0. Einführung & Motivation

Ansatz: "C++ für Java-Kenner"

- Konzentration auf semantische Unterschiede 'gleichartiger' Konzepte
- Erörterung der C++ -spezifischen Konzepte (Overloading, Templates)

Anspruch auf Vollständigkeit

Sprache laut Standard **ISO / IEC 14882 von 1998**
incl. Standardbibliothek (***STL and more***)

Schwerpunkt auf Diskussion von Konzepten anhand von Beispielen

0. Einführung & Motivation

Warum noch eine OO-Sprache?

- die zudem
  - syntaktisch sehr ähnlich zu Java ist
  - älter ist als Java ...
  - 'gefährlicher' ist als Java ...

Weil C++ eine Sprache ist

- die
  - syntaktisch sehr ähnlich zu Java ist
  - älter ist als Java ...
  - 'gefährlicher' ist als Java ...

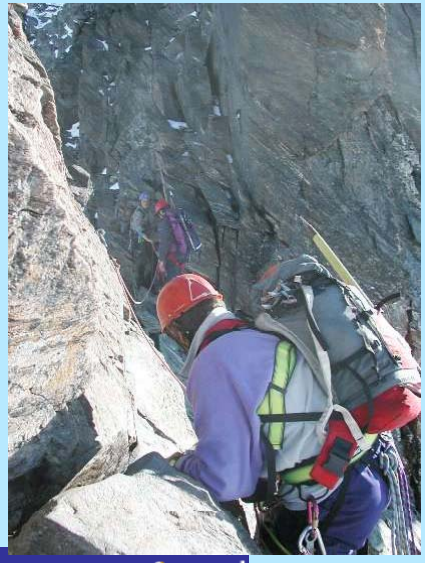
-- und potenziell effizienteren Code ermöglicht

http://www.research.att.com/~bs/applications.html

0. Einführung & Motivation

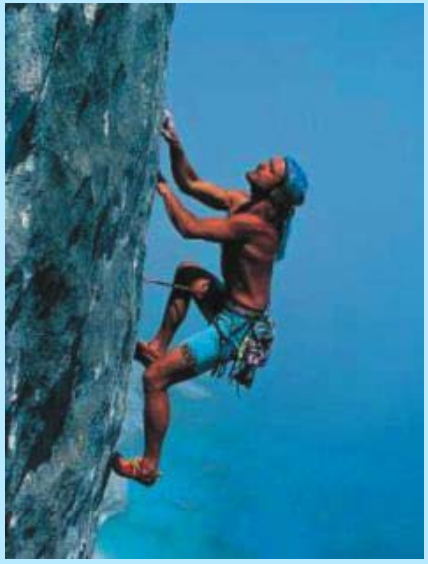
Java

- zahlreiche Sicherheitsvorkehrungen kosten Zeit & Raum
- *virtual machine*
- architekturneutral



C++

- keinerlei Sicherheitsvorkehrungen Reserven für Zeit & Raum
- *native code*
- architekturabhängig



Ein erster Blick: Hello World

Java

Hello.java

```
class Hello {
        public static void main(String s[]) {
                if (s.length < 1) return;
                Hello h = new Hello(", " + s[0]);
                h.speak();
        }
        String what;
        void speak() {
                System.out.println("Hello" + what);
        }
        Hello(String s) {
                this.what = s;
        }
        protected void finalize() {
                System.out.println("bye, bye");
        }
}
```

Ein erster Blick: Hello World

C++

h.cc

```
#include <iostream>
#include <string>
class Hello {
public: static void main(int c, char* v[]) {
                    if (c < 2) return;
                    Hello h = Hello(", "+std::string(v[1]));
                    h.speak();
        }
private: std::string what;
        virtual void speak() {
                    std::cout << "Hello" + what << std::endl;
        }
        Hello(std::string s) {
                    this->what = s;
        }
        ~Hello() {
                    std::cout << "bye, bye" << std::endl;
        }
}; // !!!!!!!!!!
```

Java-Style

Ein erster Blick: Hello World

C++

h.cc

Java-Style

```
// ... continued

int main(int argc, char* argv[])
{
        Hello::main(argc, argv);
}
```

Ein erster Blick: Hello World

C++

h0.cc

C - Style

```
#include <cstdio>

int main(int argc, char* argv[])
{
        if (argc > 1)
           std::printf("Hello, %s\n", argv[1]);
}
```

Ein erster Blick: Hello World

- in C++ kann man offenbar Java-*like* (OO) programmieren, muss es aber nicht:
  - C++ ist eine sog. *multi-paradigm*-Sprache

- Abweichungen in syntaktischen Feinheiten

- semantische Unterschiede

```
Java:           Hello h = new Hello(....);        // reference !
                h.speak();
C++:            Hello h = Hello(....);            // value !
                h.speak();
```

Ein erster Blick: Hello World

- In C++ muss jeder verwendete Bezeichner zuvor (oder in der gleichen Klasse) deklariert werden!
  (auch Bezeichner aus der Standard-Bibliothek)

```
#include <string> .... std::string
```

statt → /usr/include/g++/string

```
String (--->java.lang.String)
```

Ein erster Blick: Hello World

- In C++ gibt es globale Funktions- (Variablen- und Typ-) Deklarationen
- Es gibt geschachtelte Gültigkeitsbereiche (Klassen und namespaces) aber ohne implizite Abbildung auf eine hierarchische Verzeichnisstruktur
- Ein Compilerlauf behandelt GENAU EINE Quelldatei pro Aufruf! (.... make !)
- Dies entspricht dem klassischen Paradigma der Übersetzung von C-Programmen: ermöglich Migration, Portierbarkeit, Unix-Konformität

Ein erster Blick: Hello World

- In C++ gibt es Zeiger, Felder sind de facto Zeiger - keine Objekte, this ist ein Zeiger !
- Konvention des Programmaufrufs ist etwas anders
- Virtualität ist explizit zu spezifizieren
- Es gibt sog. Destruktoren
- Syntax von Zugriffsrechten ist etwas anders

Der zweite Blick: Effizienz

Problem 1: integer - Arithmetik

$$3^{10^n}$$

(modulo int-overflow)

Problem 2: double - Arithmetik

$$e \sim (1 + 1/n)^n$$

[n = 10e8, 10e9]

Der zweite Blick: Effizienz

```
class i {
        public static void main(String []s)
        {
                int i=1;

                for(int n=1; n<=100000000;++n)
                        i*=3;
                System.out.println( i );
        }
}
```

Java

Der zweite Blick: Effizienz

C++

```
#include <iostream>

class i {
public:
static void main()
        {
                int i=1;

                for(int n=1; n<=100000000;++n)
                        i*=3;
                std::cout << i << std::endl;
        }
};

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

Der zweite Blick: Effizienz

```
class d {

        public static void main(String []s)
        {
                double e=1;

                for(int n=1; n<=100000000;++n)
                        e*=1.00000001;
                System.out.println( e );
        }
}
```

Java

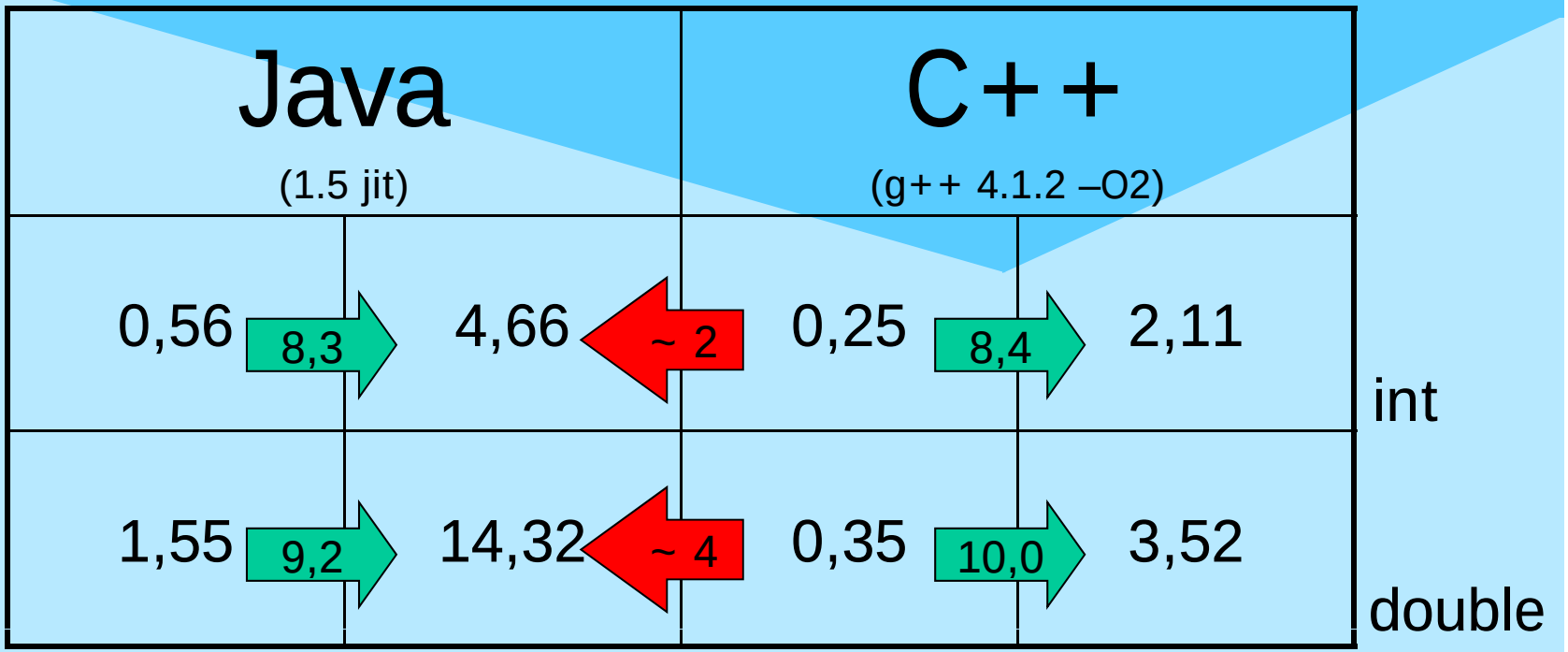
Der zweite Blick: Effizienz

C++

```
#include <iostream>

class d {
public:
static void main()
        {
                double e=1;

                for(int n=1; n<=100000000;++n)
                        e*=1.00000001;
                std::cout << e << std::endl;
        }
};

int main() {
        d::main();
        return 0; // not needed but good style
}
```

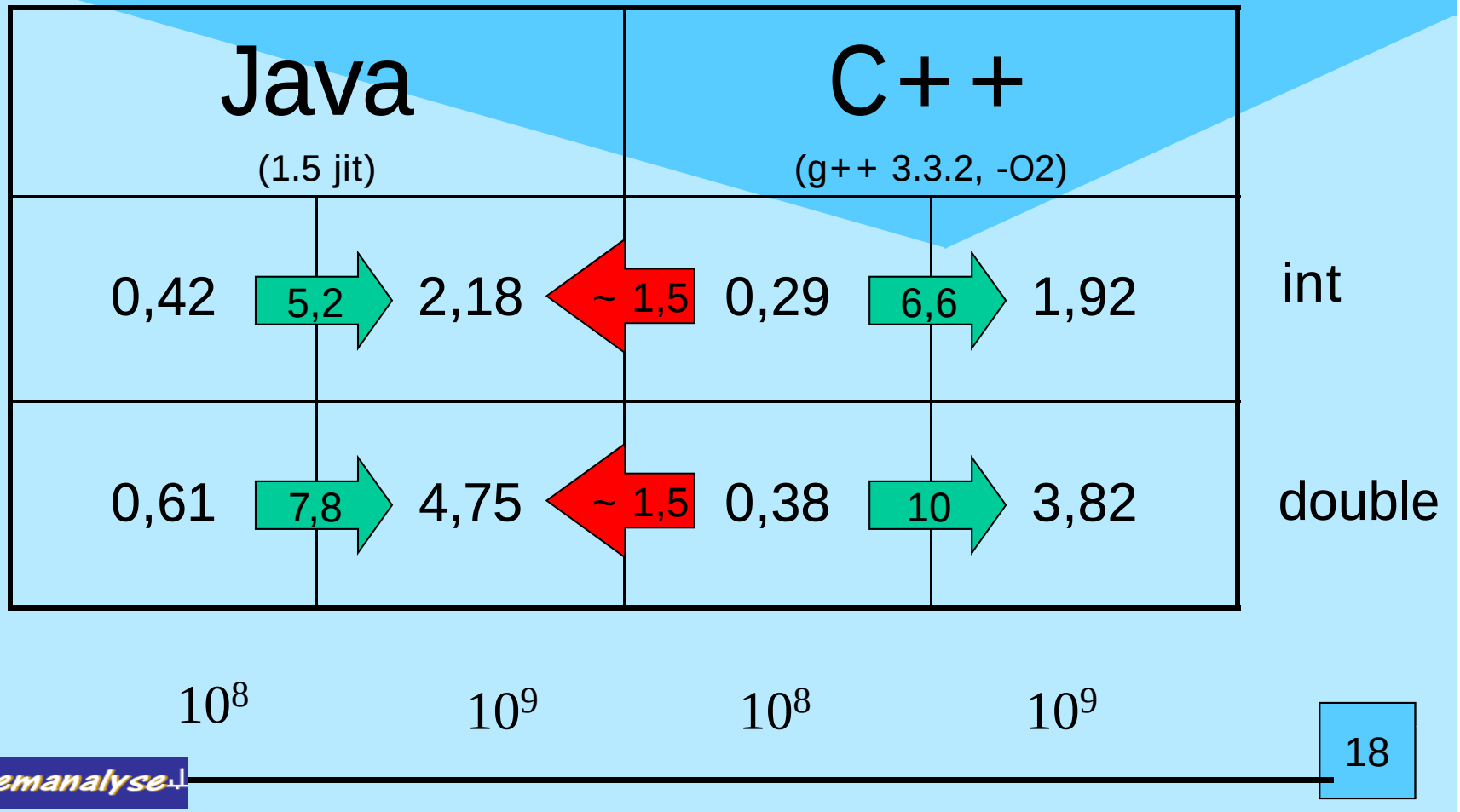
Der zweite Blick: Effizienz

Laufzeiten (Debian Linux, Pentium4 2 GHz)



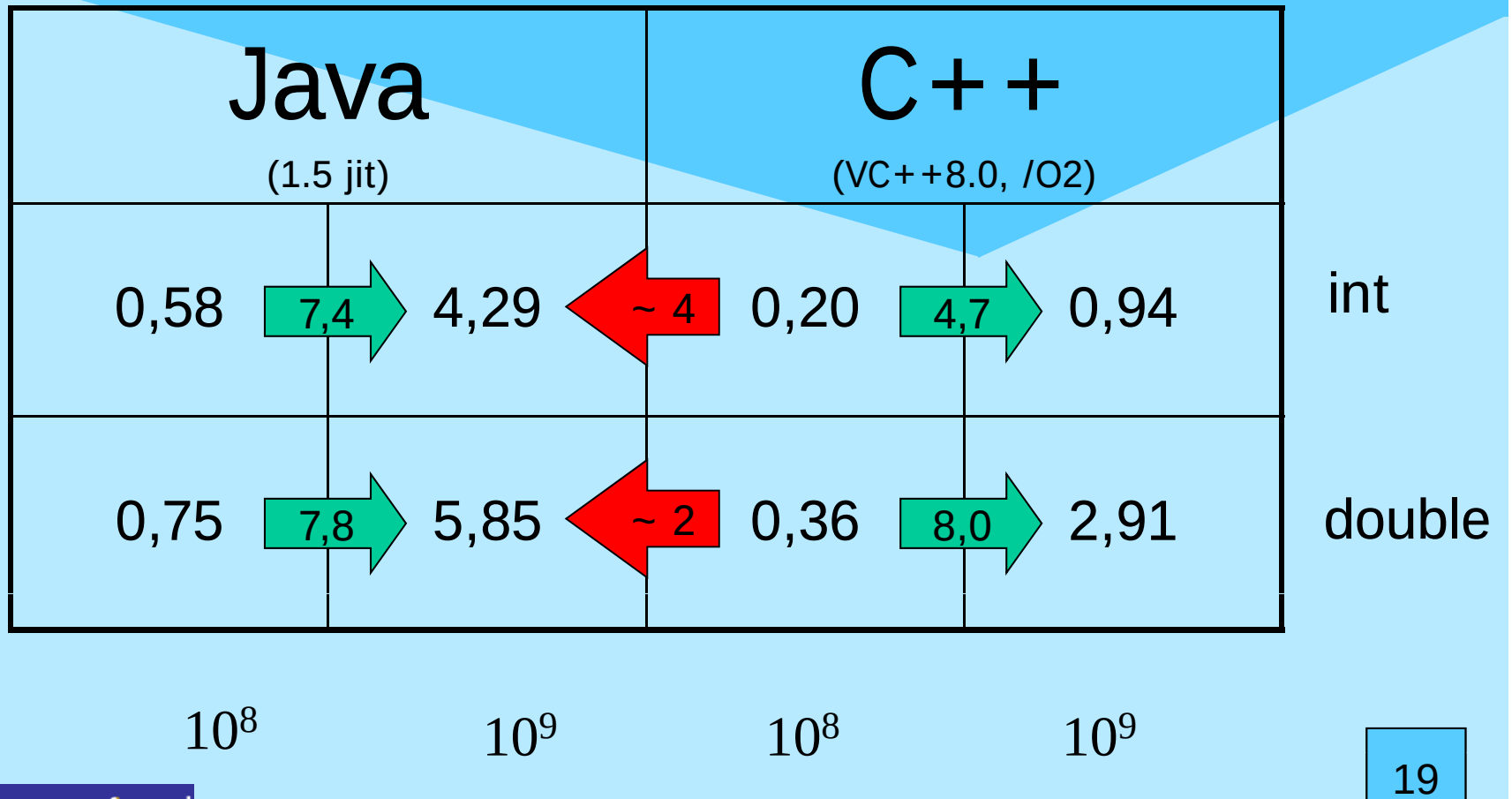
Der zweite Blick: Effizienz

Laufzeiten (Solaris 5.8, UltraSparc [8x]1050 MHz)



Der zweite Blick: Effizienz

Laufzeiten (Win XP, Pentium M 1,8 GHz)



C++ Historie

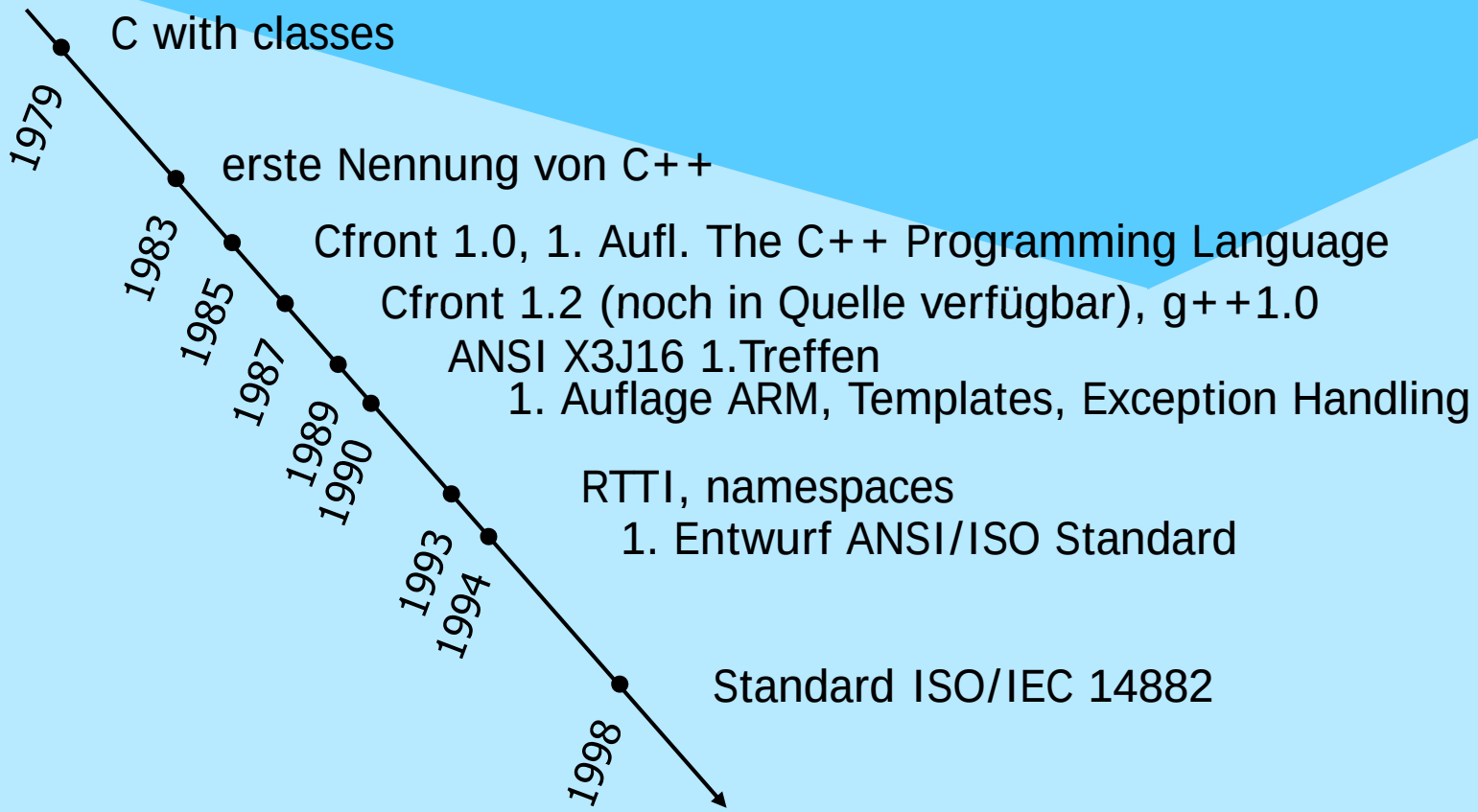
- Bjarne Stroustrup Ph.D. Arbeit 1978/79 an der Universität Cambridge: „Alternative Organisationsmöglichkeiten der Systemsoftware in verteilten Systemen“
- erste Implementation in Simula auf IBM360 (Simula67, NCC Oslo)
- Stroustrup: *„Die Entwicklung des Simulators war das reinste Vergnügen, da Simula nahezu ideal für diesen Zweck erschien. Besonders beeindruckt wurde ich durch die Art, in der die Konzepte der Sprache mich beim Überdenken der Probleme meiner Anwendung unterstützten. Das Konzept der Klassen gestattete mir, die Konzepte meiner Anwendung direkt einzelnen Sprachkonstrukten zuzuordnen. So erhielt ich Programmcode, der in seiner Lesbarkeit allen Programmen anderer Sprachen überlegen war, die ich bisher gesehen hatte.“*
- Simula - Compiler damals mit extrem schlechten Laufzeiteigenschaften

C++ Historie

- S.: *„Um das Projekt nicht gänzlich abzubrechen - und Cambridge ohne Ph.D. zu verlassen -, schrieb ich den Simulator ein zweites Mal in BCPL ... . Die Erfahrungen, die ich während des Entwickelns und der Fehlersuche in BCPL sammelte, waren grauenerregend.“*

- erste Ideen zu C++ im Kontext von Untersuchungen Lastverteilung in UNIX-Netzen bei den Bell Labs Murray Hill, New Jersey: Stroustrup: *„Ende 1979 hatte ich einen lauffähigen Präprozessor mit dem Namen Cpre geschrieben, der C um Simula-ähnliche Klassen erweiterte.“ -> C with classes*

C++ Historie


1979
C with classes
1983
erste Nennung von C++
1985
Cfront 1.0, 1. Aufl. The C++ Programming Language
1987
Cfront 1.2 (noch in Quelle verfügbar), g++1.0
1989
ANSI X3J16 1.Treffen
1990
1. Auflage ARM, Templates, Exception Handling
1993
RTTI, namespaces
1994
1. Entwurf ANSI/ISO Standard
1998
Standard ISO/IEC 14882
?
Standard C++0x

Java vs. C++: Different by Design

- Java
  - starke Anlehnung an C++
  - *Deployment* Schema: Interpretation
  - OO ist (nahezu) zwingend
  - primäres Kriterium: Komfort

    diverse (und zumeist nicht abschaltbare) implizite *Overheads* zu Lasten der Effizienz
    - Prüfung von Feldgrenzen
    - *Reflection*
    - *Garbage Collection*
    - *Objects by Reference* Semantik

Java vs. C++: Different by Design

C++

- starke Anlehnung an C
- *Deployment* Schema: Compilation
- OO ist möglich, nicht zwingend
- primäres Kriterium: Effizienz

  keinerlei impliziter *Overhead* zu Lasten der Effizienz
  - keine Prüfung von Feldgrenzen
  - (fast) kein Laufzeitabbild von Klassen
  - keine automatische Speicherverwaltung
  - *Objects by Value* Semantik

Objects by Reference

Java:

- Variablen vom Klassentyp sind IMMER Referenzen

```
X x;  // implizit == null !!
x = new X();
X y = x;  // ein Objekt mit zwei Referenzen!!!
```

- Objekte werden IMMER dynamisch (auf dem Heap) erzeugt

Objects by Reference

```
class A {
        private int i;
        public void foo() {
             i++;
        }
        public void out() {
             System.out.print(i);
        }
        public A() {
             i=0;
        }
        public static void bar(A a){
             a.foo();
        }
```

```
        public static void
        main(String s[]) {
           A a1 = new A();
           A a2 = a1;

           a1.foo();
           a2.foo();

           a1.out();
           a2.out();

           bar(a2);

           a1.out();
           a2.out();
        }
}
```

```
$ javac A.java
$ java A
????
```

Objects by Reference

```
class A {
        private int i;
        public void foo() {
             i++;
        }
        public void out() {
             System.out.print(i);
        }
        public A() {
             i=0;
        }
        public static void bar(A a){
             a.foo();
        }
```

```
        public static void
        main(String s[]) {
           A a1 = new A();
           A a2 = a1;

           a1.foo();
           a2.foo();

           a1.out();
           a2.out();

           bar(a2);

           a1.out();
           a2.out();
        }
}
```

```
$ javac A.java
$ java A
2233$
```

Objects by Value

C++:

- Variablen vom Klassentyp sind (primär) Werte

```
X x; // ein Objekt !
X y = x;  // ein weiteres Objekt als Kopie des ersten!!!
```

- Objekte können global, (Stack-) lokal und dynamisch erzeugt werden

- Es gibt auch Objektreferenzen und -Zeiger

Objects by Value

```
#include <iostream>

class A {
        int i;
public:
        void foo() {
            i++;
        }
        void out() {
            std::cout << i;
        }
        A() {
            i=0;
        }
        static void bar(A a) {
                a.foo();
        }
};
```

```
int main()
{
        A a1=A();
        A a2=a1;

        a1.foo();
        a2.foo();

        a1.out();
        a2.out();

        A::bar(a2);

        a1.out();
        a2.out();
}
```

```
$ g++ -o a a.cc
$ a
????
```

Objects by Value

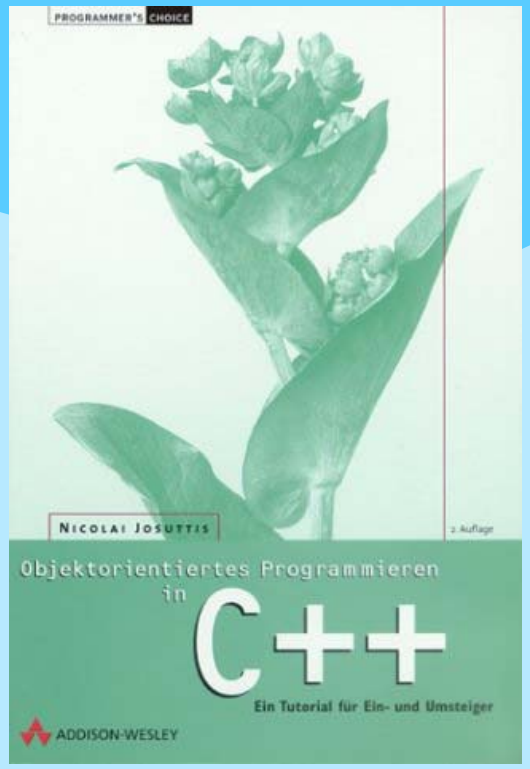
```
#include <iostream>

class A {
        int i;
public:
        void foo() {
            i++;
        }
        void out() {
            std::cout << i;
        }
        A() {
            i=0;
        }
        static void bar(A a) {
                a.foo();
        }
};
```

```
int main()
{
        A a1=A();
        A a2=a1;

        a1.foo();
        a2.foo();

        a1.out();
        a2.out();

        A::bar(a2);

        a1.out();
        a2.out();
}
```

```
$ g++ -o a a.cc
$ a
1111$
```

1. Elementares C++

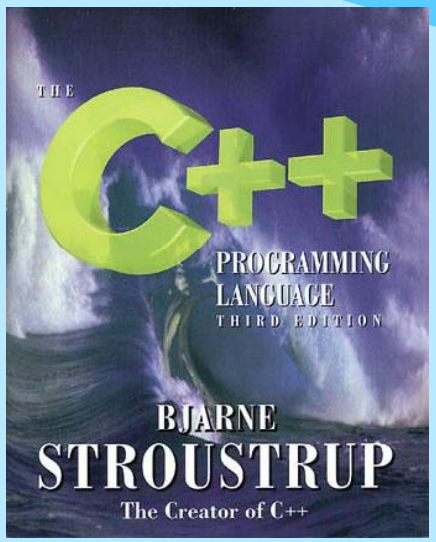
BTW: C++ Literaturempfehlungen *beginner level*

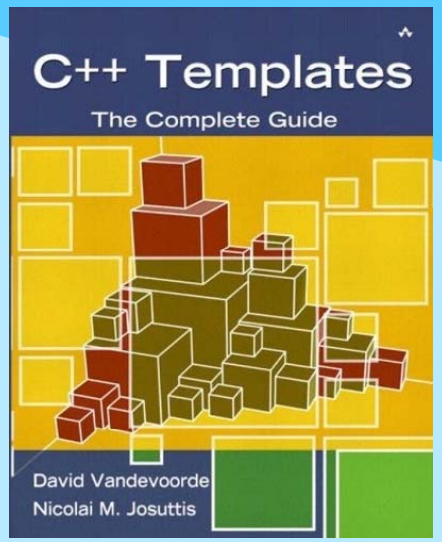


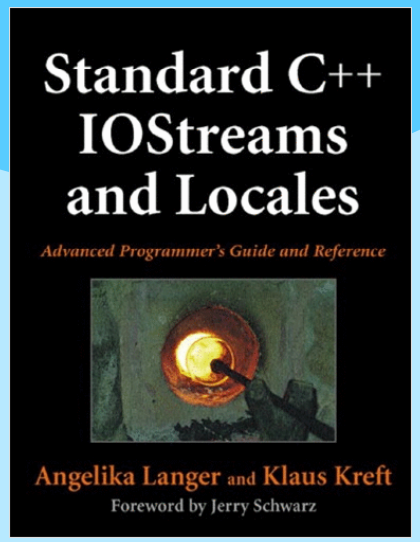
Kaufen: „Bafög-Ausgabe“ 19,95 €

Ausleihen: im Handel leider vergriffen ☹

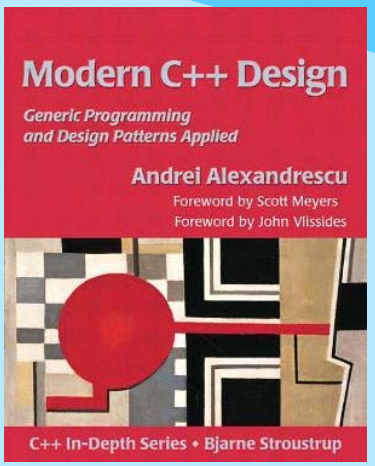
BTW: C++ Literaturempfehlungen *expert level*

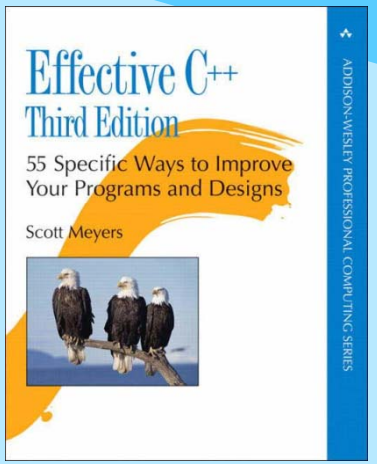


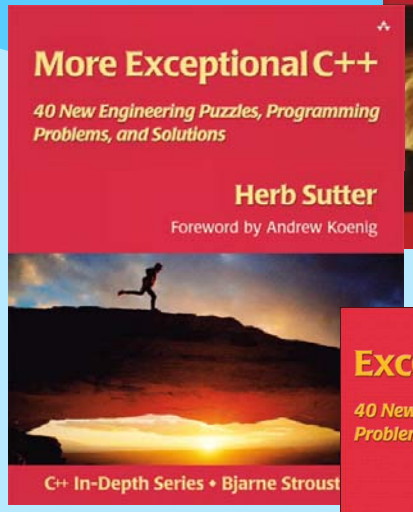


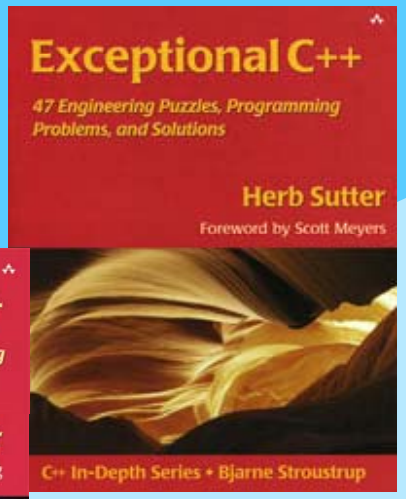


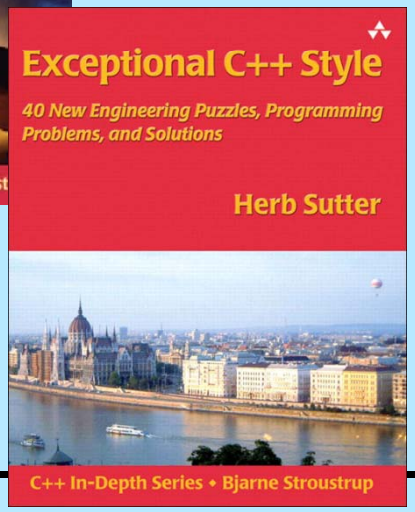
BTW: C++ Literaturempfehlungen *guru level*











1. Elementares C++

1.1. Lexik

- Kommentare wie Java

```
//  this line
/*   no
     nesting
     allowed */
```

- kein spezielles doc-Kommentarformat, aber von einigen tools unterstützt (z.b. doxygen)

- *free format:* whitespaces (space, newline, comment) beliebig zur Trennung von Token: `int a;  <---> inta;`

1.1. Lexik

Schlüsselwörter:

```
asm do if return typedef                (C: 32) (Δ C++: 31)
auto double inline short typeid
bool dynamic_cast int signed typename
break else long sizeof union
case enum mutable static unsigned
catch explicit namespace static_cast using
char export new struct virtual
class extern operator switch void
const false private template volatile
const cast float protected this wchar t
continue for public throw while
default friend register true
delete goto reinterpret_cast try
```

1.1. Lexik

Operatoren:

`+ - * / % < <= > >= == != && || !` wie üblich (Java)

`<< >> & ^ | ~` bitweise left-, right-Shift, and, or, xor, Komplement

`= *= /= %= += -= <<= >>= &= ^= |=` x*?*=y <--> x = x *?* y

`++ --` als Prefix und Postfix

`sizeof(Typname)` oder

`sizeof(Expression)` **oder** Größe in Bytes

`sizeof Expression`

`,` Kommaoperator: Gruppierung von Ausdrücken, der letzte Teilausdruck legt den Wert des Gesamtausdruchs fest!

ACHTUNG: foo(1,2,3) vs. foo((1,2,3))

1.1. Lexik

Bezeichner: wie in Java (incl. _ als Buchstabe)
Groß-/Kleinschreibung wird unterschieden

übliche Konventionen:

sog. Macros durchgängig groß: #define A_MACRO
nutzerdef. Typnamen beginnen groß: MyType
Variablen druchweg klein: MyType myvar;

1.2. Datentypen

build-in Typen:

```
char, int, short (int), long (int), (un)(signed)(long)
  int, void, float, double, bool(!)
```

- ACHTUNG: long ist kein eigener Typ, sondern Kürzel für long int
- ACHTUNG: es gibt KEINE Vorgaben zur Größe von Variablen dieser Typen: 1==sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
  sizeof(float)<=sizeof(double)

- literale Werte dieser Typen nach den »üblichen« Regeln:

```
'A'     '\n'   '\\'   '\000'   '\0x12'
123     -45    0123   0x123    0XCDEF
12U     23u    123L   0l       0x12345L
1.234   .5f    45.    1.1e12   -2.3E-5
true    false
```

1.2. Datentypen

Enumerations: Aufzählungstypen == benannte Werte

```
enum Season {spring, sommer, fall, winter};

Season now = spring; ... if (now == winter) ...
```

Felder: mehrere Objekte (Variablen) direkt hintereinander im Speicher,
ein Feld ist selbst KEIN Objekt, --> KEIN Längenattribut

```
int f [n];
```

f zeigt auf den Beginn eines Feldes von n int's, n muss eine vom Compiler errechenbare Konstante sein !

1.2. Datentypen (Felder)

```
int p[];
```

nur in Argumentlisten von Funktionen:
int-Feld unbekannter == beliebiger Länge, Größeninformation ist separat bereitzustellen

```
double m[3][4];
```

12 doubles hintereinander !

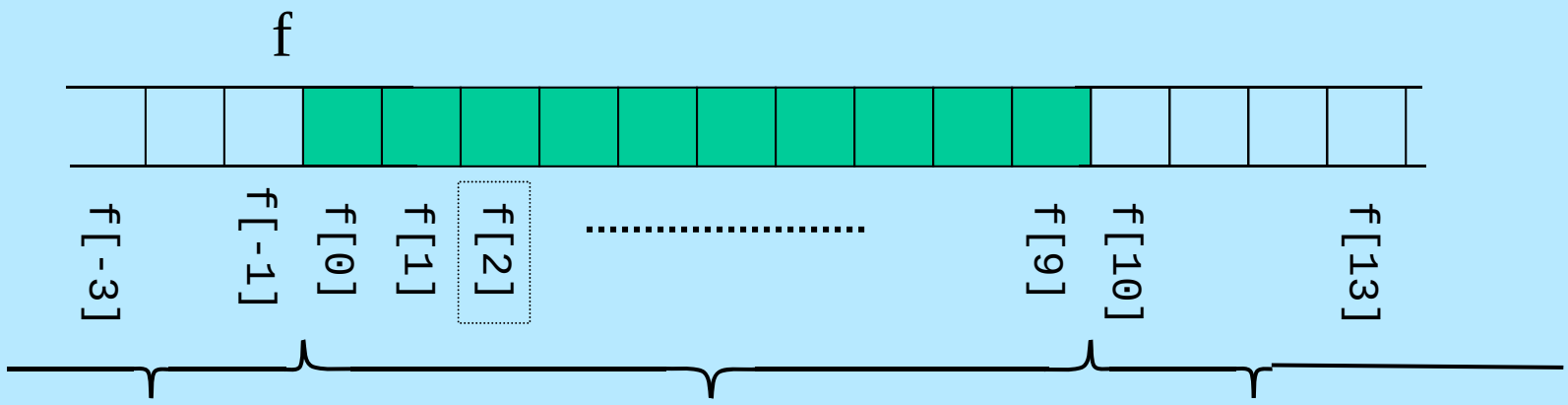
Typedefs: Synonyme für (u.U.) komplexe Typkonstrukte

```
typedef double V4[4];
V4 m[3]; // entspricht obigem Feld
```

1.2. Datentypen (Felder)

es gibt keine Prüfung auf Einhaltung der Feldgrenzen bei Zugriffen:

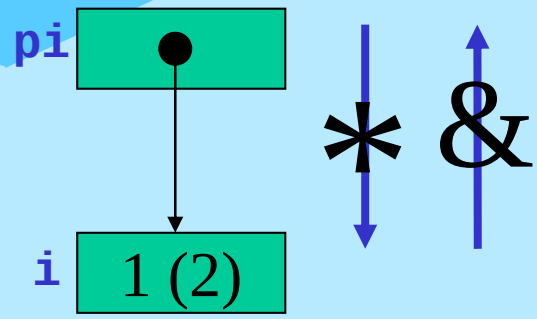
```
T f [10];
```



1.2. Datentypen

Zeiger: Indirektion per Speicheradresse!

```
int* pi; // ein u.U. NICHT initialisierter Zeiger

int i=1;
pi = & i; // Adressoperator !
```



Zeiger sind vollständig typisiert:

```
double x;
// ERROR: pi = & x;
```

Umkehroperation zu & ist die sog. Dereferenzierung:

```
*pi = 2;
```

1.2. Datentypen (Zeiger)

```
int* pi = new int; // ein neues anonymes int auf dem Heap
                   // pi ist (bislang) der einzig Zugang
                   // no more C: int* pi = malloc(sizeof(int));
int* ap = pi; // 2 Verweise, 1 Objekt !

pi = 0; // ausgezeichneter Zeigerwert <==> KEIN Objekt

ap = 0; // letzte Referenz weg: KEINE garbage collection
        // sondern ein memory leak
```

daher zuvor:

```
delete ap; // kein leak !
           // no more C: free(pi);
```

1.2. Datentypen (Zeiger)

ACHTUNG: nach `delete zeiger;` ist u.U. in `zeiger` immer noch die gleiche Adresse enthalten, jeder Zugriff darüber ist jedoch undefiniert !
Empfehlung: `delete pi; pi=0;`

auch Felder können dynamisch erzeugt werden:

```
int* pf = new int [100]; // pf zeigt auf erstes von 100 int's
```

Zeiger auf Felder sind mit `delete[]` zu deallokieren:

```
delete[] pf; pf=0;
```

1.2. Datentypen (Zeiger)

Zeiger auf Klassentypen (~ Java-Objektsemantik)

```
class X {
public:
  X() {std::cout<<"hi\n";}
  ~X(){std::cout<<"bye\n";}
};

void foo() { X* px = new X; } // hi & leak
void bar() { X* px = new X; delete px; } // hi & bye
```

auch: `new X();` möglich aber nicht zu empfehlen !

In jeder (nicht static) Memberfunktion ist `this` ein Zeiger auf das Objekt, an dem der Aufruf der Funktion erfolgte (anders als in Java !)

1.2. Datentypen (Zeiger)

From: Nicola Musatti <nicola.musatti@gmail.com>
Newsgroups: comp.lang.c++.moderated
Subject: Re: Why doesn't new return a reference?

> Hi,
> It's my understanding that new will never return NULL in modern
> implementations, so then why does it return a pointer? Is it just a
> historical artifact?

Because returning a reference would be misleading. References are usually bound to objects over whose ownership and lifetime you have no control. On the other hand you should definitely deal with the ownership and lifetime of the objects returned by new()!

What you propose would only make sense if garbage collection was the only way to deal with dynamic memory. Luckily it is not.

Cheers,
Nicola Musatti

1.2. Datentypen (Zeiger)

From: "Andrew Koenig" <ark@acm.org>
Newsgroups: comp.lang.c++.moderated
Subject: Re: Why doesn't new return a reference?
...
> It's my understanding that new will never return NULL in modern
> implementations, so then why does it return a pointer?

One common usage for "new" is to allocate an array with a size that is known only at run time. Because the size is not known at compile time, it is impossible for "new" to return a reference to the array as its result, because the size of an array is part of its type and types are always fixed during compilation.

Therefore, instead of returning a reference to the array, "new" returns a pointer to the initial element. This pointer can be used as an iterator to access the elements of the array. References do not support the iterator operations; hence are not as useful as pointers in this context.
--

1.2. Datentypen (Zeiger)

Java - **new** vs. C++ - **new**

```
class X {}

class Main {
        public static void main(String s[]) {
                // X x = new X; nicht ohne leere Parameterliste
                X x = new X();
                // int i = new int; new nur für Klassen erlaubt
                // int i = new int(); auch so nicht
                int i[] = new int[10]; // Felder sind Objekte
        }
}
```

1.2. Datentypen (Zeiger)

Java - **new** vs. C++ - **new**

```
class X {};

int main() {
                X *x1 = new X;          // besser so
                X *x2 = new X();        // als so
                int *i1 = new int;      // di
                int *i2 = new int();    //   to
                // int i[] = new int[10]; so nicht:
                // Feldvariablen sind Konstanten & die Größe
                // von i ist unbestimmt
                int *i = new int[10]; // ein Zeiger kann
                // eins oder viele referenzieren !
}
```

Warum besser keine Klammern bei parameterlosen Konstruktorrufen ?

```
T* pt = new T(); // ok

T t = T(); // ok, aber redundant
T t ();    // auch ok, aber kein Objekt vom Typ T !
```

?

```
void foo(); // Funktionsdeklaration !!!
T    t  (); //   dito

T t;
```

1.2. Datentypen (Zeiger)

Felder sind konstante Zeiger:

```
T someTs [30];

T* pt = someTs;
T* qt = &someTs[0];
```

```
using std::cout;...
cout<<1["]<<2["]<<endl;
Korrektes C++ ?
wenn ja, was wird ausgegeben ?
```

`pt[i]` ist nur eine abkürzende Notation von `*(pt+i)`

Die Zeigerarithmetik erfolgt modulo `sizeof(T)`

`pt` ist ein Zeiger auf's erste `T` im Feld

`pt+1` ist ein Zeiger auf's zweite `T` im Feld ...

1.2. Datentypen

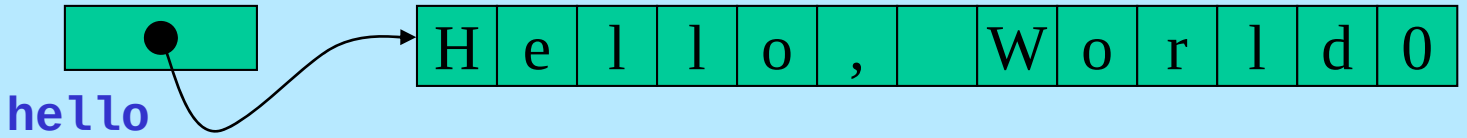
Zeichenketten:

Zeichenkettenliterale wie »üblich«

```
"eine Zeichenkette\nmit Doppelapostroph \" und Backslash \\"
```

werden als 0-terminierte **char**-Felder realisiert !

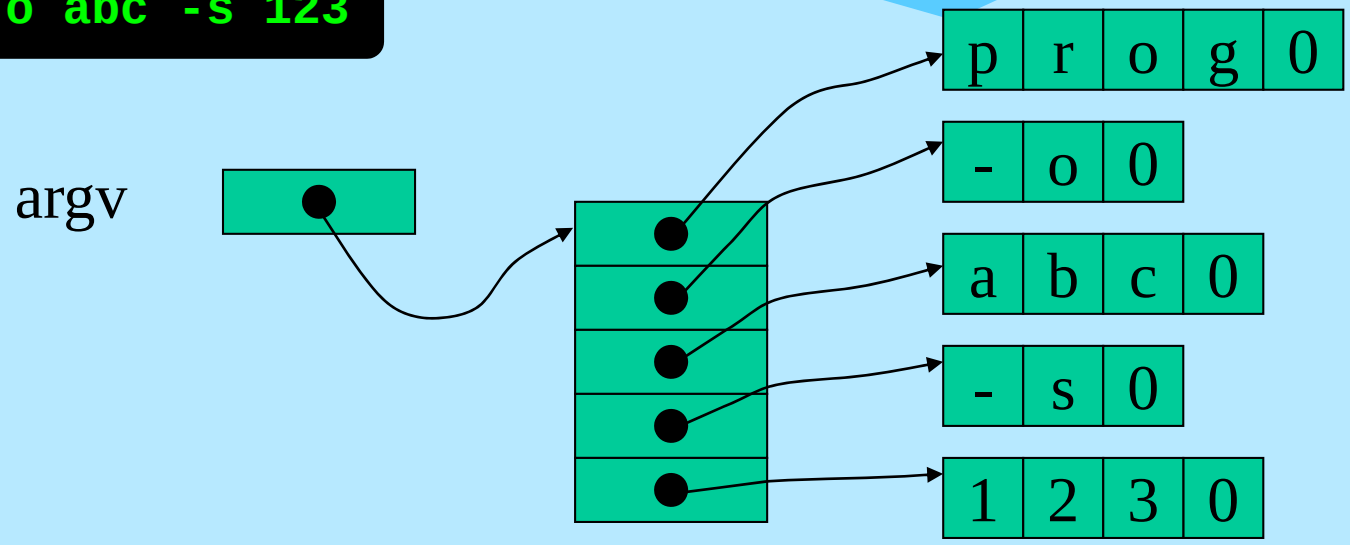
```
char* hello = "Hello, World";
```



1.2. Datentypen (Zeichenketten)

```
int main (int argc, char* argv[]) // bzw.
int main (int argc, char** argv)
```

```
$ prog -o abc -s 123
```



1.2. Datentypen (Zeichenketten)

Damit ist bereits der Umgang mit Zeichenketten implizit mit allen Problemen der Zeiger belastet (und zusätzlich mit allen *buffer overflow* Problemen bei Operationen auf Zeichenfeldern)

Außerdem sind die möglichen Operationen auf **`char[]`** C-legacy (<cstring>) und primitiv, z.B. **`strcpy`** == Kopieren von Zeichenketten

etwa:

```
void stringcopy (const char* source, char* dest)
{ while (*dest++=*source++); }
```

1.2. Datentypen (std::string)

AUSWEG: Datentyp **std::string** (<string>)

- eine Standardklasse zur Verarbeitung von Strings
- etwa auf dem Niveau von java.lang.String mit der
- Möglichkeit der Initialisierung aus C-Strings

```
std::string vorname = "bjarne";
```

- und einer Vielzahl von Operationen (a la Java):

```
std::string nachname = "Stroustrup";
```

1.2. Datentypen (std::string)

```
std::string name; // noch leer !

vorname[0]='B'; // unchecked !
vorname.at(0) = 'B'; // checked
name = vorname + " " + nachname;
if (name != "")
  std::cout << name << std::endl;
int l = name.length(); // ohne 0-Byte !

"hallo" + ", World\n"; // ERROR
string("hallo") + ", World\n"; // OK

const char* cstring = name.c_str();
```

... Vergleich, Suche, I/O ... ↳ http://www.dinkumware.com

1.2. Datentypen

Referenztypen:

Eine Neuerung gegenüber C, Aliasnamen für Objekte mit Referenzsemantik ähnlich zur primären Objektsemantik von Java, aber

– für alle Typen (incl. *build-in* Typen)

– es gibt KEINE 'Nullreferenz'

in Anlehnung an die Syntax von Zeigervereinbarungen

```
int i=42;
// int& ri;
// ERROR: Referenzen MÜSSEN initialisiert werden
int& ri = i; // i alias ri
```

1.2. Datentypen

Konstantentypen:

Ein Typ **T** wird durch den 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

1. die (nachprüfbare) Zusicherung gibt, dieses Argument NICHT zu verändern
2. beim Aufruf für das Argument auch konstante Objekte benutzt werden dürfen (was für non-const nicht erlaubt ist, weil ja die Funktion keine Zusicherung gegeben hat und daher ...)

```
const double pi=3.1415926; double someMathFkt(double);
const double x = someMathFkt(pi);  // call by value !
```

1.2. Datentypen (Konstantentypen)

Konstante Objekte müssen initialisiert werden (weil eine spätere Zuweisung nicht erlaubt ist)

konstante Objekte können auch über Zeiger nicht verändert werden, weil die Adresse einer `const T` Variablen vom Typ `const T*` ist

```
double* dp = &pi; // ERROR
*dp = 33.3;

const double* cdp = &pi;
*cdp = 33.3; // ERROR
```

1.2. Datentypen (Konstantentypen)

bei Zeigern ist wohl zu unterscheiden zwischen der *constness* des Zeigers selbst

```
int * const constant_pointer = &someint;
```

und der *constness* des referenzierten Objektes (Feldes)

```
const int * pointer_to_constant;
```

```
const int * const constant_pointer_to_constant = …;
```

1.2. Datentypen (Konstantentypen)

Referenzen (selbst) sind implizit const, es gibt jedoch Referenzen auf Konstantentypen

Wichtigste Anwendung: *call by reference in-parameter*

```
T t;
void foo(T& pt)
{
        pt.change();
}
foo(t); // call by reference: t itself changes
const T ct;
foo(ct); // ERROR
```

1.2. Datentypen (Konstantentypen)

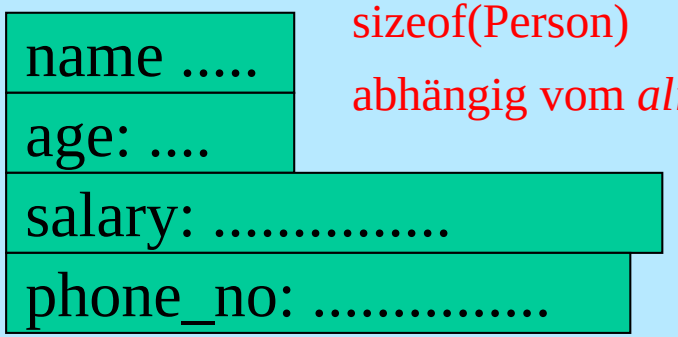
```
void foo(const T& pct)
{
        // pct.change(); ERROR
        pct.read_only();
}

const T ct;
foo(ct); // OK

class X{ public: void foo() const; void bar(); };
X x; const X cx; x.foo(); x.bar();
cx.foo(); /* OK */  cx.bar() // ERROR !
```

1.2. Datentypen

Strukturtypen (a la C):

heterogene Wertekombinationen unter einem Typnamen

```
struct Person {
       std::string name;
       int age;
       double salary;
       long phone_no;
} p;
```



sizeof(Person)

abhängig vom *alignment* !

1.2. Datentypen (Strukturtypen)

```
p.name = "Willibald Wusel";
```

Kombination mit Zeigern (dynamische Strukturobjekte)

```
Person* aNewPerson = new Person;
aNewPerson->age = 32;
// short hand for:
(* aNewPerson).age = 32;
```

Kombination mit Referenzen

```
void raise_salary (Person &p, int percentage) {
        p.salary *= 1 + percentage/100.0; // ? why .0 ?
}
raise_salary (p, 3);
```

1.2. Datentypen (Strukturtypen)

Strukturen sind in C++ *de facto* Klassen ohne Memberfunktionen und öffentlichem Zugriff auf alle Memberdaten!

```
struct Person {
        std::string name;
        int age;
        double salary;
        long phone_no;
};
```

```
class Person {
public:
        std::string name;
        int age;
        double salary;
        long phone_no;
};
```

1.2. Datentypen (Strukturtypen - Unions)

Es gibt noch die C-Variante, bei der alle Bestandteile eines solchen zusammengesetzten Typs an der gleichen Adresse (am Objektanfang) beginnen -> sog. *Unions* (spielen in C++ eine untergeordnete Rolle !)

```
union HACK {
        double d; // double precision ieee
        struct {
                unsigned :1,
                e:11;
        } s;
};
int NaN(double x) {
        HACK h; h.d = x; return h.s.e == 0x7ff;
}
```

1.3. Ausdrücke

ähnlich zu Java:

- Literale und Variablen `1.234` `"Huhh..."` `i`
- Anwendung von Operatoren auf Operanden
  `x+1` `std::cout<<4` `x=y` `foo(3,bar(7),&a)` `p->name[0]`

ABER:

- Reihenfolge der Berechnung **undefiniert** (bis auf && und ||) !
- jeder Ausdruck liefert einen Wert (ggf. den leeren Wert bei Funktionen mit Rückgabetyp **`void`**), ein nicht-leerer Wert kann, muss aber nicht weiterverwendet werden (wie in Java)
- ein Ausdruck wird durch nachfolgendes Semikolon zu einer Anweisung !

```
f(3); // Ergebnis wird ignoriert
int k=f(4); // Ergebnis wird weiterverwendet

int main() { 42; } // KORREKTES C++ ???
```

1.4. Funktionen

- Memberfunktionen von Klassen oder außerhalb von Klassen (global, bzw. namespace-lokal)
- Unterscheidung in Deklaration (Angabe der Signatur) und Definition !

```
void foo(int); // optional: Parameternamen
void foo(int x) { .... }
class X { public: int foo(); // int foo(void) !
                  X& bar() {return *this; }
                  X(int); };
X::X(int i){ .... }
```

- jede Definition ist auch eine Deklaration
- Jede Funktion muss deklariert sein, bevor sie verwendet wird; Deklaration einer Memberfunktionen wirkt ab Klassenbeginn

1.4. Funktionen

- mehrfache Deklarationen sind erlaubt
- für jede Funktion muss es (GENAU) eine Definition geben, ansonsten *linker error [the one definition rule ODR]*
- Deklarationen in `*.h` - Files, Definitionen in `*.cpp` - Files:

```
// foo.h:
int foo(int,int);
```

```
// foo.cpp
#include "foo.h"
int foo(int a, int b)
{return a+b;}
```

```
// prog.cpp:
#include "foo.h"
int main() { foo(123,456); }
```

1.4. Funktionen

Es gibt (anders als in Java) keine vollständige Analyse des Kontrollflusses:

Java

```
class flow {
  int foo(int i){
   if (i<0) return 42;
   if (i>=0) return 24;
  } // ERROR: Missing return statement
}
```

C++

```
int foo(int i){
   if (i<0) return 42;
   if (i>=0) return 24;
} // OK !
```

ABER: Verlassen einer (non-void) Funktion ohne Rückgabewert:
undefined behaviour

1.4. Funktionen

– können **`static`** sein:
  1. Memberfunktionen: Klassenmethoden wie in Java (kein **`this`**)
  2. globale Funktionen: *file scope*

– können **`static`** (lokale) Daten enthalten:

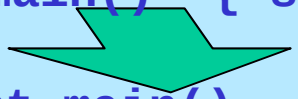
```
int foo() { static int i=2; return i*=i; }
int main(){
for (int i=0; i<3; ++i) std::cout<<foo(); // 416256
}
```

ACHTUNG: **`std::cout<<foo()<<foo()<<foo();`** **???**

1.4. Funktionen

– können **inline** sein: kein Aufruf, sondern (Seiteneffekt-freie und typgerechte) Substitution auf Quelltextebene:

```
inline int square(int i){return i*i;}
int main()  { std::cout << square(4); }
```



inline substitution

```
int main()  { std::cout << 4*4; } // u.U. sogar 16
```

– Ziel: Effizienz, auch wenn *call overhead* > 'Nutzeffekt' der Funktion

– Memberfunktionen, die im Klassenkörper definiert werden, sind implizit **inline** ! (gute Kandidaten, weil meist kurz)

Tony Hoare: "Premature optimization is the root of all evil !„

siehe auch
www.ddj.com (search for: *inline redux*) und www.gotw.ca/gotw/033.htm

1.4. Funktionen

– können *default arguments* haben: ein Endstück der Argumentliste einer Deklaration mit Wertevorgaben

```
int atoi (const char* string, int base = 10);
// ascii to int on radix base
atoi ("110"); // --> atoi("110", 10) --> 110
atoi ("110", 2); // --> atoi("110", 2) --> 6
```

Vorsicht Falle 1: `void foo(char*=0);`

`void foo(char* =0);`

1.4. Funktionen

Vorsicht Falle 2:

```
int f(int);
int f(int, int=0);
f (1); // mehrdeutig: f(1) oder f(1,0)
```

- variable Argumentlisten a la `printf` in C++: `...` *ellipsis*

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

`extern "C"` ist eine sog. *linkage* Direktive: hier kein *name mangling*

1.4. Funktionen

– können überladen werden: gleicher Name, unterscheidbare Signatur (Rückgabetyp spielt KEINE Rolle!)

name mangling

```
class X{ public:
   X();                        __1X
   X(int);                     __1Xi
   int foo();                  foo__1X
   int foo() const;            foo__C1X
   int foo(const X&);          foo__1XRC1X
};
int foo(int);                  foo__Fi
double foo(double);            foo__Fd
void foo(char*, int);          foo__Fpci
int printf(const char*, ...);    printf__FPCce
```

```
$ g++ -c foo.cc
$ nm foo.o
00000000 W __1X
00000000 W __1Xi
....
$ nm foo.o | c++filt
00000000 W
X::X(void)
00000000 W X::X(int)
....
```

1.5. Strukturierte Anweisungen

(fast) wie in Java:

while, do, for, if, switch, break, continue, return

Deklaration in Blöcken sind Anweisungen: Deklaration von Objekten *am Ort des Geschehens* (wie in Java)

```
void foo()
{
      int i=0;
      bar(i); ....
      int j=3;
      bar(j); ....
}
```

Vorsicht Falle:

```
if (x=1) ....
```

1.5. Strukturierte Anweisungen

echtes Lokalitätsprinzip lokaler Blöcke in C++ (nicht in Java):

```
class varscope {
  static void bar(int i){}
  void foo() {
        for (int i=0; i<10; ++i)
                bar(i);
        for (int i=0; i<10; ++i)
                bar(i);
        int i=123;
        bar(i);
        {
                int i=234;
// varscope.java:12: Variable 'i' is already defined in this method.
                bar(i);
        }
  }
}
```

1.5. Strukturierte Anweisungen

echtes Lokalitätsprinzip lokaler Blöcke in C++ (nicht in Java):

```
int i = 666;
static void bar(int i){}
void foo(){
        for (int i=0; i<10; ++i)
                bar(i);
        for (int i=0; i<10; ++i)
                bar(i);
        int i=123;
        bar(i);
        {
                int i=234;  // hides all outer i's
                bar(i);
                bar(::i);   // global i
        } // i==123 !
}
```

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

Wo und wie lange leben Objekte ?

	globale Objekte	lokale Objekte	dynamische Objekte
entstehen durch ...	globale Objektvereinbarung: `T o;`	blocklokale Objektvereinbarung: `{ .. T o; ..}`	durch expliziten Aufruf von new: `T*op=new T[N];`
Objekte sind initialisiert	*builtin*-Typen: ja, auf 0 Klassentypen: durch Aufruf eines Konstruktors (*	*builtin*-Typen: nein ! Klassentypen: durch Aufruf eines Konstruktors (*	*builtin*-Typen: nein ! Klassentypen: durch Aufruf eines Konstruktors (*
werden vernichtet ...	automatisch nach (!) Programmende	automatisch beim Verlassen des Blockes Sonderfall: ***temporaries*** (**	durch expliziten Aufruf von delete: `delete[] pi;`
residieren im ...	globalen Datenbereich (bereits vom Compiler geplant und vor Programmstart bereitgestellt)	***Stack*** (dehnt sich dynamisch und sequentiell aus)	***Heap*** (dehnt sich dynamisch und nicht sequentiell aus)

(* u.U. ohne nutzerspezifische Initialisierung (s. default ctor)
(am nächsten *sequence point* (typischerweise ;)**

1.5. Strukturierte Anweisungen

Switch-Anweisung (C++): **`switch`** (*expression*) **`statement`**

statement i.allg. strukturiert mittels case: / default: aber mit mehr Freiheiten als in Java

Beispiel: *Duff's Device*

(Tom Duff 1983)

```
void send
(register short *to,
register short *from,
register count)
{ do *to = *from++; while(--count>0); }

// to: some device register
```

```
void send (register short *to, register short *from,
           register count) {
        register n = (count+7)/8;
        switch (count%8){
                case 0:  do{      *to = *from++;
                case 7:           *to = *from++;
                case 6:           *to = *from++;
                case 5:           *to = *from++;
                case 4:           *to = *from++;
                case 3:           *to = *from++;
                case 2:           *to = *from++;
                case 1:           *to = *from++;
                        } while(--n > 0);
        }
}
```

1.5. Strukturierte Anweisungen

Switch-Anweisung (C++):

Initialisierungen dürfen nicht 'übersprungen' werden:

```
switch (i) {
       int v1 = 2;  // ERROR: jump past initialized variable
case 1:
       int v2 = 3;
       // ....
case 2:
       if (v2 == 7) // ERROR: jump past initialized variable
       // ....
}
```

1.5. *Un : -)* Strukturierte Anweisung

goto - *the don't use statement*

```
loop:
// ....
goto loop;
```

```
goto skip:
// ....
skip: //....
```

Initialisierungen dürfen auch nicht 'übersprungen' werden:

1.5. Strukturierte Anweisungen

Exception Handling : syntaktisch wie in Java (kein `finally`)

```
try {
        // things that may throw or not
}
catch ( Exception1 e ) {
        // handle e
}
catch ( Exception2 e ) {
        // handle e
} ......
```

bei Auftreten einer Ausnahme wird der `try`-Block verlassen und zu einem passenden (ggf. übergeordneten) `catch`-Block verzweigt, zuvor werden alle Destruktoren lokaler Objekte gerufen, die erfolgreich konstruiert wurden !

1.5. Strukturierte Anweisungen: Exception Handling

```
#include <iostream>

using std::cout; using std::endl;

class X { public:
                X(int=0){cout<<"X("<<i<<")\n";}
                ~X(){cout<<"~X()\n";}
};

void foo(int i) {
        try {   X local;
                if (i==1) throw "ooops";
                else if (i==2) throw 42;
        }
        catch (const char* why) {
                cout<<why<<endl;
        }
}
```

1.5. Strukturierte Anweisungen: Exception Handling

```
//... cont.
int main() {
        try {
                X x1(1);
                foo(1);
                X x2(2);
                foo(2);
        }
        catch (int r) {
                cout<<"exception: code "<<r<<endl;
        }
        catch (...) {
                cout<<"something thrown: don't know what\n";
        }
}
```

```
X(1)
X(0)
~X()
ooops
X(2)
X(0)
~X()
~X()
~X()
exception: code 42
```

1.5. Strukturierte Anweisungen: Exception Handling

- Exceptions sind Objekte beliebigen Typs
- *stack unwinding* ruft ggf. Destruktoren
- in einem `catch`-Block kann eine Exception mittels `throw;` 're-thrown' werden

```
....  catch (...) {
                  void handleAll(); // proto
....              handleAll();
....  }

void handleAll() {
   try { throw; }
   catch (double x) { cout << x << endl; } // z.B.
}
```

1.5. Strukturierte Anweisungen: Exception Handling

- wird eine Exception nirgends 'gefangen', so endet dass Programm durch aufruf von `std::terminate()`(* (dies ruft wiederum `std::abort()`)
- mittels `std::set_terminate()` kann man dieses Verhalten ändern:

Prototyp `void (*set_terminate(void (*term_handler)())) ();`

?????

oder leichter nachvollziehbar:

```
typedef void (*TH)();
TH set_terminate(TH);
```

(* es ist *implementation-defined*, ob dabei *stack-unwinding* stattfindet !!!

1.5. Strukturierte Anweisungen: Exception Handling

- `std::terminate()` wird auch gerufen, wenn während der Behandlung einer Ausnahme eine weitere Ausnahme auftritt
- Funktionen können mit sog. *exception specifications* ausgestattet sein, (entspricht den throws-Klauseln von Java aber)

Java: `void foo(); // lässt keine Exceptions 'raus'`

C++: `void foo(); // lässt beliebige Exceptions 'raus'`

```
void foo () throw (dies, das, nochwasanderes);
```

Java: vollständige Flussanalyse zur Compile-Zeit

C++: keinerlei Flussanalyse, aber Überwachung zur Laufzeit

- tritt eine Exception auf, die nicht spezifiziert wurde, wird `std::unexpected()` (dies ruft wiederum `std::terminate()`) gerufen

1.5. Strukturierte Anweisungen: Exception Handling

- mittels `std::set_unexpected()` kann man dieses Verhalten ändern
- Aber: Herb Sutter (Exceptional C++, Item 13) und auch Boost (Exception-specification rationale):

Never write an exception specification !

- Destruktoren sollten NIEMALS Ausnahmen erzeugen:

```
X::~X() throw ();
```

WARUM

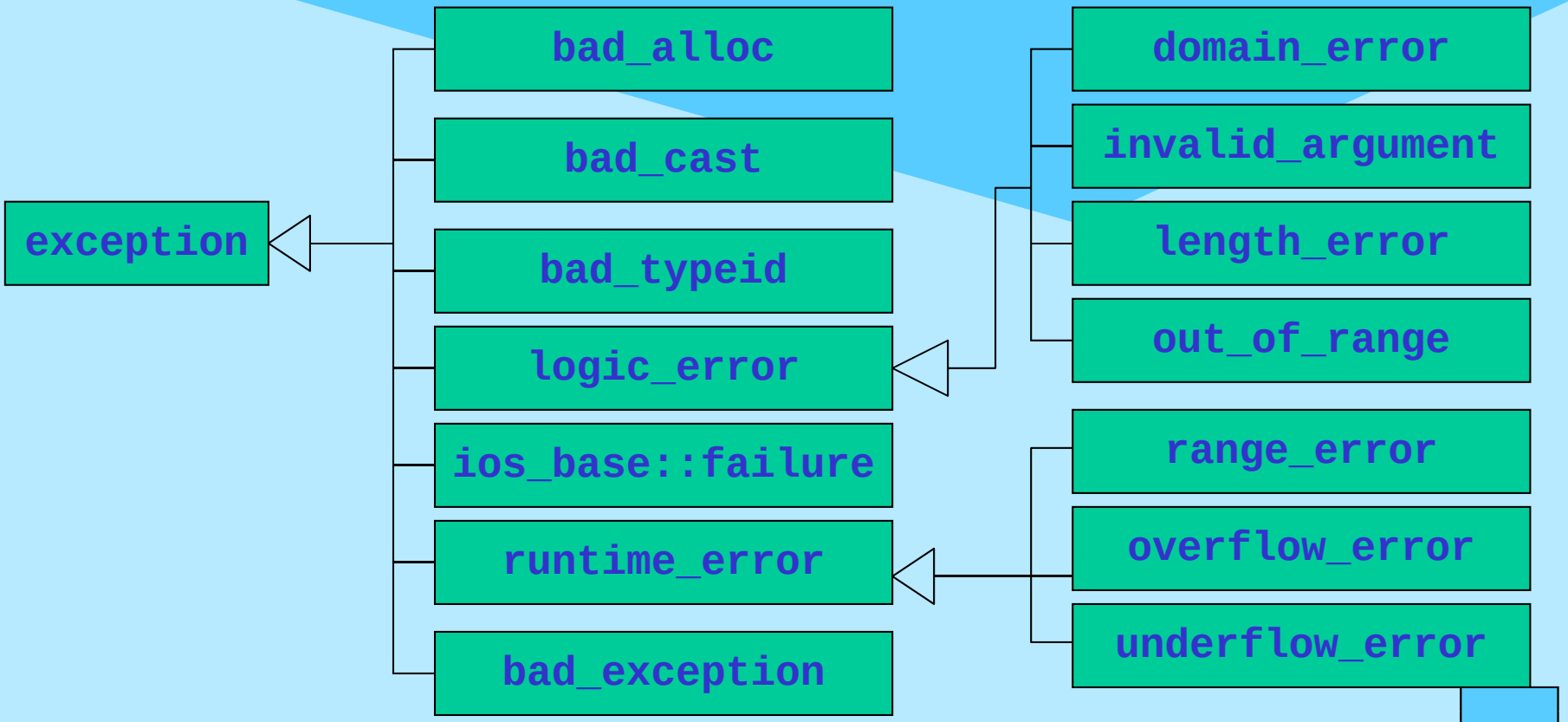
1.5. Strukturierte Anweisungen: Exception Handling

- es gibt die Möglichkeit eine ganze Funktion als `try`-Block zu implementieren:

```
int foo(int i)
try {
   may_throw(i); return 0;
}
catch (int ex) {
   return -1;
}
```

- Es gibt eine Reihe vordefinierter Ausnahmen

1.5. Strukturierte Anweisungen: Exception Handling



2. Klassen in C++

Um das Klassenkonzept ranken sich
alle wichtigen (oo) Konzepte:

- abstrakte Datentypen (Daten & Operationen)
- Zugriffsschutz
- nutzerdefinierte Operatoren
- Vererbung, Polymorphie & Virtualität
- generische Typen (Templates)

2. Klassen in C++ back -->

```
// Stack.h
#ifndef STACK_H
#define STACK_H
class Stack {
protected:
        int *data;
        int top, max;
public:
        Stack(int = 100);
        Stack(const Stack&);
        ~Stack();
        void push (int);
        int pop();
        int full() const;
        int empty() const;
};
#endif
```

prevents multiple inclusion !

ein neuer Typ !

Memberdaten

Memberfunktionen

Konstruktoren (u.u. viele)

Destruktor (einer !)

const Memberfunktion: Zusage, das Objekt nicht zu verändern

Zugriffsmodi

2. Klassen in C++

```
// Stack.cc
#include "Stack.h"
#include <cstdlib>

Stack::Stack(int dim): max(dim), top(0), data(new int[dim]) { }
Stack::Stack(const Stack & other) // Copy-Konstruktor
: max(other.max), top(other.top), data(new int[other.max]) {
        for (int i=0; i<top; ++i)
                data[i]=other.data[i];
}
Stack::~Stack() {
        delete [] data; // Feld statt einzelnem Objekt !
}
void Stack::push (int i) {
        if (!full()) data[top++]=i;
        else std::exit(-1);
} // if (!this->full()) this->data[this->top++]=i;
```

initializer list

NICHT: max

Scope resolution

2. Klassen in C++

```
int Stack::pop () {
        if (!empty()) return data[--top];
        else std::exit(-1);
}
int Stack::full() const { return top == max; }
int Stack::empty() const { return top == 0; }
```

- alternativ Memberfunktionen im Klassenkörper: dann implizit inline
- oder außerhalb des Klassenkörpers mit expliziter inline-Spezifikation (im Headerfile !)

```
// Nutzung:
#include "Stack.h"
void foo() {
Stack s1 (1000); s1.push(123);
Stack *sp = new Stack; sp->push(321);
delete sp; // ansonsten memory leak !
}
```

- Wann immer Objekte entstehen, läuft automatisch ein (passender) Konstruktor !
- Wann immer Objekte verschwinden, läuft automatisch der Destruktor !
- Klassen ohne nutzerdefinierten Konstruktor/Destruktor besitzen implizit
  - den sog. *default constructor* `X (){}`
  - den sog. *default copy-constructor* `X (const X&){ ... }` und
  - den sog. *default destruktor* `~X() {}`

memberweise Kopie !

- sobald nutzerdefinierte Konstruktor-Varianten vorliegen, gibt es nur noch den impliziten Copy-Konstruktor (wenn dieser nicht auch explizit definiert wird)

- Jedes Objekt enthält seine eigene Realisierung der Memberdaten (NICHT der Memberfunktionen!)
- Die Identität eines Objektes ist mit seiner Adresse verbunden !

```
bool Any::same(Any& other){
        return this == &other;
}
```

?

Beispiel: Jedes **`Stack`**-Objekt hat das folgende Layout unabhängig davon, wie es entstanden ist !

kein overhead durch Meta-Daten !

- es sind auch sog. unvollständige Klassendeklarationen erlaubt, von einer solchen Klasse können jedoch bis zu ihrer vollständigen Deklaration lediglich Zeiger & Referenzen benutzt werden:

```
class B;
class A {B * my_B; ....}; // oder ... class B* my_B;
class B {A * my_A; ....};
```

- strukturell identische Klassen mit verschiedenen Namen bilden verschiedene Typen (es gibt jedoch die Möglichkeit, nutzerdefiniert Kompatibilität herbeizuführen s.u.):

```
class X { public: int i; } x0;
class Y { public: int i; } y0;
X x1 = y0; Y y1 = x0; // beides falsch !!!
x0 = y0; y0 = x;      // beides falsch !!!
```

- Konstruktorparameter sind beim Anlegen von Objekten (geeignet) anzugeben, d.h es muss einen entsprechenden Konstruktor geben

```
// direct initialization:
X x0;                  // needs X::X();
X x1(1);               // needs X::X(int);
X x2 = X(2,0);         // needs X::X(int,int);
X *pb = new X (5,true);  // needs X::X(int, bool);
X x3(1, "zwei", '3'); // needs X::X(int,[const] char*,char);

// copy initialization:
X x3 = 3;    // X tmp(3); X x3(tmp); ggf. elision
```

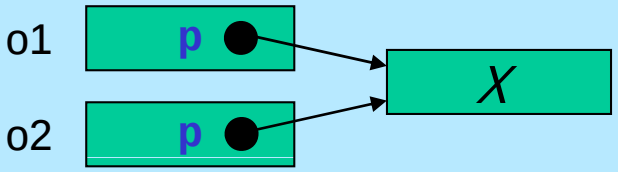
Copy-Konstruktoren

`X::X(const X&); // kanonische Form !`

shallow copy

(*default copy ctor*)

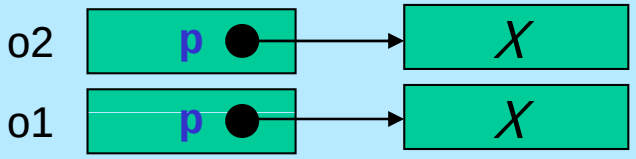
```
class SC {
  X* p;
public:
  SC(): p(new X) {}
};
SC o1;
SC o2=o1;
```



deep copy

(nutzerdefinierter *copy ctor*)

```
class DC {
   X* p;
public:
        DC(): p(new X) {}
        DC(const DC& src)
        : p(new X) {...copy X };
};
DC o1;
DC o2=o1;
```



- Konstruktoren können auch mit einem *function try block* implementiert werden, auch wenn ein passender *handler* vorliegt, wird die Ausnahme immer *re-thrown* !!!

```
struct Y {
   X* p;
   Y(int i) try : p(new X)
   { if (i) throw "huhh"; }
   catch(...)
   { /* delete p; NOT ALLOWED !!! */
      /* throw "huhh"; implicitly */}
   ~Y() { delete p; }
};
```

15.3 (10): Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.

2. Klassen in C++

```
http://www.gotw.ca/gotw/066.htm
   {
     Parrot parrot;
   }
```

In the above code, when does the object's lifetime begin? When does it end? Outside the object's lifetime, what is the status of the object? Finally, what would it mean if the constructor threw an exception? Take a moment to think about these questions before reading on.

Q: When does an object's lifetime begin?

A: When its constructor completes successfully and returns normally. That is, control reaches the end of the constructor body or an earlier return statement.

Q: When does an object's lifetime end?

A: When its destructor begins. That is, control reaches the beginning of the destructor body.

Q: What is the state of the object after its lifetime has ended?

A: As a well-known software guru ☺ once put it, speaking about a similar code fragment and anthropomorphically referring to the local object as a "he":

He's not pining! He's passed on! This parrot is no more! He has ceased to be! He's expired and gone to meet his maker! He's a stiff! Bereft of life, he rests in peace! If you hadn't nailed him to the perch he'd be pushing up the daisies! His metabolic processes are now history! He's off the twig! He's kicked the bucket, he's shuffled off his mortal coil, run down the curtain and joined the bleedin' choir invisible! THIS IS AN EX-PARROT! - Dr. M. Python, BMath, MASc, PhD (CompSci)

Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.

Leider wird dies von vielen Compilern dennoch erlaubt ☹ : g++, VisualStudio2005/2008

Initialisierung vs. Zuweisung:

= im Kontext einer Objektdeklaration: Initialisierung

```
X x = something; // initialize
```

= nicht im Kontext einer Objektdeklaration: Zuweisung

```
x = something; // assign !
```

```
class X {
  const int c;
public:
  X(int i): c(i) {} // ok, aber
  // X(int i) {c=i;} // falsch
};
```

Prefer initialization !

Initialisierung vs. Zuweisung:

```
#include <iostream>

class A {
public:
   A(int i){ std::cout<<"A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i) { std::cout<<"B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); } // valid C++ ?????
```

Initialisierung vs. Zuweisung:

Prefer initialization !

```
#include <iostream>

class A {
public:
   A(int i){ std::cout<<"A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i) { std::cout<<"B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); }
```

```
Error init.cpp 11: Cannot find default constructor to
initialize member 'B::myA' in function B::B(int)
```

Initialisierung vs. Zuweisung:

Prefer initialization !

```
#include <iostream>

class A {
public:
   A(int i){ std::cout<<"A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i) { myA = i; std::cout<<"B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); }
```

```
Error init.cpp 11: Cannot find default constructor to
initialize member 'B::myA' in function B::B(int)
```

Initialisierung vs. Zuweisung:

Prefer initialization !

```
#include <iostream>

class A {
public:
   A(int i = 0 ){ std::cout << "A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i) { myA = i; std::cout << "B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); }
```

```
$ init
A(1)
A(0)
A(2)
B(2)
```

?

Initialisierung vs. Zuweisung:

Prefer initialization !

```
#include <iostream>

class A {
public:
   A(int i){ std::cout << "A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i): myA(i) { std::cout << "B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); }
```

```
$ init
A(1)
A(2)
B(2)
```

C++ *idiom:* Resource Acquisition Is Initialization (*

```
void doDB() { // from Steven C. Dewhurst: C++ Gotchas (gotcha #67)
   lockDB();
   // do stuff with database ... but could throw !?
   unlockDB();
}

void doDB() {
   lockDB();
   try {// do stuff with database ...
   }
   catch ( ... ) { unlockDB(); throw; } // ugly
   unlockDB();
}
```

(* of an object !

C++ *idiom:* Resource Acquisition Is Initialization

```
// better:
class DBLock {
   public:
        DBLock() { lockDB(); }
        ~DBLock() { unlockDB(); }
};

void doDB() {
   DBLock lock;
   // do stuff with database ...
}
```

Fallen:

```
// NOT: DBLock lock();
// NOT: DBLock();
```

2. Klassen in C++

C++ *idiom:* *Resource Acquisition Is Initialization*

```
struct X {
        X() { cout<<"X()\n"; }
        ~X() { cout<<"~X()\n"; }
};

struct Xpointer { // a (not very) smart pointer
        X* pointer;
        Xpointer(X* p): pointer(p){}
        ~Xpointer(){delete pointer;}
};

struct Y {
        Xpointer p;
        Y(int i) try : p(new X)
        { if (i) throw "huhh"; }
        catch(...)
        { cout<< "catched local\n";}
        ~Y() {}
};

int main() try {
        cout<<"sizeof(Y)="<<sizeof(Y)<<endl;
        Y y0(0);
        Y y1(1);
}
catch(...) { cout<<"catched final\n";}
```

```
#include <iostream>
using std::whatever;
```

```
sizeof(Y)=4
X()
X()
~X()
catched local
~X()
catched final
```

```
#include <iostream>
using std::whatever;
```

C++ idiom: Resource Acquisition Is Initialization

```
class Trace { // C++ Gotchas, dito #67
public:
    Trace (const char* msg): m_(msg) {cout << "Entering " << m_ << endl;}
    ~Trace() {cout << "Exiting " << m_ << endl;}
private:
    const char* m_;
};
Trace a("global");
void foo(int i) {
    Trace b("foo");
    while (i--) { Trace l("loop"); /* ... */ }
    Trace c("after loop");
}
int main() { foo(2); }
```

```
$ t
Entering global
Entering foo
Entering loop
Exiting loop
Entering loop
Exiting loop
Entering after loop
Exiting after loop
Exiting foo
Exiting global
```

C++ *idiom:* Resource Acquisition Is Initialization

```
#include <iostream>
#include <ctime>
using std::whatever;
```

```
class Timer {
   long start, stop;
   void report()
       {cout<<(stop-start)/1000000.0<<"s"<<endl;}
public:
   Timer():start(clock()){}
   ~Timer(){ stop=clock(); report();}
};
```

- Klassen können auch sogenannte *static Member* enthalten, diese werden nur einmal pro Klasse angelegt !
- **`static`** Memberfunktionen dürfen (implizit) nur auf static Memberdaten zugreifen, (sie haben keinen **`this`**-Zeiger!)
- **`static`** Memberdaten sind nicht Teil des Objekt-Layouts
- **`static`** Memberdaten sind (einmalig) zu initialisieren !

```
$ s
Es gibt jetzt 3 A-Objekte
```

```
class A {
   static int count;
public:
   static int c(){ return count; }
   static const double A_specific_const; // NOT HERE = 123.456;
   A() {count++;}
   A(const A&) {count++; /* and copy */} // Kopien mitzählen !
   ~A(){count--;}
} a1, a2, a3;
int A::count = 0; // hier erst definiert !
const double A::A_specific_const = 123.456; // dito
int main() {
   double x = A::A_specific_const; // class access
   // A::A_specific_const = 1.23; // Fehler: const !
   cout << "Es gibt jetzt "<< a1.c()<<" A-Objekte\n";
// a1.count ist private,auch a2.c() oder a3.c() oder A::c() möglich
}
```

- neben den traditionellen C-Zeigern gibt es in C++ auch spezielle Zeigertypen für Zeiger auf Member(-daten und -funktionen)

✪

```
class X { public: int p1,p2,p3; };
void foo() {
  X x; X* pp=&x;      // ein C-Zeiger auf ein X
  int X::*xp=&X::p2; // xp ist ein Zeiger auf ein int in X
// xp = &x.p2;
// error: bad assignment type: int X::* = int *
  int *p;
// p = &X::p2;
// error: bad assignment type: int * = int X::*
  p = &(x.*xp); // ok, ohne Klammern falsch: (&x).*xp
  pp->*xp = 1; } // .* und ->* sind neue Operatoren
```

```
class Y {
public:
  void f1(){std::cout<<"Y::f1()\n";};
  void f2(){std::cout<<"Y::f2()\n";};
  static void f3(){cout<<"static Y::f3()\n";}
  typedef void (Y::*Action)();
  void repeat(Action=&Y::f1, int=1);//...(void(Y::*)(),int)
};
void Y::repeat (Action a, int count) {
  while (count--) (this->*a)();
}
int main() {
  Y y; Y* pp=&y;
  void (Y::*yfp)();
  // Zeiger auf Memberfkt. in Y mit Signatur void->void
```

```
yfp=&Y::f1; // nicht yfp =Y::f1 !(trotz vc++6.0, bcc32, icc)
// yfp();
// object missing in call through pointer to memberfunction
(y.*yfp)(); // Y::f1()
yfp=&Y::f2;
(pp->*yfp)(); // Y::f2()
// yfp=&Y::f3;
// bad assignment type: void (Y::*)() = void (*)()static
// aber:
void (*fp)()=&Y::f3;
fp(); // besser (*fp)();
y.repeat(yfp, 2);
}
```

```
$ mp
Y::f1()
Y::f2()
static Y::f3()
Y::f2()
Y::f2()
```

Vererbung: Grundprinzip von OO

- Übernahme von Eigenschaften aus einer Klasse
- Erweiterung / Modifikation

Beispiel: ein Stack mit *Buchführung*

```
class CountedStack : public Stack // IST EIN STACK
{
   int min, max, n, sum; // zusätzliche Attribute
public:
   CountedStack(int dim = 100);
   void push (int i); // redefined !
   int minimum(); // neu
   int maximum(); // neu
   double mean(); // neu
   double actual_mean();// neu
// pop, empty, full aus der Basisklasse !
};
```

2. Klassen in C++ back -->

```
CountedStack::CountedStack(int dim):Stack(dim),n(0),sum(0){}

void CountedStack::push(int i) {
  sum+=i;
  if (!n++) { min = max = i; }
  else { min = (i<min) ? i : min; max = (i>max) ? i : max; }
  Stack::push(i); // use base functionality NOT push(i)
}

double CountedStack::actual_mean() {
  if (top) { int s=0;
   for (int i=0; i<top; i++) s += data[i];
   return double(s)/top; // direct access to base members
  } else std::exit(-4);
}
```

Ist ein (nutzerdefinierter) Copy-Konstruktor erforderlich ?

Nein, weil der implizite Copy-K. die Copy-K.en aller Basisklassen ruft und für die Erweiterung **`CountedStack`** *shallow copy* ausreichend ist:

```
// implizit bereitgestellt:
CountedStack::CountedStack(const CountedStack& other)
  :
  Stack(other) {/* real copy */}
```

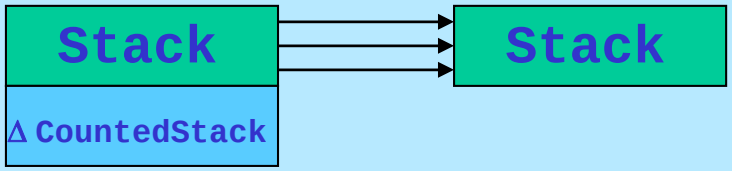
Der (nutzerdefinierte) **`Stack`**-Copy-K. erwartet allerdings eine
`const Stack& ????`

– Jedes **CountedStack** - Objekt **IST EIN** **Stack**-Objekt

```
CountedStack cs; ... cs.pop();
void foo (Stack&); ... foo (cs);
```

– von der Ableitung zur Basisklasse ist implizit eine Projektion definiert

```
void bar (Stack); ... bar(cs); // slicing
```



– nur bei `public` Vererbung gilt die **IST EIN** Relation

non-`public` Vererbung

```
class Deriv1 : private Base { .... };
```

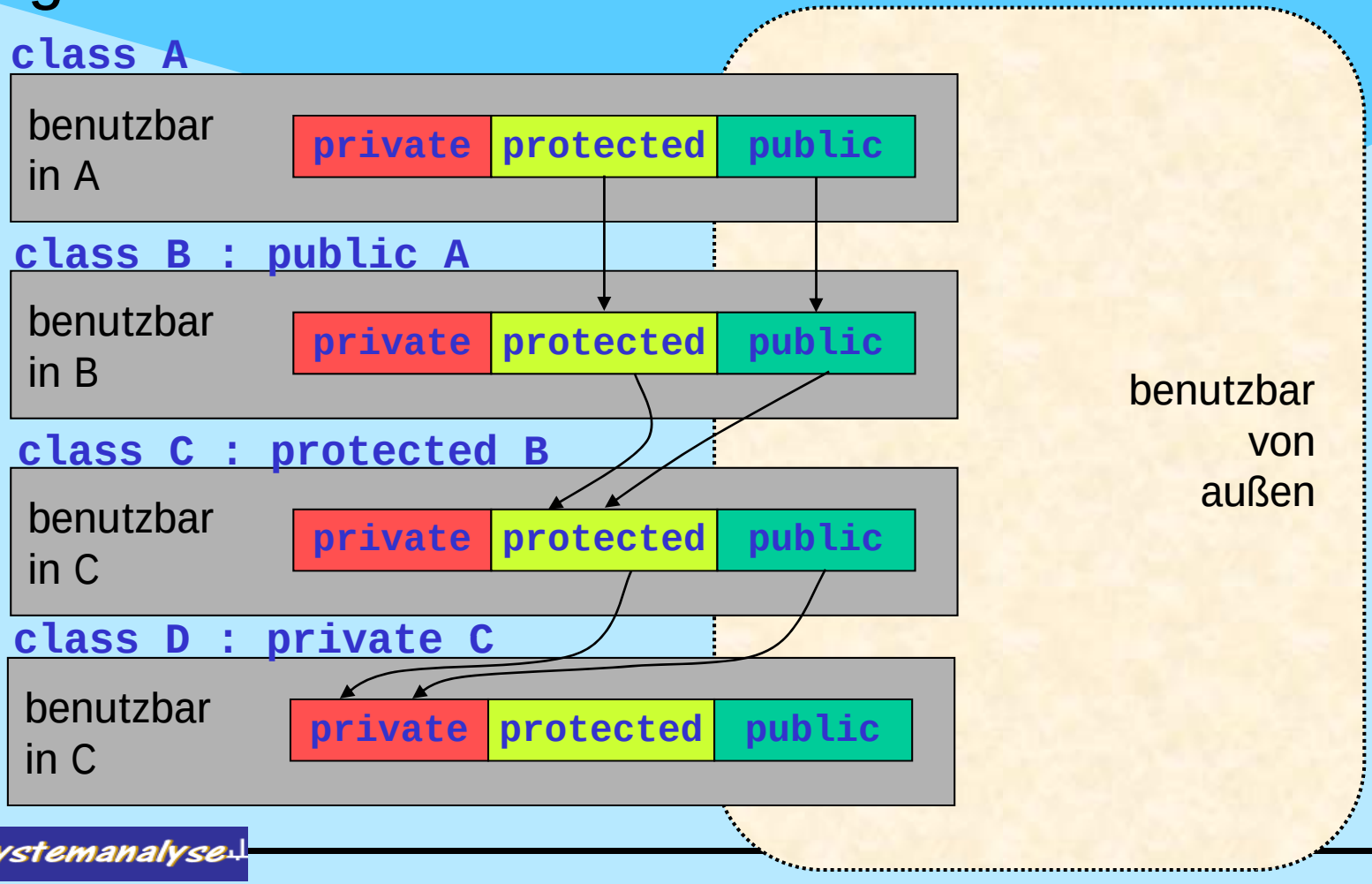
`Deriv1` IST nirgends EIN `Base` == die Vererbung ist ein (nicht erkennbares) Implementationsdetail

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

`Deriv2` IST nur in Ableitungen von `Deriv2` EIN `Base` == die Vererbung ist nur Ableitungen `Deriv2` von bekannt

das Layout von Objekten abgeleiteter Klassen wird von der Art der Vererbung **NICHT** beeinflusst !

Zugriffsrechte in C++



`struct` ist implizit `public`, `class` ist implizit `private`

Depricated:

`struct` erbt implizit `public`, `class` erbt implizit `private`

Beim *lookup* von Funktionsnamen erfolgt
overload resolution **VOR** *access check* !

```
class X {
        foo(int);
public:
        foo(int, int = 0);
};

int main(){  X x;
             x.foo(1); //call of overloaded `foo(int)' is ambiguous
}
```

Warum besteht bei **private**-Vererbung die IST EIN - Relation nicht ?

```
class A {                       class B : private A {
public:                                 ....
      int i;                    };
};

B b;
b.i = 1; // ERROR: `class B' has no member named `i'
// Wenn ein B ein A wäre:
A* pa = &b;
pa->i = 1; // sollte aber gerade geschützt werden !
// ergo, b ist kein A
A* pa = &b; // ERROR: `A' is an inaccessible base of `B'
```

Friends

oftmals ist die Entscheidung zwischen Alles (`public`) oder Nichts (`private`) zu restriktiv --> Möglichkeit, speziellen Klassen/Funktionen Zugriff einzuräumen, indem diese als `friend` deklariert werden

```
class B { public:void f(class A*); };
class A {
        int secret;
public:
        friend void trusted_function(A& a) // globale funktion !!!
        {... a.secret .... }               // inline !!!
        friend B::f(A*);
};
void B::f(A* pa) { .... pa->secret ....}
```

Friends

friend-Funktionen sind keine Memberfunktionen der Klasse, die die **friend**-Rechte einräumt

macht man eine ganze Klasse zum **friend**, werden alle Memberfunktionen dieser zu **friends**

Vorsicht bei unterschiedlichen Kontexten für **inline**- und „outline“-Funktionen

```
typedef char* T;
class S {
        typedef int T;
        friend void f1(T) { .... } // void f1(int);
        friend void f2(T);         // void f2(int);
};
void f2(T) { .... } // void f2(char*); also kein friend !
```

Friends

Die **friend**-Relation ist nicht symmetrisch, nicht transitiv & nicht vererbbar

```
class ReallySecure {
        friend class TrustedUser;
        ....
};
class TrustedUser {
        // can access all secrets
};
```

```
class Spy: public TrustedUser {
        // if friend relation would be inherited: aha !
};
```

Die Position einer **friend**-Deklaration in einem Klassenkörper (**private/protected/public**) ist ohne Bedeutung, dennoch sollte man **friend**-Deklarationen in einem **public** Abschnitt unterbringen (Schnittstelle der Klasse!)

```
// intermezzo: what's wrong with this code ?
#include <string>
#include <iostream>

class A{
public:
  const std::string& txt;
  A(const char *);
};
A::A(const char* chr) : txt(chr) {}
int main(){
  const char * foo = "foo";
  A test(foo);
  std::cout << test.txt << std::endl; // doesn't print "foo"
}
```

```
$ /opt/intel/compiler70/ia32/bin/icc -o z z.cpp
$ z

$ /opt/intel/compiler70/ia32/bin/icc -o z z.cpp -Wall
z.cpp(9): remark #383: value copied to temporary,
reference to temporary used
A::A(const char* chr) : txt(chr) {}
                              ^
```

Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

Unikate - Objekte, die man nicht kopieren kann:

```
class U { // wie Unikat
        U(const U&); // ohne Definition
        U& operator=(const U&); // dito
public:
        ...
};

U u1; // ein Unikat
U u2; // noch eines
U u3 (u1); // ERROR U::U(const U&)' is private within this context
void foo(U);
void bar () { foo(u1); }  // ERROR dito
```

Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

Singletons - Objekte, die es nur einmal gibt

```
class S { // wie Singleton, mit lazy creation
        S( some parameters ) { .... }
        S(const S&);                // inhibit copy
        S& operator=(const S&);     // inhibit assign
        static S *it_;
public:
        static S& instance() {
                if (! it_) it_ = new S( parms );
                return *it_;
        }
};  // in S.h
S* S::it_ = 0; // in S.cpp, so nötig obwohl privat !

S::instance(); // gibt stets eine Referenz auf dasselbe Objekt

// Attn.: NOT thread safe
http://www.devarticles.com/c/a/Cplusplus/C-plus-in-Theory-Why-the-Double-Check-Lock-Pattern-Isnt-100-ThreadSafe/
```

Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

Factory - Objekte, die andere Objekte am Fließband produzieren

```
class P { // … wie Produkt
        // alles privat
public:
        friend class P_Factory;
};

class P_Factory { // sinnvollerweise zugleich singleton
public:
        P* generate () { .... return new P; }
};
....
P_factory::instance().generate();
```

Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

'No' - Objekte, die es (an sich) nicht gibt

```
class No { // keine Objekte sind erzeugbar
protected:
       No::No() { .... }
public: ...
};

No n; // ERROR NO::No() not accessible
```

Besseres Sprachfeature, um dies auszudrücken sind *abstract base classes*
- Klassen die sich nur für Vererbung, nicht für Objekterzeugung eignen (s.u.)

Zeiger und Referenzen können *polymorph* sein (Objekte NICHT) !

```
Stack* sp = new CountedStack;
Stack& sr = *sp;
Stack s = *sp; // slicing
```

beim Aufruf (nicht-virtueller) Memberfunktionen entscheidet die statische Qualifikation (Eintrittspunkt zur wird zur Compile-Zeit ermittelt --> *early binding*)

```
sp->push (42); // Stack::push ! ???  --> Stack.h
sr .push (42); // Stack::push ! ???
// obwohl es ein eigenes CountedStack::push gibt und
// in beiden Fällen CountedStack-Objekte vorliegen
```

Memberfunktionen können jedoch (in der Basisklasse) als *virtuell* deklariert werden

dann entscheidet die dynamische Qualifikation (Eintrittspunkt wird zur Laufzeit ermittelt --> *late binding*)

```
class Stack' {...
public: virtual void push(int); ...};

Stack* sp = new CountedStack;
Stack& sr = *sp;
sp->push (42); // CountedStack::push !!!
sr .push (42); // CountedStack::push !!!
```

Um die Entscheidung in die Laufzeit vertagen zu können, muss eine Typinformation im Objekt hinterlegt werden

Ziel für C++: Mechanismus mit hoher Zeit- und Platzeffizienz

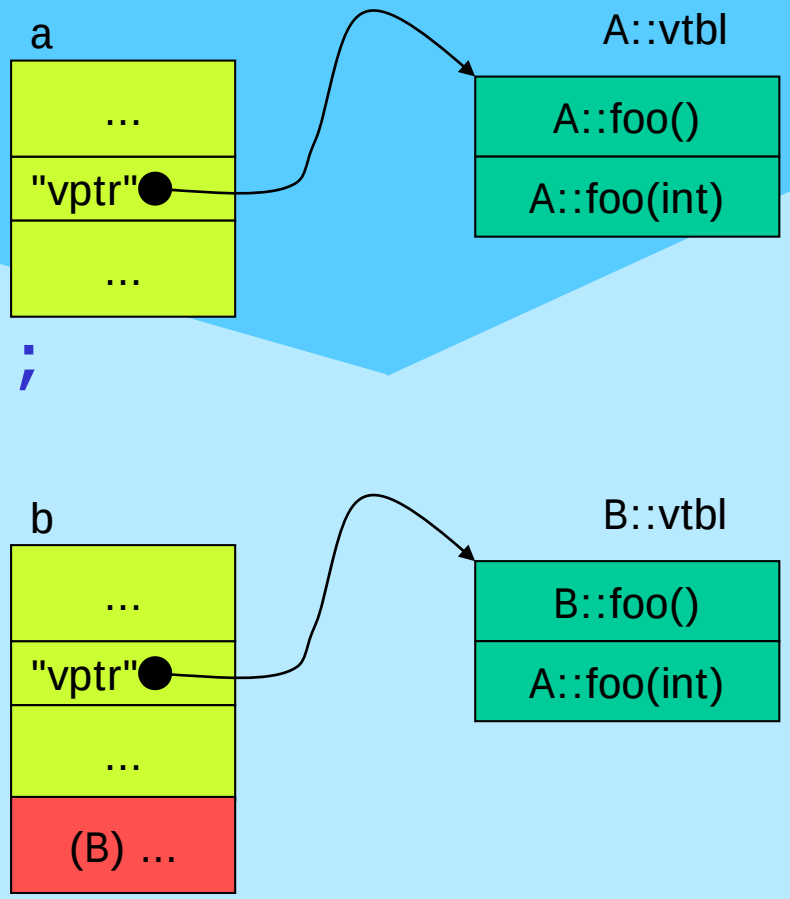
Realisierung (nicht normativ aber de facto Standard):

- ein (verborgener) Zeiger (**vptr**) pro Objekt +
- eine Adress-Substitution beim Aufruf virtueller Funktionen

damit ist *late binding* (geringfügig) teurer -- wie immer gilt das Prinzip *»Aufpreis nur auf Anfrage«*

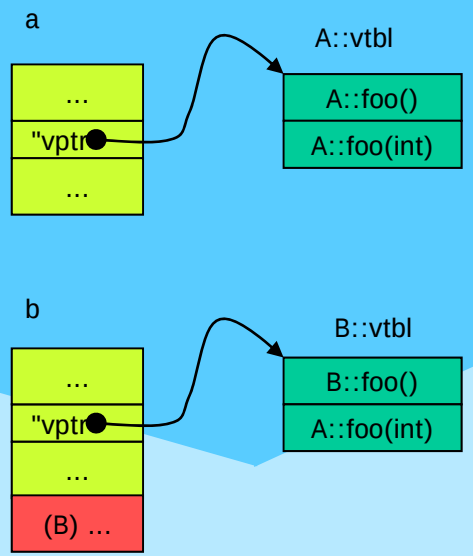
Beispiel

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

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



2. Klassen in C++

Beispiel (Fortsetzung)




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

ao.foo();     // A::foo (&ao); NOT LATE!
ap->foo();    // (ap->vptr[0])(ap); LATE BINDING
bo.foo();     // B::foo (&bo); NOT LATE!
bp->foo();    // (bp->vptr[0])(bp); LATE BINDING
app->foo();   // (app->vptr[0])(app); LATE BINDING
app->foo(1);  // (app->vptr[1])(app); LATE BINDING
bp->foo(1);   // Warning W8022 ab.cpp 14: 'B::foo()' hides virtual function 'A::foo(int)'
              // Error E2227 ab.cpp 30: Extra parameter in call to B::foo() in function main()
```

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

Durch eine initiale Operation bei der die Typzugehörigkeit des Objektes bekannt ist: **Konstruktoren** *'wissen, was sie gerade konstruieren'*

```
A::A() // impliziter default-ctor
{ » this -> vptr = &A::vtbl; « }
B::B() : A() // impliziter default-ctor
{ » this -> vptr = &B::vtbl; « }
```

nicht jeder Aufruf einer virtuellen Funktion wird spät gebunden:

- Aufruf an einer Objektvariablen (s.o. `ao.foo();`)
- Aufruf mit scope resolution: --> CountedStack::push
- Aufruf in einem Konstruktor/Destruktor !

```
inline virtual void foo();
```

erlaubt, aber `inline` xor `virtual` pro Aufruf

```
static virtual void foo();
```

nicht erlaubt

Die »Planung« von austauschbarer Funktionalität muss in einer Basisklasse erfolgen, unterhalb dieser Basis ist die Funktionalität nicht verfügbar

Eine Redefinition einer virtuellen Funktion liegt nur vor, wenn die Signatur exakt mit dem ursprünglichen Prototyp übereinstimmt

Ausnahme: kovariante Ergebnistypen

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

`virtual <returntyp> fkt` und `<returntyp> virtual fkt` sind synonym (bevorzugt 1. Variante)

»einmal virtuell, immer virtuell« (sofern die gleiche Funktion vorliegt), erneute `virtual` Deklaration in Ableitungen eigentlich redundant, aber empfohlen

Vorsicht: virtuelle Funktionen können u.U. »überdeckt « werden

```
#define O(X) std::cout<<#X<<std::endl;

struct A {
       virtual void foo() { O( A::foo() ); }
};
struct B : public A {
       void foo (int=0)   { O( B::foo(int) ); } // non virtual
};
struct C : public B {
              void foo()  { O( C::foo() ); }   };
```

```
int main()
{
        C c;
        B* p = &c;

        c.foo();
        p->foo();
}
```

```
C:\tmp>bcc32 hide.cpp
...
Warning W8022 hide.cpp 15:
'B::foo(int)' hides virtual
function 'A::foo()'
...
C:\tmp>hide
C::foo()
B::foo(int)
```

g++ (auch 3.x) warnt nicht [nicht mal bei -Wall]

Konstruktoren können nicht virtuell sein (warum nicht?)

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

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

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

Überladung wird auch bei abgeleiteten Klassen lokal zu einer Klasse berechnet (ausgehend vom Bezugspunkt !):

```
#define O(X) std::cout<<#X<<std::endl;

struct X {
        void foo(int){O(X::foo(int));}
        void foo(char){O(X::foo(char));}
};

struct Y : public X {
        void foo(int){O(Y::foo(int));}
        void foo(double){O(Y::foo(double));}
};
```

Überladung wird auch bei abgeleiteten Klassen lokal zu einer Klasse berechnet (ausgehend vom Bezugspunkt !):

```
int main () {
              X x;
              x.foo(1);
              x.foo('1');
//            x.foo(1.0); Ambiguity between 'X::foo(int)' and 'X::foo(char)
              Y y;
              y.foo(1);
              y.foo('1');
              y.foo(1.0);
}
```

```
C:\tmp>lookup
X::foo(int)
X::foo(char)
Y::foo(int)
Y::foo(int)
Y::foo(double)
```

```
class X1 {
public:
        f(int);
};
// chain of derivations Xn : Xn-1 without f
class X9 : public X8 {
public:
        void f(double);
};
void g(X9* p) { p->f(2); } // X9::f or X1::f ? X9::f !
```

ARM: *Unless the programmer has an unusually deep understanding of the program, the assumption will be that p->f(2) calls X9::f - and not X1::f declared deep in the base class. Under the C++ rules, this is indeed the case. Had the rules allowed X1::f to be chosen as a better match, unintentional overloading of unrelated functions would be a distinct possibility.*

Wenn aber doch `X1::f` gemeint ist?

```
class X1 {
public:
      f(int);
};
// chain of derivations Xn : Xn-1 without f
class X9 : public X8 {
public:
      void f(double);
      void f(int i) { X1::f(i); } // inline !
};
void g(X9* p) { p->f(2); } // X1::f
```

Java dagegen betrachtet bei Überladung alle Funktionen aus der gesamten Vererbungslinie !

```
class X {
      void O(String s){System.out.println(s);}
      public void foo(int i) {O("X::foo(int)");}
      public void foo(char c){O("X::foo(char)");}
};

class Y extends X {
      public void foo(int i){O("Y::foo(int)");}
      public void foo(double d){O("Y::foo(double)");}
};
```

```
public class lookup {
        public static void
        main (String s [])
        {
                X x = new X();
                x.foo(1);
                x.foo('1');
                // x.foo(1.0); cannot find symbol foo(double)
                Y y = new Y();
                y.foo(1);
                y.foo('1'); // bis 1.4 Fehler:
                        // Reference to foo is ambiguous
                y.foo(1.0);
        }
}
```

```
C:\tmp>java lookup
X::foo(int)
X::foo(char)
Y::foo(int)
X::foo(char)
Y::foo(double)
```

Ziel: maximales Code-Sharing -- Weg: gemeinsame (aber ggf. in Ableitungen variierende) Funktionalität in Basisklassen festlegen

Problem: die so entstehenden Basisklassen sind oft so rudimentär, dass Objekterzeugung nicht sinnvoll und Implementation einiger Memberfunktionen (noch nicht) möglich ist:

abstract base class (ABC)

pure virtual function

Beispiel:

```
struct AbstractShape {
       virtual void draw() = 0;
       virtual void erase()= 0;
};
// no objects allowed:
// AbstractShape aShape; ERROR
AbstractShape *any; // ok
any = new Circle (Point(0,0), 100);
```

```
struct Circle : // real Shape
 public AbstractShape {
  virtual void draw() {...}
  virtual void erase() {...}
};
```

2. Klassen in C++

```
class abstractBase { public:
        virtual void pure() = 0;
        void notPure() { pure(); }
        abstractBase() { notPure(); }
        virtual ~abstractBase() { notPure(); }
};

class concrete: public abstractBase { public:
        void pure() {}
        concrete() {}
};

int main() {
        cout<<"buggy:"<<endl;
        concrete c;
/*
        g++: pure virtual method called
        terminate called without an active exception
        Abort
*/
}
```

Scott Meyers, *Effective C++* :
Item 9: "Never call virtual functions during construction or destruction."

2. Klassen in C++

```
class abstractBase2 { public:
        virtual void pure() = 0;
        void notPure() { pure(); }
        abstractBase2() { /* no virtual function call ! */ }
        virtual ~abstractBase2() {/* no virtual function call ! */ }
};

class concrete2: public abstractBase2 { public:
        void pure() {}
        concrete2() {}
};

int main() {
        cout<<"buggy too:"<<endl;
        abstractBase2 *p = new concrete2;
        abstractBase2 *q = p;

        delete q;
        p->notPure();    // pure virtual method called (z.B. auf amsel)
}
```

Scott Meyers, *Effective C++* :
Item 9: "Never call virtual functions during construction or destruction."

2. Klassen in C++

[http://www.artima.com/cppsource/pure_virtual.html]

This is another classic blunder: going indirect on a "dangling" pointer. That's a pointer to an object that's been deleted, or memory that's been freed, or both. C++ programmers never write such code ... unless they're clueless (unlikely) or rushed (all too likely).

So now p points to an ex-object. What does that thing look like? According to the C++ standard, it's "undefined". That's a technical term that means, in theory, anything can happen: the program can crash, or keep running but generate garbage results, or send Bjarne Stroustrup e-mail saying how ugly you are and how funny your mother dresses you. You can't depend on anything; the behavior might vary from compiler to compiler, or machine to machine, or run to run. In practice, there are several common possibilities (which may or may not happen consistently):

- *The memory might be marked as deallocated. Any attempt to access it would immediately be flagged as the use of a dangling pointer. That's what some tools (BoundsChecker, Purify, valgrind, and others) try to do. As we'll see, the Common Language Runtime (CLR) from Microsoft's .NET Framework, and Sun Studio 11's dbx debugger, work this way.*
- *The memory might be deliberately scrambled. The memory management system might write garbage-like values into the memory after it's freed. (One such value is "dead beef": 0xDEADBEEF, unsigned decimal 3735928559, signed decimal -559038737.)*
- *The memory might be reused. If other code was executed between the deletion of the object and the use of dangling pointer, the memory allocation system might have created a new object out of some or all of the memory used by the old object. If you're lucky, this will look enough like garbage that the program will crash immediately. Otherwise the program will likely crash sometime later, possibly after curdling other objects, often long after the root cause problem occurred. This is the kind of problem that drives C++ programmers crazy (and makes Java programmers overly smug).*
- *The memory might have been left exactly the way it was.*

The last is an interesting case. What was the object "exactly the way it was"? In this case, it was an instance of the abstract base class; certainly that's the way the vtbl was left. What happens if we try to call a pure virtual member function for such an object? "Pure virtual function called".

Im Kontext von Klassen können Operatoren mit nutzerdefinierter Semantik implementiert werden:

```
//Complex.h:                ∃  std::complex<T>
#include <iosfwd>
class Complex {
        double re, im;
public:
   Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
   friend Complex operator+(const Complex&, const Complex&);
   friend Complex operator*(const Complex&, const Complex&);
   friend bool operator==(const Complex&, const Complex&);
   friend bool operator!=(const Complex&, const Complex&);
   Complex& operator+=(const Complex&); // Member !
   Complex operator-(); // Member !
   friend std::ostream& operator<<(std::ostream&, const Complex&);
   friend std::istream& operator>>(std::istream&, Complex&);
....};
```

```
//complex.cpp: Auswahl
Complex operator+(const Complex& c1, const Complex& c2) {
        return Complex(c1.re+c2.re, c1.im+c2.im);
}
bool operator==(const Complex& c1, const Complex& c2) {
        return (c1.re==c2.re && c1.im==c2.im);
}
Complex& Complex::operator+=(const Complex& c) {
        re += c.re; im += c.im;
        return *this;
}
Complex Complex::operator-() {
        return Complex(-re, -im);
}
std::ostream& operator<< (std::ostream& o, const Complex &c) {
        return o << c.re << "+i*" << c.im;
}
```

```
//usecomplex.cpp:

int main() {
        Complex z1 (3, 4);
        Complex z2 (5, 6);
        Complex z3;
        cout << "z1=" << z1 << endl << "z2=" << z2 << endl;
        cout << "z1+z2=" << z1+z2 <<endl;
        cout << "gimme a Complex: ";
        cin >> z3;
        cout << "z3=" << z3 << endl;
}
```

Die Semantik von Operatoren kann nutzerdefiniert überladen werden, nicht dagegen deren Signatur, Priorität und Assoziativität

Es ist nicht möglich, neue Operatoren einzuführen (** %$@# ....)

Überladbar sind die folgenden Operatoren:

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

nicht überladbar sind dagegen `.  .*  .->  ::  ?:`

Die vordefinierte Semantik von Operatoren für *built in* - Typen bleibt erhalten

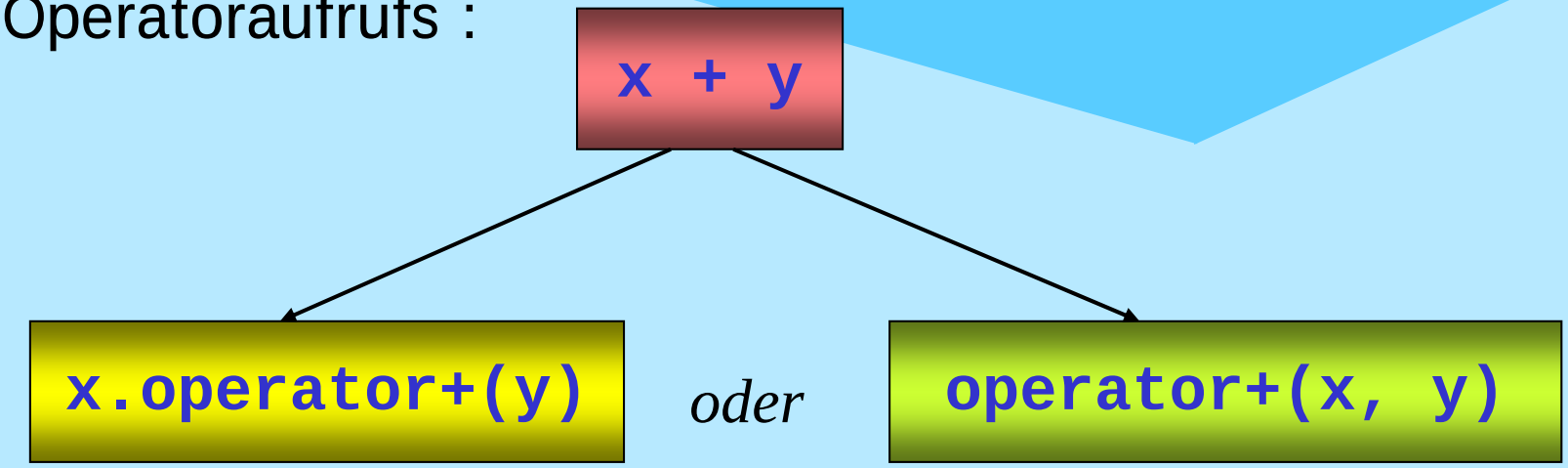
```
// falsch:
// int operator+ (int i, int j) {return i - j;}
```

durch die Forderung:

Ein Operator kann nur dann überladen werden, wenn in seiner Deklaration mindestens ein Parameter von einem Klassentyp (ggf. auch const / &) ist (dies kann auch das implizite `this`-Argument einer Memberfunktion sein) !

Member oder Friend (globale Funktion) ?

generell gibt es zwei Möglichkeiten der Auflösung eines Operatoraufrufs :



x muss von einem Klassentyp sein
(nur) y wird u.U. Typumwandlungen unterzogen

x oder y muss von einem Klassentyp sein
x und y werden u.U. Typumwandlungen unterzogen

Operatoren können NICHT static sein !

Syntax	unäre Operatoren	binäre Operatoren

Member	`class X { public:`
	` T operator ⊙ ();`
	`};`
	
	`⊙x; // Ergebnis: T`
	`// (x).operator ⊙ ();`
		`class X { public:`
		` T1 operator ⊙ (T2);`
		`};`
		
		`x ⊙ y; // Ergebnis: T1`
		`// (x).operator ⊙(y);`

Friend	`class X { public:`
	` friend T operator ⊙`
	`([const]X[&]);`
	`};`
	
	`⊙x; // Ergebnis: T`
	`// operator ⊙(x);`
		`class X { public:`
		` friend T1 operator ⊙`
		`([const]X[&], T2);`
		` friend T1 operator ✹`
		`(T2, [const]X[&]);`
		`};`
		`x ⊙ y; // Ergebnis: T1`
		`// operator ⊙(x, y);`
		`y ✹ x; // Ergebnis: T1`
		`// operator ✹(y, x);`

`X x; T2 y;`

Operator	empfohlene Variante
alle einstelligen	Member
= () [] ->	müssen Member sein !
alle der Form @=	Member
alle anderen zweistelligen	Friend

Sonderfälle

```
T X::operator [] (IndexT);          X x; IndexT i; T t;
t = x[i]; // (x).op[](i)
T X::operator () (T1, T2, ....Tn); T1 t1; ... Tn tn;
t = x(t1,t2,....tn); // (x).op()(t1,t2, ....tn)
```

funktionale Objekte

```
X& X::operator++ ();     ++x;
X  X::operator++ (int); x++;
```

☺ syntaktischer Hack

```
T X::operator->(void);
x->selector  // (x).operator->()->selector
```

kanonischer Zuweisungsoperator *copy assignment*

```
X& X::operator= (const X&);          X x1, x2;
```

wird implizit bereitgestellt* (mit *shallow assignment* Semantik), kann neu definiert werden, dann ist die komplette Semantik von "Zuweisung" nutzerdefiniert zu implementieren, incl. Zuweisung von enthaltenen Objekten bzw. Basisklassenbestandteilen

```
class A { public: /* copy assignment implicit or explicit */ };
class B : public A {public: B& operator= (const B&); };
B& B::operator=(const B& src) { // assign B member
       // how to assign the A-part ???
       A::operator= (src); // oder
       A* thisA = this;
       *thisA = src;
       return *this;
}
```

* nicht, wenn die Klasse konstante Member oder Referenzen enthält, oder Basis-=-Operator(en) nicht aufrufbar ist !

wenn ein nutzedefinierter Copy-Konstruktor vorliegt, ist zumeist auch der Zuweisungsoperator nutzerdefiniert zu implementieren

```
Stack& Stack::operator= (const Stack& src)
{     if (&src==this) return *this; // self assignment
      top = src.top;
      max = src.max;
      // NOT: data = src.data;   as the implicit one does
      // leak in this->data, data sharing afterwards
      delete[] data;
      data = new int[max];
      for (int i=0; i<top; ++i) data[i]=src.data[i];
      return *this;
}
```

Copy-Konstruktor und Copy-Assignment-Operator sind semantisch verwandt: meist gemeinsam bereitzustellen!

Kanonische und *exception safe* Implementation:

GotW #59 (Sutter: mxC++ Item 22)

What is the canonical form of strongly exception safe copy assignment?

What are the three common levels of exception safety? Briefly explain each one and why it is important.

The canonical Abrahams(* Guarantees are as follows.

1. Basic Guarantee: If an exception is thrown, no resources are leaked, and objects remain in a destructible and usable -- but not necessarily predictable -- state. This is the weakest usable level of exception safety, and is appropriate where client code can cope with failed operations that have already made changes to objects' state.

2. Strong Guarantee: If an exception is thrown, program state remains unchanged. This level always implies global commit-or-rollback semantics, including that no references or iterators into a container be invalidated if an operation fails. In addition, certain functions must provide an even stricter guarantee in order to make the above exception safety levels possible:

3. Nothrow Guarantee: The function will not emit an exception under any circumstances. It turns out that it is sometimes impossible to implement the strong or even the basic guarantee unless certain functions are guaranteed not to throw (e.g., destructors, deallocation functions).

(* http://www.boost.org/more/generic_exception_safety.html

Exception safe copy assignement → two steps:

First, provide a nonthrowing Swap() function that swaps the guts (state) of two objects:

```
void T::Swap( T& other ) // throw()
{ /* ...swap the guts of *this and other... */ }
```

Second, implement operator=() using the "create a temporary and swap“ idiom:

```
T& T::operator=( const T& other ) {
  T temp( other ); // do all the work off to the side
  Swap( temp );    // then "commit" the work using
                   // nonthrowing operations only
  return *this;
}
```

2. Klassen in C++

Beispiel Stack: stack.h

```
class Stack {
   int max, top;
   int *data;
   void swap(Stack&); // throw ();

protected:
   int* get_data() const {return data;}
   int  get_top() const  {return top;}
   int  get_max() const  {return max;}
public:
    explicit Stack(int dim=100);
    Stack(const Stack&);
    Stack& operator=(const Stack&);

    virtual ~Stack();
    virtual void push (int i);
    int pop();
    int full() const;
    int empty() const;
};
```

2. Klassen in C++

Beispiel Stack: stack.cc

```
//...
Stack::Stack(int dim): max(dim), top(0), data(new int[dim]) { }

Stack::Stack(const Stack& o):max(o.max),top(o.top),data(new int[o.max]) {
        for (int i=0; i<top; ++i) data[i] = o.data[i];
}

#include <algorithm>
void Stack::swap(Stack& other) {       // never fails:
   std::swap(max, other.max);          // swapping
   std::swap(top, other.top);          // buildin types
   std::swap(data, other.data);        // always succeeds
}

Stack& Stack::operator=(const Stack& src) {
   Stack temp (src);  // in case of failure: no change to this
   swap(temp);        // succeeds always
   return *this;
}
//...
```

`const` reicht zur Unterscheidung von überladenen Funktionen (auch Operatoren) aus

typisches Idiom:

```
class Vector { int* data; int dim;
        void check(int i) const // WHY const?
        {if (i<0 || i>=dim) throw std::out_of_range("Vector");}
public:
        Vector(int d, int val=0): dim(d), data(new int[d])
        {for (int i=0; i<dim; ++i) data[i]=val;}
        Vector(const Vector&); // deep copy
        Vector& operator=(const Vector&); // deep assign
        int  operator[] (int i) const { check(i); return data[i]; }
        int& operator[] (int i)       { check(i); return data[i]; }
};
const Vector cv(20, 3); int i = cv[11]; // NOT cv[11] = 3;
Vector v(20, 4); int j = v[13]; v[13] = 7;
```

Nutzerdefinierte Ein- und Ausgabe

- warum **friend** ?
- warum **i/o-stream** Referenzen?
- warum **SomeClass** Referenzen?

```
class SomeClass { ...

friend std::ostream& operator<<
            (std::ostream&, [const] SomeClass [&]);

friend std::istream& operator>>
            (std::istream&, SomeClass &c)

};
SomeClass o;
cout<<o<<endl;      // op<< ( op<< ( cout, o ), endl );
cin>>o;             // op>> ( cin, o );
```

```
// another intermezzo: what's wrong with this code ?
// © www.gimpel.com: bug of the month january 2003
#include <iostream>
struct WhiteHouse{
      int *p;
      WhiteHouse(int n): p(new int) { *p = n; }
      ~WhiteHouse() { delete p; }
};
WhiteHouse ww(1912);
void f() { WhiteHouse fdr(1932); fdr = ww; }
int main() {
      f(); WhiteHouse gwb(2000);
      std::cout << *ww.p; return 0;
}
```

Überladung von **new** und **delete**

1. *Replacement* der impliziten globalen Operatoren

   sämtliche Anforderungen und Freigaben von dynamischem Speicher nutzt dann diese: tiefer Eingriff in Laufzeitsystem, nichts für den Gelegenheitsprogrammierer

```
// Definition einer der impliziten Operationen
void* operator new(std::size_t) throw(std::bad_alloc);
void* operator new[](std::size_t) throw(std::bad_alloc);
void operator delete(void*) throw();
void operator delete[](void*) throw();
void* operator new(std::size_t, const std::nothrow_t&) throw();
void* operator new[](std::size_t, const std::nothrow_t&) throw();
void operator delete(void*, const std::nothrow_t&) throw();
void operator delete[](void* , const std::nothrow_t&) throw();
```

1. *Replacement* der impliziten globalen Operatoren

```
T* t = new T;      // ::operator new(sizeof(T)); !
delete t;          // ::operator delete(t);
t = new T[n];      // ::operator new[](sizeof(T)*n); !
delete[] t;        // ::operator delete[](t);

T* t = new (std::nothrow) T;  // returns 0 if it fails
delete(std::nothrow) t;
t = new (std::nothrow) T[n];  // returns 0 if it fails
delete[] (std::nothrow) t;

                   ! throws std::bad_alloc if it fails
```

2. Neudefinition von globalen Operatoren

```
void* operator new/new[] (std::size_t, weitereParameter);
void operator delete/delete[] (void*, weitereParameter);
```

außer den sog. *placement*-Operationen, die nicht *displaceable* sind:

These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library.

```
void* operator new/new[] (std::size_t, void*);
void operator delete/delete[] (void*, void*);
...
char place[sizeof(Something)];
Something* p = new (place) Something();
delete (place) p;
```

2. Neudefinition von globalen Operatoren

```
// Beispiel: allocation trace
void* operator new (std::size_t s, const char* info = 0) {
        if (info)
                printf("%s\n", info); // NOT cout<<info<<endl;
        void* p = calloc(s, 1); // zero-initialized
        if (!p) { ...some rescue action ... }
        return p;
}
void operator delete (void* p, const char* info = 0) {
        if (info)
                printf("%s\n", info);
        if (p) free(p);
}
```

3. Klassenlokale Operatoren `new` und `delete`

nur für dynamische Objekte dieses Typs, Vorteil: alle sind gleich groß

--> Pool Allocators

```
class X  {
public:
      void* operator new (std::size_t); // bzw. Varianten
      void operator delete (void* p);   // bzw. Varianten
};

X* px = new X; // X::operator new(sizeof(X));
delete px;     // X::operator delete(px);
```

Typumwandlungen

Konstruktoren (die mit einem Argument aufrufbar sind) fungieren als Typumwandler (von 1. Argumenttyp in den Klassentyp)

```
class X  {
public:
      X(int, double = 0);              // int ---> X
      X(char*, int = 1, int = 2);      // char* ---> X
};
void f(X);
X g() { return 0;}
class Y       {
public: Y(X); };    // X ---> Y
```

Typumwandlungen

```
int main()
{
        X x1 = 1;               // 1 --> X
        X x2 = "ein X";         // "ein X" --> X
        f(2);                   // 2 --> X
        x2 = g();               // 0 --> X
        Y y = 0; //  ERROR: Cannot convert 'int' to 'Y'
}
```

Es kommt maximal EINE nutzerdefinierte Typumwandlung zum Einsatz !

Typumwandlungen

Falls automatische Umwandlung per Konstruktor unerwünscht ist, kann man solche als *explicit* spezifizieren:

```
class X  {
public:
        explicit X(int, double = 0);
        ...
};

...
f(2);          // ERROR: keine implizite Umwandlung 2 --> X
f(X(2));       // OK
```

Typumwandlungen

Ziel der Umwandlung durch Konstruktoren ist immer ein Klassentyp

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

```
class Bruch  { int z, n;
public:
        Bruch (int zaehler = 0, int nenner = 1)
                : z(zaehler), n(nenner) {}
        operator double() { return double(z)/n; }
        ...
};
Bruch halb(1,2); std::sqrt(halb); ....
```

kein Rückgabetyp ! keine Argumente !

Typumwandlungen durch *Conversion Operators* sind normalerweise mit Informationsverlust verbunden :-)

Umwandlung per Konstruktion und Konversion sind gleichberechtigt, jede Mehrdeutigkeit ist ein statischer Fehler!

```
class B { public: operator int(); };
class C { public: C(B); };
C operator+ (C c1, C c2) {return c1; } // mal kein friend !

C foo (B b1, B b2) { return b1+b2; }
// Ambiguity between 'operator +(C,C)' and 'B::operator int()' in function foo(B,B)
```

Ziel einer Konversion kann ein beliebiger Typ sein (z.B. auch ein Zeigertyp)

```
struct X {
  virtual operator const char*() { return "X"; }
};
struct Y : public X {
 virtual operator const char*() { return "Y"; };
};
int main() {
              X* p = new Y;
              cout <<  p << endl;
              cout << *p << endl;
}
```

```
C:\tmp>conv2
007B33E0
Y
```

Konversionen sind nicht 'abschaltbar' (wie *explicit ctors*) ggf. Memberfunktionen **`toType()`** bevorzugen !

sogar eine Konversion nach **void*** kann u.U. sinnvoll sein

```
// bcc32: ios.h (ähnlich in anderen Impl.)
inline basic_iosT::operator void*() const {
    return fail() ? (void*)0 : (void*)1;
}

// basic_iosT ist Basiklasse von ostream, ofstream

ofstream output ("file.txt");
// wenn die Dateien nicht zum Schreiben eröffnet
// werden konnte, ist das intern in einem Status
// vermerkt, den fail() abfragt:
if (output.fail()) ...  // ODER: VIEL KOMPAKTER
if (!output) ...
```

sämtliche Typumwandlungen (Konstruktion und Konversion) werden bei Bedarf implizit (außer bei *explicit ctors*) veranlasst, aber auch bei expliziten *Cast*-Operationen

```
T1 t1;
T2 t2 = (T2) t1; // oder auch
T2 t2 = T2 (t1); // falls T2 ein Typname (kein Typkonstrukt) ist
```

Casts sind syntaktisch eher unauffällig, werden in unterschiedlichsten Absichten (und z.T. mit nicht erkennbarem Risiko!) eingesetzt

```
X = 2 / double(3);  // OK
class B: public A {....};
A *pa = new B; B* pb = (B*)pa; // OK
cout << (void*)pa; // OK
int *pi = new int; int i = int(pi); // ???
const X x; X* px = (X*)&x;  // ???
class X{}; class Y{};
X *px = new X; Y* py = (Y*)px; // ???
```

Um die Semantik besser ausdrücken zu können (und dem Compiler mehr Prüfmöglichkeiten zu geben) bietet C++ vier spezielle *Cast*-Operatoren

```
T1 t1;
T2 t2 = const_cast<T2> (t1);
T2 t2 = static_cast<T2> (t1);
T2 t2 = reinterpret_cast<T2> (t1);
T2 t2 = dynamic_cast<T2> (t1);
```

const_cast<T>

» die Konstantheit eines Objektes ignorieren «

verletzt eigentlich die "Spielregeln": alle schreibenden Zugriffe nach Brechung der *constness* haben *undefined behaviour*

aber manchmal aus praktischen Gründen unumgänglich

```
// use std::string instead of [const] char*

string s ("simsalabim");

// but:
extern "C" void someOldCfunction (char*);
...
someOldCfunction(s.c_str()); // ERROR: const ignored
someOldCfunction(const_cast<char*>(s.c_str())); // OK
```

für einige häufige Anwendungsszenarien bietet C++ eine bessere Variante: **mutable**

```
class X {
      int copies;
public:
      X(): copies(0){}
      // Copy-Ctor: one of
      X(X& other) { other.copies++; }
      // kann keine Kopien von Konstanten machen
      // or:
      X(const X& other) { other.copies++; }
                                          ^ Cannot modify a const object
      int cc() const {return copies;}
};
```

const_cast<X&>(other).copies++;

```
class X {
        mutable int copies;
public:
        X(): copies(0){}
        X(const X& other) { other.copies++; }
        // kann Kopien von Konstanten machen und dabei
        // dennoch other.copies ändern !
        int cc() const {return copies;}
};
```

`mutable` immer benutzen, wenn Objekte logisch konstant, aber in (Implementations-) Details veränderlich sein sollen (z.B. Objekte mit *lazy evaluation* gewisser Eigenschaften)

static_cast<T>

» den Compiler überreden, verwandte Typen verträglich zu verwenden «
das Ergebnis kann ohne erneute Umwandlung verwendet werden

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

// eine Y& ist auch immer eine X&
Y o;
X& x1 = o; // implizite Anpassung der Typen
X& x2 = static_cast<X&> (o); // dasselbe

// manchmal ist eine X& auch eine Y&
Y& y1 = static_cast<Y&> (x1); // ok, weil x1 ein Y ref.
// aber eben nicht immer:
X& x3 = *new X; Y& y2 = static_cast<Y&> (x3); // Crash ahead
```

reinterpret_cast<T>

» den Compiler überreden, nicht verwandte Typen verträglich zu verwenden «

das Ergebnis kann nur nach erneuter Rückumwandlung verwendet werden

die unveränderte Bitbelegung wird anders interpretiert; zumeist nicht portabel

```
int *pi = &someint;
void *v = reinterpret_cast<void*>(pi);
// don't use v, but:
int *p = reinterpret_cast<int*>(v);
*p = 337; // OK: sets someint
```

dynamic_cast<T>

» zur Laufzeit verwandte Typen verträglich und sicher verwenden «

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


// manchmal ist eine X& auch eine Y&
Y& y1 = dynamic_cast<Y&> (x1); //ok,weil x1 ein Y ref.
// aber eben nicht immer:
X& x3 = *new X;
Y& y2 = dynamic_cast<Y&> (x3); // NO Crash ahead
// exception std::bad_cast !!!
```

dynamic_cast<T>

Implementation setzt offenbar Auswertung von Laufzeittypinformationen (RTTI - *run time type identification*) voraus

Funktioniert für Zeiger und Referenzen polymorpher Typen (es muss virtuelle Funktionen in der Basisklasse geben!)

Ein *downcast* (von einem Zeiger/einer Referenz auf eine Basisklasse auf einen Zeiger/eine Referenz einer Ableitung) gelingt, wenn das referenzierte Objekt vom Typ der Ableitung oder einer Ableitung dieser ist.

Bei Zeigern liefert **`dynamic_cast`** den Wert 0, bei Referenzen wird die Ausnahme **`std::bad_cast`** geworfen, wenn die dynamische Typ nicht ausreicht

dynamic_cast<T>

```
class A {
public:  virtual void needed () {}
};
class B: public A {public: int i;};
class C: public B {public: int j;};
int main() {
        A *pa = new B;
        B *pb = dynamic_cast<B*>(pa);
        if (pb) pb->i = 12345; // ok, es ist ein B
        C *pc = dynamic_cast<C*>(pb);
        if (pc) pc->j = 54321; // wird nicht ausgefuehrt
        pa = new C;
        pb = dynamic_cast<B*>(pa);
        if (pb) pb->i = 12345; // ok, es ist ein B
}
```

Darüber hinaus kann man die Typidentität direkt abfragen:

dazu existiert der Operator **`typeid`** (wie **`sizeof`** vom Compiler umgesetzt und nicht überladbar), der eine (vergleichbare) Struktur des Typs **`type_info`** liefert, der Vergleich von **`type_info`** gelingt, wenn exakt der gleiche Typ vorliegt

auf **`type_info`** ist wiederum die Funktion **`name()`** definiert, die einen Klarnamen der Klasse (nicht notwendig identisch mit dem Klassennamen) erzeugt

`#include <typeinfo>` ist erforderlich

Die beteiligten Typen müssen wiederum polymorph sein, d.h. mindestens eine virtuelle Funktion in der gemeinsamen Basis besitzen

2. Klassen in C++

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

class A { virtual void any () {} };
class B: public A { };
class C: public A { };
void check (A* p) {
        if (typeid(*p)==typeid(A))
                { cout << "es ist ein A\n"; return; }
        if (typeid(*p)==typeid(B))
                { cout << "es ist ein B\n"; return; }
        cout << "weder A noch B\n";
}
const char* get_name(A* p) {
        return typeid(*p).name();
}
```

2. Klassen in C++

```
int main()
{
        A *p;
        p = new A;
        check (p);
        cout << get_name(p) << endl;
        p = new B;
        check (p);
        cout << get_name(p) << endl;
        p = new C;
        check (p);
        cout << get_name(p) << endl;
}
```

```
C:\tmp>rtti
es ist ein A
A
es ist ein B
B
weder A noch B
C
```

dynamic_cast<T> ist manchmal nicht zu vermeiden

```
class B {
        // no functionality 'foo'
};
class D: public B {
        virtual void foo();
};
void register (B*);
B* next();
...
register(new D);
...
B* n = next();

// how to call foo ?
dynamic_cast<D*>(n)->foo();
```

RTTI nur in Ausnahmefällen explizit benutzen

statt *spaghetti code*

```
Shape * s;
if (typeid(*s)== typeid("Circle"))
((Circle*)s)->Circle::draw();
else
if (typeid(*s)== typeid("Rectangle"))
      ((Rectangle*)s)-> Rectangle::draw();
else ...
```

benutze

```
Shape * s;
s->draw(); // late bound virtual
```

Mehrfachvererbung (*multiple inheritance*)

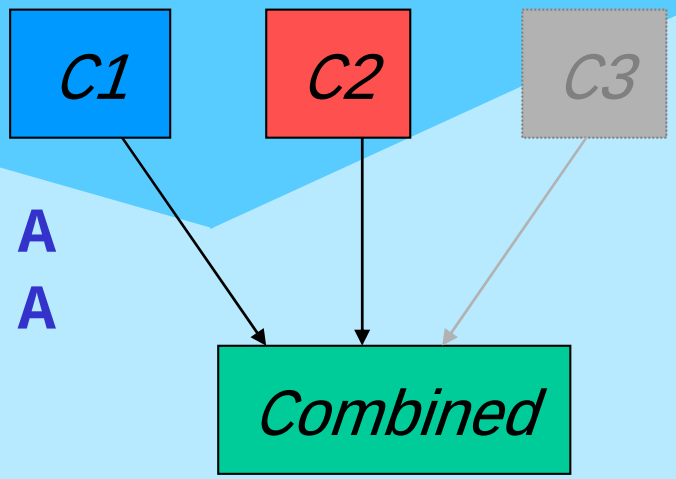
eine Klasse kann mehrere Basisklassen haben -->
'freie' Kombination von Konzepten:

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

jedes **`Combined-`**Objekt IST EIN **`Concept1`** und
IST EIN **`Concept2`** (**`Concept3`**-Abstammung ist ein Implementationsdetail)

Mehrfachvererbung (*multiple inheritance*)
Polymorphie bleibt erhalten:

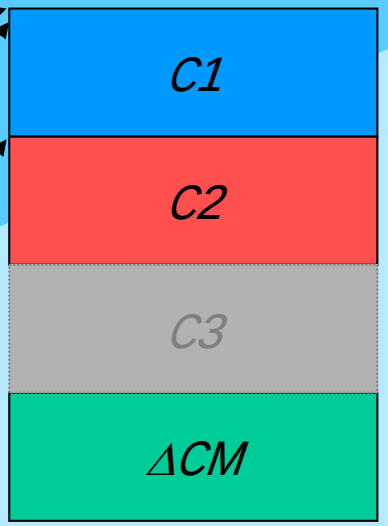
```
Combined obj;
Combined * cm = &obj;
Concept1 * c1 = cm; // IS A
Concept2 * c2 = cm; // IS A
// NOT c2=c1=cm;

// it's the same Object:
if (c1 == cm) // yes !
if (c2 == cm) // yes !
if (c1 == c2) // ERROR: uncomparable
```



Mehrfachvererbung (*multiple inheritance*)

Layout muss linearisiert werden:

```
Combined obj;
Combined * cm = &obj;
Concept1 * c1 = cm; // IS A
Concept2 * c2 = cm; // IS A
```




```
// but the (numeric) adresses may differ:
if(reinterpret_cast<void*>c1==reinterpret_cast<void*>cm)
      // yes or no!
if(reinterpret_cast<void*>c2==reinterpret_cast<void*>cm)
      // yes or no!
```

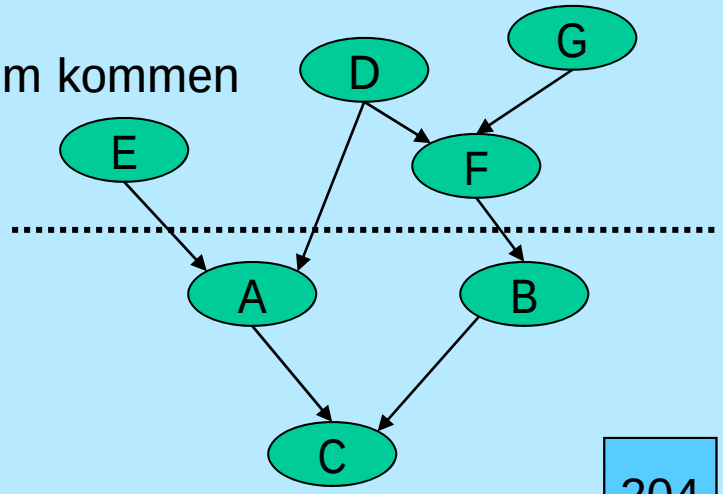
Mehrfachvererbung (*multiple inheritance*)

eine Klasse kann eine andere nicht direkt mehrfach einerben

```
struct A { int i; };
class B: public A, public A {  // NOT ALLOWED
       void foo(){ i = 0; /* which i ? A::i ? which A ? */ }
};
```

ansonsten kann es durchaus zu Maschen im Baum kommen

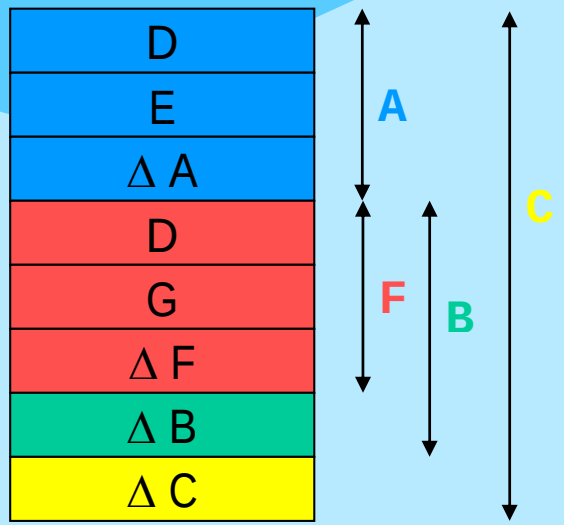
```
class A: public D, public E {...};
class F: public D, public G {...};
class B: public F {...};
-----------------------------------
class C: public A, public B {...};
```



Mehrfachvererbung (*multiple inheritance*)

ob dabei der mehrfach eingeerbte Basisklassenanteil dupliziert oder unifiziert im resultierenden Objekt erscheint ist steuerbar:

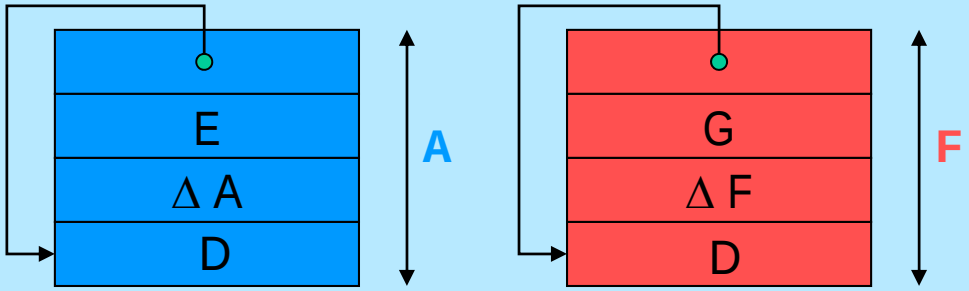
```
class A: public D, public E {...};
class F: public D, public G {...};
class B: public F {...};
class C: public A, public B {...};
```

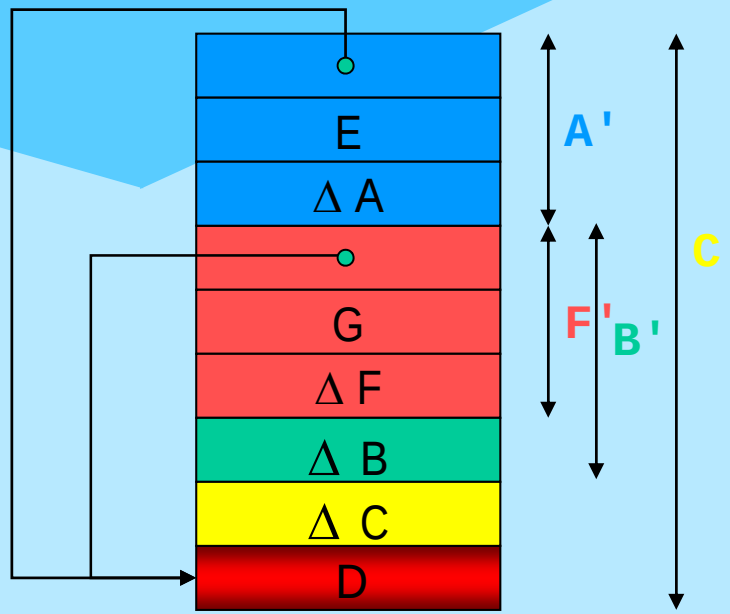


Mehrfachvererbung (*multiple inheritance*)

ob dabei der mehrfach eingeerbte Basisklassenanteil dupliziert oder unifiziert im resultierenden Objekt erscheint ist steuerbar:

```
class A: virtual public D,
                public E {...};
class F: virtual public D,
                public G {...};
class B: public F {...};
class C: public A, public B {...};
```





Mehrfachvererbung (*multiple inheritance*)

für beide Szenarien gibt es sinnvolle Anwendungen:

```
// class Listable {/* Listeneigenschaften */ };
class A: public Listable { ... };
// A's können in einer Liste erfasst werden
class B: public Listable { ... };
// B's können in einer Liste erfasst werden
class C: public A, public B { ... };
// C's können in zwei separaten Listen (als A und als B)
// erfasst werden
-------------------------------------------------------
class Person {...};
class Angestellter: public virtual Person {...};
class Student: public virtual Person {...};
class Werkstudent: public Angestellter, public Student
{...}; // ein und dieselbe Person !!!
```

non virtual

virtual

Mehrfachvererbung (*multiple inheritance*)

Durch die freie Kombination kann es leicht zu Mehrdeutigkeiten kommen

Falls diese nicht auflösbar sind, liegt ein statischer Fehler vor (s.o. `B: A,A`)
Aber auch:

```
class A {public: int i;};      class B: public A{};

class C: public A, public B {}; // ERROR
// i ... which i ? A::i ? which A::i ?
```

Mehrdeutigkeiten, die durch *scope resolution* auflösbar sind, sind erlaubt

```
struct A { int i; };  struct B { int i; };
class C: public A, public B {     i=1; // ERROR
                                  A::i=1; // OK
                                  B::i=1; // OK
};
```

Mehrfachvererbung (*multiple inheritance*)

Selbst bei virtuellen Basisklassen kann es auf Grund der Maschenbildung sein, dass ein Name eines Members auf mehreren "Wegen" auflösbar ist und zu verschiedenen Membern führt, Eindeutigkeit liegt dann vor, wenn es (genau) einen kürzesten Weg gibt *»Dominanzregel«* (ansonsten muss ebenfalls qualifiziert werden)

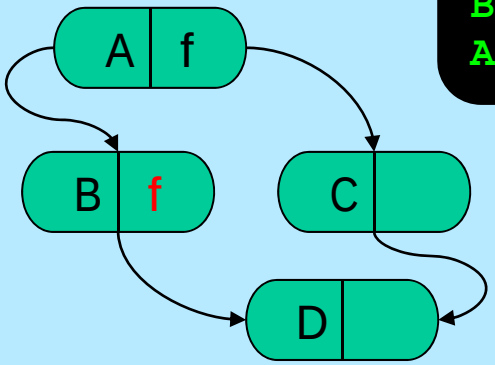
```
class A { public: void f(){cout<<"A::f()\n";}  };
class B: public virtual A {
          public: void f(){cout<<"B::f()\n";}
};
class C: public virtual A {
          public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C {};
int main() {   D d;
               // d.f(); ERROR: ambiguous access of 'f'
               d.A::f(); // ok
               B *pb = &d; pb->f(); // ok
               C *pc = &d; pc->f(); // ok
}
```

Mehrfachvererbung (*multiple inheritance*)

Selbst bei virtuellen Basisklassen kann es auf Grund der Maschenbildung sein, dass ein Name eines Members auf mehreren "Wegen" auflösbar ist und zu verschiedenen Membern führt, Eindeutigkeit liegt dann vor, wenn es (genau) einen kürzesten Weg gibt *»Dominanzregel«* (ansonsten muss ebenfalls qualifiziert werden)

```
class A { public: void f(){cout<<"A::f()\n";}  };
class B: public virtual A {
           public: void f(){cout<<"B::f()\n";}
};
class C: public virtual A {};
class D: public B, public C {};
int main() {   D d;
               d.f();
               d.A::f();
               B *pb = &d; pb->f();
               C *pc = &d; pc->f();
}
```

```
B::f()
A::f()
B::f()
A::f()
```

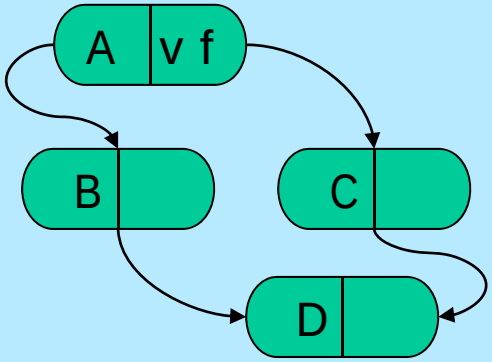


Mehrfachvererbung (*multiple inheritance*)

Mehrfachvererbung und virtuelle Funktionen sind miteinander kombinierbar, im Falle von virtuellen Basisklassen stehen u.U. ebenfalls mehrere Wege der Auflösung zur Verfügung: fall keine dominate Implementation existiert, muss in der am weitesten abgeleiteten Klasse eine Redefinition erfolgen

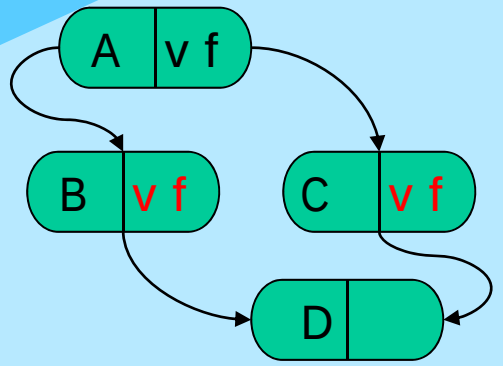
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A { };
class C: public virtual A { };
class D: public B, public C { };
main(){         D d;
                C *pc = &d;
                pc->f();
}
```

```
A::f()
```



Mehrfachvererbung (*multiple inheritance*)

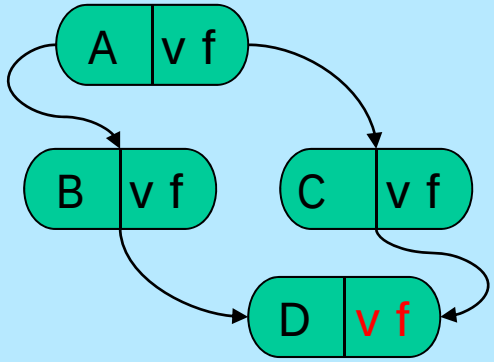
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A {
  public: void f(){cout<<"B::f()\n";}
 };
class C: public virtual A {
  public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C { };
// ERROR: no unique final overrider for f() in D
```



Mehrfachvererbung (*multiple inheritance*)

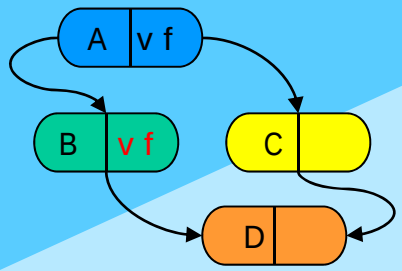
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A {
  public: void f(){cout<<"B::f()\n";}
 };
class C: public virtual A {
  public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C {
  public: void f(){cout<<"D::f()\n";}
};
... D d; C *pc = &d; pc->f(); ...
```

```
D::f()
```

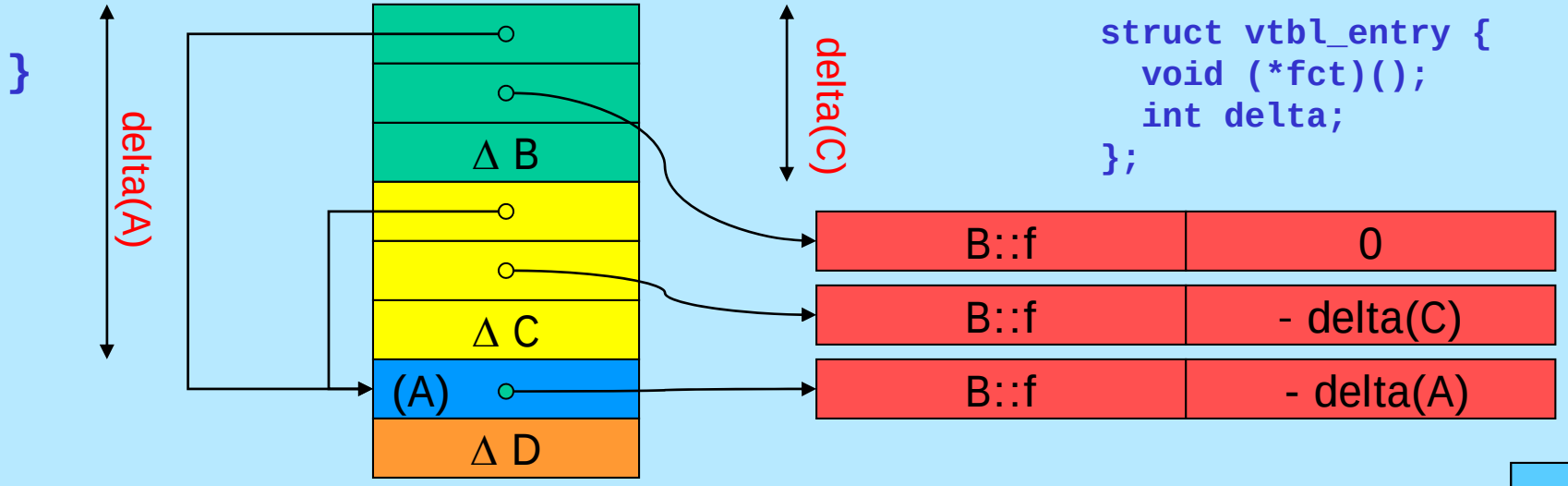


Mehrfachvererbung (*multiple inheritance*)

Implementation von virtuellen Funktionen wird *'slightly more complicated'*




```
main(){        D d;
               C *pc = &d;
               pc->f();
}
```




```
struct vtbl_entry {
  void (*fct)();
  int delta;
};
```

Mehrfachvererbung (*multiple inheritance*)

Implementation von virtuellen Funktionen wird *'slightly more complicated'*

```
typedef struct vtbl_entry {
  void (*fct)();
  int delta;
} VE;
```

```
D d;
A* pa = &d; B* pb = &d; C* pc = &d;
pa->f();
// VE* vt = &pa->vtbl[index(f)];
// (*vt->fct)((B*)((void*)pa + vt->delta));
pb->f();
// VE* vt = &pb->vtbl[index(f)];
// (*vt->fct)((B*)((void*)pb + vt->delta));
pc->f();
// VE* vt = &pc->vtbl[index(f)];
// (*vt->fct)((B*)((void*)pc + vt->delta));
```

Mehrfachvererbung (*multiple inheritance*)

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

```
class A { public: A(int); };
class B: public virtual A {
  public: B(): A(1){ .... }
};
class C: public virtual A {
  public: C(): A(2){ .... }
};

class D: public B, public C {
// public: D() { .... } // ERROR: no matching function for call to `A::A ()'
   public: D(): A(3) { .... }
};
```

Mehrfachvererbung (*multiple inheritance*)

Potentielle Mehrdeutigkeiten werden unabhängig von Zugriffsrechten lokalisiert !

```
class A {
  private: void m();
};
class B {
  public:  void m();
};
class C: public A, public B {
   void f() {
     // m(); // Fehler: Mehrdeutigkeit
     A::m(); // Fehler: kein Zugriff
     B::m(); // ok
   }
};
```

Mehrfachvererbung (*multiple inheritance*)

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

```
class A { public: int i; };
```

```
class B: private virtual A {};
class C:  public virtual A {};
class D: public B, public C {
  void f() { i++; /* erlaubt, da B: .... public A */ }
};
-------------------------------------------------------
class B: private A {};
class C: public A {};
class D: public B, public C {
void f() {
  // i++; // Fehler: Mehrdeutigkeit
  C::i++; // ok
  // B::i++; // Fehler: kein Zugriff
};
```

Namespaces

Problem: Namenskollision im globalen Namensraum, Klassen sind zwar ein Hilfsmittel zur Entlastung des globalen Namensraumes, Klassennamen sind ihrerseits jedoch (zumeist) wiederum globale Bezeichner `string, String, XtString, QtString, Matrix`

Lösung `namespace`: Deklaration wie Klassen, Verschachtelung erlaubt (aber keine Vererbung, Zugriffsrechte, ...)

```
namespace Humboldt_Universitaet {
        class Fachbereich { //...
        };
        class Student;
        void registriere(Fachbereich&, Student&);
} // ; muss hier nicht stehen im Gegensatz zu class !
```

Namespaces dürfen beliebige Deklarationen und Definitionen enthalten (auch Namespaces), Klassen dürfen lokale Klassen enthalten aber keine Namespaces, Typen (Klassen) dürfen nach ihrer Verwendung nicht lokal neu definiert werden

```
namespace X {
        namespace Y {
                typedef int B;
                class A {
                        B i;
// ERROR:                 class B {};     // changes meaning of 'B' from
                                           // 'typedef int X::Y::B'
                        class C {};
                public:
                        class D {};
                };
        }
}
// ERROR:       X::Y::A::C c; // 'X::Y::A::C' is not accessible
                X::Y::A::D d; // OK
```

namespace reopening erlaubt zusätzliche Deklarationen, fehlende Definitionen, logische Verteilung über separate Dateien (nicht für **`namespace std`** erlaubt)

```
namespace Humboldt_Universitaet { // ...
       void registriere (Fachbereich& f, Student& s)
       {
         // how this is done ...
       }
} // gehört zum gleichen namespace
```

Definitionen auch im umhüllenden *namespace* möglich

```
class Humboldt_Universitaet::Student {
   //...
};
```

Namen von äußeren *namespaces* sind wiederum globale Gebilde

--> spricht für lange (und damit) eindeutige Namen

praktische Verwendung

--> spricht für kurze Namen

Lösung: **`namespace`** Aliasnamen

```
namespace HU = Humboldt_Universitaet;
// as I'll refer it further
```

Es gibt zwei Möglichkeiten der "Bereitstellung" von Elementen aus **namespaces**

1. Mit einer **using**- Deklaration wird ein Name aus einem Namensbereich direkt in den Geltungsbereich eingeführt, in dem die **using** - Deklaration erfolgt (als wäre es dort deklariert worden).

```
void doit(){
   using HU::registriere;
   registriere(Informatik, Markus_Mustermann);
}
```

2. Durch eine **using**-Direktive können sämtliche Namen des angegebenen Namensbereichs für den Geltungsbereich zugreifbar gemacht werden, in dem die **using** - Direktive enthalten ist. Die **using** -Direktive wirkt sich dabei so aus, als seien alle Elemente außerhalb ihres Namensbereichs deklariert, und zwar an der Stelle, an der die Namensbereich-Definition tatsächlich steht.

```
using namespace Humboldt_Universität;
Fachbereich Informatik;
Student *Markus_Mustermann;
```

Achtung

`using namespace N;` und **`using N::`*`name`*** (∀ *name* in N)

sind nicht äquivalent:

```
namespace X {
      int i;
      double x;
}
int main() {
      int i = 1;
      X::i = 10;
      // using X::i;
      // redefinition !!
      using X::x;
      i = 42;
      return 0;
}
```

```
namespace X {
      int i;
      double x;
}

int main() {
      int i = 1;
      X::i = 10;
      using namespace X;
      i = 42;
      return 0;
}
```

Achtung

`using N::anEnumType;` stellt nicht die `enum`-Literale bereit

`using`- Direktiven sollten nie in unbekannten Kontexten (d.h. in denen nicht klar ist, welche Symbole definiert sind) verwendet werden, weil sie dazu führen können, dass Mehrdeutigkeiten oder Verhaltensänderungen entstehen können (Overloading)

--> Vorsicht bei ihrer Verwendung, Benutzung in Header-Files ist "untragbar schlechtes Design" (*Josuttis*) !!

using- Deklarationen können auch benutzt werden, um Zugriff auf Basis-Member abweichend von den sonst geltenden Regeln zu erlauben:

```
class A {
private:   int a1;
protected: int a2;
           void f(char){}
public:    void f(int){}
           int a3;
};
class B: private A {
public:
  using A::a2;
  using A::f; // all f's
};
```

```
int main() {
        A a;
        B b;
// erlaubt ist:
        a.a3 = 3;
        a.f(0);
        b.a2 = 2;
        b.f('A');
        b.f(1);
}
```

alles was sichtbar ist kann per **using** -Deklaration 'weitergereicht' werden

bei überladenen Funktion müssen alle Varianten zugreifbar sein, sonst liegt ein statischer Fehler vor

Anonyme Namensräume als Ersatz für (static) Objekte mit *file scope*.

```
namespace {
  int counter = 0;
  void inc();
}
int main(){inc();}

namespace{
  void inc() { counter++;}
}
```

```
namespace { /*body*/ }
            ==
namespace uniqueForThisFile{}
using namespace uniqueForThisFile;
namespace uniqueForThisFile{/*body*/}
```

Lookup unqualifizierter Namen: zunächst lokal (incl. `using`-Deklarationen) und sonst in allen sichtbaren Namespaces (gleichberechtigt)

Namensauflösung erfolgt **immer** in der Reihenfolge
1. lookup
2. overload resolution
3. access check

```
namespace A {
        void f(){cout<<"A::f()\n";}
        void g(){cout<<"A::g()\n";}
}

namespace B {
        void f(char*){cout<<"B::f(char*)\n";}
}

namespace C {
        using namespace A;
        void f(int){cout<<"C::f(int)\n";}
}
```

```
void f(double){cout<<"::f(double)\n";}

int main()
{
      using namespace B;
      using namespace C;

      f(1);
      f(1.0);
      f();
      g();
      f("Hoho");
}
```

```
C::f(int)
::f(double)
A::f()
A::g()
B::f(char*)
```

2. Klassen in C++

```
// wie zuvor

int main(){
        using B::f;
        using namespace C;

//      f(1);           // ERROR: passing `int' to argument
                        // 1 of `B::f(char *)' lacks a cast
//      f(1.0);         // ERROR: argument passing to `char *'
                        // from `double'

//      f();            // ERROR: too few arguments to
                        // function `void B::f(char *)'
        g();            // OK
        f("Hoho");      // OK
}
```

Lookup qualifizierter Namen: im jeweils benannten Scope beginnend rekursiv(* in weiteren bis der Name gefunden wird

(* **wird bei der Bildung der transitiven Hülle dasselbe Objekt mehrmals gefunden, liegt KEIN Fehler vor**

```
namespace A {
        void f(){cout<<"A::f()\n";}
        void g(){cout<<"A::g()\n";}
}

namespace B {
        void f(char*){cout<<"B::f(char*)\n";}
}

namespace C {
        using namespace A;
        void f(int){cout<<"C::f(int)\n";}
}
```

unverändert !

```
// wie zuvor

int main()
{
//    using namespace B;
//    using namespace C;

      C::f(1);
      ::f(1.0);
      A::f();
//    C::f();      // ERROR:  Too few parameters in
                   // call to 'C::f(int)'
      C::g();
      B::f("Blah");
}
```

ändert gar nichts!

Lookup unqualifizierter Namen hat noch einen wichtigen Sonderfall:

Koenig-Lookup *alias* ADL (argument dependent lookup)

```
namespace N {
  class T {
  public:
    void foo() { N::foo(*this); }
    friend std::ostream& operator<<(std::ostream&, const T&);
    friend void foo (const T&);
  };
}

std::ostream& N::operator<<(std::ostream& o, const T&) {
        return o<<"T-Object"<<std::endl;
}

void N::foo(const T& t) {  std::cout << t; }
```

Koenig-Lookup *alias* ADL (argument dependent lookup)

aus allen Parametertypen eines Funtionsaufrufs wird (rekursiv) eine Menge sog. *associated namespaces/classes* ermittelt, in denen dann die gesuchte Funktion eindeutig gefunden werden muss

```
int main() {
        N::T t;
        t.foo();        // OK: scope durch t festgelegt!
        foo(t);         // wäre ohne ADL fehlerhaft !
                        // dank ADL ok:
        N::foo(t);      // wäre ohne ADL noch akzeptabel
        // nicht aber:
        N::operator<<(std::cout, t); // anstelle von:
        std::cout << t;  // nur mit ADL korrekt
}
```

5x **T-Object**