Zusammenfassung: C++

Robert Hilbrich

5. Oktober 2007

Inhaltsverzeichnis

1 Einführung & Motivation **5**
1.1 Historie . . . 5
1.2 Objects by Value . . . 5

2 Elementares C++ **6**
2.1 Lexik . . . 6
2.2 Datentypen . . . 6
2.2.1 Enumerations . . . 6
2.2.2 Felder . . . 6
2.2.3 Typedefs . . . 7
2.2.4 Zeiger . . . 7
2.2.5 Zeichenketten . . . 8
2.2.6 Referenztypen . . . 9
2.2.7 Konstantentypen . . . 9
2.2.8 Strukturtypen . . . 10
2.3 Ausdrücke . . . 11
2.4 Funktionen . . . 11
2.4.1 static . . . 12
2.4.2 inline . . . 12
2.4.3 Default Argumente . . . 12
2.4.4 Ellipsis . . . 13
2.4.5 Überladung . . . 13
2.5 Strukturierte Anweisungen . . . 13
2.5.1 Wo und wie lange leben Objekte . . . 13
2.5.2 Switch & Goto . . . 14
2.5.3 Exception Handling . . . 14

3 Klassen in C++ **16**
3.1 Wichtige Grundaussagen . . . 16
3.1.1 Initialisierung vs. Zuweisung . . . 17
3.1.2 Resource Acquisition is Initialization (of an Object) [RAII] . . . 18
3.1.3 Copy-Konstruktoren . . . 18
3.1.4 Static-Member . . . 18
3.1.5 Member-Zeiger . . . 19
3.2 Vererbung . . . 19
3.2.1 Public Vererbung . . . 19

3.2.2 Non-Public Vererbung 20
3.3 Zugriffsrechte in C++ 20
3.3.1 Friends 21
3.3.2 Unikate 22
3.3.3 Factory 22
3.4 Early and Late Binding 22
3.4.1 VTBL und VPTR 23
3.4.2 Inline, Virtual und Static 24
3.4.3 Virtuelle Konstruktoren 25
3.4.4 Virtuelle Destruktoren 25
3.5 Überladung und Ableitung 26
3.6 Abstract Base Classes 26
3.7 Überladung von Operatoren 27
3.7.1 Member oder Friend 27
3.7.2 Copy Assignment 28
3.7.3 Nutzerdefinierte Aus- und Eingabe 28
3.7.4 Überladung von New und Delete 29
3.7.5 Typumwandlungen 29
3.8 Neue Cast-Operatoren 30
3.8.1 Die Konstantheit eines Objektes ignorieren - $const_cast < T >$. 30
3.8.2 Den Compiler überreden verwandte Typen verträglich zu machen - $static_cast < T >$ 31
3.8.3 Den Compiler überreden NICHT verwandte Typen verträglich zu machen - $reinterpret_cast < T >$ 31
3.8.4 Zur Laufzeit verwandte Typen verträglich machen und sicher verwenden - $dynamic_cast < T >$ 32
3.9 Mehrfachvererbung (Multiple inheritance) 33
3.9.1 Einführung 33
3.9.2 Regeln 33
3.9.3 Mehrdeutigkeiten 34
3.9.4 Mehrfachvererbung und virtuelle Funktionen 34
3.9.5 Konstruktoren 34
3.9.6 Zugriffsrechte 35
3.10 Namespaces 35
3.10.1 Problem 35
3.10.2 Was sind Namespaces 35
3.10.3 Namespace-Reopening 35
3.10.4 Namespace Aliase 36
3.10.5 Bereitstellung von Elementen aus Namespaces 36
3.10.6 Anonyme Namensräume 37
3.10.7 Namensauflösung 37

4 Generische Programmierung in C++ **38**
4.1 Einführung 38
4.2 Ankündigung eines Meta-Parameters mit Typename 39
4.3 Template Parameter 39
4.3.1 Parameter-Voreinstellungen 39
4.3.2 Individuelle Implementationen für spezielle Parameter 39
4.3.3 Sonstiges zu Parametern 40
4.4 Template Code und Compiler 40
4.5 Function Templates 40
4.6 Generische Algorithmen 41

1 Einführung & Motivation

C++ ist eine *multi-paradigm*-Sprache, d.h. man kann Java-like programmieren, muß es aber nicht. Aber: C++ hat prinzipiell *Value*-Semantik (im Gegensatz zur *Reference*-Semantik von Java. OO ist bei C++ möglich, aber nicht zwingend. Primäres Kriterium ist eher *Effizienz*, so soll es keinen *impliziten Overhead* zu Lasten der Effizienz geben.

Daraus ergeben sich die Folgen: keine Feldgrenzenprüfung, fast kein Laufzeitabbild von Klassen, keine automatische Speicherverwaltung und die Objects-by-Value-Semantik.

Man muß hier aus Performance-Gründen *Virtualität* explizit spezifizieren. Eine Klasse ohne Member-Variablen ist mindestens 1 Byte groß. Dieses Byte stellt die Identität des Objektes dar, auf das ein `this`-Zeiger zeigen könnte.

1.1 Historie

Bjarne Stroustrups Ph.D. Arbeit benötigte einen Simulator, so entstand Simula auf einer IBM360. Leider hatte Simula sehr schlechte Laufzeiteigenschaften, so dass er den Simulator erneut in BCPL schrieb. Dann wurde es ein Präprozessor für C - `Cpre` - der C um Simula-ähnliche Klassen erweiterte: *C with Classes.*

1983 wurde C++ das erste Mal erwähnt. 1998 wurde C++ als ISO-Standard spezifiziert.

1.2 Objects by Value

Variablen vom Klassentyp sind (primär) Werte. So ist die Zuweisung `X y = x;` die Erstellung eines weiteren Objektes als Kopie des ersten und keine weitere Referenz (wie bei Java!).

Objekte können *global*, *lokal* und *dynamisch* erzeugt werden. Alternativ zu diesem Konzept kann man aber auch Objekt-Referenzen oder Objekt-Zeiger nutzen.

2 Elementares C++

2.1 Lexik

Eigentlich ist alles wie in C. Übliche Konvention für Bezeichner ist die, das Variablen durchweg klein geschrieben werden und nutzerdefinierte Typen in CamelCase.

2.2 Datentypen

Long ist kein eigener Datentyp, es ist mehr ein Kürzel für `long int`. Keine Standardisierung für die größe dieser Variablen, nur eine Rangfolge:

$$1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)$$

$$sizeof(float) <= sizeof(double)$$

2.2.1 Enumerations

Enumerations sind Aufzählungstypen, also benannte Werte:

```
enum Season { Sommer, Winter, Herbst, Frühling };
Season Zeit = Sommer;
if (Zeit == Winter) { ... }
```

2.2.2 Felder

Felder sind einfach mehrere Objekte, die direkt hintereinander im Speicher stehen. (Felder sind im Gegensatz zu Java keine Objekte und haben kein Längenattribut.)

Beispiel: `int f[n];` Mehrdimensionale Felder stehen im Prinzip auch einfach im Speicher hintereinander.

In der Signatur von Funktion kann auch ein Feld angegeben werden. Wenn die Länge nicht bekannt ist, so kann dort auch einfach ein `(..., int f[], ...)` stehen. Dann ist die Längeninformation allerdings separat bekannt zu geben.

Bei Feldern findet keine Prüfung auf die Einhaltung der Feldgrenzen statt! Der Zugriff ist dann einfach undefiniert.

2.2.3 Typedefs

Mittels `typedef` werden Synonyme für (komplexe) Typkonstrukte angelegt:

```
typedef double V4[4];
V4 einFeld;
```

2.2.4 Zeiger

Zeiger beinhalten die Indirektion per Speicheradresse. Zeiger sind vollständig typisiert, so dass eine falsche Zuweisung einen statischen Analysefehler bringt.

```
int* ptr_i;
ptr_i = &double_var; /* Fehler */
ptr_i = &int_var;
```

Der `*`-Operator ist die Dereferenzierung. Die Umkehrung - der Adressoperator - ist das `&`. Der Wert `0` ist ein ausgezeichneter Zeigerwert und bedeutet *kein Objekt*.

Aufpassen, sobald Objekt mit `new` dynamisch auf dem Heap angelegt wurden. Wird der Zeiger umgebogen ist das Objekt nicht mehr erreichbar und es kommt zu einem *Memory Leak*. Daher sollte auf jedes `new` auch ein `delete` folgen.

Ein doppeltes `delete` führt zu *undefined behaviour*, außer ein `delete 0` ist immer erlaubt. Daher immer das Päarchen `delete` und dann `ptr = 0;` nutzen. Sonst entsteht ein *dangling pointer*, also ein Zeiger auf einen Speicherbereich, der schon freigeben wurde.

Auch Felder können mit `new` dynamisch angelegt werden, müssen dann aber mittels `delete[]` (!) abgeräumt werden!

```
int* ptr = new int[10];
...
delete[] ptr; ptr = 0;
```

Lösungsansatz gegen Memory-Leaks: Einfach einen Zeiger in ein Objekt kapseln (*Ressource Aquisition is Object Instantiation*).

Bei einer Speicheranforderung mittels `new` wird der Speicher noch nicht physisch bereitgestellt. Erst bei einem Zugriff wird dieser Speicher wirklich bereitgestellt.

In jeder (nicht static) Memberfunktion ist `this` ein Zeiger auf das Objekt, an dem der Aufruf erfolgte.

Achtung: Der *parameterlose Konstruktor* sollte besser ohne Klammern gerufen werden! Sonst könnte es auch eine Funktionsdeklaration sein. Bei min. 1 Parameter tritt das Problem nicht auf, da Werte und nicht Typen übergeben werden.

```
T* pt = new T();   /* dynamisch */
T t = T();         /* redundant */
T t ();            /* Ist Funktionsdeklaration */
                   /* (eigentlich aber okay)   */
```

Felder und Zeiger sind eng verwandt: `T* pt = &someTs[0];`. Die dahintersteckende Zeigerarithmetik erfolgt immer modulo `sizeof(T)`. So ist `pt+1` ein Zeiger auf das zweite `T` im Feld.

`pt[i]` ist nur eine Kurzform von `*(pt+i)` und kann gemäß Addition auch vertauscht werden!

```
cout << 1[ "]<<2[" ]<< endl;
```

String `"]<<2["` und dann `+1`, also wird `<` ausgegeben.

2.2.5 Zeichenketten

Wie üblich als `0`-terminierte Felder. In `argv[0]` steht der Programmname drin! Danach die Argumente.

Damit ist bereits der Umgang mit Zeichenketten implizit mit allen problem der Zeiger belastet und zusätzlich auch mit allen *buffer-overflow* Problemen bei Operationen auf Zeichenfeldern.

Legacy-C-String-Routinen sind durch Einbinden von `<cstring>` nutzbar.

Einen Ausweg stellt der Datentyp `std::string` dar, mit der Möglichkeit der Initialisierung aus einem C-String heraus. `std::string name = "Robert Hilbrich";`.

2.2.6 Referenztypen

Primär wird in C++ die *Wertesemantik* verwendet. Mittels der Referenztypen können nun Aliasnamen für Objekte mit Referenzsemantik für alle Typen verwendet werden. Es gibt *keine Nullreferenz* und Referenzen *müssen initialisiert werden.*

```
int   i = 42;
int& ri = i;   /* Alias für i */
```

Eine Vereinbarung einer Referenz verlangt eine Zuweisung. Besonders nützlich ist dies bei Funktionsparametern, um nicht zu viele Strukturen auf den Stack kopieren zu müssen.

2.2.7 Konstantentypen

Ein Typ `T` wird durch das Präfix `const` zu einem *Konstantentyp.* Objekte solcher Typen sind *unveränderlich* (per statischer Kontrolle durch den Compiler).

Für Argumente von Funktionen bedeutet dies, dass die Funktion die nachprüfbare Zusicherung gibt, dieses Argument *nicht zu verändern* und dass beim Aufurf auch konstante Objekte *benutzt werden dürfen.*

```
const double pi = 3.14;
double func(double);
const double x = func(pi); /* call by value! */
```

Konstante Objekte müssen initialisiert werden, weil eine spätere Zuweisung nicht erlaubt ist. Konstante Objekte *können auch über Zeiger nicht verändert* werden, weil der Typ inkompatibel ist (`const T*` statt `T*`).

Referenzen selbst sind implizit `const`, es gibt jedoch Referenzen auf Konstantentypen! Wichtigste Anwendung sind hierbei die *call by reference in-parameter.*

```
foo(const int i);  /* überflüssig, da call by value */
foo(const int& i); /* notwendig, da über Referenz eigentlich veränderbar */
```

Bei Zeigern ist ebenfalls wohl zu unterscheiden zwischen der *constness* des Zeigers selbst (`int* const cptr = &ival;`) und der *constness* des referenzierten Objektes: `const int* ptr_to_const;`. Oder alles verknüpft:

```
const int* const ptr_to_const = &ival;
```

String Literale sind meist `const char*`, aber das const kann auch weggelassen werden. Compiler können so Speicher sparen und gleiche Literale nur einmal im Speicher ablegen.

Als Funktionsparameter wird einfach der Konstantentyp eingetragen, um anzudeuten, dass eine Funktion das Argument nicht verändert.

Um anzudeuten, dass eine Memberfunktion, den Wert ihres Objektes nicht verändert, wird auch hier `const` angegeben. (Welche Funktion ist *objekt-verändernd*, welche nicht?

```
class X {
  public: void foo() const;
          void bar();
};
```

Warum wird das `const` nach dem Namen geschrieben? Das `const` am Ende richtet sich an den `this`-Zeiger, der mit dem Funktionsaufruf implizit mit übergeben wird!

Mittels eines expliziten *casts* kann `const` ausgetrickst werden.

2.2.8 Strukturtypen

wie in C, als heterogene Zusammenfassung von Wertekombinationen unter einem Typnamen.

```
struct Person {
   std::string name;
   int         alter;
} p;
```

Wenn man hier `sizeof()` anwendet, so stellt man fest, dass die Größe vom *Alignment* im Speicher abhängig ist. Dabei spielt auch die Reihenfolge der Mitglieder eine Rolle (Speicherreihenfolge!). `sizeof` arbeitet für Typen **und** Werte.

Verwendet wird hier der Punkt-Operator bzw. der Pfeil-Operator bei Zeigern (ist Sternchen und Punkt und Klammern zusammen).

Strukturen sind in C++ eigentlich Klassen ohne Memberfunktionen und öffentlichem Zugriff auf alle Memberdaten.

Es gibt auch noch *Unions* vom C-Erbe her, aber die spielen nur eine untergeordnete Rolle.

2.3 Ausdrücke

Eigentlich sind sie ähnlich zu Java. Literale und Variablen sind Ausdrücke, sowie die Anwendung von Operatoren auf Ausdrücke ergibt wieder einen Ausdruck.

ABER:

- die Reihenfolge der Berechnung der Ausdrücke (bis auf `&&`, ...) ist *undefiniert*
- jeder Ausdruck liefert einen Wert, der aber nicht verwendet werden *muß* (void = leerer Wert).
- ein Ausdruck wird durch Angabe eines Semikolons zur Anweisung

2.4 Funktionen

Funktionen können außerhalb von Klassen sein (global bzw. *namespace-lokal*) oder als Memberfunktionen innerhalb von Klassen agieren. Man unterscheidet *Deklaration* (Angabe einer Signatur) und *Definition*. Jede Definition ist auch eine Deklaration. Jede Funktion muß deklariert sein, bevor sie verwendet werden kann.

Mehrfache Deklarationen sind erlaubt, aber es darf nur eine *Definition* geben, sonst gibt es einen Linker-Fehler. Die Deklaration erfolgt in den `h`-Files, die Definition in den `cpp`-Files.

Es gibt keine vollständige Analyse des Kontroll-Flusses, also ist das Verlassen einer non-void-Funktion ohne Rückgabe möglich, aber *undefiniert*.

2.4.1 static

Funktionen können `static` sein. Damit werden globale Funktionen beschränkt auf den *file-scope*. Memberfunktionen werden Klassenmethoden (ohne `this`-Zeiger).

Auch können Funktionen *static-Daten* enthalten. Diese werden dann zu globalen Variablen mit Funktionen-Scope. (Nur einmal angelegt!)

2.4.2 inline

Funktionen können als `inline` deklariert werden, um den gesamten Aufruf-Overhead zu vermeiden. Es findet kein Aufruf stat, sondern eine seiteneffekt-freie und typgerechte Substitution auf Quelltextebene.

Der große Vorteil liegt im Geschwindigkeitszuwachs, allerdings kann man dann auch keinen Breakpoint mehr in diesen Funktionen setzen.

Die meisten Compiler machen inline automatisch und sehen die explizite Angabe mehr als einen Hinweis. Besonders gute inline-Kandidaten sind Memberfunktionen, die im Klassenkörper definiert werden, da sie meist kurz sind.

Es dürfen wie gesagt, keine Seiteneffekte auftreten!

2.4.3 Default Argumente

Bei einer Funktionsdeklaration kann ein *Endstück* der Argumentenliste eine Deklaration mit Wertevorgaben sein.

```
int atoi(const char* string, int base = 10);
```

Achtung Falle bei: `void foo(char *= 0)`. (Es gibt ja den Operator `*=`).

Typischerweise werden diese Argumente in die Headerfiles gesteckt. Beim Überladen von Funktionen kann es in Verbindung mit Default Argumenten zu Mehrdeutigkeiten kommen!

2.4.4 Ellipsis

Wie bei `printf` gibt es auch die *variable Argumentenliste*.

```
extern "C" int printf(const char* fmt, ...);
```

Hier ist `extern "C"` eine Linker-Direktive, es findet kein *Name-Mangling* statt. Damit werden konstante Einstiegspunkte für die C-Runtime-Library geschaffen.

Zugriff auf die Argumente der Ellipsis ist mit den Makros `VA_list`, `VA_start` möglich.

2.4.5 Überladung

Funktionen können überladen werden, also gleicher Name, aber eine unterscheidbare Signatur (Rückgabetyp spielt **keine Rolle!**.

Mittels nm foo.o — c++filt kann man das Name-Mangling sehen. Name-Mangling an sich ist kein C++-Standard, jeder Hersteller macht da eigene Sachen.

2.5 Strukturierte Anweisungen

Fast wie in Java. Aber wir haben ein echtes Lokalitätsprinzip von Blöcken.

Objekte deklarieren wenn sie gebraucht werden; sie vernichten (lassen), sobald sie nicht mehr gebraucht werden.

2.5.1 Wo und wie lange leben Objekte

Globale Objekte

Sie entstehen durch eine globale Objektvereinbarung und sind bei Built-in Typen mit 0 initialisiert. Klassentypen werden durch den Ctor Aufruf initialisiert. Vernichtet werden sie automatisch bei Programm-Ende. Sie residieren im globalen Datenbereich, der bereits vom Compiler geplant und vor Programmstart bereitgestellt wird.

lokale Objekte

Sie entstehen durch eine blocklokale Objektvereinbarung. Built-in-Typen werden **nicht initialisiert.** Bei Klassentypen wird der Ctor gerufen. Bei Verlassen des Blocks werden sie vernichtet. Außer *temporaries*, diese werden am nächsten *Sequence-Point* vernichtet. Sonst residieren sie im Stack, der sich dynamisch und sequentiell ausdehnt.

dynamische Objekte

Sie entstehen durch den Aufruf von `new`. Built-in-Typen werden nicht initialisiert, bei Klassen-Typen wird der Ctor gerufen. Vernichtet werden sie durch den Aufruf von `delete[]`. Sie residieren im **Heap**, der sich dynamisch und nicht sequentiell ausdehnt.

2.5.2 Switch & Goto

Bei beiden dürfen Initialisierung nicht übersprungen werden!

2.5.3 Exception Handling

Wie bei Java, aber kein finally.

```
try { .... }
catch (Exception e1) { ... }
catch (Exception e2) { ... }
catch (...)          { /* catch all */ }
```

Bei Auftreten einer Ausnahme wird der try-Block verlassen und zu einem passenden catch-Block verzweigt. *Zuvor werden alle Destruktoren lokaler Objekte gerufen, die erfolgreich konstruiert wurden*! Wirklich nur lokale!!!!

Mittels `throw "oops"` kann eine Exception geworfen werden. Sie kann in einem Catch-Block auch wieder *re-thrown* werden. (`throw` ist auch ein Operator!!)

Exceptions sind Objekte eines beliebigen Typs. Das *stack unwinding* ruft die Destruktoren der *lokalen Objekte.*

Wird eine Exceptions **nirgends gefangen**, so endet das Programm durch den Aufruf von `std::terminate()`.

`std::terminate()` wird auch gerufen, wenn während der Behandlung einer Ausnahme eine weitere Ausnahme auftritt.

Exception specification deklarieren, welche Exceptions eine Funktion werfen kann. Per Default werden alle Execptions raus gelassen. Tritt eine nicht spezifizierte Exception auf, wird `std::unexpected()` gerufen, was wiederum `std::terminate()` ruft. Aber: **Never write an exception specification.**

```
void foo() throw (dies, das) {...}
```

Destruktoren sollten niemals Ausnahmen erzeugen. Sonst terminiert eventuell das ganze Programm!

Wenn ein Funktionskörper nur aus einem Try-Catch-Block besteht, so können die { und }-Klammern des Funktionskörpers weggelassen werden.

Es gibt eine Reihe vordefinierter Exceptions, inkl. einer Ableitungshierachie. Prinzipiell können Exceptions aber beliebig sein.

3 Klassen in C++

Um das Klassenkonzept ranken sich alle wichtigen OO-Konzepte:

- abstrakte Datentypen (Daten & Operationen)
- Zugriffsschutz (Kapselung)
- Nutzerdefinierte Operatoren
- Vererbung, Polymorphie und Virtualität
- Generische Typen

Neu ist u.a. die *Initializer Liste*:

```
Stack::Stack(int dim) : max(dim), data(new int[dim]) { }
```

und auch der Scope-Resolution Operator `::`.

Memberfunktionen können innerhalb des Klassenkörpers definiert werden (implizit inline) oder außerhalb des Klassenkörpers definiert werden mit eine eventuell expliziten `inline`-Spezifikation im Headerfile!

3.1 Wichtige Grundaussagen

Wann immer Objekte entstehen läuft automatisch ein passender *Konstruktor* (CTor). Wann immer Objekte verschwinden, läuft automatisch ein *Destruktor* (DTor). Klassen ohne nutzerdefinierten Ctor/Dtor besitzen implizit:

- den *default constructor*: `X() {}`
- den *default copy-constructor*: `X(const X&) { /*member copy*/}`

- den *default destructor*: `~X() {}`

Sobald nutzerdefinierte Konstruktoren vorliegen (mit Außnahme vom Copy-Konstruktor) gibt es nur noch den impliziten default-copy-constructor.

Jedes Objekt enthält seine eigene Realisierung der Memberdaten, aber **nicht** der Memberfunktionen. Die *Identität eines Objektes* ist mit seiner Adresse verbunden! (Es gibt kein Overhead durch *Meta-Daten*.)

Es sind auch forward Deklarationen von Klassen erlaubt, allerdings lassen diese lediglich die Erstellung von Zeigern oder Referenzen zu.

Strukturell gleiche Klassen bilden trotzdem verschiedene Typen (allerdings läßt sich nutzerdefinierte Kompatibilität einstellen.)

Konstruktorenparameter sind beim Anlegen von Objekten geeignet anzugeben, d.h. es muß einen entsprechenden Konstruktor geben.

```
X x0;               /* Ctor X::X()         */
X x1(1);            /* Ctor X::X(int)      */
X x2 = X(1,2);      /* Ctor X::X(int, int) */
X* xp = new X(1,2); /* Ctor X::X(int, int) */

X x3 = 3;
/*  X tmp(3);   // Temporary bauen
    X x3(tmp);  // Default Copy-Constructor anwerfen */
```

3.1.1 Initialisierung vs. Zuweisung

Im Kontext einer Objektdeklaration handelt es sich um eine *Initialisierung*, aber ohne diesen Kontext handelt es sich um eine *Zuweisung*.

Die einzige Stelle, wo in C++ sinnvoll initialisiert werden kann, ist in der *initializer list* vom Konstruktor: `X::X() : myz(3) {}`.

Sonst ist keine Initialisierung im Klassenkörper möglich, denn es werden nur Typen vereinbart, die aber an der Stelle nicht initialisiert werden. Folgender Code geht **NICHT**:

```
class Z { public: Z(int i){} };
class X { Z myZ(3);          };
```

Es geht nur in der Initializer-Liste:

```
class Z { public: Z(int i){} };
class X { Z myZ;
          Z() : myZ(3) {};   };
```

3.1.2 Resource Acquisition is Initialization (of an Object) [RAII]

Wenn man zum Beispiel mit einem Datenbank-Lock arbeiten will, dann muß man daran denken, dass Lock danach auch wieder ordnungsgemäß abzuräumen - auch im Fehlerfall!

Wenn man nun eine Klasse `DBLock` schreibt, die im Ctor ein Lock setzt und im Dtor das Lock aufgibt, so ist das ganze viel schöner.

Eine lokale Variable `DBLock lock;` gilt nur für ihren Block und sowohl bei einer Exception, als auch beim Verlassen des Blocks wird ihr Destruktor garantiert gerufen. Auf diese Weise kann ein Zugriff bequem gesichert werden.

Ähnlich läßt sich auch ein Quellcode-Tracer realisieren. Einfach eine lokale Variable vom Klassentyp mit entsprechenden Meldungen im Konstruktor und Destruktor schreiben.

Alternativ auch ein Timer!

3.1.3 Copy-Konstruktoren

Die kanonische Form: `X::X(const X&);`

Der default copy constructor führt nur eine *shallow copy* durch, d.h. alle Elemente werden kopiert. Damit auch von Pointern referenzierte Speicherbereiche geklont werden, muß eine *deep copy* mit einem *nutzerdefinierten copy constructor erreicht werden.*

3.1.4 Static-Member

Klassen können auch sogenannte *static member* enthalten, diese werden nur einmal pro Klasse abgelegt. *static memberfunktionen* dürfen **nur** auf static member zugreifen; sie haben keinen `this`-Zeiger.

Static memberdaten sind **NICHT** Teil des Objektlayouts und sind einmalig zu initialisieren.

```
class A { static int count; }
int A::count = 3;             /* hier erst definiert */
```

3.1.5 Member-Zeiger

Neben den traditionellen C-Zeigern gibt es auch neue Zeiger auf Membervariablen und Funktionen! Dazu gibt es die neuen Operatoren `.*` und `->*`.

```
class X { public: int p1, p2, p3; }

void foo() {
   X x;
   X*       pp = &x;      /* traditioneller C-Zeiger auf ein X */
   int X::* xp = &X::p2;  /* xp ist ein Zeiger auf ein int in X */

   pp->*xp = 1;
}
```

Es gibt wie gesagt auch Zeiger auf Memberfunktionen in Objekten, aber die Syntax ist sehr komplex!

3.2 Vererbung

Ist ein Grundprinzip von OO und beinhaltet die Übernahme von Eigenschaften aus einer Klasse bei gleichzeitiger Ermöglichung von Erweiterungen und Modifikationen.

3.2.1 Public Vererbung

Man spricht hier auch von der sogenannten *Implementationsvererbung*.

```
public CountingStack : public Stack            /* = ist ein Stack */
{
    CoutingStack(int dim): Stack(dim), n(0) {}  /* Base Class Ctor */
```

```
void push(int i) {  sum++;                  /* Redefine a Method */
                    Stack::push(i);         /* but use base Class functionality */
...
```

Es gibt auch einen direkten Zugriff auf *Base-Members*, so als wären sie in diesem Namensraum deklariert worden.

Es ist nun auch kein eigener Copy-Construktor notwendig, denn der implizite Copy-Construktor **ruft die Copy-Konstruktoren aller Basisklassen**! (Shallow Copy ist hier ausreichend.)

Nur bei der *Public-Vererbung* gilt die **Ist-Ein-Relation**!

Daher ist jeder CountingStack ist ebenfalls ein Stack! Deswegen kann dieses Ding auch an alle Methoden übergeben werden, die nur einen Stack erwarten.

Von der Ableitung zur Basisklasse ist implizit eine *Projektion* definiert, d.h. durch das *slicing* wird nur der Teil sichtbar gemacht, der auch zum Stack gehört, auch wenn ein CoutingStack übergeben wurde!

3.2.2 Non-Public Vererbung

Hier spricht man von der sogenannten *Interfacevererbung.* `class Deriv1 : private Base { .... };`

`Deriv1` ist **nirgends** ein `Base`, d.h. die Vererbung ist ein (nicht erkennbares) Implementationsdetail. Die *IST EIN*-Relation besteht nicht. Man kann auch nicht durch ein geschicktes Pointer-Slicing auf den Base-Teil zugreifen (Compiler-Fehler!).

`class Deriv2 : protected Base { .... };`

`Deriv2` ist **nur in Ableitungen** ein `Base`, d.h. die Vererbung ist nur Ableitungen von `Deriv2` bekannt.

3.3 Zugriffsrechte in C++

Prinzipiell gibt es die Zugriffsrechte:

- *private* - nur in der Klasse selber nutzbar
- *protected* - nur in der Klasse und in ihren Ableitungen zugreifbar

- *public* - überall sichtbar

Bei einer *Public-Vererbung* gehen alle Zugriffsrechte ganz normal in die Zugriffsrechte der abgeleiteten Klasse über.

Bei einer *Private-Vererbung* werden die ehemals private-Deklarationen unsichtbar und die ehemaligen protected und public-Deklarationen werden nun **private**.

Bei der seltenen *Protected-Vererbung* werden die privaten Sachen wieder unsichtbar und die ehemals protected und public-Deklarationen werden nun **protected**.

Ein `struct` ist implizit **public**, `class` ist implizit **private**.

Ganz wichtig: Beim Lookup von Funktionsnamen erfolgt **Overload resolution vor access-check**!

```
class X {
   private: foo(int);
   public:  foo(int, int = 0);
};
X x;
x.foo(1); /* call of overloaded function is AMBIGUOUS */
```

3.3.1 Friends

Friends bieten die Möglichkeit, speziellen Klassen oder Funktionen Zugriff auf Private Daten einzuräumen.

```
class A {
   private: int secret;
   public:  friend void friendfunc(); /* globale Funktion */
            friend B::f();  /* Funktion einer anderen Klasse als Friend */
... };
```

Friend-Funktionen sind keine Memberfunktionen der Klasse, die die Rechte einräumt! Macht man eine ganze Klasse zum Friend, so werden auch alle Memberfunktionen zu `friends`.

Die Friends-Relation ist **nicht symmetretisch, nicht transitiv und nicht vererbbar**.

Die Position einer Friend-Deklaration im Klassenkörper ist eigentlich unwichtig, aber man sollte sie im Public-Teil unterbringen.

3.3.2 Unikate

Setzt man den Kopie-Konstruktor auf *private*, so können Kopien verhindert werden!

Kann man zum Beispiel für *Singletons* benutzen, also eine Static-Klasse, die einen Zeiger auf *genau eine Instanz* des Objektes enthält und einen privaten Kopy-Konstruktor.

3.3.3 Factory

sind Objekte, die andere Objekte am Fließband herstellen.

```
class P { private: /* alles private */
           public: friend class P_Factory;
          };

class P_Factory {  /* am besten auch singleton */
    public: P* generate() { return new P; }
}

P_Factory::instance().generate();
```

3.4 Early and Late Binding

Zeiger und Referenzen auf Objekte können *polymorph* sein unter Ausnutzung von *slicing.*

Achtung: Für Objekte gilt dies nicht. Objekte können nicht polymorph sein!

```
Stack* sp = new CoutingStack;
sp->push(9); /* ruft Stack::push auf! */
```

Beim Aufruf von nicht-virtuellen Memberfunktionen entscheidet die statische Analyse. Der Zeiger ist von diesem Typ, dann rufe ich die Funktion von diesem Typen auch auf! (**early-binding**)

Sind die Funktionen allerdings in der Basis-Klasse als *virtuell deklariert*, dann entscheidet die dynamische Qualifikation, d.h. der Eintrittspunkt wird zur Laufzeit ermitteln - sogenanntes *late binding*.

```
class Stack { public virtual void push(int); };
Stack* sp = new CountingStack;
sp->push(9); /* ruft CountingStack::push auf! */
```

Damit die Entscheidung auf die Laufzeit vertagt werden kann, muß eine *Typinformation* für das Objekt hinterlegt werden. Woher soll der Zeiger sonst wissen, welche Methoden da sind?

3.4.1 VTBL und VPTR

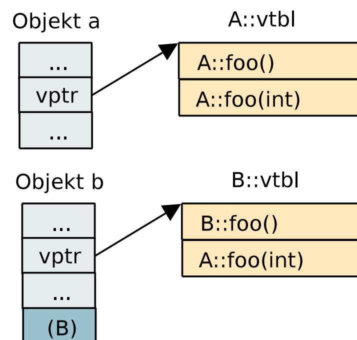
Häufig verwendeter Mechanismus in C++ ist:

- **vptr** - ein verborgener Zeiger pro Objekt und
- eine **Adress-Substitution** beim Aufruf von virtuellen Funktionen

Damit ist das late-binding etwas teuerer, aber eben nur auf *Anfrage*.

```
struct A {
    void bar();
    virtual void foo();
    virtual void foo(int);
} a;

struct B: public A {
    void bar();
    virtual void foo();
} b;
```




```
A ao;             B bo;
A *ap = new A;    B *bp = new B;

A *app = new B;
```

Jetzt kommen die Aufrufe:

```
ao.foo();       /* A::foo        EARLY BINDING */
ap->foo();      /* (ap->vptr[0]) LATE BINDING  */

bo.foo();       /* B::foo        EARLY BINDING */
bp->foo();      /* (bp->vptr[0]) LATE BINDING  */

app->foo();     /* app->vptr[0]   LATE BINDING */
app->foo(9);    /* app->vptr[1]   LATE BINDING */
```

Wie gelangt der richtige `vptr`, der die korrekte dynamische Typinformation widerspiegelt, in ein Objekt?

Antwort: Durch die initiale Operation bei der die Typzugehörigkeit bekannt ist, denn die Konstruktoren wissen, wa sie gerade konstruieren. Das heißt, bei dem Default Constructor `A::A()` wird folgendes gesetzt: `this -> vptr = &A::vtbl;`

Nicht jeder Aufruf wird spät gebunden! Ausnahmen sind:

- Aufruf an einer Objektvariablen (`ao.foo();`)
- Aufruf mit `::` (ScopeResolution)
- Aufruf in einem Ctor/Dtor

3.4.2 Inline, Virtual und Static

`inline virtual void foo();` ist sehr wohl erlaubt, denn wenn der Aufruf der Funktion `foo` einer Objektvariablen erfolgt, so ist der Typ sehr wohl bekannt und kann ge-`inline`-d werden. Ist der Typ nicht bekannt, so zieht die Suche in der Typ-Hierachie (`virtual`). Also entweder `inline` oder `virtual` pro Aufruf.

`static virtual void foo()` ist dagegen **NICHT ERLAUBT**. Das macht ja auch keinen Sinn, denn *virtual* zieht nur im Zusammenhang mit Zeigern und Referenzen auf ein Objekt, was *late-binding* ermöglicht! Der Aufrufkontext von *Static-Funktionen* ist aber immer der Klassen-Scope und daher ist auch bekannt, welche Funktion gerufen werden soll.

Die Planung von austauschbarer Funktionalität muß in der Basisklasse erfolgen, unterhalb der Basis ist diese Funktionalität nicht verfügbar!

Eigentlich heißt es: *einmal virtuell, immer virtuell*, aber das Hinschreiben der eigentlich redundanten Formulierung wird empfohlen.

Eine Redefinition einer Funktion liegt nur dann vor, wenn die Signatur exakt übereinstimmt. Außer wenn es *kovariante Ergebnistypen* gibt, d.h. wenn die Ergebnistypen, der Spezialisierungsrichtung folgend, angepasst werden.

```
class X ( virtual X* clone() { return new X(*this); }
class X ( virtual X* clone() { return new X(*this); }
X x, *px = x.clone();
Y y, *py = y.clone();
```

3.4.3 Virtuelle Konstruktoren

... **sind nicht zulässig**. Denn der Konstruktoraufruf wird durch den Compiler generiert und in diesem Kontext sind die Typen klar. Denn aus der Variablendefinition bzw. aus dem mit `new` verwendeten Klassennamen geht eindeutig der Typ des zu erzeugenden Objekts hervor. Statische Bindung reicht aus. Dynamische ist unötig und unzulässig.

3.4.4 Virtuelle Destruktoren

Destruktoren können virtuell sein (und sollten dies auch sein, wenn in der Klassen ansonsten mindestens eine virtuelle Methode vorkommt!).

```
class X { ~X(); };
class Y : public X { ~Y(); };
X *px = new Y;
delete px; /* undefined behaviour, meist nur X::~X(); */
```

daher besser folgendes machen:

```
class X { virtual ~X(); };
class Y : public X { virtual ~Y(); };
X *px = new Y;
delete px; /* ruft nun Y::~Y() */
```

3.5 Überladung und Ableitung

Überladung wird auch bei abgeleiteten Klassen lokal zu einer Klasse berechnet (ausgehend vom Bezugspunkt!). Das heißt, wenn ich irgendwie durch Typumwandlungen in einer Klasse schon Funktionen finde, die ich für den Aufruf nutzen kann, dann nehme ich die und gucke nicht noch weiter in der Hierachie, ob ich irgendwo noch *bessere* finde.

Das heißt, auch wenn ich irgendwo in der Hierachie eine Funktion finden würde, die perfekt passt (Integer Parameter), aber in der lokale Funktion gibt es eine, wo ich implizit umwandeln kann, dann nehme ich die (Double-Parameter!).

Wenn das nicht so wäre, dann könnte sogenanntes *unintentional overloading* von nicht in Beziehung stehenden Funktionen resultieren.

Falls aber doch die ursprüngliche Funktion, ganz weit oben in der Hierachie gemeint war, so kann in der ganz unten aufgelösten Funktion auch der Scope-Operator `BASECLASS::Func()` benutzt werden.

Also: Im Gegensatz zu Java, werden bei C++ bei Überladung nicht alle Funktionen betrachtet, sondern nur die lokalen einer Klasse, ausgehend vom Bezugspunkt! (Man geht nicht davon aus, dass der Programmierer ein äußerst tiefes Verständnis der gesamten Vererbungslinie hat und damit unintendiert eine ganz andere Funktion ruft!).

3.6 Abstract Base Classes

Eine ABC ist eine Klasse, die mindestens eine *pure virtual function* enthält und daher eine Objekterzeugung nicht sinnvoll erscheint!

```
class ABC {
   virtual void draw() = 0;  /* pure virtual funktion */
}
class X : ABC { };
ABC a;    /* ERROR */
ABC *ap;  /* OK, die Daten sind ja bekannt */

ap = new X;
```

3.7 Überladung von Operatoren

Im Kontext von Klassen können Operatoren mit nutzerdefinierter Semantik überladen werden. Das Überladen der Signatur, Priorität oder Assoziativität ist nicht möglich! Das Einführen von neuen Operatoren ist auch nicht möglich.

Alle bisherigen Operatoren sind überladbar, bis auf `. .* .-> :: ?:` Diese verlangen meistens Namen von Typen statt Werte!

Die vordefinierte Semantik von Operatoren für built-in-Typen bleibt erhalten. Man kann also nicht die Addition für int neu erfinden.

Sehr Wichtig: Ein Operator kann nur dann überladen werden, wenn in seiner Deklaration mindestens ein Parameter von einem Klassentyp ist!

3.7.1 Member oder Friend

Generell gibt es zwei Möglichkeiten der Auflösung eines Operatoraufrufs für `x+y`:

- `x.operator+(y)` (Memberfunktion)

  Dabei muß x von einem Klassentyp sein und y wird u.U. Typumwandlungen unterzogen.

- `operator+(x,y)` (globale Funktion)

  x oder y muß ein Klassentyp sein und x oder y werden Typumwandlungen unterzogen

Alle einstelligen Operatoren und alle der Form `@=` sollten Memberfunktionen werden. Alle der Form: `= () [] ->` müssen Member sein. Bei denen soll garantiert werden, dass es sich beim ersten Operator um ein *Lvalue* handelt!

Alle anderen zweistelligen sollten Friends werden.

Für `++x` nutzt man `X& X::operator++();`
Für `x++` nutzt man `X X::operator++(int);` (syntaktischer Hack)

3.7.2 Copy Assignment

Bezeichnet den Kanonischen Zuweisungsoperator
`X& X::operator=(const X&)`

Dieser wird mit Shallow-Assigment Semantik implizit bereitgestellt, kann aber neu definiert werden. Falls ein nutzerdefinierter Kopy-Konstruktor vorliegt, ist zumeist auch der Zuweisungsoperator nutzerdefiniert zu implementieren!

Da sie beide zusammen bereitgestellt werden, gibt es auch eine kanonische Form die ebenfalls *exception safe* ist. *What is the canonical form of strongly exception safe copy assignments?*

Als erst muß eine Swap Funktion bereitgestellt werden, die keine Exception wirft:
`void T::swap(T& other) { /* Swap *this and other */ }`

Dann wird der Operator nach dem Motto *erstelle eine temporäre Variable und swappe* programmiert:

```
T& T::operator(const T& other) {
     T temp(other);   /* Copy Constructor */
     swap(temp);      /* Commit current work */
     return *this; }
```

3.7.3 Nutzerdefinierte Aus- und Eingabe

```
class SomeClass {
 friend ostream& operator<< (ostream&, const SomeClass&);
 friend istream& operator>> (istream&, SomeClass&);
}
```

Warum `friend`? Weil wir an die Implementation von `ostream` nicht mehr rankommen, wenn wir sie als Member machen wollten.

Warum Referenzen? Damit man das ganze Kaskadieren kann:
`cout << o << endl; /* op<< ( op<<(cout, o), endl); */`

Warum SomeClass Referenzen? Damit man beim Einlesen auch das Objekt verändern kann! Bei der Ausgabe macht man das eher aus Performance Gründen!

3.7.4 Überladung von New und Delete

Wenn man ein Replacement der impliziten globalen Operatoren vorhat, dann ist das ein sehr tiefer Eingriff ins Laufzeitsystem - also nichts für den Gelegenheitsprogrammierer.

```
void *operator new(std::size_t) throw (std::bad_alloc);
void operator delete(void*) throw();
```

Bis auf bestimmte Operatoren (`new(size_t, void*)`) lassen sich so auch nur bestimmte globale Operatoren neudefinieren.

```
/* allocation tracer */
void* operator new(std::size_t s, const char* info = 0) {
     printf(info...);
     void *p = calloc...
     return p;
}
```

Es lassen sich aber auch die **klassenlokalen Operatoren new/delete** überladen! Sie gelten nur für dynamische Objekte von diesem Typ. Der Vorteil liegt darin, dass alle gleich groß sind und wir uns nicht um die Besorgung des Speicherplatzes kümmern müssen!

```
class X {
    public: void* operator new(std::size_t);
            void  operator delete(void* p);
};
X* px = new X;     delete px;
```

3.7.5 Typumwandlungen

Konstruktoren, die mit einem Argument aufrufbar sind, fungieren als Typumwandler vom Argumenttyp zum Klassentyp.

```
 class X {
X(int, double=0); // int  -> X
X(char);   // char -> X
```

Achtung: Es kommt *maximal eine* nutzerdefinierte Typumwandlung zum Einsatz. (Effizienzgedanke!)

Wenn man keine automatischen Typumwandlungen möchte, dann spezifiziert man einfach `explicit`, also

```
class X { explicit X(int); };
f(2);    /* keine implizite Umwandlung, ERROR */
f(X(2)); /* explizite Umwandlung gefordert    */
```

Ziel der Umwandlung ist eigentlich immer ein Klassentyp.

Es gibt aber noch eine zweite Kategorie von nutzerdefinierten Umwandlungsoperationen, bei denen die Quelle der Umwandlung immer ein Klassentyp ist: *Conversion Operators*

```
class Bruch {
   operator double() { return zähler/nenner; }
```

Wichtig ist hierbei, dass keine Argumente und kein Rückgabetyp angegeben wird! Nicht mal void!!!

Beide Möglichkeiten der Umwandlung sind gleichberechtigt, daher äußern sich Mehrdeutigkeiten auch in einem statischen Fehler!

Ziel einer Konversion kann auch ein Zeigertyp sein, d.h. man kann eine Konversion nach `const char*` definieren: `operator const char*() {return "X";}`. Damit könnte das Objekt ausgegeben werden, wie ein normaler C-String.

Diese Konversionen sind aber nicht abschaltbar.

Sowohl bei impliziten Typumwandlungen, aber auch bei expliziten *CAST-Operationen* werden diese Conversionen implizit benutzt. Cast sind in C++ aber eher unüblich.

3.8 Neue Cast-Operatoren

3.8.1 Die Konstantheit eines Objektes ignorieren - $const_cast < T >$

... verletzt eigentlich die Spielregeln, da alle schreibenden Zugriffe nach Brechnung der constness ein *undefined behaviour* nach sich ziehen.

Aber manchmal ist das aus praktischen Gründen unumgänglich.

```
string s("Hallo");
extern "C" oldCFunc(char *);

oldCFunc(s.c_str());                   /* ERROR const ignored */
oldCFunc(const_cast<char*>(s.c_str())); /* OK */
```

Alternativ kann man hier `mutable` nutzen, wenn ein Objekt zwar logisch konstant ist, aber in Implementationsdetails veränderlich sein sollte.

Wenn man zum Beispiel eine `const X&` übergeben kriegt und in `X` ein Member mutable ist, dann kann dieser trotz konstanter Referenz verändert werden!

3.8.2 Den Compiler überreden verwandte Typen verträglich zu machen - $static_cast < T >$

... das Ergebnis kann ohne erneute Umwandlung verwendet werden.

```
class X {};
class Y : public X {};

Y yo;
X& x1 = yo;                   /* implizite Anpassung der Typen */
X& x2 = static_cast<X&>(yo); /* dasselbe */
```

Achtung, die IST-EIN-Relation ist nicht symmetrisch. Jedes Y ist immer auch ein X, aber kein X ist auch ein Y!

3.8.3 Den Compiler überreden NICHT verwandte Typen verträglich zu machen - $reinterpret_cast < T >$

Das Ergebnis kann hier aber nur nach erneuter *Rückverwandlung* verwendet werden! Dabei wird die unveränderte Bitbelegung einfach anders interpretiert!

Daher ist dieser Cast meist auch nicht potabel.

```
int *pi = &someInt;
```

```
int i = reinterpret_cast<int>(pi);
   /* dont use i now! */
int p* = reinterpret_cast<int*>(i);
*p = 3121; /* Now Okay */
```

3.8.4 Zur Laufzeit verwandte Typen verträglich machen und sicher verwenden - $dynamic_cast < T >$

```
class X { virtual void foo(); };
class Y : public X {};

Y& y1 = dynamic_cast<Y&>(x1); /* OK, weil x1 ein Y referenziert */

X& x3 = *new X;
Y& y2 = dynamic_cast<Y&>(x3); /* EXCEPTION: std::bad_cast */
```

Die Implementation dieses Casts setzt offensichtlich auf die Auswertung von Laufzeitinformationen (RTTI = Run Time Type Identification).

Sobald eine Klasse eine virtuelle Funktion hat, ist sie polymorph (sie hat vptr und vtbl). Daher gibt es die RTTI auch nur für Zeiger und Referenzen von *polymorphen* Typen.

Es gelingt ein *Downcast* (Von Zeiger/Referenz einer Basisklasse auf Zeiger/Referenz einer Ableitung), wenn das referenzierte Objekt vom Typ der Ableitung oder einer Ableitung dieser ist.

Im Fehlerfall liefert dieser Cast bei Zeigern eine 0 und bei Referenzen die Exception std::bad_cast zurück.

Trotzdem kann man eine Basisklasse nicht einfach zur Ableitung casten.

Man kann auch die Typidentität direkt abfragen, dazu existiert `typid` (wie `sizeof` vom Compiler umgesetzt und nicht überladbar), was eine Struktur `type_info` liefert. Der Vergleich dieser beiden Strukturen ist möglich!

In `Type_info` gibt es auch einen Klarnamen der Klasse (nicht notwendigerweise identisch mit dem Klassennamen).

Achtung: Damit das funktioniert müssen beide Funktionen **polymorph** sein.

Dennoch sollte die RTTI nur in Ausnahmefällen explizit benutzt werden! Statt dessen lieber late-bound Virtual-Funktionen nehmen! (Da ist die RTTI implizit!)

3.9 Mehrfachvererbung (Multiple inheritance)

3.9.1 Einführung

Eine Klasse kann mehrer Basisklassen haben, d.h. es ist eine freie Kombination von Konzepten möglich.

```
class Combined : public Concept1,
                 public Concept2,
                 private Concept3
{ };
```

Damit ist jedes COMBINED-Objekt auch ein CONCEPT1 und ein CONCEPT2. Aber CONCEPT3 ist nur ein *Implementationsdetail*.

Dabei bleibt die Polymorphie erhalten. So kann man Basisklassenzeiger und abgeleitete Klassen-Zeiger vergleichen, aber die unterschiedlichen Basisklassenzeiger sind *unvergleichbar*.

Im Speicher wird das Layout *linearisiert*, das heißt die Bestandteile der einzelnen Klassen liegen hintereinander im Speicher. Wenn Slicing vorgenommen wird, dann kann es sein, dass Teile ausgeblendet werden, weil die Slice nicht zu Beginn des Combined-Objekts beginnt, sondern eher ein Teil mittendrin ist.

3.9.2 Regeln

Eine Klasse kann eine andere **nicht** direkt mehrfach einerben:

```
class B: public A, public A {...};
```

Dennoch kann es im Laufe der Vererbungen zu gewollten oder ungewollten Maschen/Diamonds im Vererbungsbaum kommen. Ob der dabei mehrfach eingeerbte Basisklassenanteil dupliziert oder unifiziert im neuen Objekt erscheinen soll, ist steuerbar.

Jede Klasse, die nur einmal auftauchen soll, wird mittels `virtual` eingeerbt. Realisiert wird das ganze nicht über Blockkopien, sondern über Zeiger auf den letztendlichen virtual-Teil.

Für beide Szenarien gibt es sinnvolle Anwendungen!

3.9.3 Mehrdeutigkeiten

Es kann hier schnell zu Mehrdeutigkeiten kommen, die zu einem statischen Fehler führen. Wenn sich diese Mehrdeutigkeiten durch Scope-Resolution auflösen lassen, dann ist alles okay.

Es ist auch möglich, dass es mehrere Wege zur Auflösung eines Namens gibt. Es dominiert hier der *kürzeste Weg*. Liegen mehrere gleich lange Wege vor, so gibt es Mehrdeutigkeiten und es muß qualifiziert werden.

Beispiel: `B b; b.A::f();`

Potentielle Mehrdeutigkeiten werden auch *unabhängig* von ihren Zugriffsrechten lokalisiert! Auch wenn sie durch die Rechte eigentlich deutlich werden müssten, so ist AccessCheck als letztes der Fall!

3.9.4 Mehrfachvererbung und virtuelle Funktionen

...sind miteinander kombinierbar! Im Falle von virtuellen Basisklassen stehen u.U. mehrere Wege der Auflösung zur Verfügung. Falls keine dominante Implementation existiert, muß in der am weitesten abgeleiteten Klasse eine Redefinition erfolgen.

Fehlermeldung: Funktion needs a final overrider!

3.9.5 Konstruktoren

Konstruktoren virtueller Basisklassen müssen in der am weitesten abgeleiteten Klasse direkt gerufen werden.

3.9.6 Zugriffsrechte

Wird eine virtuelle Basisklasse sowohl private als auch public vererbt, so dominiert public!

Bei nicht virtueller Vererbung gilt für jedes Auftreten einer Basisklasse das Zugriffsrecht entsprechend der direkten Vererbung.

3.10 Namespaces

3.10.1 Problem

Namenskollisionen im globalen Namensraum sind in großen Projekten unvermeidbar. Klassen helfen zwar etwas zur Entspannung, aber die Namen der Klassen sind wiederum im globalen Namensraum vertretbar.

Die Lösung liegt in der Deklaration von Klassen in Namespaces!

3.10.2 Was sind Namespaces

Namespaces dürfen beliebige Deklarationen und Definitionen enthalten (auch Namespaces, die so geschachtelt werden können).

Klassen dürfen lokale Klassen enthalten, aber keine Namespaces!

Namen von äußeren Namespaces sind wiederum globale Gebilde.

3.10.3 Namespace-Reopening

. . . erlaubt zusätzliche Deklarationen, fehlende Definitionen und eine logische Verteilung über separate Dateien! (**Natürlich ist das nicht für STD erlaubt!**

```
namespace HU {
 void register () {};
}
```

Man kann auch Dinge direkt im umhüllenden Namespace definieren:

```
class HU::Register {}; /* Namespace HU */
```

3.10.4 Namespace Aliase

Lange Namen sind pregnant aber unpraktisch, daher Aliase:

```
namespace HU = Humboldt_Universitaet;
```

3.10.5 Bereitstellung von Elementen aus Namespaces

Using-Deklaration

Mit einer `using`-Deklaration wird ein Name aus einem Namensbereich direkt in den Geltungsbereich eingeführt, in dem die using-Deklaration erfolgt (als wäre das Element dort deklariert worden):

```
void doit() {
 using HU::register;
 register(Bla, Blub);
}
```

Using-Direktive

Durch eine `using`-Direktive können sämtliche Namen des angegebenen Namensbereiches für den Geltungsbereich zugreifbar gemacht werden.

Die Direktive wirkt dabei so, also seien alle Elemente außerhalb ihres Namensbereichs deklariert und zwar an der Stelle, an der die Namensbereich-Definition *tatsächlich* steht.

```
using namespace Humboldt_Universitaet;
Fachbereich Informatik;
Student* Markus;
```

Wenn nicht klar ist, welche Symbole definiert sind, sollte die using-Direktive nicht verwendet werden. Dadurch können Mehrdeutigkeiten entstehen! Aus diesem Grund sollten sie auch nicht in Headerfiles eingebunden werden - **KEIN USING** in Headerfiles!

Mittels der using-Deklaration können auch private-eingeerbte Funktion wieder öffentlich zugänglich gemacht werden!

```
class B : private A {
 public: using A::f;
}
```

3.10.6 Anonyme Namensräume

Es gibt auch anonyme Namensräume mit File-Scope.

```
namespace { /* body */ }

/* is äquivalent to */

namespace UniqueNS{}
using namespace UniqueNS;
namespace UniqueNS{ /* Body */ }
```

3.10.7 Namensauflösung

Ist zunächst *Lokal* (inkl. Using) und sonst in allen sichtbaren Namespaces!

Prinzipiell erfolgt die Namensauflösung immer erstmal *lookup*, dann *overload resolution* und dann *Access check*.

Lookup qualifizierter Namen erfolgt im jeweils benannten Scope beginnend rekursiv!

Lookup *unqualifizierter* Namen hat noch einen Sonderfall: *Koenig Lookup* oder auch Argument Dependent Lookup genannt. Aus allen Parametertypen eines Funktionsaufrufs wird rekursiv eine Menge sogenannter *associated namespaces / classes* ermittelt, in denen dann die gesuchte Funktion eindeutig gefunden werden muß.

4 Generische Programmierung in C++

4.1 Einführung

Vererbung erlaubt die Wiederverwendung von Programmcode, virtuelle Funktionen erlauben die Austauschbarkeit. Klassen erlauben aber **nicht** die Wiederverwendung durch Parametrisierung (und Optimierung zur Compile-Zeit). Jede abgeleitete Klasse bräuchte dann auch einen neuen Namen.

C++ erlaubt die Definition von generischen Klassen, die noch von Typen oder Werten abhängen.

```
/* stack.h */
template <class T, int D> /* variabel in T und D */
class Stack {
    private: T *data;
             int max;
    public:  Stack(int dim = D) : max(dim) {}
             push(T i);
 };

 /* stack.cpp */
 template <class T, int D>
 void Stack<T, D>::push(T i) {
  data[top++] = i;
 }
```

Eine Template Klasse definiert ein allgemeines Muster für beliebig viele konkrete Klassen. Aus dem Template selbst läßt sich aber kein Code erstellen. Dies geschieht erst bei der Instantiierung des Templates.

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

Daher muß bei der Instantiierung einer Template Klasse der Quelltext aller benutzten Funktionen zur Verfügung stehen. Übliche Verfahrensweise:

```
/* stack.h */
template <class T, int D>
class { /* wie oben */ };
 #include "stack.cpp"
```

Die Instantiierung einer konkreten Klasse geschieht auf explizite Anforderung oder man instantiiert auf Vorrat!

```
template class Stack<int, 10>;
```

4.2 Ankündigung eines Meta-Parameters mit Typename

`<class T>` bedeutet **NICHT**, dass nur mit Klassentypen instantiiert werden darf. Vielmehr kündigt `class` hier einen *Meta-Typ-Parameter* an. Alternativ kann aber auch das neue Schlüsselwort `typename` verwendet werden.

Wenn es kein `typename` gäbe, dann wäre eine Deklaration `T::X * x` mehrdeutig: Multiplikation oder Zeigertyp!

```
template <typename T>
class Beispiel {
 typename T::X x; // X muß in T ein Typname sein
};
```

4.3 Template Parameter

4.3.1 Parameter-Voreinstellungen

```
template <class T = int, int D = 20>
```

4.3.2 Individuelle Implementationen für spezielle Parameter

```
template <> /* Alle Parameter gebunden */
class Stack<std::string, 100> { ... /* special implementation */ };
```

```
template <>
class Stack<std::string, 100>::push(const std::string& s) {}
```

So kann man auch für gleiche Parameter konfigurieren:

```
template <typename T>
class MyClass<T, T> {}   /* zwei gleiche Typen */
class MyClass<T, int> {} /* T2 ist int         */

// class MyClass<int, int> {}  /* ist nicht eindeutig!!! */
```

4.3.3 Sonstiges zu Parametern

Einzelne Memberfunktionen können eigene Template-Parameter (über die Klasse hinaus) besitzen (*member templates*).

Template-Parameter können selbst wieder Template-Typen sein! *template template parameters*.

4.4 Template Code und Compiler

Template Code muß zunächst nur syntaktisch korrekt sein. Erst bei der Instantiierung wird semantische Korrektheit geprüft.

Für Klassentemplates werden nur die Funktionen instantiiert, die auch benötigt werden und nur diese müssen auch fehlerfrei übersetzbar sein!

Substitution failure is not an error! Das heißt, wenn im Ersetzungsprozess ein Fehler auftritt, dann wird die Funktion einfach nicht in den Kreis der für eine Overload-Resolution betrachteten Funktionen aufgenommen. Der Substitution Fehler (weil ein Member für einen Parameter nicht verfügbar ist) ist kein Error!

Auf diese Weise läßt sich die Zuweisbarkeit von Typen *statisch* entscheiden!

4.5 Function Templates

Auch Funktionen können als Templates implementiert werden.

```
template <typename T>
const T& max(const T& a, const T& b) {
 return a > b ? a : b;
}
```

Dies ist quasi eine *universell überladene Funktion*. Natürlich wird die Überladung zunächst in non-template Code aufgelöst. Danach, falls eine Template-Funktion exakt passt. Ansonsten, wenn Typumwandlungen nötig sind, wird die Schablone benutzt.

Der Funktionskörper muß für Compiler zugänglich sein.

Inlining *en passant*.

Ein Funktionsaufruf ist auch ohne explizite Instantiierung möglich, wenn vorliegende Parameter diese eindeutig machen.

```
max(3,5)         -> non-template max
max<>(3,5)       -> max<int>
max(3.0, 4.9)    -> max<double>
max('a', 23.45) -> non-template (keine gleichen Typen!)
max(7, 23L)      -> non-template max
```

Es gibt **keine Default-Template-Parameter**, **keine Template Template Argumente** und **keine partielle Spezialisierung** für Funktionstemplates.

Wenn eine Funktion spezialisiert werden soll, dann bitte überladen!

4.6 Generische Algorithmen

Dadurch können auch Algorithmen generisch werden, also unabhängig vom Typ. Der entstehende Programmcode muß natürlich semantisch korrekt sein. Also Operatoren, wie $<$, müssen definiert worden sein!