse

### 64.6.3 File-Organisation #2 bei Klassen-Templates

---

**Hinweis:** Sowohl die Deklaration als auch die Definition des Templates sind im File stack.h vorhanden. Dadurch gibt es keine Probleme mit IDEs.  
Nachteil: Deklaration und Definition sind nicht getrennt.

---

## 64.7 Fazit

Sowohl Variante #1 als auch #2 haben Vor- und Nachteile.  
Wählen sie eine aus! (Ich bevorzuge #2)

## Teil XIII

# Exceptions („Ausnahmen“)

## 65 Exception vs. Error

Bitte unterscheiden Sie zwischen Exception (Ausnahme) und Error (Fehler)

- **Error (Fehler):** Abweichung zur Spezifikation ("falsch implementiert"). Errors sollten bei der Verifikation (Testen) entdeckt werden.
- **Exception (Ausnahme):** abnormale (aber vorhersehbare und mögliche) Bedingung bei der Programmausführung.
- Wir sprechen hier über Exception Handling (der Ausdruck Error Handling ist aus oben genannten Gründen eigentlich falsch, obwohl er sehr häufig verwendet wird). Leider heissen auch die Standardklassen in C++ häufig `xyz_error` statt `xyz_exception` (schade, ist nicht einmal konsistent)

## 66 Mögliche Reaktionen auf Ausnahmen

- **Ignorieren**  
Motto: "Augen zu und durch"  
Eine sehr risikoreiche Variante
- **Programmabbruch**  
Merkt immerhin, dass etwas nicht in Ordnung ist, die Reaktion ist aber unbefriedigend. Ist Exception Detection aber nicht eigentlich Exception Handling.
- **Exceptioncodes** (nicht Fehlercodes)  
Funktionen geben als Rückgabewert, als Parameter oder global einen Ausnahmecode an.

## 67 Exceptioncodes als Rückgabewert

```
if (eOk == s.push(elem))
{
 ...
}
else
{
 ... // handle exception
}
```

- Rückgabewert wird von Exceptioncodes belegt (ist unschön)
- Geht nicht bei Konstruktoren (Ctors haben keinen Rückgabewert)
- Bei differenzierten Exceptioncodes gibt es stark verschachtelte if ... else bei jedem Aufruf → unleserlich

## 68 Exceptioncodes als Referenzparameter

```
s.push(elem, excpt);
if (eOk == excpt)
{
 ...
}
else
{
 ... // handle exception
}
```

- Aufruf wird mit zusätzlichem Referenzparameter für den Exceptioncode erweitert (ist unschön)



- Bei differenzierten Exceptioncodes gibt es stark verschachtelte if ... else bei jedem Aufruf → unleserlich

## 69 Globaler Exceptioncode

```
s.push(elem); // globalException is set here
if (eOk == globalException)
{
 ...
}
else
{
 ... // handle Exception
}
```

- Führt aufgrund des globalen Exceptioncodes zu schwer les- und wartbaren Programmen.
- Bei differenzierten Exceptioncodes gibt es stark verschachtelte if ... else bei jedem Aufruf → unleserlich

## 70 Wo sollen Exceptions behandelt werden?

- Fact: Exceptions können irgendwo im Programm entstehen
- Wie soll auf Exceptions reagiert werden?
- Eine angemessene Reaktion kann häufig nicht ausschliesslich an der Stelle des Auftretens gemacht werden
- Die Reaktion muss auch weiter "nach oben" gereicht werden können, bis auf die Applikationsebene, wo allenfalls eine Mitteilung an den Benutzer getätigt wird.
- 

---

**Hinweis:** Grundsatz: Nur das regeln, was sinnvoll ist und auf einer bestimmten Stufe wirklich entschieden werden kann, sonst nach oben weiterreichen.

---

### 70.1 Beispiel: Übertragungssystem

- Übertragungssysteme werden häufig in Schichten aufgeteilt (OSI-Modell). Die unterste Schicht ist der Physical Layer, die oberste der Application Layer.
- Annahme: In einer tiefen Schicht (Kanalcodierung) werden den Nutzbits noch Paritätsbits hinzugefügt, um Übertragungsfehler erkennen zu können.
- Wie soll diese Schicht auf einen Fehler reagieren?
  - Wenn genügend Paritätsbits mitgegeben werden, kann allenfalls das fehlerhafte Bit festgestellt und korrigiert werden.
  - Wenn nur festgestellt werden kann, dass etwas falsch ist, nicht aber das fehlerhafte Bit selbst: ???
  - Dann sollte diese Exception nach oben gereicht werden. Dort wird evtl. entschieden, den Block nochmals anzufordern, eine Meldung auszugeben, etc.

## 71 Ziel für Exception Handling

- "Normaler" Programmablauf (Schönwetterfall) wird durch das Exception Handling nicht tangiert
- Der Normalfall soll einfach gelesen werden können
- Der Ausnahmefall ist klar und einfach geregelt
- Der Overhead soll möglichst klein sein
- Die Weiterreichung an die nächsthöhere Funktion im Call Stack soll einfach sein

## 72 Exception Handling in C++

- Exceptions werden in Form eines Objekts am Ort ihres Auftretens ausgeworfen (explizit oder auch "automatisch") (werfen = to throw)
- Exception Handler versuchen, diese Exception-Objekte aufzufangen (to catch)

## 72.1 Exception Handling in C++: Syntax

```
try
{
 ... // Code, der eine Exception auswerfen koennte
}
catch (const MyExceptionClass& exc)
{
 ... // wenn ein Objekt der Klasse MyExceptionClass oder einer Unterklasse
 // davon ausgeworfen wurde, dann kann dieser Handler das Objekt fangen
}

// u.U. weitere Catches
```

## 72.2 Auslösen (Werfen) von Ausnahmen

- Ausnahmen können mit dem Schlüsselwort **throw** explizit ausgeworfen werden
- Nach einem throw-Befehl wird das Programm abgebrochen und beim ersten passenden umgebenden Handler fortgesetzt
- Dabei werden alle lokalen Objekte wieder automatisch zerstört (Stack unwinding)
- Geworfen werden kann ein beliebiges Objekt (üblich: ein spezifisches C++-Ausnahmeobjekt)
- (Ausschliesslich) innerhalb eines Exception Handlers ist auch die Form **throw;** erlaubt. Dadurch wird die Exception an den nächsten Handler weitergereicht (Exception propagation).

### 72.2.1 Beispiel für Exception Handling: unübliche Variante

```
class Xcpt
{
public:
 Xcpt(const char* text);
 ~Xcpt();
 const char* getDiagStr() const;
private:
 const char* diagStr;
};

void allocateFoo()
{
 b1();
 if (0 == allocation())
 throw Xcpt("Allocation failed!");
 b2();
}
```

```
// Testprogramm
void testFoo()
{
 a1();
 try
 {
 a2();
 allocateFoo();
 a3();
 }
 catch (const Xcpt& exc)
 {
 cout << "Caught exception. Text: " << exc.getDiagStr() << endl;
 }
 a4();
}
```

## 72.3 Vordefinierte Ausnahmeklassen

- Ausnahmeobjekte können beliebigen Typs sein (z.B. auch int). Meist werden jedoch spezifische hierarchisch organisierte C++-Ausnahmeklassen verwendet.
- Vordefinierte Standardklasse: **exception**

## 72.4 Exception-Hierarchie in C++

### 72.5 Laufzeit- vs. Logische „Fehler“

- Logische "Fehler"(logic\_error)
  - Ausnahmen im Programmablauf, die bereits zur Entwicklungszeit ihre Ursachen haben.
  - Theoretisch könnten diese Ausnahmen verhindert werden.
- Laufzeit"fehler"(runtime\_error)
  - Nicht vorhersehbare Ausnahmen wie z.B. arithmetische Überläufe
  - Diese Ausnahmen treten erst zur Laufzeit auf, z.B. durch eine nicht erlaubte Benutzereingabe

## 72.6 Exceptions und ihre Header-Dateien

## 72.7 Exception Handler

- Ein oder mehrere Exception Handler können hintereinander definiert werden
- Die einzelnen catch-Handler müssen sich in den Parametern unterscheiden
- Wenn eine Exception geflogen kommt, wird **der erste passende Handler** genommen. Ein passender Handler macht ein catch auf genau diese Exception oder auf eine Basisklasse derselben.
- 

---

**Achtung:** Deshalb (**sehr wichtig**):

Der allgemeinste Handler (am weitesten oben in der Hierarchie) muss der letzte catch-Handler sein.

---

### 72.7.1 Handler mit beliebige Exceptions ohne Hierarchie

### 72.7.2 Handler mit System-Exceptions

### 72.7.3 Handler mit schlechter Reihenfolge

## 72.8 Exception Handler 2

- Wenn kein Handler passt, dann wird im Aufrufstack nach oben gesucht, ob ein passender Handler vorhanden ist.
- Wenn auch dort keiner gefunden wird, dann wird die Funktion `terminate()` aufgerufen.
- `terminate()` beendet das Programm, kann aber auch selbst definiert werden.  
Demo: `uncaught.cpp`
- Catch all  
Der folgende Handler fängt ausnahmslos alle Exceptions ab (und muss wenn gewünscht deshalb immer als letzter aufgeführt werden):

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

Demo: `cuaghtAll.cpp`

## 72.9 Exception Propagation

- Innerhalb eines Exception Handlers kann eine Exception mittels **throw**; weitergereicht werden.
- Die Exception wird dann auch nicht etwa an das nächste **catch(...)** weitergeleitet, sondern an die aufrufende Funktion.  
Demo: `propagate.cpp`

## 72.10 Exception Specification

```
void foo() throw(/* Liste der Exceptions */);
```

- Die Liste spezifiziert, welche Exceptions von einem Aufrufer von `foo()` erwartet werden müssen.
- Aber: garantiert auch, dass das Programm abstürzt, wenn eine andere als eine der spezifizierten Exceptions ausgeworfen wird, d.h. `foo()` muss dafür sorgen, dass wirklich nur die aufgelisteten Exceptions ausgeworfen werden.
- Genauer: falls eine nicht spezifizierte Exception ausgeworfen wird, dann wird die Funktion `unexpected()` aufgerufen, welche üblicherweise das Programm abbricht.
- `unexpected()` kann selbst definiert werden.

### 72.10.1 Exception Specification: Beispiele

```
void foo1() throw(specificXcpt1, specificXcpt2);
// die zwei angegebenen Exceptions muessen vom Aufrufer von
// foo1() erwartet werden.

void foo2() throw();
// KEINE Exceptions koennen geflogen kommen

void foo3();
// beliebige Exceptions muessen erwartet werden
```

### 72.10.2 Exception Handling in der Praxis

- Exceptions sollen nur für Ausnahmen, nicht für den normalen Ablauf verwendet werden
- Exceptions sollen nicht vorbeugende Abfragen ersetzen
- Ein Programm soll nur gegen entscheidende Ausnahmen abgesichert werden
- Wenn eine Exception ausgeworfen wird, dann wird normalerweise eine der vordefinierten Exceptionklassen oder eine (evtl. selbst definierte) Unterklasse davon genommen
- Exception specifications werden, wenn überhaupt, nur bei ausgewählten (Schnittstellen-)Funktionen definiert
- **Always throw exceptions by value, and catch them by const reference.**

### 72.10.3 Throw exceptions by value, catch them by const reference

Was ist der Unterschied zwischen den folgenden Varianten?

- catch (Xcpt& e)
- catch (const Xcpt& e)
- catch (Xcpt e)

### 72.10.4 Handling von System Exceptions

```
int Calculator::divide()
{
 if (0 == nr2)
 throw runtime_error("Division_durch_Null");

 return nr1 / nr2;
}
```

Q: In obiger Variante wird verhindert, dass überhaupt erst eine Division durch Null ausgeführt wird. Könnte die Division nicht einfach probiert werden? Das System sollte ja wenn nötig eine Runtime Exception selbständig auswerfen?

```
int Calculator::divide()
{
 return nr1 / nr2;
}
```

### 72.10.5 Handling-Strategie von System Exceptions bei verschiedenen Programmiersprachen

- In Java und C# gelangen die System Exceptions in die Sprache", d.h. eine Low-Level Exception wird in eine Exception der Programmiersprache "gemappt".

- Die Sprache C++ betreibt kein solches Exception Mapping, d.h. C++ wirft keine Low-Level Exceptions und diese können auch nicht mit catch(...) abgefangen werden.
- Der Hauptgrund dafür ist einmal mehr Effizienz. Wenn ständig Exceptions herumfliegen (auch wenn sie nicht abgefangen werden), dann beeinträchtigt das die Performance.
- Einzelne Systemumgebungen betreiben dennoch Exception Mapping in C++ (z.B: Microsoft in Visual C++). (Visual C++ ist auch nicht C++)

#### 72.10.6 Betreibt meine Umgebung Exception Mapping?

- Mit Hilfe des folgenden Codeausschnitts kann einfach überprüft werden, ob eine bestimmte Umgebung Exception Mapping betreibt:

```
try
{
 int a = 5;
 int b = a/0;
}
catch (...)
{
 cout << "Caught_exception_if_exception_is_mapped" << endl;
}
```

- Unter Umständen muss die Null über cin (zur Laufzeit) eingegeben werden, da "freundliche" Compiler allenfalls darauf hinweisen, dass eine Division durch Null nicht geht.

## Teil XIV

## Preprocessor

## 73 Eigenschaften des Preprocessors

- Wird vor der eigentlichen Übersetzung aktiviert
- Führt textuelle Manipulationen von Source-Dateien durch
  - Makro-Substitution
  - bedingte Übersetzung
  - Einfügen von Dateien
- Der Output des Preprocessors wird dem eigentlichen Compiler übergeben
- Die Direktiven des Preprocessors beginnen immer mit #
- Der Preprocessor ist zeilenorientiert

## 74 Preprocessor-Direktiven und Bedingungsanweisungen

| Direktiven | Bedingungsanweisungen |
|------------|-----------------------|
| #define    | #if                   |
| #undef     | #ifdef                |
| #include   | #ifndef               |
| #line      | #elif                 |
| #error     | #else                 |
| #pragma    | #endif                |

## 74.1 #define

- Definition von Makros (Gefahren betrachten!)

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

- Definition von symbolischen Konstanten (Gefahren beachten!)

```
#define PI 3.14159265358
```

- Es erfolgt eine reine Textersetzung ohne Typenprüfung. Diese kann erst der Compiler vornehmen
- #defines haben keinen Scope
- Simple Definition eines Symbols (z.B. bei Include-Guard)

```
#define FOO_H_
```

## 74.2 #undef

Definition eines Symbols rückgängig machen

```
#undef MAX
#undef PI
#undef FOO_H_
```

### 74.3 #include

Vollständiges Einfügen einer Datei

```
#include <iostream>
// Datei iostream wird in den definierten Include-Verzeichnissen
// gesucht und eingefuegt

#include "foo.h"
// Datei foo.h wird im aktuellen Verzeichnis gesucht und eingefuegt

#include "../drivers/doo.h"
// Pfadangaben muessen immer relativ zum aktuellen Verzeichnis sein.
// Niemals absolute Pfade verwenden!
```

### 74.4 #line

Direktes Setzen der Nummerierung von Sourcecode-Zeilen. Durch die optionale Angabe eines Dateinamens lässt sich der Compiler einen neuen Dateinamen unterschieben.

Das kann z.B. bei vom Compiler erstellten Dateien nützlich sein.

```
#line 67 "main.cpp"
// die naechste Zeile im Sourcecode erhaelt die Nummer 67
// die Sourcedatei erhaelt den Namen "main.cpp"
```

### 74.5 #error

Sofortiger Abbruch des Compiler-Vorgangs und Ausgabe einer Fehlermeldung

```
#ifndef MODEL
#error MODEL ist nicht definiert
#endif
```

### 74.6 #pragma

- #pragma-Direktiven erlauben die Verwendung von Implementierungs-spezifischen Direktiven. Entwicklungsumgebungen können somit ihre eigenen Anweisungen definieren.  
**Diese Direktive ist damit per Definition nicht portabel.**
- Konflikte entstehen keine, da eine Compiler unbekannte #pragma-Direktiven ignoriert.
- Die Portabilität ist damit aber nicht gewährleistet. Ein anderer Compiler versteht die Direktive u.U. nicht und erzeugt deshalb nicht den gewünschten Code.

### 74.7 Bedingungsanweisungen

- Bedingungsanweisungen sind nach folgendem Schema aufgebaut

Bedingungsprüfung

Direktiven

beliebig viele #elif-Gruppen mit Direktiven

optional ein #else mit Direktiven #endif

- Mögliche Bedingungsprüfungen sind
  - #if gibt true, falls Bedingung true ist
  - #ifdef SYMBOL gibt true, falls SYMBOL definiert ist
  - #ifndef SYMBOL gibt true, falls SYMBOL nicht definiert ist



### 74.7.1 Beispiele für Bedingungsanweisungen

```
#if INT_MAX > 32767
 int i;
#else
 long i;
#endif

#ifdef TESTVERSION
 printf("Zeilennummer: %d\n", __LINE__);
 printf("Dateiname: %s\n", __FILE__);
#endif

#ifndef CALCULATOR_H_
#define CALCULATOR_H_
// ...
#endif
```

```
#if 0
 // Auskommentieren des gesamten hier stehenden Codes.
 // Ist sehr nuetzlich waehrend der Entwicklung.
```

### 74.8 Weitere Features des Preprocessors

- #-Operator (stringizing-Operator): Argument wird in String konvertiert
- ##-Operator (token-past-Operator): Zeichenfolge links und rechts des Operators wird zusammengezogen
- und weiteres (siehe Dokus)

```
#define SHOW(var, nr) printf(#var #nr " = %.1f\n", var ## nr)

// Annahme: es gibt eine Variable x5 mit dem Wert 16.4
SHOW(x, 5);
// wird umgesetzt in printf("x" "5" " = %.1f\n", x5);
// d.h. in printf("x5 = %.1f\n", x5);

// Ausgabe ist x5 = 16.4
```

na ja!?

### 74.9 Kritische Würdigung des Preprocessors

- Preprocessoranweisungen sollten zurückhaltend eingesetzt werden, da der Code durch zu viele Preprocessoranweisungen sehr schnell unübersichtlich werden kann.
- Auf den ersten Blick sind Preprocessoranweisungen oft nicht nachvollziehbar.
- Häufig gibt es Alternativen, die ebenso effizient und zudem viel sicherer sind.
- Richtig eingesetzt kann der Preprocessor sehr nützlich sein.