
TERRA INCOGNITA
STL
BOOST
λ
external libraries seas
THE HOLY STANDARD TERRITORY
The C++ Lands
Its amazing creatures and weird beasts

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 2011**
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

Java-Style

```
#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;
          }
}; // !!!!!!!!!!
```

Ein erster Blick: Hello World

C++

h.cc

```
// ... continued

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

Java-Style

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, 10e10]

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

long
bei 10^{10}

10000000000L
bei 10^{10}

Der zweite Blick: Effizienz

```
#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;
}
```

C++

long bei 10^{10}

100000000000L bei 10^{10}

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

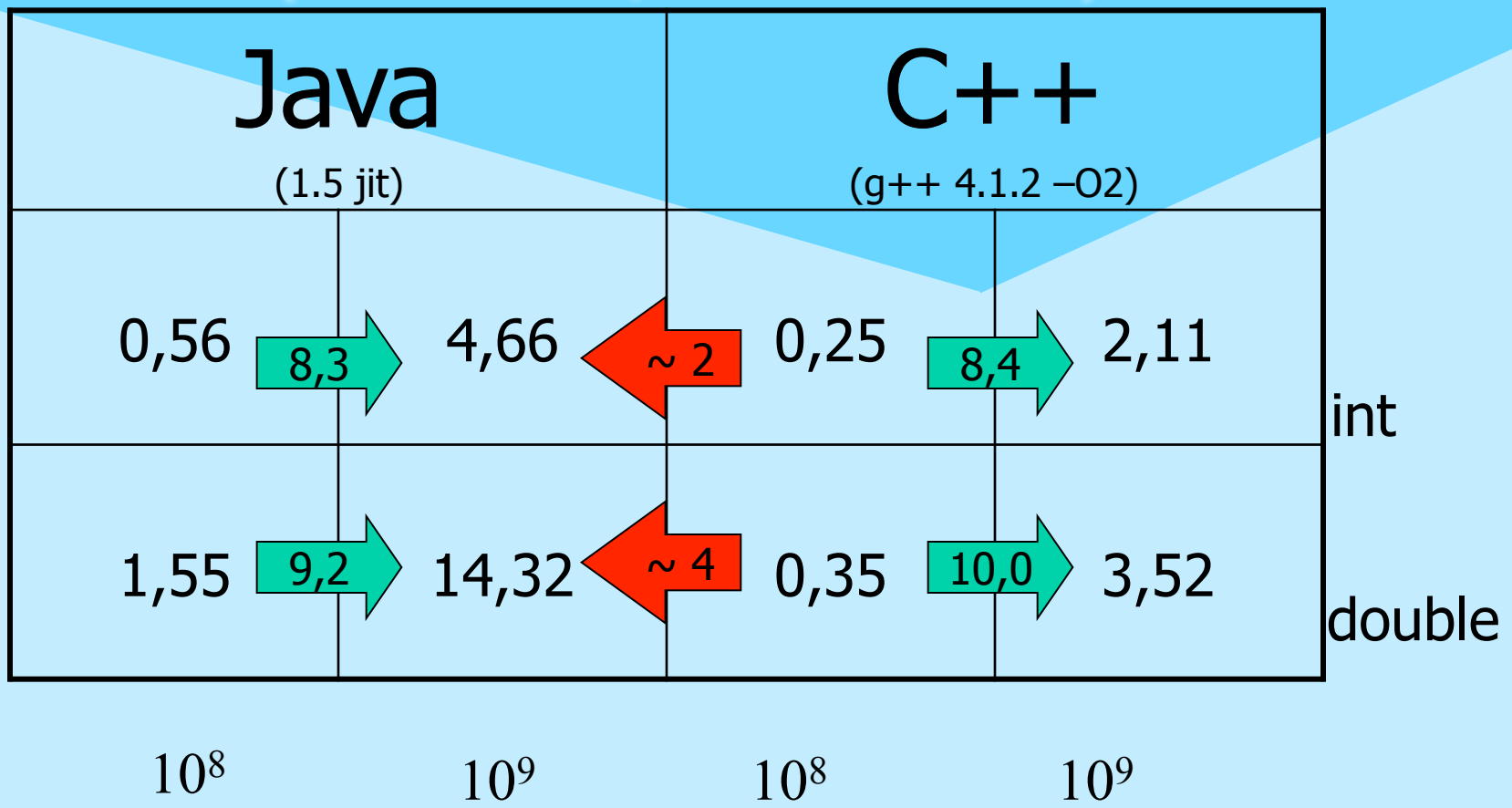
Der zweite Blick: Effizienz

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
}
```

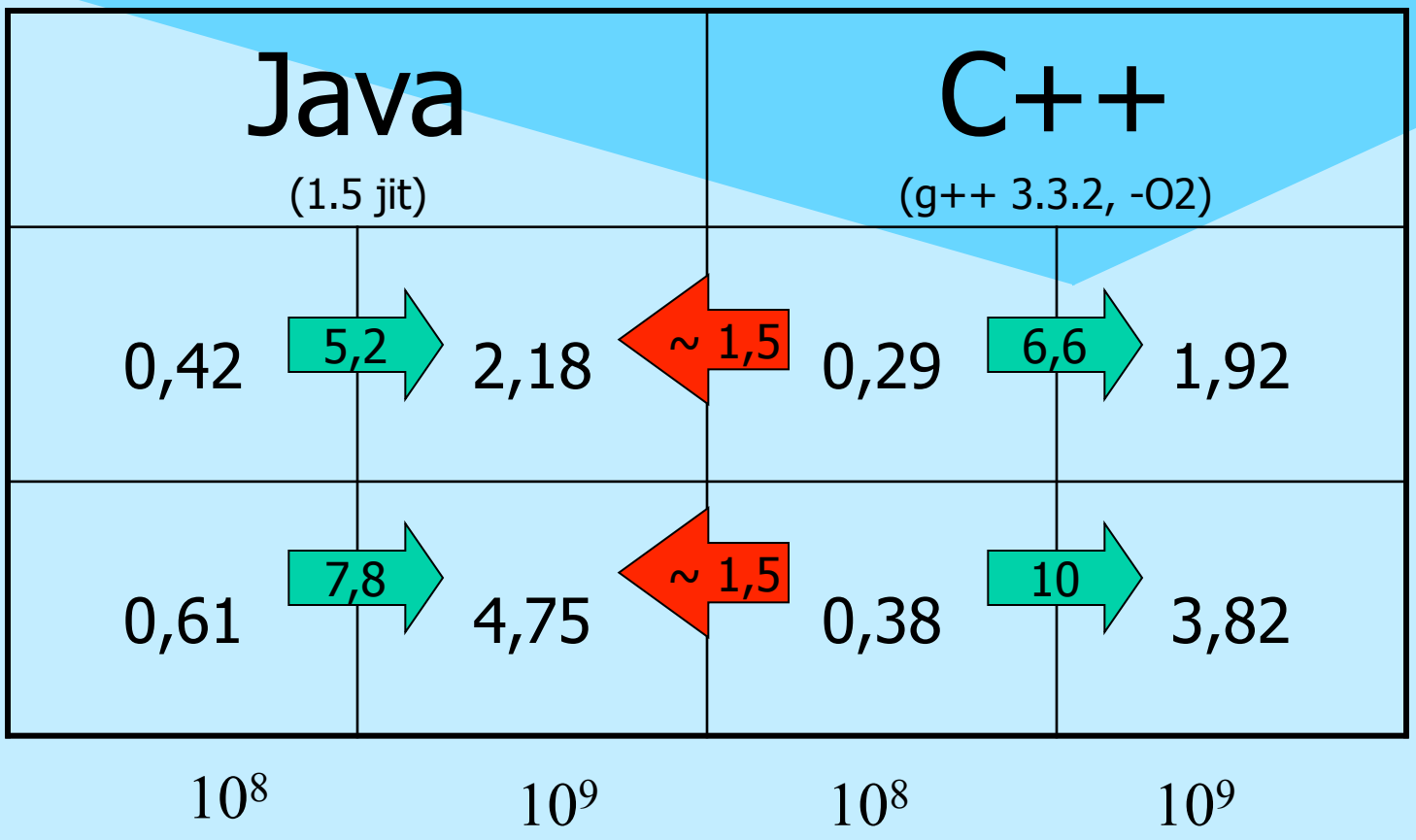
Der zweite Blick: Effizienz

Laufzeiten (Debian Linux, Pentium4 2 GHz)



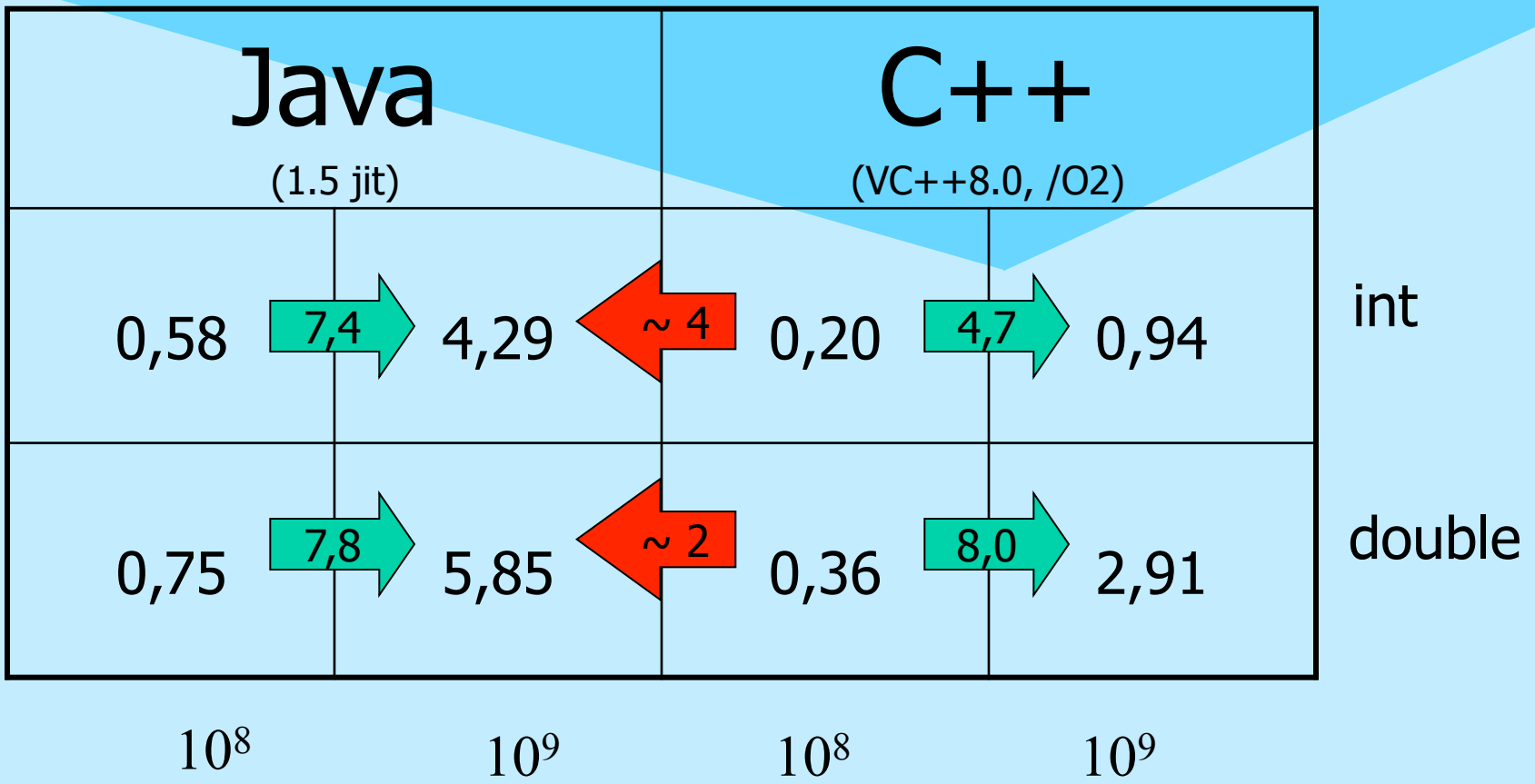
Der zweite Blick: Effizienz

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



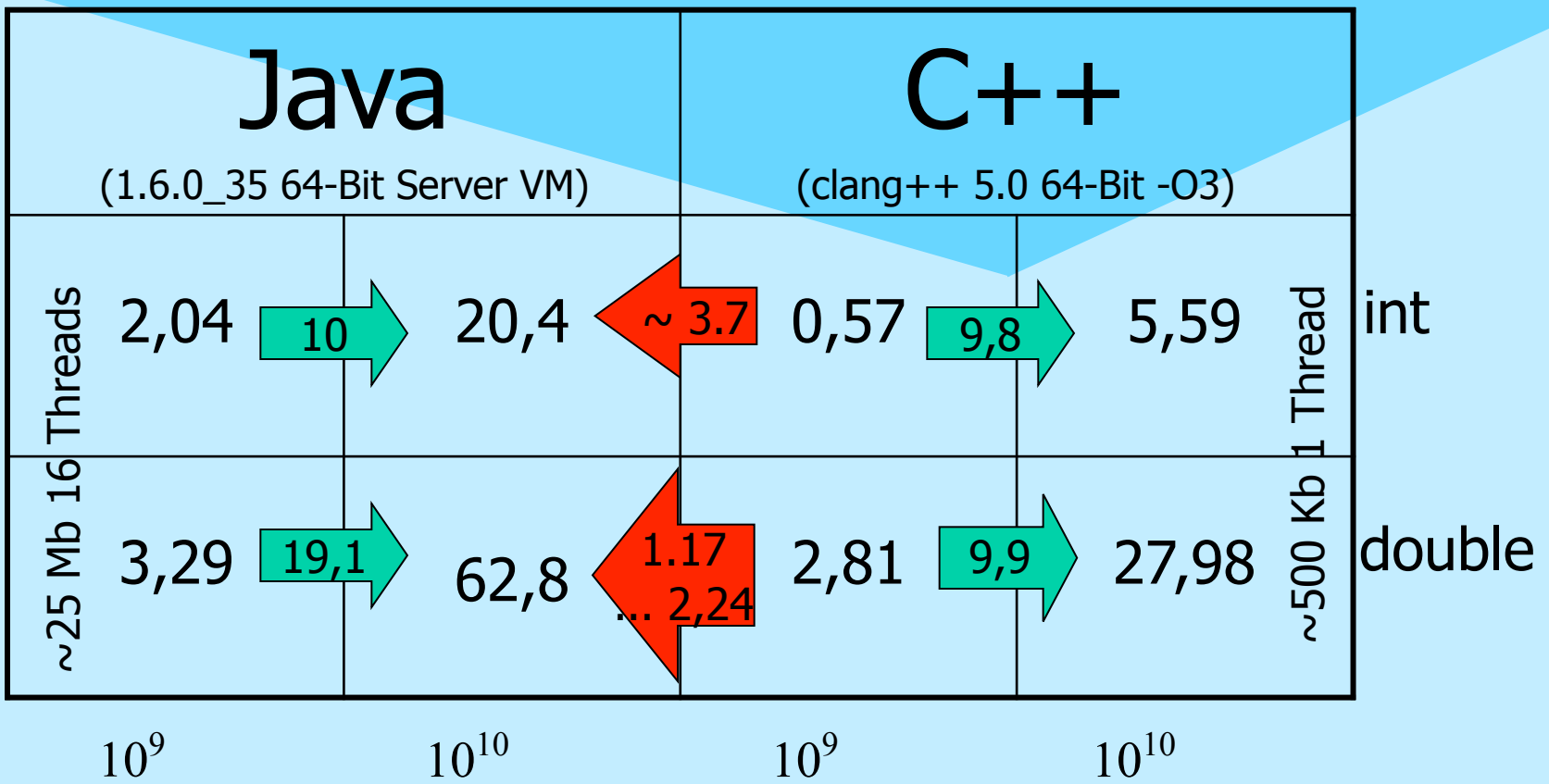
Der zweite Blick: Effizienz

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



Der zweite Blick: Effizienz (up to date :-)

Laufzeiten (Mac OS X 10.9, 1.86 GHz Intel Core 2 Duo)



C++ Historie

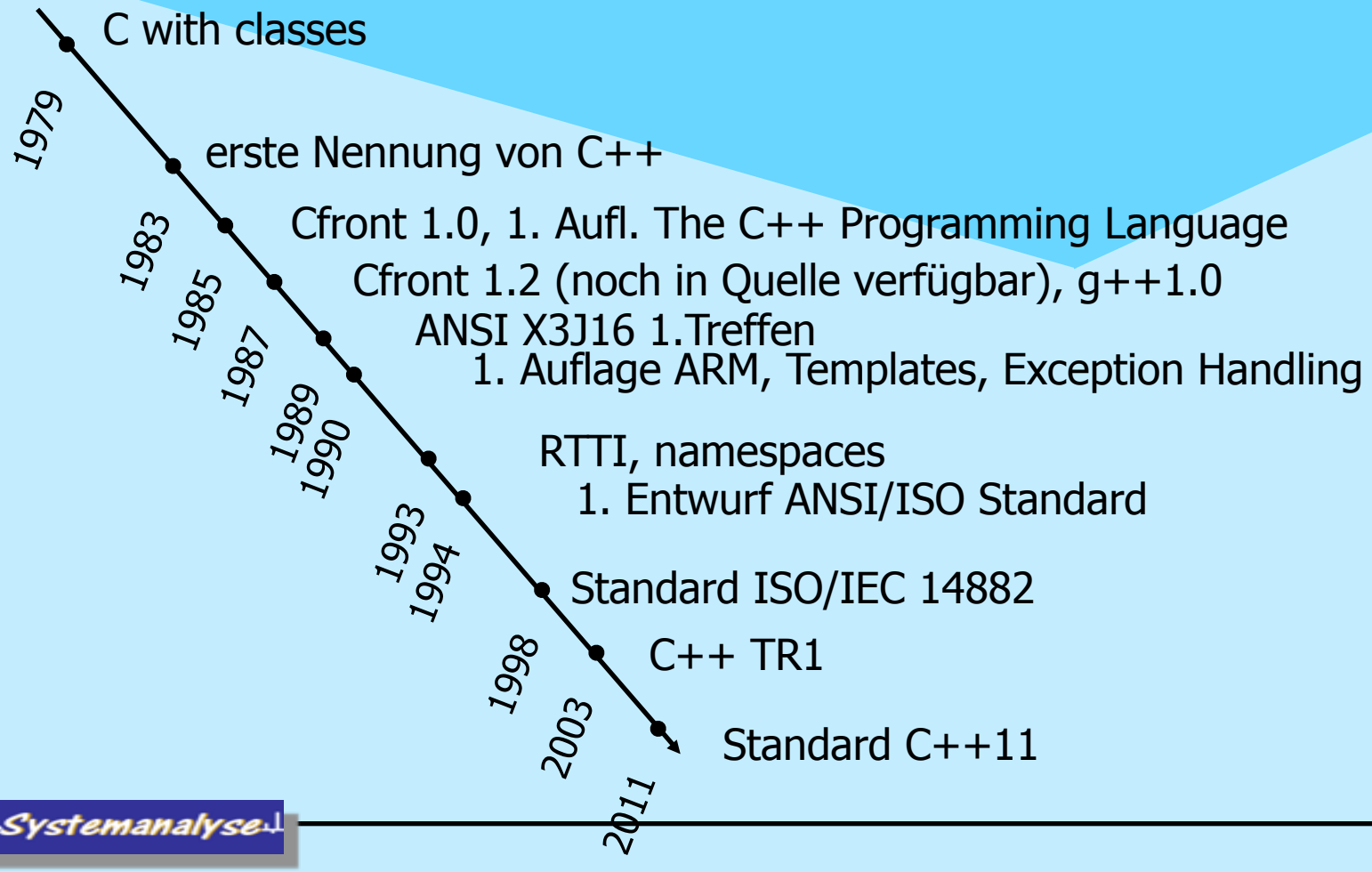
- Bjarne Stroustrup Ph.D. Arbeit 1978/79 an der Universität Cambridge: „Alternative Organisationsmöglichkeiten der Systemsoftware in verteilten Systemen“
- erste Implementation in Simula auf IBM360 (Simula67, NCC Oslo)
- Stroustrup: „Die Entwicklung des Simulators war das reinste Vergnügen, da Simula 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


C with classes
1979
erste Nennung von C++
1983
Cfront 1.0, 1. Aufl. The C++ Programming Language
1985
Cfront 1.2 (noch in Quelle verfügbar), g++1.0
1987
ANSI X3J16 1.Treffen
1989
1. Auflage ARM, Templates, Exception Handling
1990
RTTI, namespaces
1993
1. Entwurf ANSI/ISO Standard
1994
Standard ISO/IEC 14882
1998
C++ TR1
2003
Standard C++11
2011

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();
}
```

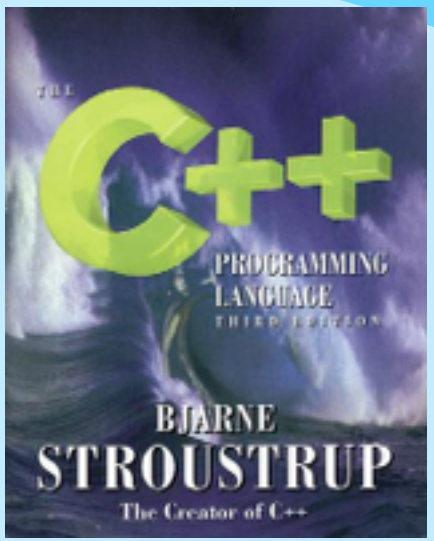
```
$ g++ -o a a.cc
$ a
1111$
```

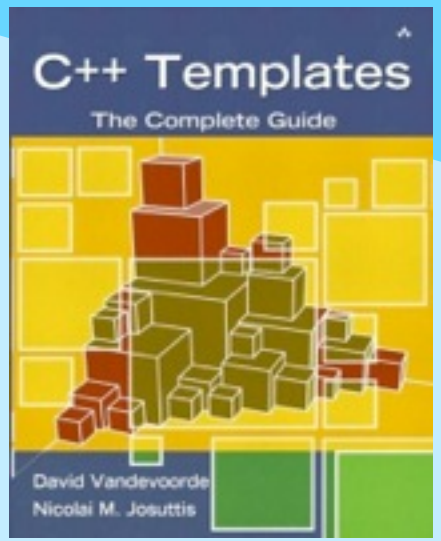
BTW: C++ Literaturempfehlungen beginners level

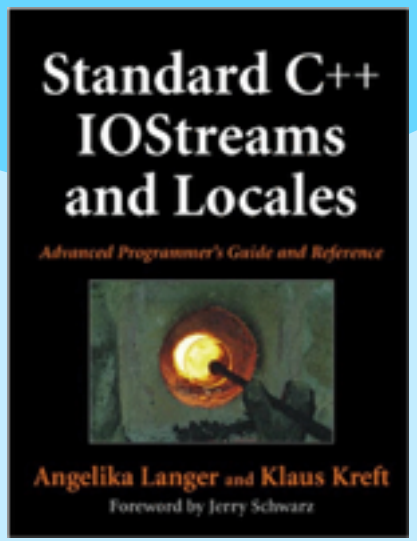
Kaufen: „Bafög-Ausgabe" 19,95 €

Ausleihen: im Handel leider vergriffen ☹

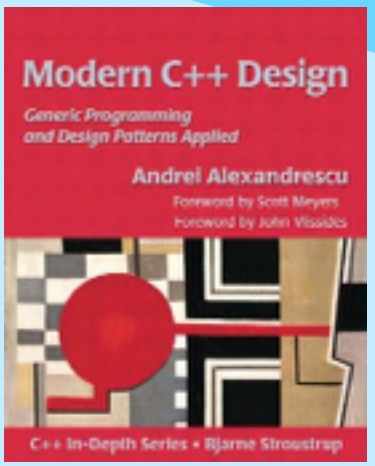
BTW: C++ Literaturempfehlungen expert level

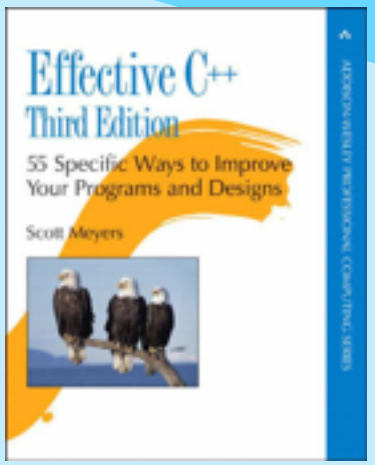






BTW: C++ Literaturempfehlungen guru level





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:

```
alignof asm auto bool break case catch char char16_t
char32_t class const constexpr const_cast continue
decltype default delete do double dynamic_cast else
enum explicit export extern false float for friend
goto if inline int long mutable namespace new noexcept
nullptr operator private protected public register
reinterpret_cast return short signed sizeof static
static_assert static_cast struct switch template this
thread_local throw true try typedef typeid typename
union unsigned using virtual void volatile wchar_t while
```

(C: 32) (Δ C++98: 31) (Δ C++11: 9)

1.1. Lexik

Operatoren:

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

`<< >> & ^ | ~` bitweise left-, right-Shift, and, xor, or, 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 durchweg 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}; //unscoped
enum class Direction {left, right, up, down}; //scoped
Season now = spring; ... if (now == winter) ...
Direction where = Direction::up;
```

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 !

neu in C++11: Typdeduktion

```
auto x = 7;
```

x ist von Typ int wegen des Typs des Literals.

```
auto x = expression;
```

x ist vom Typ des Resultats von expression.

(erlangt erst im Zusammenhang mit Templates seine volle Bedeutung)

neu in C++11: Typdeduktion

```
template<class T> void printall(const vector<T>& v) {
  for (auto p = begin(v); p!=end(v); ++p) cout << *p << "\n";
}
 statt C++98:
template<class T> void printall(const vector<T>& v) {
  for (typename vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p)
    cout << *p << "\n";
}
```

C++11

```
template<class T,class U> void f(const vector<T>& vt, const vector<U>& vu){
  // ...
  auto tmp = vt[i]*vu[i]; // whatever T*U yields
  // ...
}
```

1. Elementares C++

C - **enum**s mit Problemen:

- konvertierbar nach int
- exportieren ihre Aufzählungsbezeichner in den umgebenden Bereich (name clashes)
- schwach typisiert (z.B. keine forward Deklaration möglich)

enum classes ("strong enums") sind stark typisiert und 'scoped':

```
enum Alert { green, yellow, election, red }; // traditional enum
enum class Color { red, blue };              // scoped and strongly typed enum
                 // no export of enumerator names into enclosing scope
                 // no implicit conversion to int
enum class TrafficLight { red, yellow, green };

Alert a = 7;             // error (as ever in C++)
Color c = 7;             // error: no int->Color conversion
int a2 = red;            // ok: Alert->int conversion
int a3 = Alert::red;     // error in C++98; ok in C++0x
int a4 = blue;           // error: blue not in scope
int a5 = Color::blue;    // error: not Color->int conversion
Color a6 = Color::blue; // ok
```

Typ der Repräsentation kann spezifiziert werden

```
enum class Color : char { red, blue }; // compact representation enum class

TrafficLight { red, yellow, green };
    // by default, the underlying type is int

enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
// how big is an E?
// (whatever the old rules say;
// i.e. "implementation defined")

enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
// now we can be specific
```

C++11

forward Deklaration möglich

```
enum class Color_code : char; // (forward) declaration
void foobar(Color_code* p); // use of forward declaration
// ...
enum class Color_code : char { red, yellow, green, blue };
// definition
```

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 !

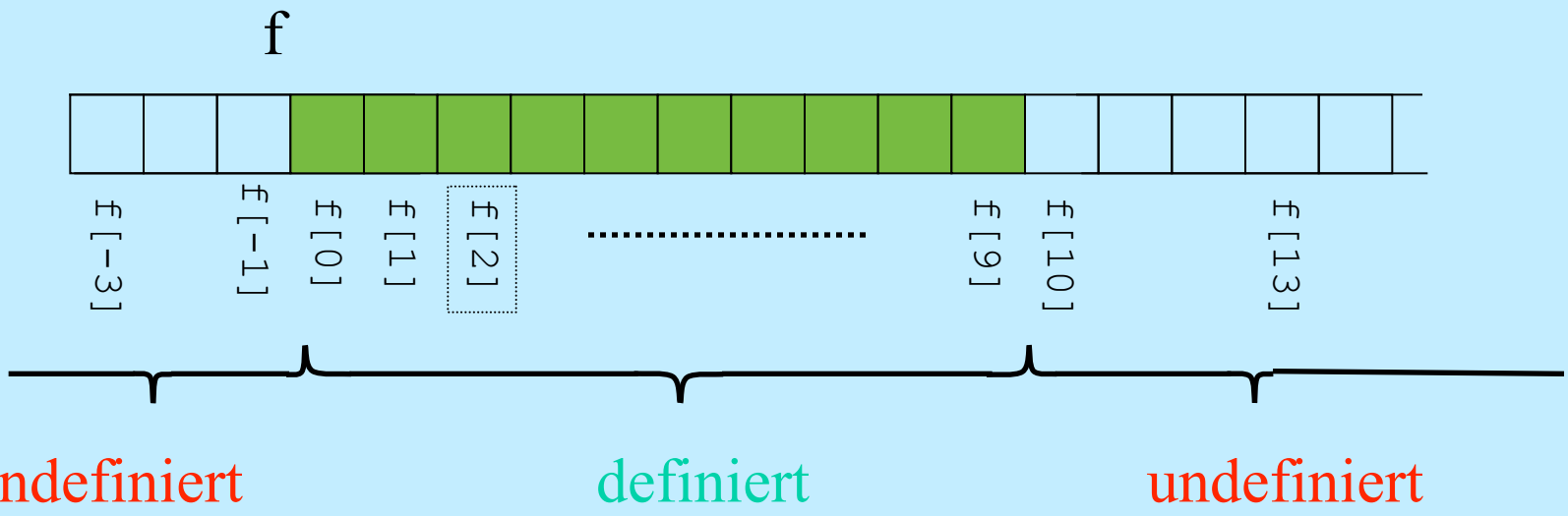
Typedefs: Synonyme für (u.U.) komplexe Typkonstrukte

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

1.2. Datentypen (Felder)

es gibt keine Prüfung auf Einhaltung der Feldgrenzen bei Zugriffen:

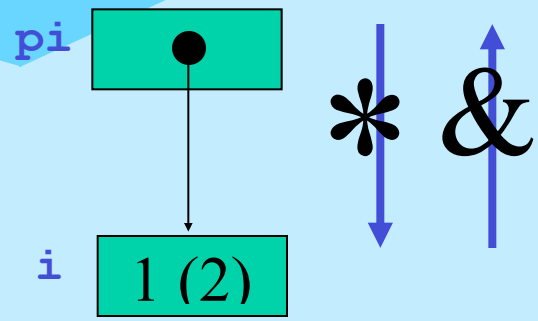
```
T f [10];
```



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)

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 !
}
```

nullptr

C++11

```
void foo(int) {std::cout<<"foo(int)\n";}
void foo(int*) {std::cout<<"foo(int*)\n";}
…
foo(0);
foo(NULL);
foo(nullptr);

int* pi = nullptr;

if (pi == nullptr) pi = new int; // same as:
if (pi) pi = new int;
```

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

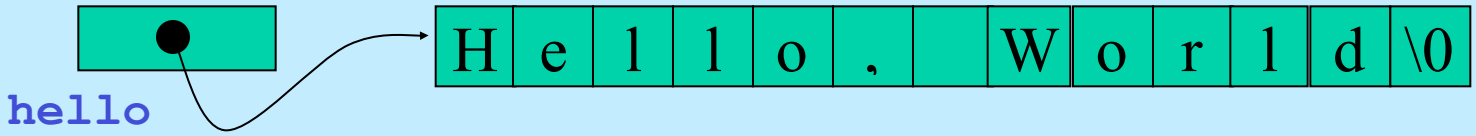
Zeichenketten:

Zeichenkettenliterale wie »üblich«

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

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

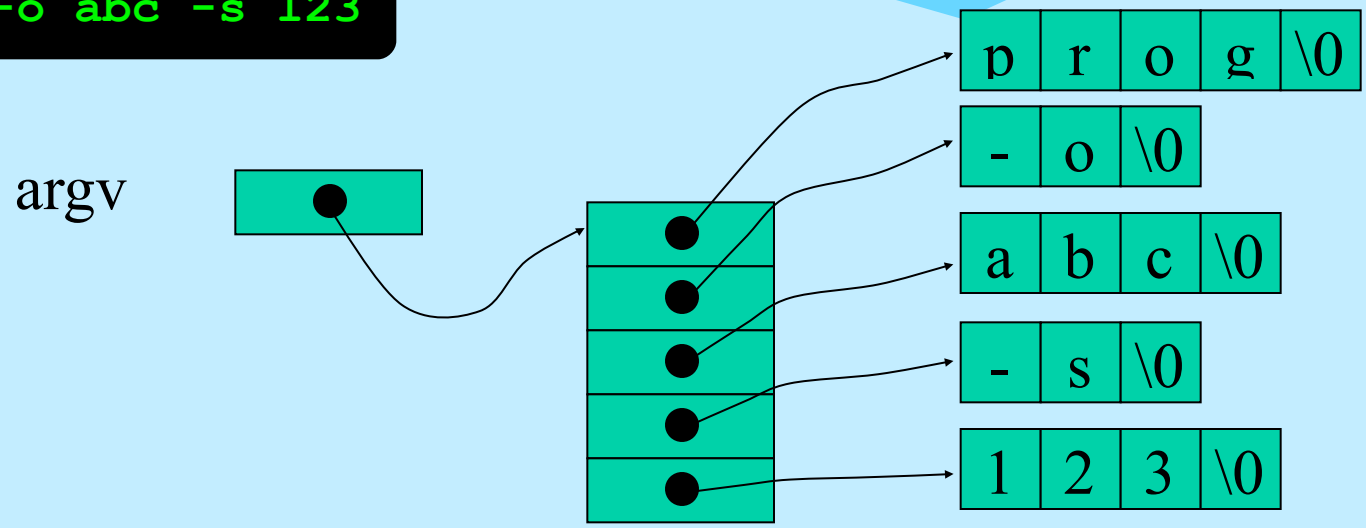
```
char* hello = "Hello, World";
```



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)

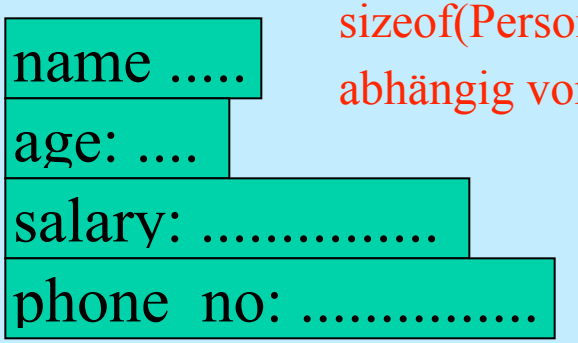
```
void foo(const T& pct)
{
        // pct.change(); ERROR
        pct.read_only();
}

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();` ???

P. S. offene Fragen

1. mehrfache Deklarationen in Klassen?

```
class X {
    void foo();
    void foo() { .... }
};
```

Nein!

Memberfunktionen können in der Klasse (genau einmal) deklariert oder definiert werden. Nur wenn nur deklariert wurde, darf außerhalb der Klasse (nur) definiert werden.

2. Warum keine Kontollflussanalyse in C++?

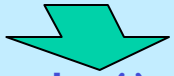
- `exit`, Exceptions http://www.gotw.ca/gotw/020.htm, `asm`-Einschlüsse
- dennoch: § 6.6.3: Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function. -- in C legal wenn Wert nicht benutzt.

  http://stackoverflow.com/questions/1610030/why-can-you-return-from-a-non-void-function-without-returning-a-value-without-pr

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) ....
```

neu in C++11: range-based for

```
int array[] = { 1, 2, 3, 4, 5 };

for (int x : array) // value
    x *= 2;

for (int& x : array) // reference
    x *= 2;
```

C++11

Ersetzung durch:

```
{
    auto && __range = range-init;
    for ( auto __begin = begin-expr,  __end = end-expr;  __begin != __end;  ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

neu in C++11: uniform initialization syntax/semantic

C++(98) hat verschiedene Wege zur Initialisierung, je nach Objekttyp und Kontext. -> fehleranfällig und nicht konsistent

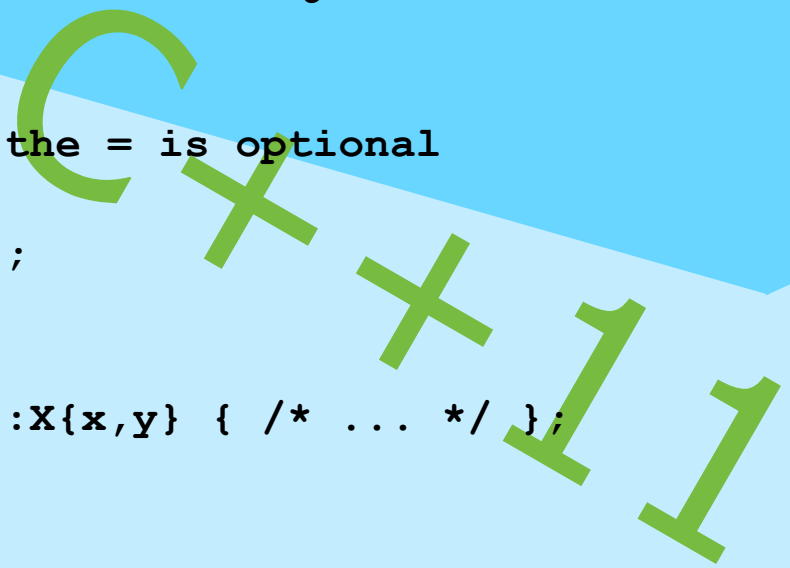

```
string a[] = { "foo", " bar" }; // ok: initialize array variable
vector<string> v = { "foo", " bar" }; // error: initializer list for non-aggregate vector
void f(string a[]); f( { "foo", " bar" } ); // syntax error: block as argument
```

und

```
int a = 2;           // assignment style
int[] aa = { 2, 3 }; // assignment style with list
complex z(1,2);      // functional style initialization
x = Ptr(y);          // functional style for conversion/cast/construction
```

neu in C++11: uniform initialization syntax/semantic

C++11 {}-initializer lists für alle Initialisierungen:




```
X x1 = X{1,2};
X x2 = {1,2}; // the = is optional
X x3{1,2};
X* p = new X{1,2};

struct D : X {
 D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
 int a[3];
 S(int x, int y, int z) :a{x,y,z} { /* ... */ };
 // solution to an old problem
};
```

neu in C++11: uniform initialization syntax/semantic

Auch ein altes (Parse-)Problem ist damit gelöst:

C++11

```
struct P
{
   P(){std::cout<<"P::P()\n";}
   P(const P&) {std::cout<<"P::P(const P&)\n";}
};

// C++ most vexing parse - what is:

P p(P()); // ???
// p: P -> P :-(

P p{P()}; // default constructed P
```

neu in C++11: uniform initialization syntax/semantic

Handliche Listen überall

C++11

```
vector<double> v = { 1, 2, 3.456, 99.99 };
vector<double> v1 { { 1, 2, 3.456, 99.99 } };

list<pair<string,string>> languages = { // parse error in C++98
 {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"} };

map<vector<string>,vector<int>> years = { // fine in C++11
  { {"Maurice","Vincent","Wilkes"},{1913,1945,1951,1967,2000} },
  { {"Martin","Ritchards"},{1982,2003,2007} },
  { {"David","John","Wheeler"},{1927,1947,1951,2004} }
};
```

neu in C++11: uniform initialization syntax/semantic

Nicht mehr nur für Felder, Argumente vom Typ `std::initializer_list<T>` möglich.

```
void f( initializer_list<int> );
f( {1,2} );
f( {23,345,4567,56789} );
f({}); // the empty list
f{1,2}; // error: function call ( ) missing
years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});

void print(std::initializer_list<int> ls) {
    for(const auto i: ls) std::cout<<i<<std::endl;
}
... print ({1,2,3,4,5,6,7,8,9});
```

Konstruktoren mit einem einzigen Argument vom Typ **std::initializer_list** heißen initializer-list Konstruktoren. Die Standardcontainer, string, regex etc. haben solche.

neu in C++11: no more narrowing

```
int x = 7.3; // Ouch!
void f(int);
f(7.3); // Ouch!
```

C++11

mit {} Initialisierung nicht:

```
int x1 = {7.3}; // error: narrowing
double d = 7;
int x2{d}; // error: narrowing (double to int)
char x3{7};
     // ok: even though 7 is an int, this is not narrowing
vector<int> vi = { 1, 2.3, 4, 5.6 };
     // error: double to int narrowing
```

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 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	***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 i=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 Destruktoren aller erfolgreich konstruierten Objekte
- 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

neu in C++11: noexcept

statisches no-throw:

```
void i_will_never_throw (int i) noexcept{ ... }  // noexcept(true)

template <typename T> bool compare(T t1, T t2) noexcept( noexcept(t1==t2))
                      return t1 == t2;
    •}
```

sämtliche Exception-Vorkehrungen können im Code entfernt werden

falls doch eine Exception auftritt: `terminate`

neu in C++11: exception nesting

Ausnahmen verpacken:

C++11

```
void lib_func (int i) {
    try {
        if (i<0) throw low_level_ex();
    }
    catch (...) {
        std::throw_with_nested(high_level_exception());
    }
} // die aktuelle Ausnahme wir in eine neue eingepackt ...

namespace std {
class nested_exception { ... // mixin class: 18.8.6
   [[noreturn]] void rethrow_nested() const;
          [[noreturn]] template<class T> void throw_with_nested(T&& t);
          template <class E> void rethrow_if_nested(const E& e);
};
```

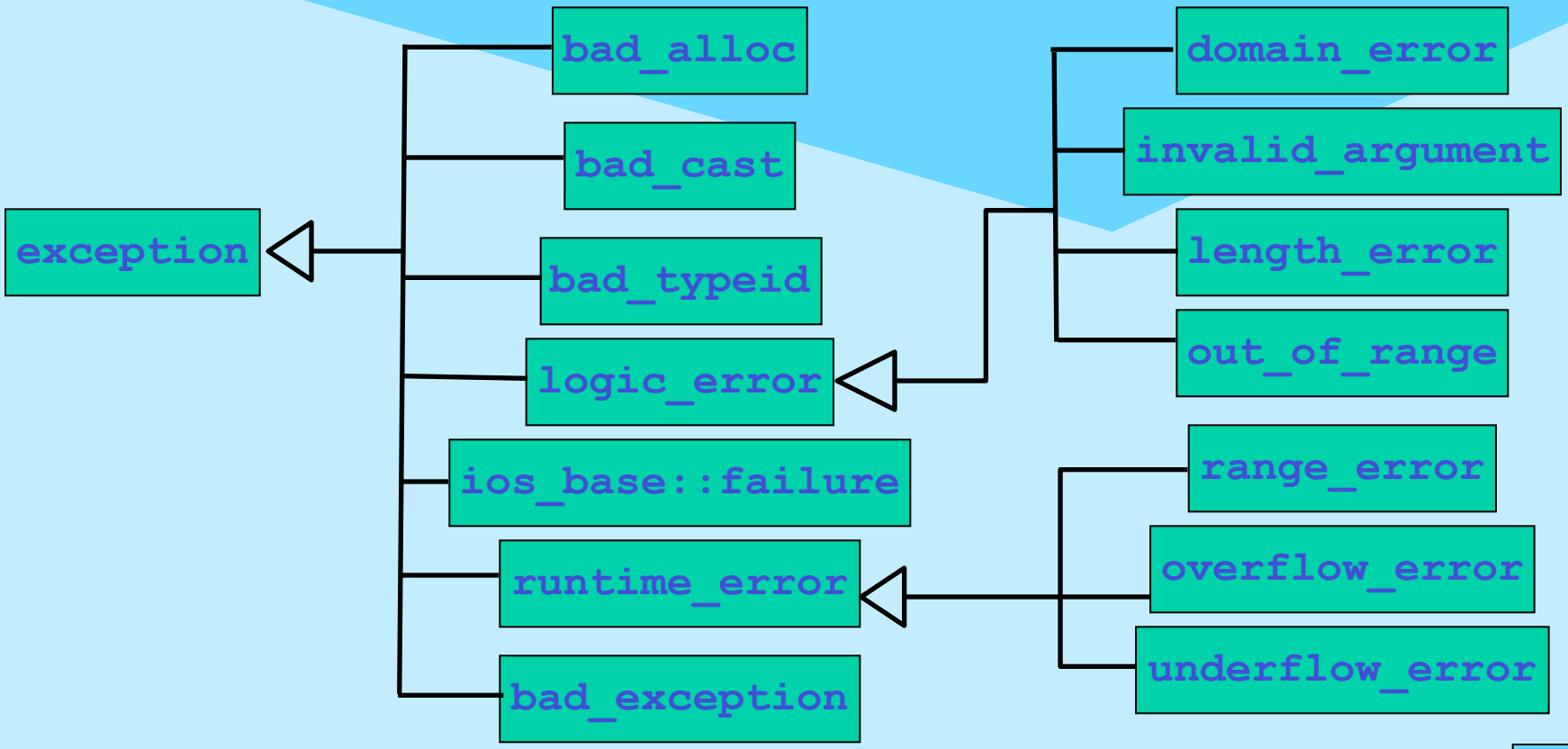
1.5. Strukturierte Anweisungen: Exception Handling

- 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)

```
// 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

```
// 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

```
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 = 1;    // X tmp(1); X x3(tmp); ggf. elision
```

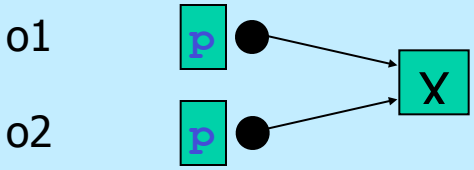
Copy-Konstruktoren

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

shallow copy

(default copy ctor)

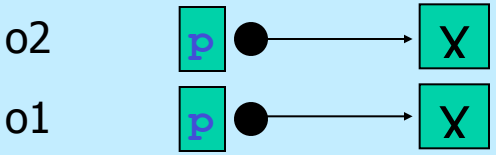
```
class SC {
   X* p;
public:
   SC(): p(new X) {}
};
SC o1;
SC o2=o1;
```



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.
```

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();
```

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
```

C++ idiom: Resource Acquisition Is Initialization

```
#include <iostream>
#include <memory>

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

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

    ~Y() {}
};

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

```
sizeof(Y)=8
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
```

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

C++ idiom: Resource Acquisition Is Initialization

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

```
#include <iostream>
#include <chrono>
```

C++ idiom: Resource Acquisition Is Initialization

```
class Timer { // conforms to C++11
    std::chrono::steady_clock::time_point start;
    std::string what;
public:
    Timer(std::string s): start(std::chrono::steady_clock::now()), what(s) {}
    ~Timer() {
        auto duration = std::chrono::steady_clock::now() - start;
        std::cout << what+":\t" <<
        std::chrono::duration_cast<std::chrono::milliseconds>(duration).count()
        << " ms" << std::endl;
    }
};
```

- 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(){cout<<"Y::f1()\n";};
  void f2(){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 !
};
```

```
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

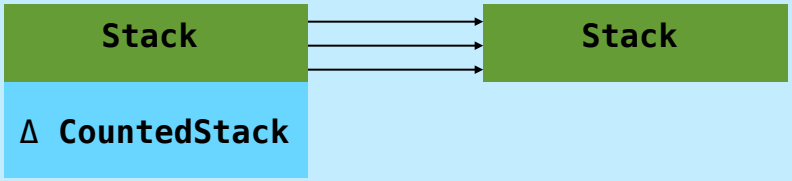
```
const Stack& ????
```

- 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 { .... };
```

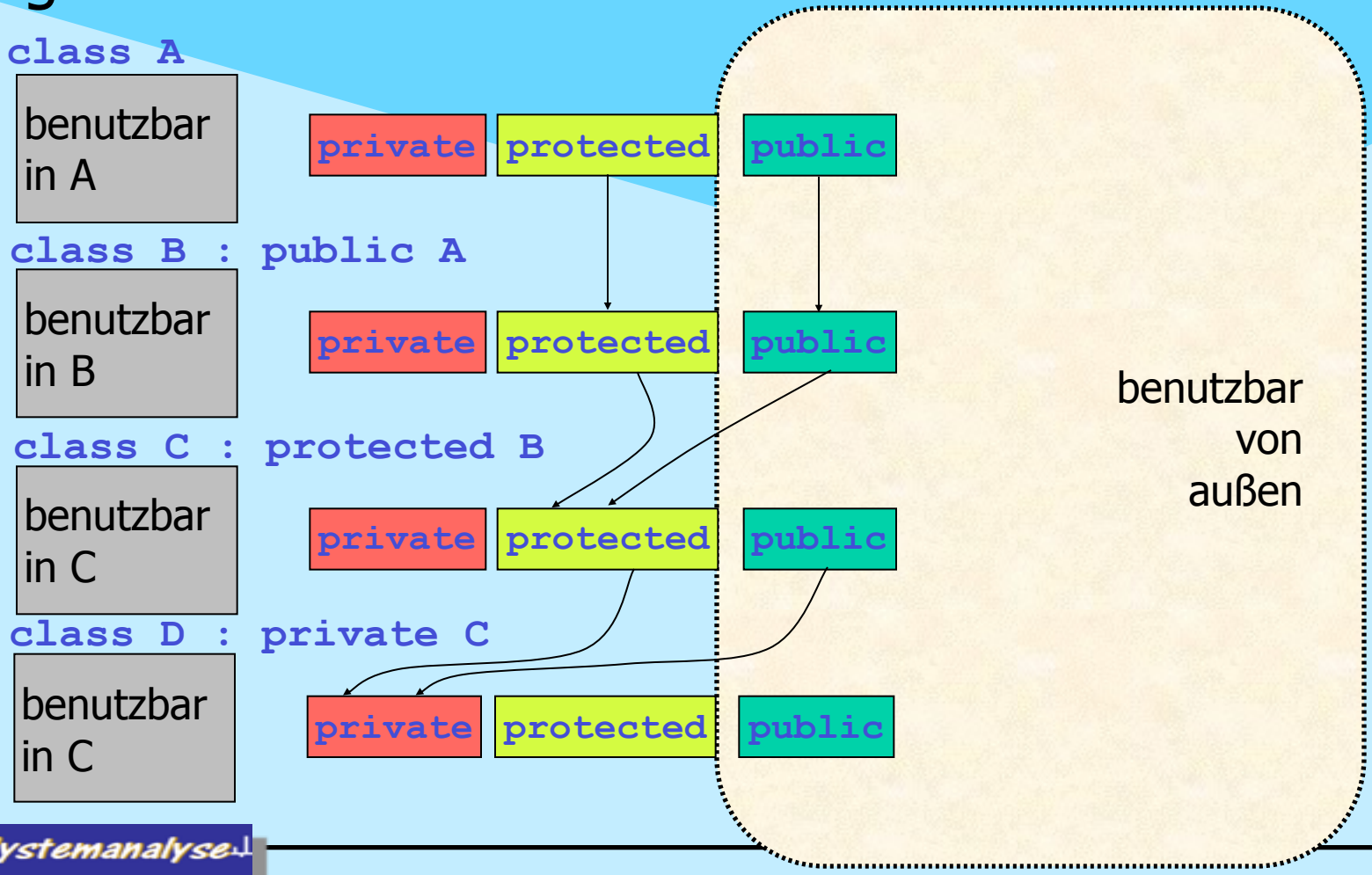
`Deriv1` IST nirgends EIN `Base` == die Vererbung ist ein (nicht erkennbares) Implementationsdetail

```
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 `friend`**s**

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

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

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

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!)

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/
```

Thread-safe Singletons

```
class Singleton {
    static std::shared_ptr<Singleton> instance_;
    static std::once_flag oflag;
    Singleton();// private !
    static void safe_create()
    { instance_.reset(new Singleton()); }
public:
    static std::shared_ptr<Singleton> instance() {
        std::call_once(oflag, safe_create); // variadic args
        return instance_;
    }
};
// in some cpp-File
std::shared_ptr<Singleton> Singleton::instance_;
std::once_flag Singleton::oflag;
```

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

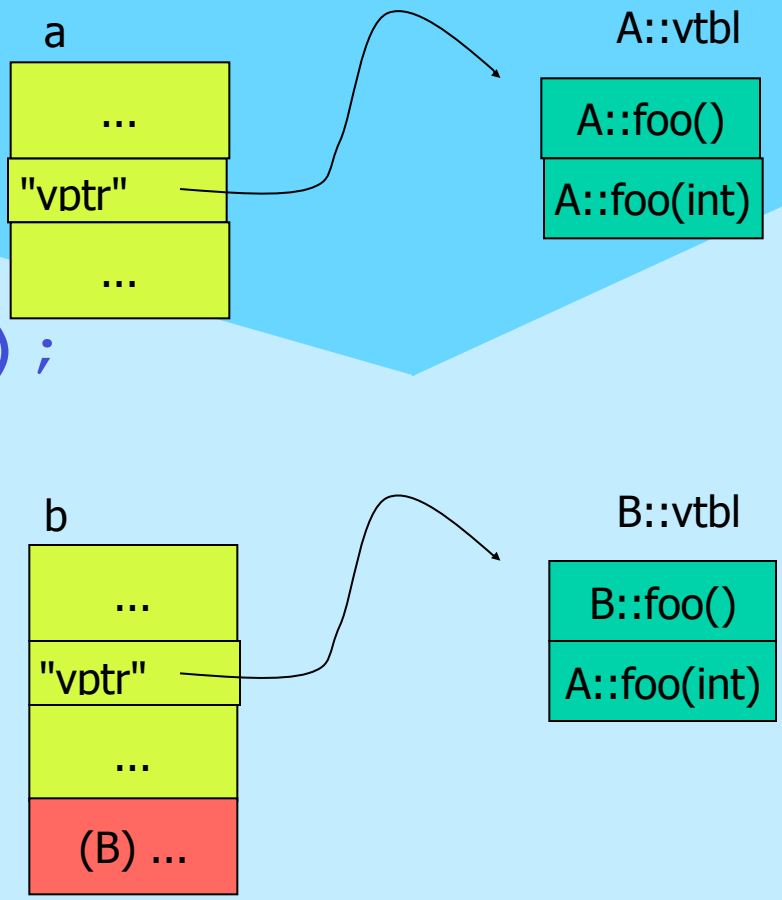
Realisierung (nicht normativ aber de facto Standard):

- 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;
```



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();
}
```

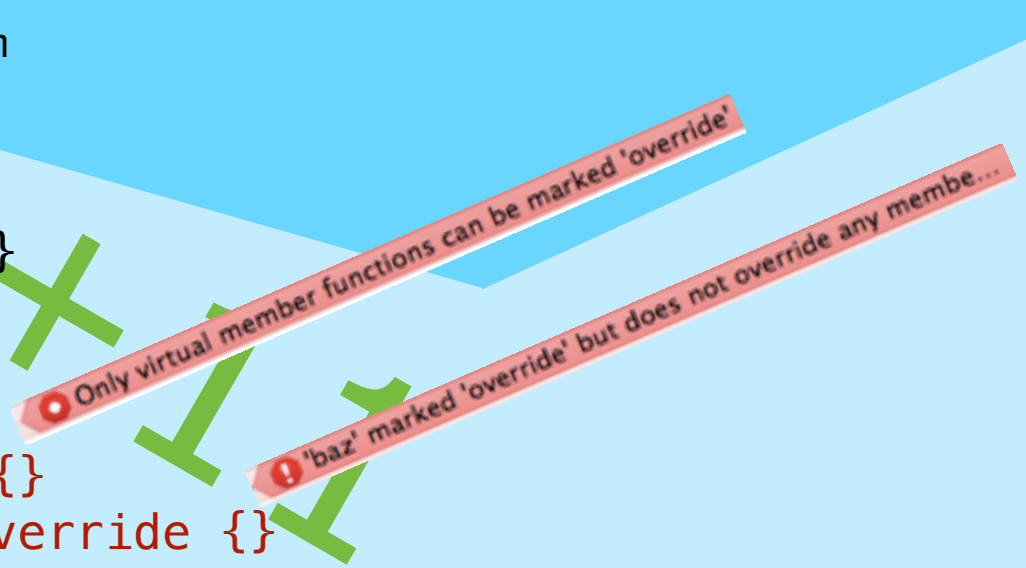
```
C:\tmp>bcc32 hide.cpp
...
Warning W8022 hide.cpp
15:
'B::foo(int)' hides
virtual
function 'A::foo()'
...
C:\tmp>hide
C::foo()
B::foo(int)
```

g++ (auch 4.x) warnt nicht [nicht mal bei -Wall]

Neu in C++11 **override** (kein! reservierter Bezeichner - ein kontextsensitives Schlüsselwort)

Ziel: Fehler bei der Redefinition besser finden

```
class B {
public:
    virtual void bar() {}
};

class D: public B {
    void bac() override {}
    virtual void baz() override {}

    virtual void bar() override {}
} override; // OK no keyword in this context
```



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);
```

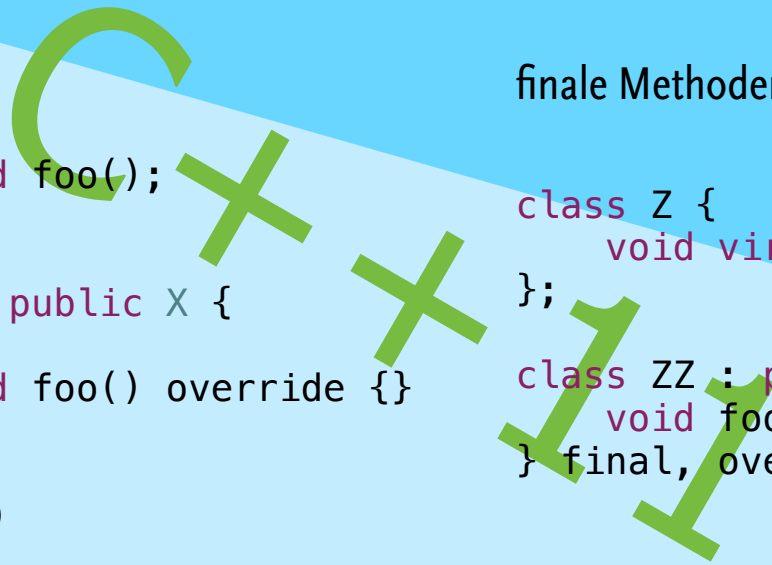
```
struct Circle : // real Shape
 public AbstractShape {
  virtual void draw() {...}
  virtual void erase() {...}
};
```

Neu in C++11 **`final`** (kein! reservierter Bezeichner - ein kontextsensitives Schlüsselwort)

finale Klassen: keine Ableitung möglich

finale Methoden: keine Redefinition in Ableitungen

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

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

void call (Y* p)
{
    p->foo(); // can bind statically !
}

// class Z : public Y {}; // not possible
```



finale Methoden müssen virtuell sein !

```
class Z {
    void virtual foo() const {}
};

class ZZ : public Z {
    void foo() const final override {};
} final, override;
```

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."

Neu in C++11 **deleted/defaulted functions** (in Anlehnung an die Syntax von pure virtual functions)

```
class X {
public:
    X() = default;
    virtual ~X() = default;
    X(const X&) = delete;
    void foo(int);
    void foo(double) = delete;
};

X x;
X x1(x);
x.foo(1);
x.foo(1.0);

// delete auch für globale Funtionen
void bar(double);
void bar(int) = delete;
bar(1.9);
bar(19);
```

Call to deleted constructor of 'X'

Call to deleted member function 'foo'

Call to deleted function 'bar'

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

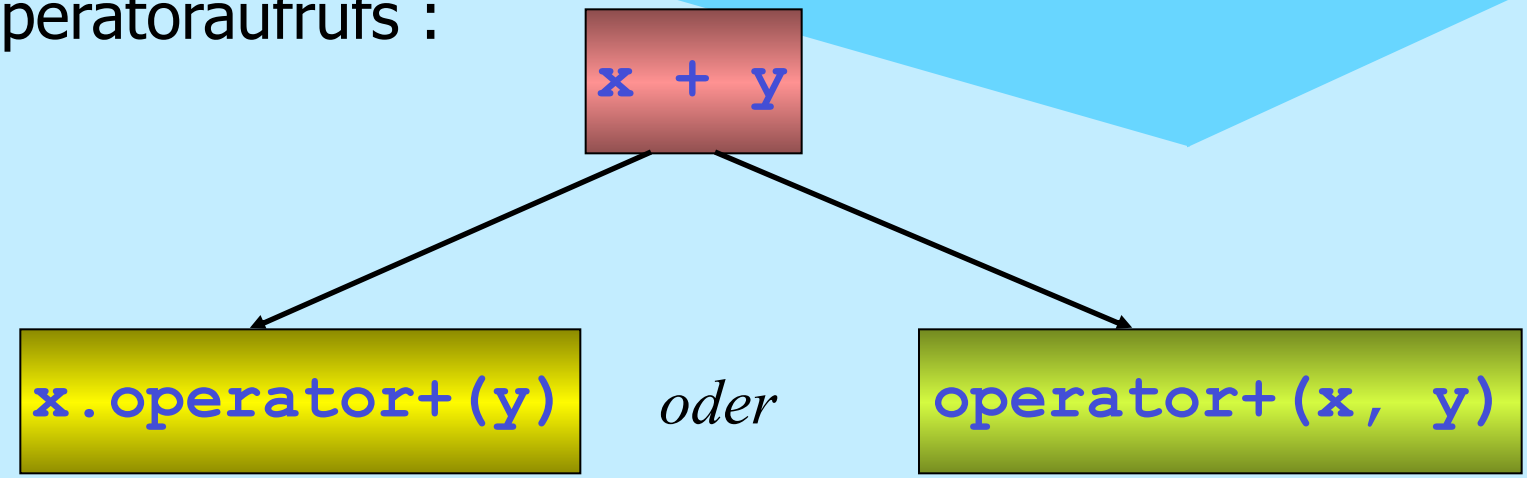
```
// falsch:
// int operator+ (int i, int j) {return i - j;}
```

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

Member

unäre Operatoren

```
class X { public:
 T operator ⊙ ();
};

⊙x; // Ergebnis: T
// (x).operator ⊙ ();
```

binäre Operatoren

```
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 assignment → 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;
}
```

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;
};
```

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 );
```

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 in C++98 nicht 'abschaltbar' (wie explicit ctors) ggf. Memberfunktionen `toType()` bevorzugen ! in C++11 auch erlaubt

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

```
#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();
}
```

```
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)

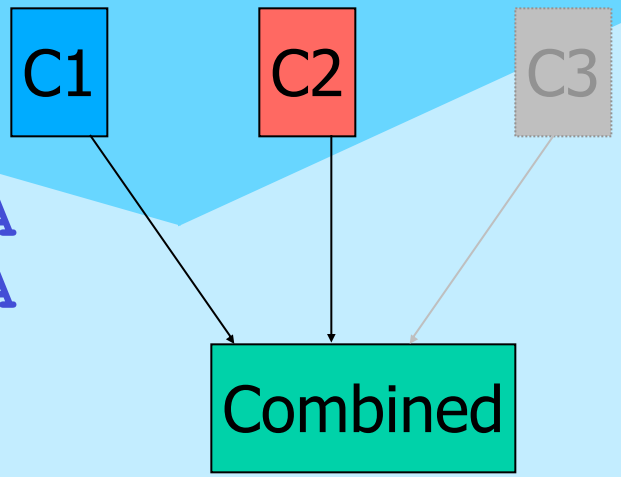
eine Klasse kann mehrere Basisklassen haben --> 'freie' Kombination von Konzepten:

```
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:

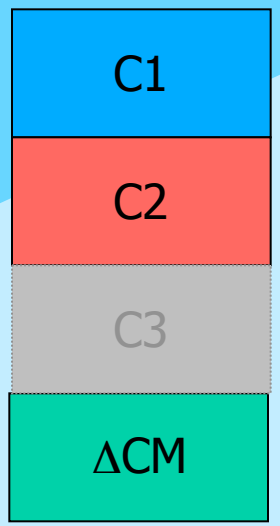
```
Combined obj;
Combined * cm = &obj;
Concept1 * c1 = cm; // IS A
Concept2 * c2 = cm; // IS A
// NOT c2=c1=cm;

// 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!
```

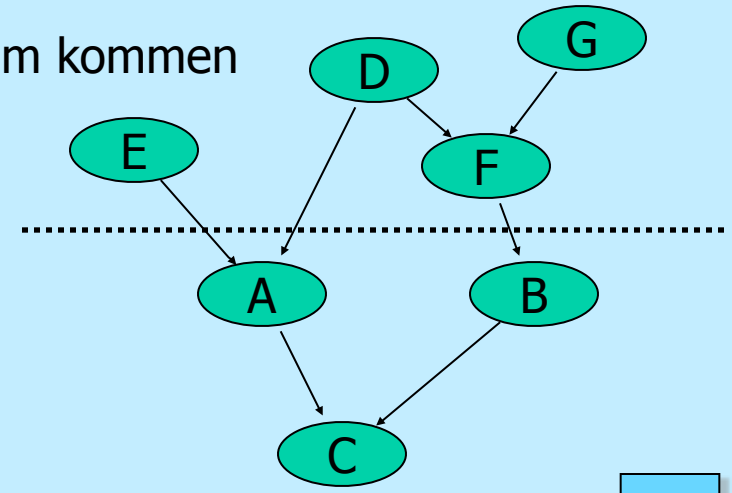
Mehrfachvererbung (multiple inheritance)

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

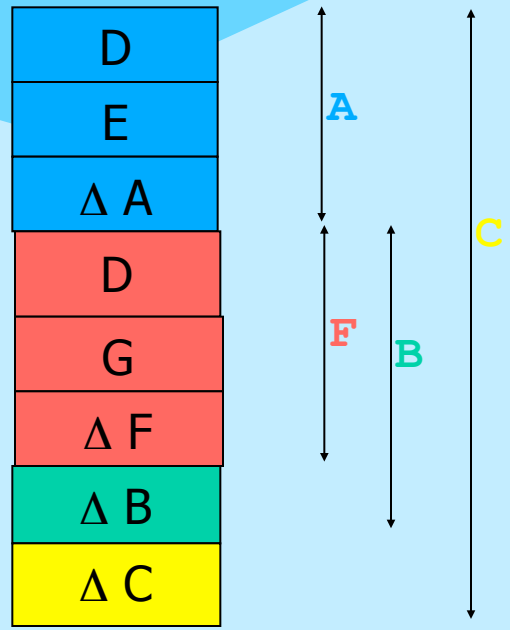
```
class A: public D, public E {...};
class F: public D, public G {...};
class B: public F {...};
-----------------------------------
class C: public A, public B {...};
```



Mehrfachvererbung (multiple inheritance)

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

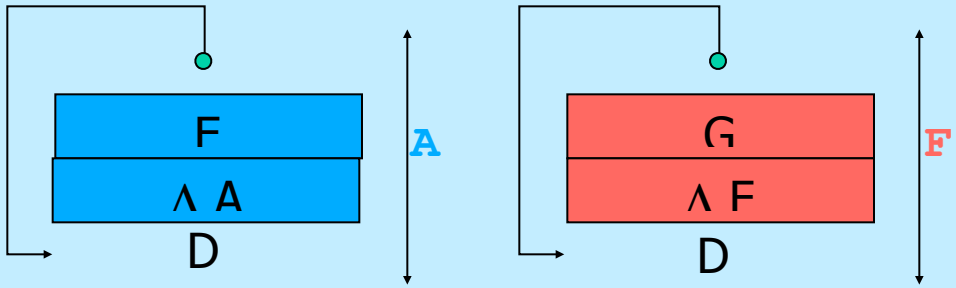
```
class A: public D, public E {...};
class F: public D, public G {...};
class B: public F {...};
class C: public A, public B {...};
```

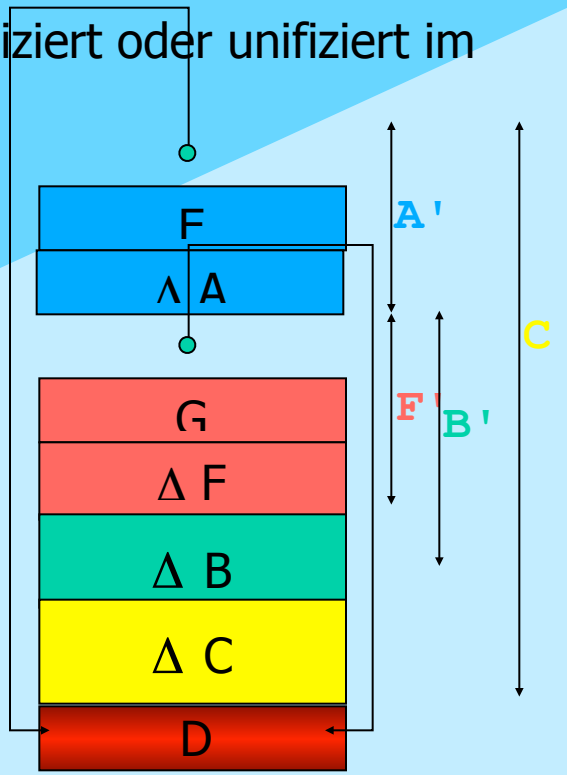


Mehrfachvererbung (multiple inheritance)

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)

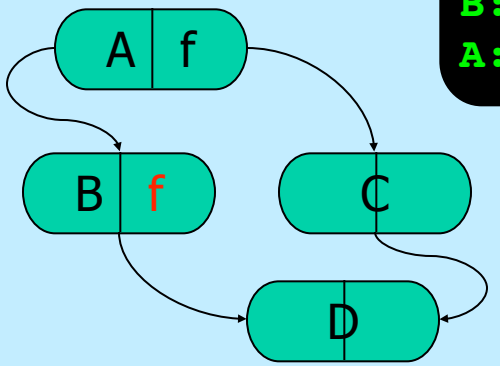

```
class A { public: void f(){cout<<"A::f()\n";}   };
class B: public virtual A {
           public: void f(){cout<<"B::f()\n";}
};
class C: public virtual A {
           public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C {};
int main() {   D d;
               // d.f(); ERROR: ambiguous 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()
```

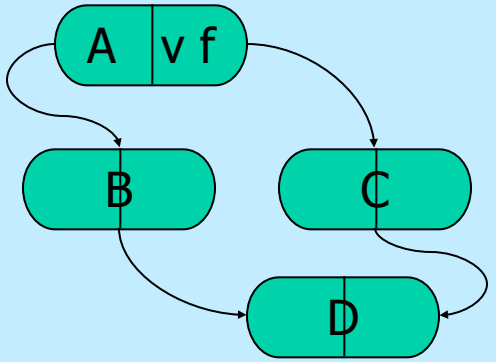


Mehrfachvererbung (multiple inheritance)

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

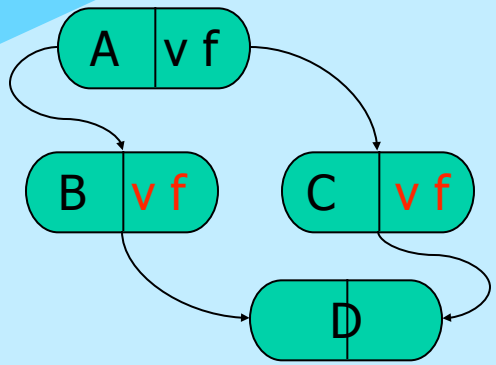
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A { };
class C: public virtual A { };
class D: public B, public C { };
main(){         D d;
                C *pc = &d;
                pc->f();
}
```

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



Mehrfachvererbung (multiple inheritance)

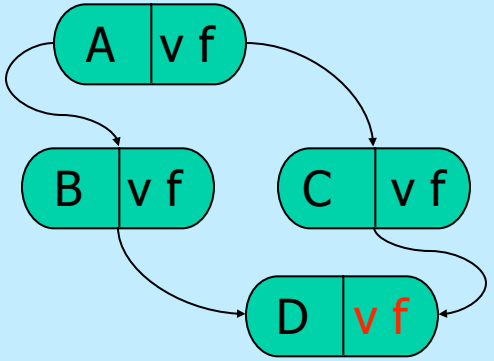
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A {
  public: void f(){cout<<"B::f()\n";}
 };
class C: public virtual A {
  public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C { };
// ERROR: no unique final overrider for f() in D
```



Mehrfachvererbung (multiple inheritance)

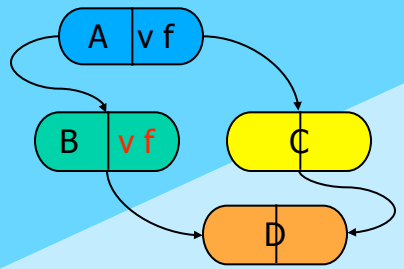
```
class A {
  public: virtual void f(){cout<<"A::f()\n";}
};
class B: public virtual A {
  public: void f(){cout<<"B::f()\n";}
 };
class C: public virtual A {
  public: void f(){cout<<"C::f()\n";}
};
class D: public B, public C {
  public: void f(){cout<<"D::f()\n";}
};
... D d; C *pc = &d; pc->f();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

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

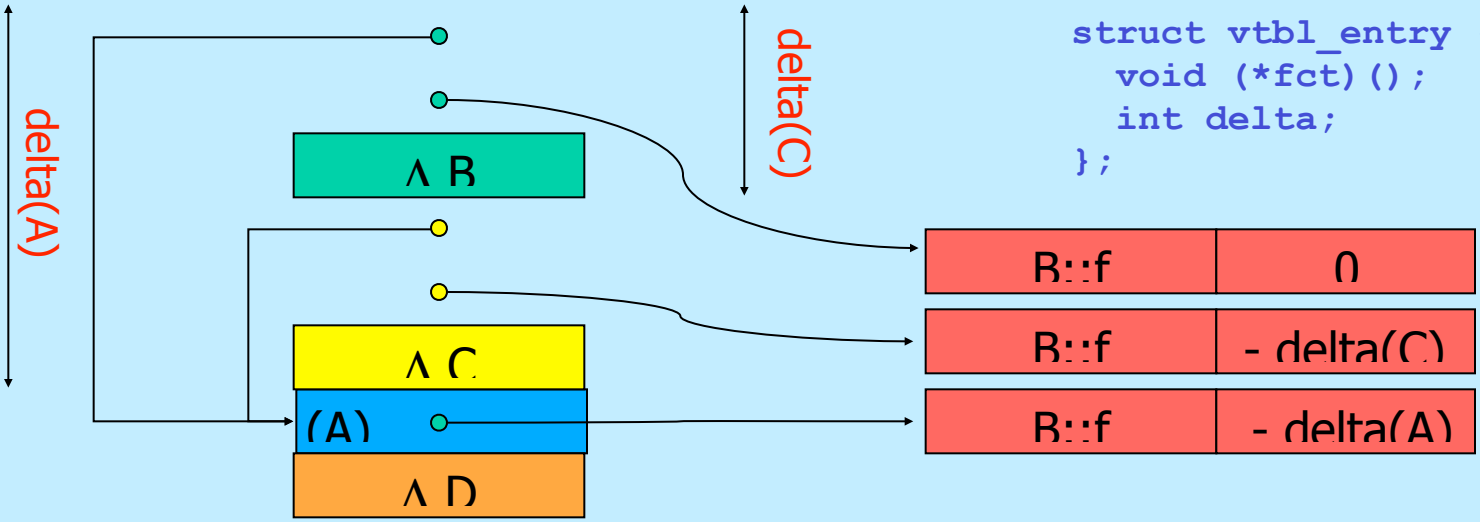


Mehrfachvererbung (multiple inheritance)

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*)
```

```
// 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
```

Problem:

Vererbung erlaubt die Wiederverwendung von Programmcode, virtuelle Funktionen ermöglichen dabei den Austausch gewisser Funktionalität

Vererbung erlaubt NICHT die Wiederverwendung durch Parametrisierung von Klassen, z.B.

```
class Stack_of_int{...};
```
und
```
class Stack_of_double{...};
```

NICHT (sinnvoll) realisierbar als:

```
class Stack { ... }; // abstract ??
class Stack_of_int: public Stack { ... };
class Stack_of_double: public Stack { ... };
```

Lösung:

C++ erlaubt die Definition von generischen Klassen:
solche, die von (noch nicht spezifizierten) Typen oder Werten abhängen !

```
// GenericStack.h :
template <class T, int D> // variabel in T und D
class Stack {
protected:
   T *data;
   int top, max;
public:
   Stack(int dim = D): data(new T[dim]), top(0), max(dim) {}
   ~Stack() { delete [] data; } // beide hier inline
   void push (T i); // outline
   T pop();         // outline
};
```

Die Definition von Memberfunktionen außerhalb von generischen Klassen hängt damit selbst von diesen Typen oder Werten ab !

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

template <class T, int D>
T Stack<T,D>::pop () {
      return data[--top];
}
```

eine Template-Klasse definiert ein allgemeines Muster für beliebig viele konkrete Klassen,

aus dem Template selbst lässt sich jedoch (i. allg.) kein Code generieren,

dies geschieht erst bei der sog. Instantiierung der Template-Klasse

```
#include "GenericStack.h"
Stack<int, 10> s1, s2;
Stack<double, 20> s3;
Stack<Stack<int, 10>, 10> ss;
void foo() {
  s1.push(1);  s2.push(2);
  ss.push(s1); ss.push(s2);
  s3.push(3.7);
}
```

Daher muss bei der Instantiierung einer Template-Klasse der Quelltext aller (benutzten) (auch der nicht-inline) Funktionen zur Verfügung stehen:

übliche Verfahrensweise:

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

die Instantiierung einer konkreten Klasse geschieht auf explizite Anforderung (erste Verwendung), eine solche kann jedoch weitere Instantiierungen nach sich ziehen

Instantiierung 'auf Vorrat' (ohne sofortige Verwendung) ist möglich:

```
template class Stack<int, 10>;
template class Stack<double, 20>;
template class Stack<Stack<int, 10>, 10>;
```

`<class XYZ>` bedeutet nicht, dass nur mit Klassentypen instantiiert werden darf, vielmehr kündigt `class` einen 'Meta-(Typ-)Parameter' an, alternativ kann hier auch das neue Schlüsselwort `typename` verwendet werden

Wozu dient das neue Schlüsselwort `typename` ?

```
template <class T> class Beispiel { ...
   T::X * x;
   // ist nach C++-Grammatik mehrdeutig
};
```

```
class T1 { public: static int X; };
```
`Beispiel<T1>`-Instantiierung: `*` ist Multiplikation

```
class T2 { public: typedef int X; };
```
`Beispiel<T2>`-Instantiierung: `*` ist Zeigertyp-Konstrukt

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

Voreinstellungen für Template-Parameter sind möglich

```
template <class T = int, int D = 20>
class Stack {....};
Stack<> s; // Stack<int,20> !
```

ACHTUNG Lexikfalle:

```
template <typename T> class X { ... };
X<X<int>> xxt;// ERROR C++98: operator >> ???
              // ok in C++11 :-)
X<X<int> > xxt;   // 98 workaround
```

Individuelle Implementationen für spezielle Template-Parameter sind möglich:

```
template <> // alle Parameter gebunden!
class Stack<std::string, 100> {
// Stacks von strings kann man u.U.
// anders/spezieller/effizienter/cleverer
// implementieren
... };
// ohne: template<> !!!
void Stack<std::string, 100>::push
          (const std::string& s) { .... }
```

Individuelle (partielle) Implementationen für spezielle Template-Parameter sind möglich:

```
template <typename T1, typename T2>
class MyClass { // allgemeinste Form
.... };
template<typename T>
class MyClass<T, T> { // T1 und T2 sind gleich
.... };
template<typename T>
class MyClass<T, int> { // T2 ist int
.... };
template <typename T1, typename T2>
class MyClass<T1*, T2*> { // beide sind Zeigertypen
.... };
          // ACHTUNG MyClass<int, int> nicht eindeutig!
```

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

```
template <typename T>
class Stack {
      // Stack<S> und Stack<T> sollen zuweisbar
      // sein, wenn S und T zuweisbar sind
      template <typename T2>
      Stack<T>& operator=(const Stack<T2>&);
.... };                  // default op= bleibt erhalten !
template <typename T>
  template <typename T2>
    Stack<T>& Stack<T>::operator=(const Stack<T2>& o)
              { for(....) someT = someT2; ... }
```

Template-Parameter können selbst wieder Template-Typen sein: template template parameters

```
template <typename T>
class Stack { ... };

template <typename T, typename Container>
class LIFO {  Container elems; };

LIFO<int, Stack<int> > l; // int redundant
//besser:
template <typename T,
    template <typename E> class Container = Stack>
class LIFO {  Container<T> elems; }
LIFO<int> l1; LIFO<int, otherStack> l2;
```

kann fehlen, weil nicht benutzt

NICHT typename

Template-Code muss zunächst nur syntaktisch korrekt sein. Erst bei der Instantiierung wird semantisch Korrektheit geprüft.

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

SFINAE: **S**ubstitution **f**ailure **i**s **n**ot **a**n **e**rror

generic programming eröffnet völlig neue Möglichkeiten

Beispiel: Zuweisbarkeit von Typen statisch entscheiden
Ist ein Typ T in einen Typ U konvertierbar ?

```
// s. Andrei Alexandrescu: Modern C++ Design

template <class T, class U>
class Conversion {
   typedef char Small;
   class Big {char dummy[2];}; // guaranteed bigger !
   static Small Test(U);       // not implemented !
   static Big   Test(...);     // not implemented !
   static T MakeT();           // not implemented !
public:
   enum {
     exists = sizeof(Test(MakeT())) == sizeof(Small),
     same = false };
};
```

```
int main() {
  using namespace std;
  cout
   << Conversion<double, int>::exists << ' '
   << Conversion<char, char*>::exists << ' '
   << Conversion<size_t, vector<int> >::exists << ' ';
}
// double ---> int ?
// Test(MakeT()) overloading zwischen int und ...
// int passt besser: Test liefert ein Small
// sizeof-Vergleich liefert true, exists ist 1
```

```
1 0 0
```

```
// even more is possible:
template <class T>
class Conversion<T,T> { // same Type
public:
   enum { exists = 1, same = 1 };
};

#define SUPERSUBCLASS(T,U) \
  (Conversion<const U*, const T*>::exists &&\
  !Conversion<const T*, const void*>::same)
// true if U inherits publicly from T
// or T and U are identically
```

Auch Funktionen können als Templates implementiert werden:
function templates

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

'universell' überladene Funktion, Überladung wird bevorzugt in non-template Code aufgelöst (SFINAE bei template Code),
wenn eine template-Funktion exakt passt, aber ansonsten Typumwandlungen nötig sind, wird die Schablone benutzt!
Funktionskörper muss für Compiler zugänglich sein
---> inlining en passant

Aufruf auch ohne explizite Instantiierung möglich (wenn vorliegende Parameter diese eindeutig machen):

```
max ( 3, 5 ); // non-template max
max<> ( 3, 5 ); // max<int>
max ('a', 23.45); // non-template max
max ( 3.1241, 5.0 ); // max<double>
     // better match als non-template max !
// max ( 7, "sieben" ); ERROR
max ( 7, 23L ); // non-template max
max<long> ( 7, 23L ); // ok
```

Spezialisierungen sind natürlich möglich (weitere explizite Überladung)

```
inline const char*
max (const char* a, const char* b) {
      return std::strcmp(a,b) > 0 ? a : b;
}
```

Der Template-Typ darf auch für lokale Variablen benutzt werden

```
template <typename T>
void swap (T& a, T& b) {
      T tmp(a); a = b; b = tmp;
}
```

keine Default-Template Argumente, keine Template Template Argumente und keine partielle Spezialisierung ist für Funktionstemplates erlaubt

Dadurch können auch Algorithmen generisch werden

```
// swap kann beliebige Objekte 'austauschen'
// Benutzung z.B.:
template <class T>
void simple_sort (T* vec, int size) {
  for (int m=0; m<size-1; m++)
    for (int n=m+1; n<size; n++)
      if (vec[m]>vec[n]) swap(vec[m], vec[n]);
}
int v [100]; ....
simple_sort (v, 100);
```

der entstehende Programmcode muss semantisch korrekt sein:

```
class Y {};
class X {
      // operator > not defined
public:
      operator int () {return 1; }
};

X x [20];  Y y [40];
simple_sort (x, 20);
simple_sort (y, 40);

/* swap.cc: In function `void simple_sort(T*, int)
[with T = Y]':
swap.cc:34:   instantiated from here
swap.cc:15: no match for `Y& > Y&' operator
*/
```

Fehlerausschriften sind z.T. sehr kryptisch

```
........
tstring.cpp: In function `void __static_initialization_and_destruction_0(int,
   int)':
tstring.cpp:12: no matching function for call to `std::basic_string<char,
   std::char_traits<char>, std::allocator<char> >::basic_string(const
char[23],
   int, int, int)'
/usr/local/lib/gcc-lib/sparc-sun-solaris2.8/3.1/include/g++/bits/
basic_string.tcc:233: candidates
   are: std::basic_string<_CharT, _Traits,
   _Alloc>::basic_string(_Alloc::size_type, _CharT, const _Alloc& = _Alloc())
   [with _CharT = char, _Traits = std::char_traits<char>, _Alloc =
   std::allocator<char>]
/usr/local/lib/gcc-lib/sparc-sun-solaris2.8/3.1/include/g++/bits/
basic_string.tcc:226:
                 std::basic_string<_CharT, _Traits,
   _Alloc>::basic_string(const _CharT*, const _Alloc& = _Alloc()) [with _CharT
   = char, _Traits = std::char_traits<char>, _Alloc =
std::allocator<char>] .....................
```

Für g++ gibt es einen sog. Decryptor (STLfilt)

```
pandora ahrens 54 ( ~/STLfilt/samples ) > ../gfilt tstring.cpp
BD Software STL Message Decryptor v2.31 for gcc (03/03/2003)
...
tstring.cpp: In function
   `void __static_initialization_and_destruction_0(int, int)':
tstring.cpp:12: No match for 'string(const char[23], int, int, int),
```

Für C++0X war eine Spracherweiterung um sog. Concepts vorgesehen, die gezieltere Prüfungen und Fehlerausgaben im Compiler erlaubt:

Concepts Extending C++ Templates For Generic Programming

„in letzter Minute" aus C++11 rausgestrichen .... :-(
in C++14 vielleicht als lite version

Weitere (syntaktische) Besonderheiten bei Templates:

- das Konstrukt `.template`

```
template <int N>
void printBits (const std::bitset<N>& bs)  {
  // std::cout << bs.to_string < char,
                                 char_traits<char>,
                                 allocator<char> > ();
  std::cout << bs.template to_string < .... > ();
}
```

- Zero Initialization

```
template <typename T>  void foo() {
      T x;   // OK: default ctor for classes
             // but uninitialized for build-in types
      T x = T(); // 0 (false) for built-in types
}
```

... and friends?

Beispiel: Ausgabeoperator für Stack<T>

```
template <class T> class Stack { //...

friend std::ostream&
    operator<< (std::ostream &, const Stack<T>&)
    { /* inline implementiert … */  }
    };
```

Wie aber, wenn nicht inline ?

... and friends?

```
template <class T> class Stack { //...

friend std::ostream& operator<< (std::ostream &, const
   Stack<T> &);
// IST NUR DEKLARIERT, WIRD ALSO NICHT INSTANZIERT

template<class S>
friend std::ostream& operator<<
   (std::ostream &, const Stack<S> &);
// ZU GENERISCH: EINE GANZE FAMILIE VON Stack<S>
// AUSGABEOPERATOREN (NICHT NUR T)
};
```

Wie dann?

... and friends?

Die Funktion muss schon vor der Klasse als generisch deklariert werden!

```
template <class T>
std::ostream& operator<< (std::ostream &, const Stack<T> &);
```

Dazu muss aber der Name der Klasse zuvor deklariert werden ☹

How to break this circle?
--- Vorabdeklaration der Template-Klasse:

```
template<class T> class Stack; // is generic !
template<class T>
std::ostream& operator<<
      (std::ostream& out, const Stack<T>& stack);

template<class T> class Stack {
friend std::ostream& operator<< <>
   (std::ostream&,const Stack<T>& );
   // final gebunden an T !!!
};
```

Weitere (semantische) Besonderheiten bei Templates:

two-phase lookup:

1. **Die Phase in der die Schablone vom Compiler erstmalig inspiziert wird (Template-Parameter sind noch unbekannt): lookup von *non-dependent names* (und zwar nur hier), Basisklassen werden dabei nicht betrachtet – unqualifizierte Namen sind (zumeist) *dependent*!**

2. **Die erneute Inspektion am *point of instantiation* (POI) unter Kenntnis der Template-Parameter. lookup von *dependent names* (per :: nur hier), sonstige dependence in beiden Phasen, ADL wird nur in Phase 2 realisiert.**

Weitere (semantische) Besonderheiten bei Templates:

- 2-phase lookup bringt u.U. unerwartete Effekte

```
template <typename T>
class Base {
  public:
    void exit(int);
};

template <typename T>
class Derived : Base<T> {
  public:
    void foo() {  exit(0); // Base<T>::exit is NOT considered
                  Base<T>::exit(1); // OK, but static bound
                  this->exit(2); // allows for late binding
    }
};
```

leider falsch implementiert in Visual C++ 2008

Weitere (semantische) Besonderheiten bei Templates: **SFINAE**

```
struct Test {
    typedef int Type;
};

template < typename T >
void f(typename T::Type) {} // definition #1

template < typename T >
void f(T) {}                // definition #2

void foo() {
    f<Test>(10); // call #1
    f<int>(10);  // call #2 without error thanks to SFINAE
}
```

Achtung: bei Überladung von Template-Funktionen trifft die explizite Instantiierung keine Auswahl, sondern es werden alle Funktionen 'zugleich' instantiiert von denen die vorliegende Parametersituation eine Funktion eindeutig auswählen muss!

```
template <typename T> int f(T)          { return 1; }
template <typename T> int f(T*)         { return 2; } // kein Spezialisierung !

template <typename T> struct C {
          static int f()                { return 1; }
};
template<typename T> struct C<T*> {
          static int f()                { return 2; }
};

int main() {
          std::cout << f ((int*)0) << std::endl;
          std::cout << f<int> ((int*)0) << std::endl;
          std::cout << f<int*>((int*)0) << std::endl;
          std::cout << C<int>::f() << std::endl;
          std::cout << C<int*>::f() << std::endl;
}
```

```
2
2
1
1
2
```

Der Prozess der Instanziierung ist 'Turing-mächtig'!

Beispiel: Fibonacci-Zahlen berechnen während der Übersetzung

```
#include <iostream>
template <int N>
class Fib {   long val;
public:
    Fib(): val( Fib<N-1>() + Fib<N-2>() ){}
    operator long () {return val;}
};
template<>
class Fib<0> {
public:       operator long () {return 1;}
};
```

```
template<>
class Fib<1> {
public:       operator long () {return 1;}
};
int main(){
       std::cout << Fib<42>() << std::endl;
}

// alternativ auch
#include <iostream>
template <int x> int f() { return f<x-2>()+f<x-1>(); }
template <>int f<1>()     { return 1;}
template <>int f<0>()     { return 1;}
int main(){   std::cout<<f<12>()<<std::endl;}
```

Template Klassen Instantiierungen können auch durch Template Funktionen 'provoziert' werden

```
template <class T1, class T2>
struct pair { // so in std !
       T1 first;
       T2 second;
       pair() {}
       pair(const T1& a, const T2& b) : first(a), second(b)
{}
};
```

```
template <class T1, class T2>
inline bool operator==
(const pair<T1, T2>& x, const pair<T1, T2>& y) {
        return x.first == y.first && x.second == y.second;
}

template <class T1, class T2>
inline bool operator<
(const pair<T1, T2>& x, const pair<T1, T2>& y) {
        return x.first < y.first ||
                (!(y.first < x.first) &&
                        x.second < y.second);
}

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y) {
        return pair<T1, T2>(x, y);
}
```

Type Traits

Generischer Code kennt Typen nicht vorab, möchte u.U. dennoch generalisiert auf Eigenschaften von Typen zugreifen:

```
…
class T1 { // dito T2, T3: classes i have control of
public:
  static const char* someProperty;
};

const char* T1::someProperty = "class T1"; // dito T2, T3

template <class T>
useProperty(T t) { cout << t.someProperty <<endl; }

int main() {
  useProperty(T1());
  useProperty(T2());
  useProperty(T3());
}
```

Type Traits

„Wenn der Typ aber nun ein **`int`** ist, lieber Heinrich ….“ oder
„Wenn der Typ aber nun nicht meiner ist, lieber Heinrich ….“

Idee: Man stellt jedem Typ einen sog. Traits-Typ (trait == Merkmal, Wesenszug) zur Seite, der diese Eigenschaften bereithält:

```
// voilà:
template <class Any>
class Traits {
public:
      static const char* someProperty;
};
```

Type Traits

... und spezialisiert diesen:

```
template <> class Traits<int> {
public:
        static const char* someProperty;
};

// ACHTUNG: NICHT template<> !
const char* Traits<int>::someProperty = "int";

class T4 { /* without someProperty */ }; // not under my control

template <> class Traits<T4> {
public:
        static const char* someProperty;
};

// ACHTUNG: NICHT template<> !
const char* Traits<T4>::someProperty = "class T4";
```

Type Traits

```
class T5 { // not under my control
public:  static const char* prop;
};

template <> class Traits<T5> {
public:    static const char* someProperty;
};
const char* Traits<T5>::someProperty = T5::prop; // mapping

template <class T>
void useProperty (T t) { Traits<T> tt; cout << tt.someProperty <<endl; }

const char* T5::prop = "class T5";
typedef int T6;

int main() {
      useProperty(T4());
      useProperty(T5());
      useProperty(T6());
}
```

Type Traits

```
// was ist nun mit T1,2,3 ? usеProperty geht nicht mehr wie oben
// entweder:
template <> struct Traits<T1> {
    static const char* someProperty;
};
const char* Traits<T1>::someProperty=T1::someProperty;

template <> struct Traits<T2> {
        static const char* someProperty;
};
const char* Traits<T2>::someProperty=T2::someProperty;

template <> struct Traits<T3> {
        static const char* someProperty;
};
const char* Traits<T3>::someProperty=T3::someProperty;

// und usеProperty von S. 275
```

Type Traits

```
// was ist nun mit T1,2,3 ? usProperty geht nicht mehr wie oben
// oder:

template <>
void useProperty<T1>(T1 t) { cout << t.someProperty <<endl; }

template <>
void useProperty<T2>(T2 t) { cout << t.someProperty <<endl; }

template <>
void useProperty<T3>(T3 t) { cout << t.someProperty <<endl; }
```

variadic templates

Templates mit variabler Argumentanzahl?

bislang in C++98 nicht direkt möglich, stattdessen (geniale Idee von A. Alexandrescu): Typlisten

```
template <class T, class U>
struct Typelist {
typedef T Head;
typedef U Tail; // U kann selbst wieder Liste sein
};
class NullType {}; // a „non“ type
// z.B.
typedef Typelist<
      char,
      Typelist<
              signed char,
              Typelist<
                        unsigned char,NullType>>> AllCharTypes;
```

variadic templates

+ Makros zur handlichen Erzeugung und Template-Magie für Listenoperationen:

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2)>
...

// Listenlänge:
template <class TList> struct Length;
template <>
struct Length<NullType> {
      enum { value = 0 };
};
template <class T, class U>
struct Length<Typelist<T,U>> {
      enum { value = 1 + Length<U>::value };
};
```

variadic templates

```
// indizierter Zugriff:
template <class Head, class Tail>
struct TypeAt<Typelist<Head, Tail>, 0>
{
        typedef Head Result;
};

template <class Head, class Tail, unsigned int i>
struct TypeAt<Typelist<Head, Tail>, i>
{
        typedef typename TypeAt<Tail, i-1>::Result Result;
};
// Wow !
```

unhandlich, unübersichtlich ;-(

besser: direkte Unterstützung innerhalb der Sprache mit Compilerunterstützung für (Typ-)Listenverarbeitung (ggf. Abb. auf ‚TypeList')

variadic templates

```
template<typename ... Types>
class simple_tuple;

template<> // triviale Spezialisierung
class simple_tuple<> {};

template<typename First, typename ... Rest> // rekursive Spezialisierung
class simple_tuple<First,Rest...>: // Vererbung !!!
    private simple_tuple<Rest...>  {
    First member;
public:
    simple_tuple(First const& f,Rest const& ... rest):
        simple_tuple<Rest...> {rest...}, member{f}   {}
    First const& head() const { return member; }
    simple_tuple<Rest...> const& rest() const { return *this; }
};
```

parameter pack

variadic templates

```
// indizierter Zugriff:
template<unsigned index, typename ... Types>
struct simple_tuple_entry; // allgemein

template<typename First, typename ... Types>
struct simple_tuple_entry<0, First, Types...> // Index 0
{
    typedef First const& type;
    static type value(simple_tuple<First,Types...> const& tuple) {
        return tuple.head();
    }
};
```

variadic templates

```
// indizierter Zugriff (cont.):
template<unsigned index,typename First,typename ... Types>
struct simple_tuple_entry<index,First,Types...> {
  typedef typename simple_tuple_entry<index-1,Types...>::type type;

   static type value(simple_tuple<First,Types...> const& tuple) {
       return simple_tuple_entry<index-1,Types...>::value(tuple.rest());
    }
};

template<unsigned index,typename ... Types>
typename simple_tuple_entry<index,Types...>::type
get_tuple_entry(simple_tuple<Types...> const& tuple) {
    return simple_tuple_entry<index,Types...>::value(tuple);
}
// zumeist in Bibliotheken
```

variadic templates

std::tupl

```
// Anwendung leicht verständlich:

int main() {
    simple_tuple<int,char,double> st {42,'a',3.141};
    std::cout<<get_tuple_entry<0>(st)<<","
             <<get_tuple_entry<1>(st)<<","
             <<get_tuple_entry<2>(st)<<std::endl;
   std::cout<<"sizeof(st)="<<sizeof(st)<<std::endl;

}
```

std::tuple<...> tup;
std::get<i>(tup)

```
$ g++ -std=c++0x -v variadic.cpp
… GNU C++ … version 4.4.3 …
$ a.out
42,a,3.141
sizeof(st)=16
```

variadic templates

std::tuple hat außerdem noch:

```
auto c0 = std::make_tuple(4, 5, 6, 7);

und

int    v4 = 1;
double v5 = 2;
int    v6 = 3;
double v7 = 4;
std::tie(v4, v5, v6, v7) = c0;

// vi == i
```

[std::tuple schon in TR1 (2003) aber noch ohne variadic templates implementiert]

```
template <typename ... Args>
auto Nto1 (Args... args) -> decltype(make_tuple(args...))
{ return make_tuple(args ...); }
```

variadic templates

Überladung anhand von parameter packs ist möglich: Douglas Gregor: **A Brief Introduction to Variadic Templates** (n2087.pdf) .... ein typsicheres printf in C++

```
template<typename T, typename... Args>
void printf(const char* s, const T& value, const Args&... args) {
    while (s) {
        if (s == '%' && ++s != '%') { // ignore the character that follows
                                      // the '%': we already know the type!
    •           std::cout << value;
            return printf(++s, args...);
        }
        std::cout << s++;
    }
    throw std::runtime error("extra arguments");
}
void printf(const char* s) {
    while (s) {
        if (s == '%' && ++s != '%')
           throw std::runtime error("missing arguments");
        std::cout << s++;
    }
}
```

Die C++ Standardbibliothek basiert wesentlich auf generischem Code

- Ursprünglich als STL (standard template library) bei HP entworfen, bei SGI weiterentwickelt **(http://www.sgi.com/tech/stl/)**
- eine »unkonventionelle« C++ -Bibliothek, bestehend aus Headerfiles, die massiven Gebrauch von Templates machen
  (also keine fertig übersetzten Bibliotheken) !
- auf den ersten Blick im Widerspruch zu einer der Grundgedanken der objektorientierten Programmierung: Trennung von Datenstrukturen und Algorithmen
- stellt elementare Containerklassen für Listen, Vektoren, Mengen, Multimengen, Stacks, Assoziationen ... unter einem einheitlichen Blickwinkel bereit: über allen Typen sind sog. Iteratoren definiert, die sich verhalten, wie Zeiger in die entsprechenden Strukturen, kanonisch in Fortsetzung von C-Feldern und C-Zeigern)
- Zeiger selbst eignen sich auch als elementare Iteratoren

C++ -Standardbibliothek:

die folgenden 51 Standard C++ headers (einschließlich der 18 zusätzlichen Standard C headers) machen eine sog. hosted implementation of Standard C++ aus:

<algorithm>, <bitset>, <cassert>, <cctype>, <cerrno>, <cfloat>,
<ciso646>, <climits>, <clocale>, <cmath>, <complex>, <csetjmp>,
<csignal>, <cstdarg>, <cstddef>, <cstdio>, <cstdlib>,
<cstring>, <ctime>, <cwchar>, <cwctype>, <deque>, <exception>,
<fstream>, <functional>, <iomanip>, <ios>, <iosfwd>, <iostream>,
<istream>, <iterator>, <limits>, <list>, <locale>,
<map>, <memory>, <new>, <numeric>, <ostream>, <queue>, <set>,
<sstream>, <stack>, <stdexcept>, <streambuf>, <string>, <strstream>,
<typeinfo>, <utility>, <valarray>, <vector>

Grundlegende Beobachtung: Algorithmen sind (oft) abstrakt, in dem Sinne, dass sie unabhängig sind von den elementaren Typen sind, auf denen sie operieren:

```
// Beispiel: Lineares Suchen
int* find (int* first, int* last, int value) {
       while (first != last && *first != value) ++first;
       return first;
}  //                          warum nicht bool find(....) ???
```

das geht genauso mit double*, XYZ*, ... : Abstraktion vom elementaren Typ und Reduzierung auf Konzepte von Anfang, Ende, Vergleich ...

```
template <class InputIterator, class T>
InputIterator find
(InputIterator first, InputIterator last, const T& value) {
       while (first != last && *first != value) ++first;
       return first;
}
```

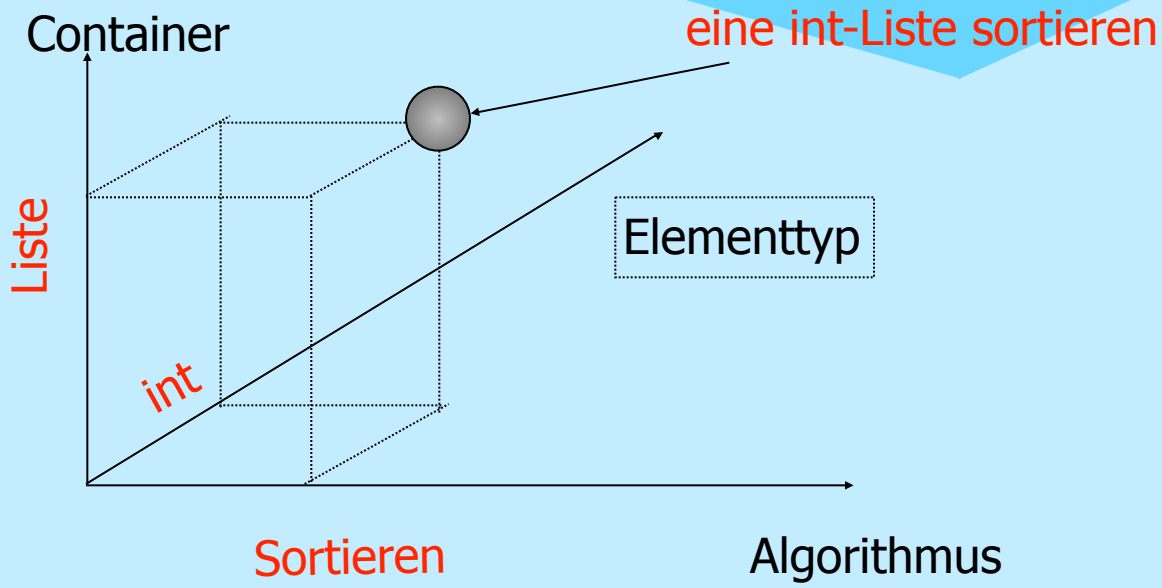
Weitere Verallgemeinerungen sind oft möglich:

```
template <class InputIterator, class Predicate>
InputIterator find_if
(InputIterator first, InputIterator last, Predicate pred) {
       while (first != last && !pred(*first)) ++first;
       return first;
}
// pred ist Verallgemeinerung von ==(. , value)
// pred muss als Funktion aufrufbar sein
```

Ziel: "Algorithmus x angewendet auf Containertyp y des Basistyps z "

Traditioneller Ansatz:

m (Algorithmen) * n (Containertypen) * t (Basistypen)



- Templates in ihrer primären Anwendung:
  m (Algorithmen) * n (template<t> Container)
- Ansatz der STL: die Algorithmen agieren auf austauschbaren (aus Templates instantiierbaren) Zugriffsoperationen
  m (Algorithmen) + n (template<t> Container)
- Iteratoren entkoppeln die Containertypen von den Algorithmen, die auf ihnen operieren
- ein Iteratorwert verkörpert eine Position, kann mittels ++ weitergesetzt werden ACHTUNG: man benutze immer Prefix-In/Dekrement (keine temporäre Variable)
- Iteratoren können auf Un-/Gleichheit geprüft werden
- * liefert das referenzierte Element (wie bei Zeigern)
- (Input-)Iteratorobjekte können kopiert werden !

Per Container gibt es lokale Typvereinbarungen, die die jeweiligen Spezifika des Containers "kennen", nach außen sich jedoch gleich als abstrakte Konzepte verhalten:

```
namespace std {
        template <class T> // (fast) so aus <vector>
        class vector { public:
                typedef T                       value_type;
                typedef value_type*             pointer;
                typedef const value_type*       const_pointer;
                typedef value_type*             iterator;
                typedef const value_type*       const_iterator;
                typedef value_type&             reference;
                typedef const value_type&       const_reference;
                typedef size_t                  size_type;
                typedef ptrdiff_t               difference_type;
        .....};
} // end namespace std
```

Benutzung muss die konkreten Typen (der Elemente und des Containers!) NICHT kennen:

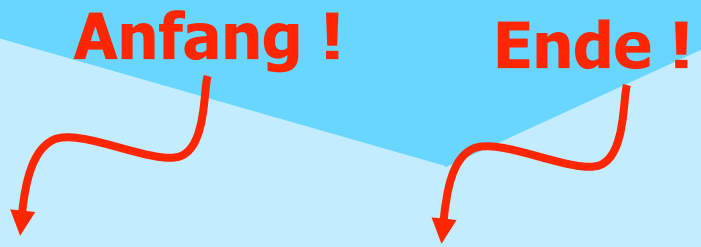
```
typedef vector<int> Ivec;
Ivec v;
// fill v ....
// iterate:

for (Ivec::iterator i = v.begin(); i != v.end(); ++i)
       do_something_with( *i );
...
// ist 42 drin:
Ivec::iterator where = find (v.begin(), v.end(), 42);
if (where != v.end())       // ja !
else                        // nein !
```

Anfang !

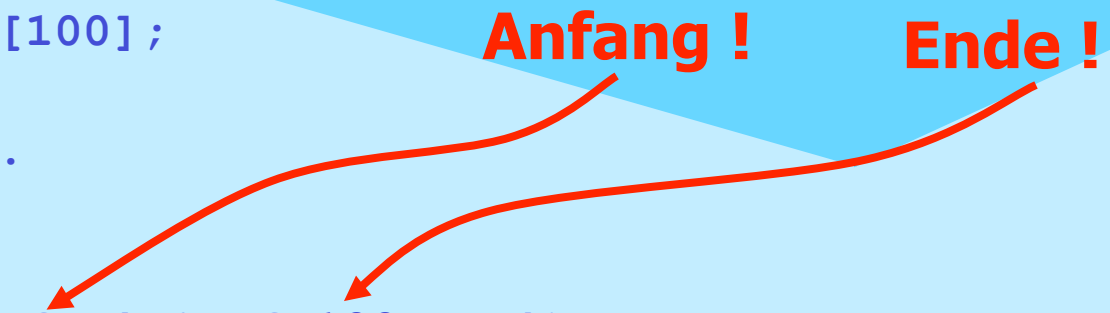
Ende !

Benutzung muss die konkreten Typen (der Elemente und des Containers!) NICHT kennen:

```
typedef set<int> Iset;
Iset s;
// fill s ....
// iterate:

for (Iset::iterator i = s.begin(); i != s.end(); ++i)
       do_something_with( *i );
...
// ist 42 drin:
Iset::iterator where = find (s.begin(), s.end(), 42);
if (where != s.end())       // ja !
else                        // nein !
```



Benutzung muss die konkreten Typen (der Elemente und des Containers!) NICHT kennen, auch C-Felder können (als triviale Container) mit ihren Iteratoren (Zeiger) verwendet werden




```
typedef int F[100];
F f;
// fill f ....
// iterate:

for (int* i = f; i != f+100; ++i)
       do_something_with( *i );
...
// ist 42 drin:
int* where = find (f, f+100, 42);
if (where != f+100)          // ja !
else                         // nein !
```

noch einheitlicher mit C++11: free functions begin/end

```
typedef int F[100];
typedef std::list<int> L;
F f; L l;
// fill f ... l ....
// iterate:

   for (auto i = begin(f); i != end(f); ++i)
        do_something_with( *i );
   for (auto i = begin(l); i != end(l); ++i)
        do_something_with( *i );
```

Anfang !

Ende !

```
// see http://herbsutter.com/2013/05/13/gotw-2-solution-temporary-objects/
```

decltype -- deduction of a type from an expression (C++11)

`decltype(E)`

ist der Typ ("declared type") des Namens oder des Ausdrucks E und kann in Deklarationen verwendet werden. Der Ausdruck wird NICHT berechnet!

```
void f(const vector<int>& a, vector<float>& b) {
  typedef decltype(a[0]*b[0]) Tmp;
  for (int i=0; i<b.size(); ++i) {
    Tmp* p = new Tmp(a[i]*b[i]);
  }
}
```

auto ist oft einfacher. **decltype** wird gebraucht, wenn man einen Typ für etwas benötigt, das keine Variable ist z. B. ein return Typ.

decltype -- deduction of a type from an expression

- decltype unterscheidet Werte und Referenzen:

```
•int val = 3;
•int& ref = val;

•decltype(val)      ------> int
•decltype(ref)      ------> int&
•decltype(ref+1)    ------> int
•decltype((val))    ------> int&
```

- (val) ist ein Ausdruck und hat eine Adresse

decltype -- deduction of a type from an expression

Wie kann man eine generische Funktion add schreiben?

```
template <typename X, typename Y>
??? add(X x, Y y)
{
    return x + y;
}
```

decltype -- deduction of a type from an expression

Wie kann man eine generische Funktion add schreiben?

```
// so nicht:
template <typename X, typename Y>
decltype(x+y) add(X x, Y y) //???
{     // unknown x and y
    return x + y;
}
```

decltype -- deduction of a type from an expression

Wie kann man eine generische Funktion add schreiben?

```
// so schon, aber extrem hässlich:
template <typename X, typename Y>
decltype((*(X*)0) + (*(Y*)0))) add(X x, Y
  y)
{
    return x + y;
}
```

decltype -- deduction of a type from an expression

Wie kann man eine generische Funktion add schreiben?

mit der neuen Funktionssyntax elegant:

```
// so:
template <typename X, typename Y>
auto add(X x, Y y) -> decltype (x+y)
{       // auto: deduce from elsewhere
    return x + y;
}
```

<iterator> free functions begin/end

```
template <class C> auto begin(C& c) -> decltype(c.begin());

template <class C> auto begin(const C& c) -> decltype(c.begin());

template <class C> auto end(C& c) -> decltype(c.end());

template <class C> auto end(const C& c) -> decltype(c.end());

template <class T, size_t N> T* begin(T (&array)[N]);

template <class T, size_t N> T* end(T (&array)[N]);
```

<iterator> free functions begin/end

```
template <class C> auto begin(C& c) -> decltype(c.begin())
    •{ return c.begin(); }

template <class C> auto begin(const C& c) -> decltype(c.begin())
    •{ return c.begin(); }

template <class C> auto end(C& c) -> decltype(c.end())
    •{ return c.end(); }

template <class C> auto end(const C& c) -> decltype(c.end())
    •{ return c.end(); }

template <class T, size_t N> T* begin(T (&array)[N])
    •{ return array; }

template <class T, size_t N> T* end(T (&array)[N])
{ return array + N; }
```

Generische Algorithmen `(#include <algorithm>)`

von speziellen Containern durch Iteratoren entkoppelt!
standardisiert ist das Verhalten (ISO 14882/1998)

25.1.1 - For each [lib.alg.foreach]

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function f);
```

-1- Effects: Applies f to the result of dereferencing every iterator in the range [first, last), starting from first and proceeding to last - 1.

-2- Returns: f.

-3- Complexity: Applies f exactly last - first times.

-4- Notes: If f returns a result, the result is ignored.

nicht die Implementation:

```
// bcc32:
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function f) {
    while (first != last) f(*first++);
    return f;
}
// g++ 3.2.2: siehe http://www.boost.org/libs/concept_check/concept_check.htm
template<typename _InputIter, typename _Function>
_Function for_each(_InputIter __first, _InputIter __last, _Function __f) {
    // concept requirements
    __glibcpp_function_requires(_InputIteratorConcept<_InputIter>)
    for ( ; __first != __last; ++__first)
      __f(*__first);
    return __f;
}
// vc++ 6.0: siehe http://www.dinkumware.com
template<class _II, class _Fn> inline
          _Fn for_each(_II _F, _II _L, _Fn _Op)
          {for (; _F != _L; ++_F)
                    _Op(*_F);
          return (_Op); }
```

Nicht-Modifizierende Algorithmen (#include <algorithm>)

`for_each`	eine (lesende) Operation auf alle Elemente anwenden
`find`	erstes Auftreten eines Wertes ermitteln
`find_if`	erste Erfüllung eines Prädikates ermitteln
`search`	erstes Auftreten einer Teilfolge ermitteln, gleiche Werte / Prädikat **! overloaded**
`find_end`	letztes Auftreten einer Teilfolge ermitteln, gleiche Werte
`find_end_if`	letztes Auftreten einer Teilfolge ermitteln, Prädikat
`find_first_of`	erstes Auftreten eines Elementes aus einem Bereich ermitteln, gleiche Werte
`find_first_of_if`	erstes Auftreten eines Elementes aus einem Bereich ermitteln, Prädikat
`adjacent_find`	erstes Auftreten benachbarter gleicher Elemente ermitteln, gleiche Werte / Prädikat **! overloaded**
`min_element`	Position des kleinsten Elements ermitteln, < / binäres bool Prädikat
`max_element`	Position des größten Elements ermitteln, < / binäres bool Prädikat
`count`	Elemente zählen, gleich Vorgabewert
`count_if`	Elemente zählen, für die Prädikat erfüllt
`equal`	Vergleich Bereich gegen Bereichsanfang, gleiche Werte / Prädikat
`lexicographical_compare`	Vergleich zweier Bereiche, elementweise gleich / Prädikat
`mismatch`	Position der ersten Abweichung bei Vergleich Bereich gegen Bereichsanfang gleiche Werte / Prädikat (Ergebnis: Iteratorpaar!)

Modifizierende Algorithmen (---> nicht für [multi]set/map !) (#include <algorithm>)

`copy`	einen Bereich an einen Bereichsanfang kopieren (Platz muss ausreichen!)
`copy_backward`	einen Bereich vor ein Bereichsende kopieren (Platz muss ausreichen!)
`swap_ranges`	Austausch Bereich gegen Bereichsanfang
`transform`	eine lesende und schreibende Operation auf alle Elemente anwenden auch mit Bereich und Bereichsanfang, die binär verknüpft werden und Ergebnisse an einen weiteren Bereichsanfang geschrieben werden (Platz muss ausreichen!)
`fill`	einen Bereich mit einem Wert füllen
`fill_n`	ab Bereichsanfang n Elemente mit einem Wert füllen (Platz muss ausreichen!)
`generate`	einen Bereich mit dem Ergebnis einer Operation füllen
`generate_n`	ab Bereichsanfang n Elemente mit dem Erg. einer Operation füllen (Platz muss ausreichen!)
`replace`	in einem Bereich alle Elemente mit altem Wert durch neuen Wert ersetzen
`replace_if`	in einem Bereich alle Elemente für die eine Prädikat gilt durch neuen Wert ersetzen
`replace_copy`	in einem Bereich alle Elemente mit altem Wert durch neuen Wert ersetzt an einen Bereichsanfang kopieren
`replace_copy_if`	in einem Bereich alle Elemente für die eine Prädikat gilt durch neuen Wert ersetzt an einen Bereichsanfang kopieren

Löschende Algorithmen (#include <algorithm>)

ACHTUNG: Größe und Ende des manipulierten Containers ändert sich NICHT!
Ergebnis ist immer das neue Ende des Containers!

`remove` in einem Bereich alle Elemente löschen, die gleich einem Wert sind

`remove_if` in einem Bereich alle Elemente löschen, für die ein Prädikat gilt

`remove_copy` aus einem Bereich alle Elemente die gleich einem Wert sind, gelöscht an einen Bereichsanfang kopieren

`remove_copy_if` aus einem Bereich alle Elemente für die ein Prädikat gilt, gelöscht an einen Bereichsanfang kopieren

`unique` in einem Bereich gleiche aufeinanderfolgende Elemente zu einem kollabieren

UND

in einem Bereich bzgl. einem Prädikat gleiche aufeinanderfolgende Elemente zu einem kollabieren

`unique_copy` aus einem Bereich gleiche aufeinanderfolgende Elemente zu einem kollabiert an einen Bereichsanfang kopieren

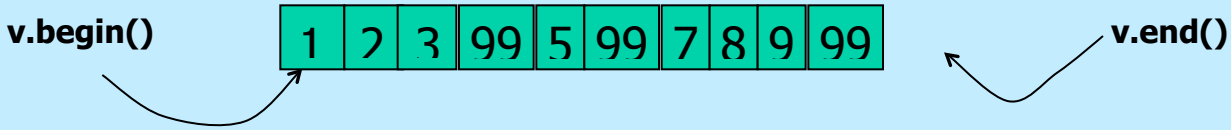
UND

aus einem Bereich bzgl. einem Prädikat gleiche aufeinanderfolgende Elemente zu einem kollabiert an einen Bereichsanfang kopieren

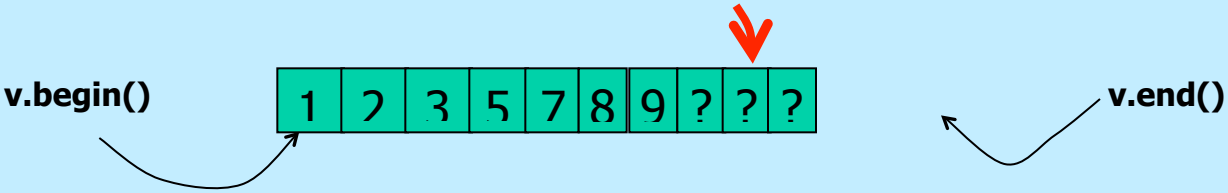
Löschende Algorithmen

ACHTUNG: Größe und Ende des manipulierten Containers ändert sich NICHT! Ergebnis ist immer das neue Ende des Containers!

Scott Meyers: Effective STL Item 32 (pp.139): "Follow `remove`-like Algorithms by `erase` if you really want to remove something."




```
vector<int>::iterator newEnd(remove(v.begin(), v.end(), 99));
```




```
v.erase((remove(v.begin(), v.end(), 99), v.end()); // !!!
```

Löschende Algorithmen

ACHTUNG: Auch wenn damit die logische Konsistenz wiederhergestellt ist, wird bei Vektoren kein Speicherplatz zurückgegeben!

Szenario: Sammle Kandidaten K in einem `vector<K> k` (~100.000) treffe Auswahl (die besten 10) entferne und lösche den Rest: der Vektor belegt immer noch 100.000 Elemente im Speicher :-(

Scott Meyers: Effective STL Item 17 (pp.77): "Use » the `swap` trick« to trim excess capacity."

```
vector<K>(k).swap(k); // wie bitte ???
```

```
// {
//     vector<K> tmp(k); // copy-ctor erzeugt so viele elemente wie nötig
//     tmp.swap(k);// k ist der neue (reduzierte) Vektor, tmp wird freigegeben
//  }
```

Mutierende Algorithmen (#include <algorithm>)

`reverse` in einem Bereich die Reihenfolge aller Elemente umkehren

`reverse_copy` aus einem Bereich alle Elemente in umgekehrter Reihenfolge an einen Bereichsanfang kopieren

`rotate` einen Bereich so rotieren, dass ein neuer Bereichsanfang zum ersten Element wird

`rotate_copy` einen Bereich rotiert, an neuem Bereichsanfang an einen Bereichsanfang kopieren

`next_permutation` in einem Bereich die nächste Permutation erzeugen, liefert false, wenn letzte (lexikogr. aufsteigende) erreicht ist, sonst true

`prev_permutation` in einem Bereich die vorhergehende Permutation erzeugen, liefert false, wenn letzte (lexikogr. absteigende) erreicht ist, sonst true

`random_shuffle` einen Bereich zufällig nach einem impliziten oder expliziten Zufallsgenerator 'verwürfeln'

`partition` in einem Bereich alle Elemente, für die ein Prädikat gilt, nach vorn schieben, Ergebnis ist die erste Position, an der das Prädikat nicht gilt

`stable_partition` in einem Bereich alle Elemente, für die ein Prädikat gilt, nach vorn schieben, Ergebnis ist die erste Position, an der das Prädikat nicht gilt, Reihenfolgen in Teilmengen bleibt erhalten

Sortierende Algorithmen (#include <algorithm>)

`sort` einen Bereich nach < oder einer Relation sortieren

`stable_sort` einen Bereich nach < oder einer Relation sortieren, gleiche Elemente bleiben in relativer Position zueinander !

`partial_sort` einen Bereich bis zu einer Position nach < oder einer Relation sortieren

`partial_sort_copy` aus einem Bereich bis zu einer Position nach < oder einer Relation sortiert in einen anderen Bereich kopieren

`nth_element` einen Bereich nach < oder einer Relation um eine Position sortieren, so dass diese an der richtigen Stelle steht (links <=, rechts >)

`make_heap` einen Bereich nach < oder einer Relation in einen Heap umwandeln (∀ i : h[i]>h[2*i+1] und h[i]>h[2*i+2])

`push_heap` das letzte Element eines Bereichs [f, l) in einen Heap [f, l-1) zu einem Heap einbauen (make_heap, push_back, push_heap) (nach < oder einer Relation)

`pop_heap` das erste (nach < oder einer Relation größte) Element mit dem letzten vertauschen und einen neuen Heap in [f, l-1) erzeugen

`sort_heap` einen Heap nach < oder einer Relation sortieren (Ergebnis ist bei < kein Heap mehr) O(N*log(N))

Algorithmen für sortierte Bereiche (`#include <algorithm>`)

`lower_bound` in einem (nach < oder einer Relation) sortierten Bereich die erste Position ermitteln, an der ein Wert sortiert nach < oder einer Relation eingefügt werden kann

`upper_bound` ein einem (nach < oder einer Relation) sortierten Bereich die letzte Position ermitteln an der ein Wert sortiert nach < oder einer Relation eingefügt werden kann

`equal_range` in einem nach < oder einer Relation sortierten Bereich die erste und letzte Position für sortiertes Einfügen als `pair<FwdIter, FwdIter>` ermitteln

`binary_search` ist ein Wert in einem nach < oder einer Relation sortierten Bereich enthalten ?

`includes` ist ein nach < oder einer Relation sortierter Bereich in einem anderen nach < oder einer Relation sortierten Bereich enthalten ?

Algorithmen für sortierte Bereiche (#include <algorithm>)

`merge` zwei nach < oder einer Relation sortierte Bereiche an einen Bereichsanfang sortiert mischen (Platz muss ausreichen!)

`inplace_merge` zwei nach < oder einer Relation sortierte Teilbereiche die direkt hintereinander liegen werden sortiert gemischt (f1, l1f2, l2)

`set_union` zwei nach < oder einer Relation sortierte Bereiche an einen Bereichsanfang sortiert mischen, Elemente, die in beiden vorkommen werden nur einmal erfasst

`set_intersection` aus zwei nach < oder einer Relation sortierten Bereichen wird die Schnittmenge an einen Bereichsanfang übertragen

`set_difference` aus zwei nach < oder einer Relation sortierten Bereichen wird die Differenzmenge (im ersten und nicht im zweiten) an einen Bereichsanfang übertragen

`set_symmetric_difference` aus zwei nach < oder einer Relation sortierten Bereichen wird die Komplementärmenge (im ersten und nicht im zweiten oder umgekehrt) an einen Bereichsanfang übertragen

Numerische Algorithmen (#include <numeric>)

`accumulate` bildet die Summe (bzw. Anwendung eines Operators) eines Initialwertes mit allen Elementen eines Bereichs

`partial_sum` bildet die Summe (bzw. Anwendung eines Operators) aller Anfangsstücke eines Bereichs und kopiert diese ab einem Bereichsanfang

`adjacent_difference` bildet die Differenzen (bzw. Anwendung eines Operators) jeweils benachbarter Elemente eines Bereichs und kopiert diese ab einem Bereichsanfang (erstes Element wird übernommen)

`inner_product` bildet die Summe (bzw. Anwendung eines Operators) aus einem Initialwert und dem Skalarprodukt (bzw. Anwendung eines weiteren Operators) eines Bereichs und eines Bereichsanfangs

Folgende Containertypen existieren schon länger (in C++98):

Typ	Header	Charakteristik	Iteratoren
`vector`	`<vector>`	dynamische Felder mit wahlfreiem Zugriff, erweiterbar am Ende	`RandomAccess`
`deque`	`<deque>`	dynamische Felder mit wahlfreiem Zugriff, erweiterbar am Anfang und am Ende	`RandomAccess`
`list`	`<list>`	doppelt verkette Listen ohne wahlfreiem Zugriff, nicht sortiert	`Bidirectional`
`set`	`<set>`	Mengen mit impliziter Sortierung, keine Duplikate	`Bidirectional`
`multiset`	`<set>`	Mengen mit impliziter Sortierung, Duplikate erlaubt	`Bidirectional`
`map`	`<map>`	Mengen von Schlüssel/Wert- Paaren mit impliziter Sortierung nach dem Schlüssel, keine Duplikate	`Bidirectional`
`multimap`	`<map>`	Mengen von Schlüssel/Wert- Paaren mit impliziter Sortierung nach dem Schlüssel, Duplikate erlaubt	`Bidirectional`

Folgende Containertypen kommen neu hinzu (in C++11):

Typ	Header	Charakteristik	Iteratoren
`array`	`<array>`	Felder fester Länge mit wahlfreiem Zugriff, erweiterbar am Ende	`RandomAccess`
`forward_list`	`<forward_list>`	einfach (vorwärts) verkette Listen ohne wahlfreiem Zugriff, nicht sortiert	`forward`
`unordered_set`	`<unorderd_set>`	unsortierte Mengen mit Hash, keine Duplikate	`forward`
`unordered_ multiset`	`<unorderd_set>`	unsortierte Mengen mit Hash, Duplikate erlaubt	`forward`
`unordered_ map`	`<unorderd_map>`	unsortierte Mengen von Schlüssel/Wert-Paaren mit Hash, keine Duplikate	`forward`
`unordered_ multimap`	`<unorderd_map>`	unsortierte Mengen von Schlüssel/Wert-Paaren mit Hash, Duplikate erlaubt	`forward`

gemeinsame (generische) Operationen aller Container-Klassen:

1. Erzeugung und Zerstörung

Ausdruck	Bedeutung
`Container<Typ> m`	erzeugt einen leeren Container ohne Elemente
`Container<Typ> m1 (m2)`	erzeugt einen Container als Kopie eines anderen (gleichen Typs)
`Container<Typ> m (f, l)` *f wie first* *l wie last*	erzeugt einen Container und initialisiert ihn mit Kopien der Elemente aus dem Bereich [f, l) member template: beliebige Quell-Container (anderer Typen)!
`m.~Container<Typ> ()`	löscht alle Elemente und gibt deren Speicherplatz frei

gemeinsame (generische) Operationen aller Container-Klassen:

2. Nichtverändernde Operationen

Ausdruck	Bedeutung
`m.size ()`	aktuelle Anzahl von Elementen
`m.empty ()`	leer? (entspricht m.size()==0 ABER SCHNELLER!)
`m.max_size ()`	maximal mögliche Anzahl von Elementen

`m1 == m2`
`m1 != m2`
`<, >, <=, >=`
	Gleichheit (== auf alle Elemente angewendet)
	Ungleichheit (dito)
	lexikografische Ordnung

gemeinsame (generische) Operationen aller Container-Klassen:

3. Zuweisende Operationen

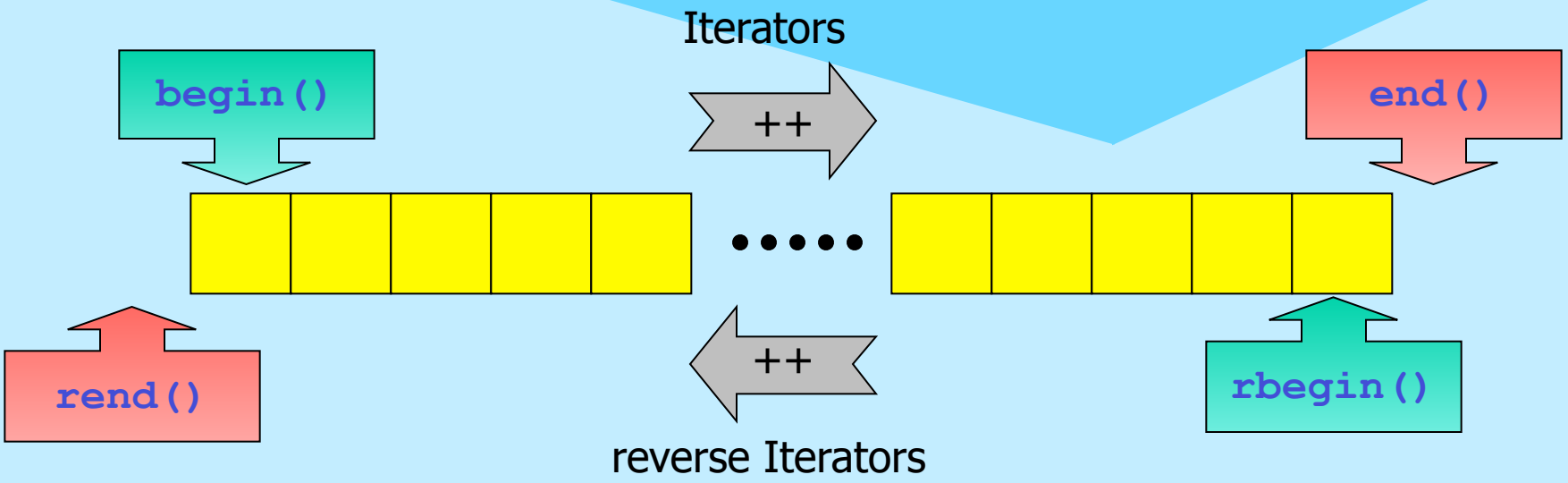
Ausdruck	Bedeutung
`m1 = m2`	Zuweisung
`m1.swap (m2)`	Vertauschen aller Elemente
`swap (m1, m2)`	!! immer schneller als m1 = m2

gemeinsame (generische) Operationen aller Container-Klassen:

4. Zugreifende Operationen

Ausdruck	Bedeutung
`m.begin ()`	Iterator auf das erste Element
`m.end ()`	Iterator hinter das letzte Element
`m.rbegin ()`	Iterator auf das letzte Element
`m.rend ()`	Iterator vor das erste Element

Abstrakte Iteratoren



alle Operationen liefern sog. const iterators für konstante Container
(über solche ist der Container nicht veränderbar)

gemeinsame (generische) Operationen aller Container-Klassen:

5. Einfügende/Löschende Operationen

Ausdruck	Bedeutung
`m.insert (pos, e)`	Kopie von e bei pos einfügen (Rückgabewert und Bedeutung von pos hängen von Containertyp ab)
`m.erase (pos)`	löscht Element an der Position pos
`m.erase (f, l)`	löscht alle Elemente aus dem Bereich [f, l) liefert die Position des Folgeelements
`m.clear ()`	leert den Container

Wichtige Eigenschaften der Container

- die Effizienz aller Operationen ist optimiert, der Standard macht Vorgaben zu oberen Grenzen
- die verfügbaren Implementationen orientieren sich am state of the art in ihrer algorithmischen Umsetzung
- die generischen (container-unabhängigen) Algorithmen sind in `<algorithm>` definiert
- häufig gibt es für spezielle Container bessere Algorithmen als Memberfunktionen (z.B. `set<T>::find`)
- nicht alle Kombinationen von Algorithmen und Containern sind möglich (sinnvoll) (z.B. `sort` auf `set`: Mengen sind bereits sortiert, `sort` auf `list` ist nicht möglich)
- Alle Container arbeiten mit einer Wertesemantik, d.h. Elemente werden immer kopiert !

Member-Algorithmen sind besser als globale:

```
#include <set>
#include <iostream>
#include <algorithm>
#include "Timer.h"
using namespace std;
typedef set<int> Set;

main() {
  Set s;  int probe = rand();
  {
        cout<<"Generating 1.000.000 nodes: "<<flush;
        Timer t;
        for(int i=0; i<500000; ++i) s.insert(rand());
        s.insert(probe);
        for(int i=1; i<500000; ++i) s.insert(rand());
        cout<<s.size()<<" Elemente in der Menge"<<endl;
  }
}
```

Timer.h

```
#include <ctime>
#include <iostream>
const double millis= 1000000.0;
class Timer { long t_;
    void report()
    { std::cout<<t_/millis<<"s"<<std::endl; }
public:
    Timer(): t_(clock()){}
    ~Timer(){ t_=clock()-t_;report();}
};
```

Member-Algorithmen sind besser als globale:

```
{
        cout<<"s.find(...): "<<flush;
        Timer t;
        for (int i=0; i<1000; ++i) s.find(probe); // member
}
{
        cout<<"find (...): "<<flush;
        Timer t;
        for (int i=0; i<1000; ++i)
                find(s.begin(), s.end(), probe); // generic
}
}
```

```
Generating 1.000.000 nodes:
999752 Elemente in der Menge
5.22s
s.find(...): 0s
find (...): 346.93s !!!
```

nicht alle Kombinationen von Algorithmen und Containern sind möglich:

```
// vector, algorithm, iostream
using namespace std;
typedef vector<int> V;

template <class CT>
void out(const CT& c) {
  for(typename CT::const_iterator i=c.begin(); i!=c.end(); ++i)
    std::cout<<*i<<' ';
  std::cout<<std::endl;
}

int main() {
  V v;
  for(int i=0; i<10; ++i) v.push_back(rand());
  out(v);
  sort(v.begin(), v.end());
  out(v);
}
```

nicht alle Kombinationen von Algorithmen und Containern sind möglich: `ll.c`

```
// list, algorithm, iostream
using namespace std;
typedef list<int> L;

template <class CT>
void out(const CT& c) {
  for(typename CT::const_iterator i=c.begin(); i!=c.end(); ++i)
    std::cout<<*i<<' ';
  std::cout<<std::endl;
}

int main() {
  L l;
  for(int i=0; i<10; ++i) l.push_back(rand());
  out(l);
  sort(l.begin(), l.end()); // ??? list hat keine random access iteratoren!
  out(l);
}
```

```
l.sort(); // OK
```

nicht alle Kombinationen von Algorithmen und Containern sind möglich:

```
mio ahrens 72 ( c++/experimente ) > make ll
/usr/include/g++/stl_algo.h: In function `void sort<_List_iterator<int,int
&,int
 *> >(_List_iterator<int,int &,int *>, _List_iterator<int,int &,int *>)':
ll.cc:17:   instantiated from here
/usr/include/g++/stl_algo.h:1320: no match for `_List_iterator<int,int &,int
*>
& - _List_iterator<int,int &,int *> &'
.....
mio ahrens 73 ( c++/experimente ) > ~/STLfilt/gfilt ll.cc
BD Software STL Message Decryptor v2.31 for gcc (03/03/2003)
stl_algo.h: In function
   `void sort<list<int>::iter>(list<int>::iter, list<int>::iter)':
ll.cc:17:   instantiated from here
stl_algo.h:1320: no match for `list<int>::iter & - list<int>::iter &'
    [STL Decryptor: Suppressed 14 more STL standard header messages]
```

Vektoren (`#include <vector>`)

- dynamische Arrays eines beliebigen Typs mit wahlfreiem Zugriff
- Abstraktion von C-Feldern, unsortiert
- alle Algorithmen sind anwendbar (RandomAccessIterator)
- sehr gutes Zeitverhalten beim Löschen und Einfügen am Ende
- ansonsten ist jedes Löschen/Einfügen mit dem Verschieben (Zuweisen) von Elementen verbunden ! (--> schlechtes Zeitverhalten)
- Vektoren wachsen dynamisch:

`size()`	Anzahl der aktuell enthaltenen Elemente
`max_size()`	Anzahl der maximal möglichen Elemente (impl.abh.)
`capacity()`	Anzahl der insgesamt (ohne) Reallokierung aufnehmbaren Elemente
`reserve(size_type)`	Freihalten von Platz

Vektoren (#include <vector>)

- beim Einfügen/Löschen werden u.U. Iteratoren ungültig, bei Reallokierung = Copykonstruktor + Destruktor am alten Platz / Element !!! werden alle Iteratoren ungültig

Ausdruck	Bedeutung
`vector<E> m (n)`	n E-Elemente per default-Konstruktor
`vector<Elem> m (n, e)`	n E-Elemente als Kopie von e
`m.capacity ()`	freier Platz ohne Reallokierung
`m.reserve (size_type)`	Platz freihalten

Ausdruck	Bedeutung
`m.assign (n, e)`	Zuweisung von n Elementen, die als Kopie von e erzeugt werden
`m.assign (f, l)`	Zuweisung von Kopien der Elemente aus dem Bereich [f, l)
`m.at (n)`	Element am Index n (mit Prüfung ob es existiert, ggf. out_of_range Exc. !)
`m[n]`	Element am Index n (ohne Prüfung ob es existiert !)
`f.front ()`	das erste Element (ohne Prüfung ob es existiert!)
`f.back ()`	das letzte Element (ohne Prüfung ob es existiert!)

Ausdruck	Bedeutung
`m.insert (pos,e)`	fügt Kopie von e an pos ein, liefert Position des neuen Elements
`m.insert (pos, f, l)`	bei pos Kopien von [f, l) einfügen Resultat void (member template !)
`m.push_back (e)`	Kopie von e hinten anhängen
`m.pop_back ()`	löscht letztes Element (gibt nichts zurück)
`m.resize (n)`	Größe auf n ändern und ggf. mit dc Elementen auffüllen
`m.resize (n, e)`	Größe auf n ändern und ggf. mit Kopien von e auffüllen

dc - default constructed

Deques "decks" [double ended queue] (#include <deque>)

- nach zwei Seiten dynamisches Array eines beliebigen Typs mit wahlfreiem Zugriff, unsortiert
- alle Algorithmen sind anwendbar (RandomAccessIterator)
- sehr gutes Zeitverhalten beim Löschen und Einfügen am Anfang und am Ende
- ansonsten ist jedes Löschen/Einfügen mit dem Verschieben (Zuweisen) von Elementen verbunden ! (--> schlechtes Zeitverhalten)
- Deques werden i.allg. in mehreren Speicherblöcken (anders als Vektoren) automatisch verwaltet:
  - es gibt keine Vorreservierung, implizite Reallokierung
  - bei jedem Einfügen werden **alle Verweise potentiell ungültig**
  - wahlfreier Zugriff ist zwar möglich aber langsamer als bei Vektoren

Ausdruck	Bedeutung
`wie vector`	Erzeugung ...
`keine weiteren`	Nicht verändernde Operationen ...
`wie vector`	Zuweisung ...
`wie vector auch at und []`	Zugriffe ...
`wie vector  +`	Einfügen / Löschen ...
`m.push_front (e)`	Kopie von e vorn anhängen
`m.pop_front ()`	löscht erstes Element (gibt nichts zurück)

Listen (#include <list>)

- doppelt verkettete Liste eines beliebigen Typs ohne wahlfreien Zugriff (man muss sich "durchhangeln")
- unsortiert
- nicht alle Algorithmen sind anwendbar (BidirektionalIterator)
- gutes Zeitverhalten beim Löschen und Einfügen an beliebiger Stelle !
- Verweise auf Elemente bleiben dabei gültig !!

Ausdruck	Bedeutung
`wie deque`	Erzeugung ...
`keine weiteren (nur diese)`	Nicht verändernde Operationen ...
`wie deque`	Zuweisung ...
`wie deque ohne at und []`	Zugriffe ...
`wie deque +`	Einfügen / Löschen ...
`m.remove (e)`	löscht alle! Elemente mit dem Wert e
`m.remove_if (op)`	löscht alle! Elemente für die op(e) gilt
`m.erase (pos)`	löscht Element bei pos und liefert Position des Folgeelementes
`m.erase (f, l)`	löscht Elemente der Bereichs [f,l) und liefert Position des Folgeelementes

Ausdruck	Bedeutung
`m.unique ()`	kollabiert Folgen von Elementen mit gleichem Wert
`m.unique (op)`	kollabiert Folgen von Elementen mit op(e1)==op(e2)
`m1.splice (pos,m2)`	verschiebt alle Elemente von m2 nach m1 vor die Position pos
`m1.splice (pos,m2,m2pos)`	verschiebt das Element von m2 an der Position m2pos nach m1 vor die Position pos (m1, m2 identisch ist erlaubt)
`m1.splice (pos,m2,m2f,m2l)`	verschiebt alle Elemente von m2 aus dem Bereich [m2f, m2l) nach m1 vor die Position pos (m1, m2 identisch ist erlaubt)

Ausdruck	Bedeutung
`m.sort ()`	sortiert nach <
`m.sort (op)`	sortiert nach op
`m1.merge (m2)`	mischt sortiertes m2 in sortiertes m1 ein
`m1.merge (m2,op)`	mischt sortiertes m2 in sortiertes m1 ein, sortiert dabei nach op
`m.reverse ()`	kehrt die Reihenfolge der Elemente um

Mengen und - Multimengen (#include <set>)

- Mengencontainer mit automatischer Sortierung der Elemente
- `set` - jedes Element kommt höchstens einmal vor
- `multiset` - Elemente können mehrfach enthalten sein (Bags)
- wegen der automatischen Sortierung muss für den Elementtyp der Operator `<` definiert sein ! Dieser legt auch die Gleichheitsrelation fest: zwei Elemente a und b sind gleich, wenn weder a<b noch b<a gilt !
- man kann bei der Instantiierung von Mengentemplates auch ein anderes Ordnungskriterium angeben:

```
namespace std {
     template < class T,
                class Compare = less<T>,
                class Allocator = allocator<T> >
     class set; /* dito multiset */          }
```

- `set <int, greater<int> >` absteigend_sortierte_Mengen

- `greater<T>` und `less<T>` sind sog. function objects (`operator()` ist definiert) `#include <functional>`

```
template <class Arg, class Result>
  struct unary_function {
    typedef Arg    argument_type;
    typedef Result result_type;
  };
template <class Arg1, class Arg2, class Result>
  struct binary_function {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
  };
```

Funktoren

```
template <class T> struct greater : public
  binary_function<T,T,bool> {
  bool operator()(const T& x, const T& y) const
  { return x > y; }
};
```

Aufruf	Operation
`negate<T>()(p)`	- p
`plus<T>()(p1, p2)`	p1 + p2
`minus<T>()(p1, p2)`	p1 - p2
`multiplies<T>()(p1, p2)` `[depricated] times<T>()`	p1 * p2
`divides<T>()(p1, p2)`	p1 / p2
`modulus<T>()(p1, p2)`	p1 % p2

Funktoren

Aufruf	Operation
`equal_to<T>()(p1, p2)`	p1 == p2
`not_equal_to<T>()(p1, p2)`	p1 != p2
`less<T>()(p1, p2)`	p1 < p2
`greater<T>()(p1, p2)`	p1 > p2
`less_equal<T>()(p1, p2)`	p1 <= p2
`greater_equal<T>()(p1, p2)`	p1 >= p2
`logical_not<T>()(p)`	! p
`logical_and<T>()(p1, p2)`	p1 && p2
`logical_or<T>()(p1, p2)`	p1 || p2

Funktoradaptoren (alt)

Aufruf	Operation
`bind1st(op, wert)`	op(wert, param)
`bind2nd(op, wert)`	op(param, wert)
`not1(op)`	!op(param)
`not2(op)`	!op(param1, param2)

Beispiel: alle geraden Zahlen aus einer Liste entfernen

```
liste.remove_if (not1(bind2nd(modulus<int>(), 2)));
```

Funktoradaptoren (neu)

`std::bind` provides support for partial function application, i.e. binding arguments to functions to produce new functions.

bind (C++11)	binds one or more arguments to a function object (function template)
is_bind_expression (C++11)	indicates that an object is `std::bind` expression or can be used as one (class template)
is_placeholder (C++11)	indicates that an object is a standard placeholder or can be used as one (class template)
Defined in namespace `std::placeholders`	
_1, _2, _3, _4, ... (C++11)	placeholders for the unbound arguments in a `std::bind` expression (constant)

`std::function` provides support for storing arbitrary function objects.

function (C++11)	wraps callable object of any type with specified function call signature (class template)
mem_fn (C++11)	creates a function object out of a pointer to a member (function template)
bad_function_call (C++11)	the exception thrown when invoking an empty `std::function` (class)

Reference wrappers allow reference arguments to be stored in copyable function objects:

reference_wrapper (C++11)	`CopyConstructible` and `CopyAssignable` reference wrapper (class template)
ref (C++11) **cref** (C++11)	creates a `std::reference_wrapper` with a type deduced from its argument (function template)

Funktoradaptoren (neu)

```
#include <random>
#include <iostream>
#include <functional>

void f(int n1, int n2, int n3, const int& n4, int n5) {
  std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
}

int g(int n1) {
    return n1;
}

struct Foo {
    void print_sum(int n1, int n2) {
        std::cout << n1+n2 << '\n';
    }
    int data = 10;
};
```

Funktoradaptoren (neu)

```
int main() {
    using namespace std::placeholders; //for _1, _2, _3...

    // demonstrates argument reordering and pass-by-reference
    int n = 7;
    auto f1 = std::bind(f, _2, _1, 42, std::cref(n), n);
    n = 10;
    f1(1, 2, 1001); // 1 is bound by _2, 2 is bound by _1, 1001 is unused

    // nested bind subexpressions share the placeholders
    auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
    f2(10, 11, 12);

    // common use case: binding a RNG with a distribution
    std::default_random_engine e;
    std::uniform_int_distribution<> d(0, 10);
    std::function<int()> rnd = std::bind(d, e);
    for(int n=0; n<10; ++n)
        std::cout << rnd() << ' ';
    std::cout << '\n';
```

Funktoradaptoren (neu)

```
    // bind to a member function
    Foo foo;
    auto f3 = std::bind(&Foo::print_sum, foo, 95, _1);
    f3(5);

    // bind to member data
    auto f4 = std::bind(&Foo::data, _1);
    std::cout << f4(foo) << '\n';
}
```

Output:

```
2 1 42 10 7
12 12 12 4 5
1 5 0 2 0 8 2 2 10 8
100
10
```

Mengen und - Multimengen (`#include <set>`)

- Implementierung typischerweise als balancierte Binärbäume (red-black-tree)
- sehr gutes Zeitverhalten beim Suchen, gutes beim Löschen und Einfügen an allen Stellen !
- die Sortierung hat zur Konsequenz, dass man Elemente nicht ändern kann, realisiert dadurch, dass alle Iteratoren Zugriffe auf `const`-Objekte bereitstellen (Wert ändern: alten Wert aufsuchen und löschen, neuen Wert einfügen)
- beim Einfügen einzelner Elemente unterscheiden sich `set` und `multiset` in ihren Rückgabewerten:

`multiset` Position des neuen (eingefügten) Elements

`set` gibt ein `pair<iterator,bool> p` zurück:
`p.second` gibt an, ob wirklich eingefügt wurde,
`p.first` ist die Position des eingefügten bzw. bereits vorhanden Elements

Ausdruck	Bedeutung
`m.count (e)`	Anzahl der Elemente mit Wert e `set`: 0..1; `multiset` 0..n
`m.find (e)`	liefert die Position des ersten Auftretens von e oder `end()`
`m.lower_bound (e)`	erste Position an der e eingefügt werden könnte (das erste Element >= e)
`m.upper_bound (e)`	letzte Position an der e eingefügt werden könnte (das erste Element > elem)
`m.equal_range (e)`	erste und letzte Position zum Einfügen gibt ein `pair<const_iterator, const_iterator>` zurück

Ausdruck	Bedeutung
`m.insert (e)`	fügt Element ein und liefert Position des neuen Elements, bzw. ob es geklappt hat
`m.insert (pos,e)`	fügt Element ein und liefert Position des neuen Elements, bzw. ob es geklappt hat (pos wird nur als Hinweis verwendet)
`m.insert (f, l)`	Elemente von [f, l) einfügen (void)
`m.erase (e)`	löscht alle Auftreten von e, return Anzahl der entfernten Elemente
`m.erase (pos)`	löscht Element bei pos
`m.erase (f, l)`	löscht Bereich [f, l) liefert Position des Folgeelements

Maps und - Multimaps (#include <map>)

- Mengencontainer für Schlüssel/Wert- Paare mit automatischer Sortierung anhand der Schlüssel (dictionaries)

```
typedef pair<const Key, T> value_type; // in [multi]map
```

- **map** jeder Schlüsselwert kommt höchstens einmal vor
- **multimap** Schlüsselwerte können mehrfach enthalten sein
- wegen der automatischen Sortierung muss für den Schlüsseltyp der Operator **<** definiert sein ! dieser legt auch die Gleichheitsrelation fest: zwei Schlüssel a und b sind gleich, wenn weder a<b noch b<a gilt !

- die Sortierung hat zur Konsequenz, dass man Schlüssel nicht ändern kann, realisiert dadurch, dass alle Schlüssel `const`-Objekte sind, der Wert zu einem Schlüssel kann geändert werden !

- man kann bei der Instantiierung von `map`-Templates auch ein anderes Ordnungskriterium angeben

```
namespace std {
  template <
    class Key,
    class T,
    class Compare = less<Key>,
    class Allocator = allocator<pair<const Key, T> >
  >
  class map; // dito multimap
}
```

Ausdruck	Bedeutung
`wie set (count ... equal_range)`	e ist ein Schlüsselwert (`Key`)
`wie set (insert ... erase)`	e ist vom Typ `[multi]map<Key,T>::value_type`
`m[key]` **(nur für map !)**	liefert eine Referenz für den Wert des Elements zu key, fügt das Element dabei ggf. neu in die `map` ein !!

– beim Einfügen einzelner Elemente unterscheiden sich `map` und `multimap` in ihren Rückgabewerten:

`multimap` Position des neuen (eingefügten) Elements

`map` gibt ein `pair<iterator,bool> p` zurück:
`p.second` gibt an, ob wirklich eingefügt wurde,
`p.first` ist die Position des eingefügten bzw. bereits vorhanden Elements

3. Generische Programmierung in C++

```
#include <iostream>
#include <string>
#include <map>

using namespace std ;

int main ( ) {
  string buf ;
  map < string , int > m ;
  while ( cin >> buf ) m [ buf ] ++ ;
  multimap < int , string > n ;
  for ( map < string , int > :: iterator p = m . begin ( ) ;
                p != m . end ( ) ; ++ p )
    n . insert ( multimap < int , string > :: value_type ( p -> second , p ->
    first ) ) ;
  for ( multimap < int , string > :: iterator p = n . begin ( ) ;
                p != n . end ( ) ; ++ p )
    cout << p -> first << "\t" << p -> second << endl ;
}
```

```
$ wc < wc.cpp
1       "\t"
1       <iostream>
…..
6       string
10      (
10      )
10      p
11      ;
```

Input - & Output – Iteratoren

```
#include <string>
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>
#include <sstream>

int main()
{
    std::istringstream s("I need your help!");

    std::vector<std::string> v( (std::istream_iterator<std::string>(s)) ,
                                std::istream_iterator<std::string>());
    std::copy(v.begin(), v.end(),
              std::ostream_iterator<std::string>(std::cout, "\n"));
}
```

```
I
need
your
help!
```

Scott Meyers „Effective STL" (Item 6:) Be alert for C++'s most vexing parse

3. Generische Programmierung in C++

	vector	deque	list	set	multiset	map	multimap
interne Datenstruktur	dynamisches Array	Menge von Arrays	doppelt verkettete Liste	Binärbaum	Binärbaum	Binärbaum	Binärbaum
Elemente-Art	Wert	Wert	Wert	Wert	Wert	Wertepaar	Wertepaar
Duplikate erlaubt	ja	ja	ja	nein	ja	nein (Schlüssel)	ja (Schlüssel)
wahlfreier Zugriff	ja	ja	nein	nein	nein	über Schlüssel	nein
Iterator-Kategorie	Random-Access	Random-Access	Bidirectional	Bidirectional (Wert konstant)	Bidirectional (Wert konstant)	Bidirectional (Schlüssel konstant)	Bidirectional (Schlüssel konstant)
Suchen/Finden von Elementen	langsam	langsam	sehr langsam	sehr schnell	sehr schnell	sehr schnell für Schlüssel	sehr schnell für Schlüssel
Einfügen/Löschen schnell	am Ende	am Anfang und am Ende	überall konstant	überall logarithmisch	überall logarithmisch	überall logarithmisch	überall logarithmisch
Verweise werden ungültig	bei Reallokierung	potenziell immer	nein	nein	nein	nein	nein
Speicher wird freigegeben	nie	manchmal	immer	immer	immer	immer	immer
Speicherreservierung möglich	ja	nein	-	-	-	-	-

Container-Adaptoren

neben den (primären) Containern gibt es einige sog. Container-Adaptoren, es handelt sich dabei um Anpassungen der Container für spezielle Anwendungen

Queues (`#include <queue>`)

FIFO-Warteschlangen (auch mittels `list` instantiierbar)

```
namespace std {
  template < class T,
             class Container = deque<T>
  >
  class queue;
}
```

Priority Queues (#include <queue>)

Warteschlangen mit Prioritäten (auch mittels `deque` instantiierbar)

```
namespace std {
  template < class T, class Container = vector<T>,
    class Compare = less<typename Container::value_type>
  >
  class priority_queue;
}
```

Stacks (#include <stack>)

Kellerspeicher (auch mittels `list` und `vector` instantiierbar)

```
namespace std {
   template < class T, class Container = deque<T> >
   class stack;
 }
```

Strings (#include <string>) (vgl. z.B. Josuttis, Kapitel 10, S.357 ff.)

```
namespace std {
  template < class charT,
     class traits = char_traits<charT>
     class allocator = allocator <charT> >
  class basic_string;
  // noch nicht auf Zeichentyp festgelegt
  typedef basic_string<char>    string;  // ASCII
  typedef basic_string<wchar_t> wstring; // Unicode
}
```

mit Einführung von `string` wurde auch die `iostream`-Bibliothek erheblich überarbeitet, um mit strings zusammenarbeiten zu können, ohne dass sich die Nutzerschnittstelle wesentlich verändert hat, ggf. ist wichtig

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

Numerische Klassen

Komplexe Zahlen (#include <complex>)

```
namespace std {
  template<class T> class complex;
  template<> class complex<float>;
  template<> class complex<double>;
  template<> class complex<long double>;
}
```

komplexe Zahlen mit allem "drum und dran" (Arithmetik und transzendente Funktionen) für float, double, long double bereits vordefiniert !

Valarrays (#include <valarray>)

```
namespace std {
  template<class T> class valarray;
}
```

Vektoren und Matrizen für numerische Operationen mit gutem Zeitverhalten (keine temporären Zwischenergebnisse) und kompakter Notation (z.B. Ausführung von Operationen auf allen Elementen, Bildung von sog. Slices, ...)

Bitsets (#include <bitset>)

```
namespace std {
  template< size_t bits > class bitset;
}
```

Bitvektoren (konstanter Länge)

vector<bool> - kein Container von bool's

```
std::vector<bool> v;

bool *pb = & v[0]; // nicht übersetzbar ☹
```

Scott Meyers „Effective STL" (Item 18:) Avoid using vector<bool>.

neue Container in C++11

Arrays (#include <array>)

```
namespace std {
  template<class T, size_t N> class array;
}
```

statische Felder mit (zur Compile-Zeit) fixierter Größe, effizient wie C-Felder, stattet diese quasi mit Memberfunktionen aus:

Iterators	Capacity	Element access
begin	size	operator[]
end	max_size	at
rbegin	empty	front
rend	**Modifiers**	back
cbegin	fill	data
cend	swap	
crbegin		
crend		

neue Container in C++11

Arrays (#include <array>)

Besonderheit: Tuple-Zugriff

```
template <size_t I, class T, size_t N> T& get (array<T,N>& arr) noexcept;
template <size_t I, class T, size_t N> T&& get (array<T,N>&& arr) noexcept;
template <size_t I, class T, size_t N> const T& get (const array<T,N>& arr)
                                                                  noexcept;
```

```
std::array<int,3> myarray = {10, 20, 30};
std::tuple<int,int,int> mytuple (10, 20, 30);

std::tuple_element<0,decltype(myarray)>::type myelement;  // int myelement

myelement = std::get<2>(myarray);
std::get<2>(myarray) = std::get<0>(myarray);
std::get<0>(myarray) = myelement;
```

am Horizont: neue Container in C++14

dynamische arrays (#include <dynarray>)

```
namespace std {
  template<class T> class dynarray;
}
```

zwischen `array` und `vector`: dynamische Festlegung der Größe zur Laufzeit bei Konstruktion, danach aber für dieses Objekt fixiert - flexibler als array aber genauso effizient

neue Container in C++11

forward lists (#include <forward_list>)

```
namespace std {
  template <class T,class Alloc=allocator<T>> class forward_list;
}
```

einfach (vorwärts) verkettete Listen: kein `push_back`, nur `push_front`

Iterators	
before_begin	zum Einfügen am Anfang
begin	
end	
cbefore_begin	zum Einfügen am Anfang
cbegin	
cend	

neue Container in C++11

unordered sets (#include <unordered_set>)

```
namespace std {
  template < class Key,                    // unordered_set::key_type
         class Hash = hash<Key>,           // unordered_set::hasher
         class Pred = equal_to<Key>,       // unordered_set::key_equal
         class Alloc = allocator<Key> // unordered_set::allocator_type
         > class unordered_set;
```

Mengen ohne Ordnung auf den Elementen, keine Duplikate:

Buckets	
bucket_count	Return number of buckets (public member function)
max_bucket_count	Return maximum number of buckets (public member function)
bucket_size	Return bucket size (public member type)
bucket	Locate element's bucket (public member function)
Hash policy	
load_factor	Return load factor (public member function)
max_load_factor	Get or set maximum load factor (public member function)
rehash	Set number of buckets (public member function)
reserve	Request a capacity change (public member function)

neue Container in C++11

unordered multisets (#include <unordered_set>)

```
namespace std {
  template < class Key,                    // unordered_multiset::key_type
         class Hash = hash<Key>,           // unordered_multiset::hasher
         class Pred = equal_to<Key>,       // unordered_multiset::key_equal
         class Alloc = allocator<Key> // unordered_multiset::allocator_type
         > class unordered_multiset;
```

Multimengen ohne Ordnung auf den Elementen, Mehrfacheinträge OK:

Buckets	
bucket_count	Return number of buckets (public member function)
max_bucket_count	Return maximum number of buckets (public member function)
bucket_size	Return bucket size (public member type)
bucket	Locate element's bucket (public member function)

Hash policy	
load_factor	Return load factor (public member function)
max_load_factor	Get or set maximum load factor (public member function)
rehash	Set number of buckets (public member function)
reserve	Request a capacity change (public member function)

neue Container in C++11

unordered maps (#include <unordered_map>)

```
namespace std {
  template < class Key,                    // unordered_map::key_type
         class T,                          // unordered_map::mapped_type
         class Hash = hash<Key>,           // unordered_map::hasher
         class Pred = equal_to<Key>,       // unordered_map::key_equal
         class Alloc = allocator<pair<const Key, T>>
                                           // unordered_map::allocator_type
         > class unordered_map;
```

Schlüssel/Wert-Mengen ohne Ordnung auf dem Schlüsseltyp, keine Duplikate (unordered_set mit pair als Element-Typ)

neue Container in C++11

unordered multimaps (#include <unordered_map>)

```
namespace std {
  template < class Key,                    // unordered_multimap::key_type
         class T,                          // unordered_multimap::mapped_type
         class Hash = hash<Key>,           // unordered_multimap::hasher
         class Pred = equal_to<Key>,       // unordered_multimap::key_equal
         class Alloc = allocator<pair<const Key, T>>
                                           // unordered_multimap::allocator_type
         > class unordered_multimap;
```

Schlüssel/Wert-Mengen ohne Ordnung auf dem Schlüsseltyp, Mehrfacheinträge OK (`unordered_set` mit `pair<const K, T>` als Element-Typ)

Allokatoren (#include <memory>)

- Separation der Speicherverwaltung für dynamische Objekte (Listen- und Baum-Knoten etc.) von den abstrakten Containern: Container enthalten als Typparameter eine Klasse, die die Speicherverwaltung komplett übernimmt
- Standardmäßig stellt jede Implementation einen default allocator in der Klasse `std::allocator` bereit, dieser verwaltet geeigntet Memory-Pools
- alternative Allokatoren (z.B. mit garbage collection, oder auf verschiedenen Speichermodellen ...) sind möglich und beeinflussen die eigentliche Funktionalität der Container in keiner Weise !!

rvalue references

non-const Referenzen kann man an *lvalues*, const Referenzen an *lvalues* oder *rvalues* binden, an non-const *rvalues* kann man keinerlei Referenzen binden. (Damit niemand den Wert von temporären Variablen ändert, die u.U. verschwinden, bevor der Wert benutzt werden kann)

```
void incr(int& a) { ++a; }
int i = 0; incr(i); // i becomes 1
incr(0); // error: 0 is not an lvalue
```

Aber was ist mit:

```
template<class T> swap(T& a, T& b) { // old style swap
  T tmp(a); // now we have two copies of a
  a = b;    // now we have two copies of b
  b = tmp;  // now we have two copies of tmp (aka a)
}
```

3. Generische Programmierung in C++

rvalue references

Wenn Kopieren für **T** teuer ist (z.B. string und vector), wird swap ebenfalls teuer (deshalb gibt es in std spezialisierte swap-Versionen)

Eigentlich sollte am **besten gar nichts** kopiert werden – die Werte von **a**, **b**, und **tmp** sollten nur **bewegt** werden

In C++11 kann man "move constructors" and "move assignments" definieren:

```
template<class T>
class vector { // ...
   vector(const vector&);             // copy constructor
   vector(vector&&);                  // move constructor
   vector& operator=(const vector&); // copy assignment
   vector& operator=(vector&&);       // move assignment
};
// note: move constructor and move assignment takes non-const &&
// they can, and usually do, write to their argument
```

&& bezeichnet eine "*rvalue reference*". Eine *rvalue reference* kann man an einen *rvalue* (aber nicht an einen *lvalue*) binden

3. Generische Programmierung in C++

rvalue references

```
X a;
X f();
X& r1 = a;      // bind r1 to a (an lvalue)
X& r2 = f();    // error: f() is an rvalue; can't bind
X&& rr1 = f();  // fine: bind rr1 to temporary
X&& rr2 = a;    // error: bind a is an lvalue
```

Ist dies das perfekte swap?

```
template<class T> void swap(T&& a, T&& b) { // perfect swap ?
  T tmp = a;
      a = b;
      b = tmp
}
```

NEIN, weil

1. dann keine lvalues übergeben werden könnten :-(und
2. alles, was einen Namen hat (auch wenn es mit && vereinbart wurde), bei der Benutzung kein rvalue ist :-(

3. Generische Programmierung in C++

rvalue references

```
•// perfect swap:
template<class T> void swap(T& a, T& b) {
  T tmp = std::move(a);   // could invalidate a
      a = std::move(b);   // could invalidate b
      b = std::move(tmp); // could invalidate tmp
}
```

move(x) bewegt **NICHTS**, sondern bedeutet "man kann **x** als *rvalue* verwenden".

Wer bewegt den inneren Zustand des Objektes? Das move assignment `T& T::operator=(T&&);` (wenn vorhanden), sonst wird mit dem copy assignment kopiert.

move() wiederum kann als template function mit einem rvalue reference Parameter implementiert werden (s.u.).

rvalue references können auch für *perfect forwarding* benutzt werden (s.u.).

In der C++11 standard library haben alle Container *move constructors* und *move assignments*. Operationen, die Elemente einfügen (wie **insert()** und **push_back()**) haben Versionen, die auf *rvalue references* arbeiten, es gibt die Platzierungsfunktion **emplace()**

rvalue references

Platzierung (*emplacement*):

```
class X  { public: X(U); ... };

std::vector<X> v; U u;

v.push_back(X(u)); // C++98: copy X, ggf. RVO (return value optimization)

v.push_back(X(u)); // C++11: move, falls X moveable, sonst copy (ggf. RVO)

// +
v.emplace_back(u); // no copy, no move: create in place
```

```
<vector>, <list>, <deque> ....
     void push_back(const T& x);
     void push_back(T&& x);
       template <class... Args> void emplace_back(Args&&... args);
     template <class... Args> iterator emplace(const_iterator position, Args&&... args);
     iterator insert(const_iterator position, const T& x);
     iterator insert(const_iterator position, T&& x);
```

rvalue references

```
#include <vector>
class X   {
    std::vector<double> data;
public:
    X():   data(100000) {}                  // lots of data

    X(X const& other):                      // copy constructor
        data(other.data) {}                 // duplicate all that data

    X(X&& other):  // move constructor
        data(std::move(other.data)) {}  // move the data: no copies

    X& operator=(X const& other) {          // copy-assignment
        data=other.data;                    // copy all the data
        return *this;   }

   X& operator=(X && other) {               // move-assignment
        data=std::move(other.data);         // move the data: no copies
        return *this;    }
};
```

rvalue references

```
// kanonische Implementation (nicht immer die schnellste)
class C   {
    Data data; // generic
public:
    friend void swap(C& a, C& b) { swap(a.data, b.data); }

    C(){}
    C(C const& other): data(other.data) {}
    •    C& operator=(C& other) {
    •        C tmp(other);
    •        swap(*this, tmp);
    •        return *this;
    •    }
    C(C&& other): C{} { swap(*this, other); }
    C& operator=(C && other) {
    •        swap(*this, other);
    •        return *this;
    •    }
};
```

rvalue references

```
C make_C()        // build a C with some data
{ return C(); }

int main()
{
    C c1;
    C c2(c1);                 // copy
    C c3(std::move(c1));      // move: c1 no longer has any data

    c1=c2;              // copy assign
    c1=make_C();        // return value is an rvalue, so move rather than copy
}
```

rvalue references

Perfect Forwarding Args&&... args ???

in Template-Code Parametertypen im Original verwenden:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg) // not perfect: extra copy
{
  return shared_ptr<T>(new T(arg));
}

// better:
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg) // no copy, but not for rvalues
{
  return shared_ptr<T>(new T(arg));
}
... factory<X>(makeX()); // error
... factory<X>(42);      // error, despite of X(int)
```

rvalue references

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg) // no copy, for lvalues
{
  return shared_ptr<T>(new T(arg));
}
//+
  template<typename T, typename Arg>
  shared_ptr<T> factory(const Arg& arg) // no copy, for rvalues
  {
    return shared_ptr<T>(new T(arg));
  }

... factory<X>(makeX()); // ok
... factory<X>(42);      // ok
```

leider auch nicht perfekt:

- 2^n Überlandungen bei n Parametern :-(
- keine move-Semantik möglich :-(

rvalue references

die Lösung: *rvalue* Referenzen selbst

bislang (C++98) war es nicht möglich Referenzen auf Referenzen zu setzen: ~~A& &~~

nun (C++11) gelten die folgenden *reference collapsing rules*:

- A& & ---> A&
- A& && ---> A&
- A&& & ---> A&
- A&& && ---> A&&

1. When called on an lvalue of type `Arg`, then `Arg` resolves to `Arg&` and hence, by the reference collapsing rules above, the argument type effectively becomes `Arg&`.
2. When called on an rvalue of type `Arg`, then `Arg` resolves to `Arg`, and hence the argument type becomes `Arg&&`.

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg) {
    return shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

rvalue references

`std::forward` ?

bei der Übergabe von Arg an den T-Konstruktor den ermittelten Typ bewahren:

dazu wird benötigt (implizit aus `<type_traits>`)

```
template< class T > struct remove_reference;      // general pattern
    •//template specializations:
    •template< class T > struct remove_reference      {typedef T type;};
template< class T > struct remove_reference<T&>  {typedef T type;};
template< class T > struct remove_reference<T&&> {typedef T type;};

und:

template<class S>
S&& forward(typename remove_reference<S>::type& a) noexcept
{
  return static_cast<S&&>(a);
}
```

rvalue references

alles zusammen:
wenn das Argument ein *lvalue* ist

```
X x; // alles was einen Namen hat ist lvalue
factory<A>(x); // Arg ---> X&
```

einsetzen ergibt:

```
shared_ptr<A> factory(X& && arg)
{
  return shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& && forward(remove_reference<X&>::type& a) noexcept
{
  return static_cast<X& &&>(a);
}
```

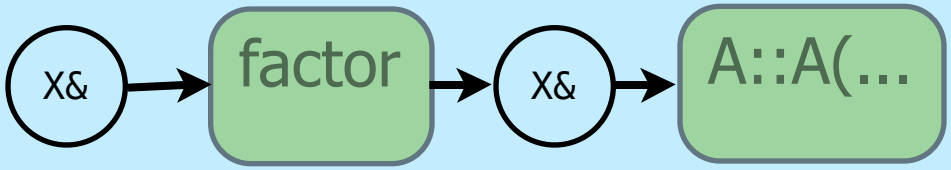
rvalue references

wenn das Argument ein *lvalue* ist

```
X x; // alles was einen Namen hat ist lvalue
factory<A>(x); // Arg ---> X&
```

einsetzen ergibt:

```
shared_ptr<A> factory(X& arg)
{
  return shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& forward(X& a) noexcept
{
  return static_cast<X&>(a);
}
```



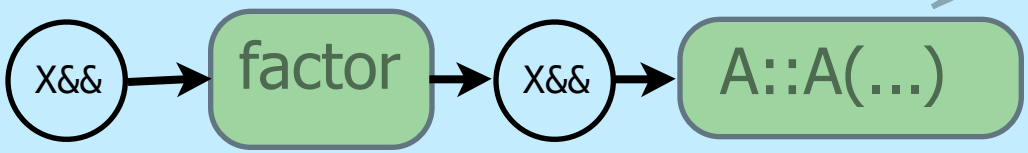
rvalue references

wenn das Argument ein *rvalue* ist

```
X foo(); // rvalue
factory<A>(foo()); // Arg ---> X
```

einsetzen ergibt:

```
shared_ptr<A> factory(X&& arg)
{
  return shared_ptr<A>(new A(std::forward<X>(arg)));
}
X&& forward(X& a) noexcept
{
  return static_cast<X&&>(a);
}
```



moving into A !!

rvalue references

Was ist mit mehr als einem Parameter?
auch parameter packs (s. variadic templates) können als rvalue Referenzen übergeben werden

Beispiel:

```
template< class... Types >
tuple<VTypes...> make_tuple( Types&&... args ); // Vtypes: „Value types“
```

Creates a tuple object, deducing the target type from the types of arguments. The deduced types are std::decay<Ti>::type (transformed as if passed to a function by value) unless application of std::decay results in std::reference_wrapper<X> for some type X, in which case the deduced type is is X&.

```
template< class T >  // more type magic
struct decay {
    typedef typename std::remove_reference<T>::type U;
    typedef typename std::conditional<
        /*if*/ std::is_array<U>::value,
        /*then*/ typename std::remove_extent<U>::type*,
        /*else*/ typename std::conditional<
            /*if*/ std::is_function<U>::value,
            /*then*/ typename std::add_pointer<U>::type,
            /*else*/ typename std::remove_cv<U>::type
    >::type>::type type;
};
```

```
template<bool B, class T, class F>
struct conditional { typedef T type; };

template<class T, class F>
struct conditional<false, T, F> { typedef F
type; };
```

rvalue references

Damit funktioniert alles „wie erwartet“:

```
#include <iostream>
#include <tuple>
#include <functional>

int main()
{
    auto t1 = std::make_tuple(10, "Test", 3.14);
    std::cout << "The value of t1 is "
              << "(" << std::get<0>(t1) << ", " << std::get<1>(t1)
              << ", " << std::get<2>(t1) << ")\n";

    int n = 1;
    auto t2 = std::make_tuple(std::ref(n), n); //std::reference_wrapper
    n = 7;
    std::cout << "The value of t2 is "
              << "(" << std::get<0>(t2) << ", " << std::get<1>(t2) << ")\n";
}
```

```
The value of t1 is (10, Test, 3.14)
The value of t2 is (7, 1)
```

rvalue references

was macht `std::move` nun wirklich?

The `move` function really does very little work. All `move` does is accept either an lvalue or rvalue argument, and return it as an rvalue *without* triggering a copy construction:

```
template<class T>
typename remove_reference<T>::type&&
std::move(T&& a) noexcept
{
  typedef typename remove_reference<T>::type&& RvalRef;
  return static_cast<RvalRef>(a);
}
```

rvalue references

```
// stealing from named objects:
class X {
    char* state;
public:
    X(const char* s): state(strdup(s)) {}
    X(X&& src) {  state = src.state;
                  src.state = nullptr; }
    X(const X& src): state(strdup(src.state)) {}
    friend std::ostream& operator<<(std::ostream& out, X& x) {
        if (x.state == nullptr)
            return out<<"nullptr"<<std::endl;
        return out<<x.state<<std::endl;
    }
    ~X() { free(state); }
};

int main (){ X x1("eins"); X x2(x1); X x3(std::move(x1));
    •            std::cout<<x1<<x2<<x3;
    •}
```

```
nullptr
eins
eins
```

auto_ptr (#include <memory>) deprecated !!!

```
namespace std {
  template<class X> class auto_ptr {
    template <class Y> struct auto_ptr_ref {};
  public:
    typedef X element_type;
    explicit auto_ptr(X* p =0) throw();
    auto_ptr(auto_ptr&) throw();
    template<class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw();
    template<class Y>
    auto_ptr& operator=(auto_ptr<Y>&) throw();
   ~auto_ptr() throw();
```

do not use it anymore

3. Generische Programmierung in C++

http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/

GotW #89 Solution: Smart Pointers

by Herb Sutter

Problem

JG Question

1. When should you use shared_ptr vs. unique_ptr? List as many considerations as you can.

Guru Question

2. Why should you almost always use make_shared to create an object to be owned by shared_ptrs? Explain.

3. Why should you almost always use make_unique to create an object to be initially owned by a unique_ptr? Explain.

4. What's the deal with auto_ptr?

http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/

GotW #89 Solution: Smart Pointers

by Herb Sutter

4. What's the deal with auto_ptr?

auto_ptr is most charitably characterized as a valiant attempt to create a unique_ptr before C++ had move semantics. auto_ptr is now deprecated, and should not be used in new code.

If you have auto_ptr in an existing code base, when you get a chance try doing a global search-and-replace of auto_ptr to unique_ptr; the vast majority of uses will work the same, and it might expose (as a compile-time error) or fix (silently) a bug or two you didn't know you had.

unique (owning) pointers #include <memory>

```
namespace std {
     template <class T, class D = default_delete<T>> class unique_ptr;
     template <class T, class D> class unique_ptr<T[],D>;
                                                    // array specialization
   }
```

smarte Zeiger mit der (einzigen) Zuständigkeit für den aufbewahrten Zeiger
nicht kopier-, aber verschiebbar, in Containern erfassbar!
beim Verschieben wechselt die Zuständigkeit an den Empfänger
der letzte der den Verweis auf das Objekt hat, besitzt es
nur eine Klassenhülle um den Zeiger (ohne stored deleter)
ansonsten zwei Zeiger: Objekt + Deleter
T& `operator*() const`, T* `operator->() const` [und `T* get() const`] sind definiert: **Benutzung wie *raw pointer***
nie wieder `delete` rufen!

unique (owning) pointers #include <memory>

nie wieder new rufen mit Hilfe von make_unique
(in C++11 „vergessen" in C++14 dabei :-)

```
/* stephantl_make_unique.h
 * Based on code by Stephan T. Lavavej at http://channel9.msdn.com/Series/
 * C9-Lectures-Stephan-T-Lavavej-Core-C-/STLCCSeries6
 */
#ifndef STEPHANTL_MAKE_UNIQUE_H_
#define STEPHANTL_MAKE_UNIQUE_H_

#include <memory>
#include <utility>
#include <type_traits>

namespace fut_stl {
  namespace impl_fut_stl {
    template<typename T, typename ... Args>
    std::unique_ptr<T> make_unique_helper(std::false_type, Args&&... args){
            return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
    }
    // ... next slide
```

3. Generische Programmierung in C++

```
template<typename T, typename ... Args>
    std::unique_ptr<T> make_unique_helper(std::true_type, Args&&... args) {
      static_assert(std::extent<T>::value == 0,
        "make_unique<T[N]>() is forbidden, please use make_unique<T[]>(),");
            typedef typename std::remove_extent<T>::type U;
            return std::unique_ptr<T>(new U[sizeof...(Args)]
                                      {std::forward<Args>(args)...});
        }
    }

    template<typename T, typename ... Args>
    std::unique_ptr<T> make_unique(Args&&... args) {
        return impl_fut_stl::make_unique_helper<T>(
            std::is_array<T>(),std::forward<Args>(args)... );
    }
}
#endif

// ohne felder reicht auch:
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

3. Generische Programmierung in C++

```
template<typename T, typename ... Args>
    std::unique_ptr<T> make_unique_helper(std::true_t                    rgs) {
      static_assert(std::extent<T>::value == 0
        "make_unique<T[N]>() is forbidde                    (),");
            typedef typename std
            return std::u

        }
    }
}
#endi

// ohn
templa                typename... Args>
std::u                <T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

really needs several C++11 new features, which all fit nicely together:

- static assertions
- type traits
- variadic templates
- move semantic
- perfect forwarding

shared (ref-counted) pointers #include <memory>

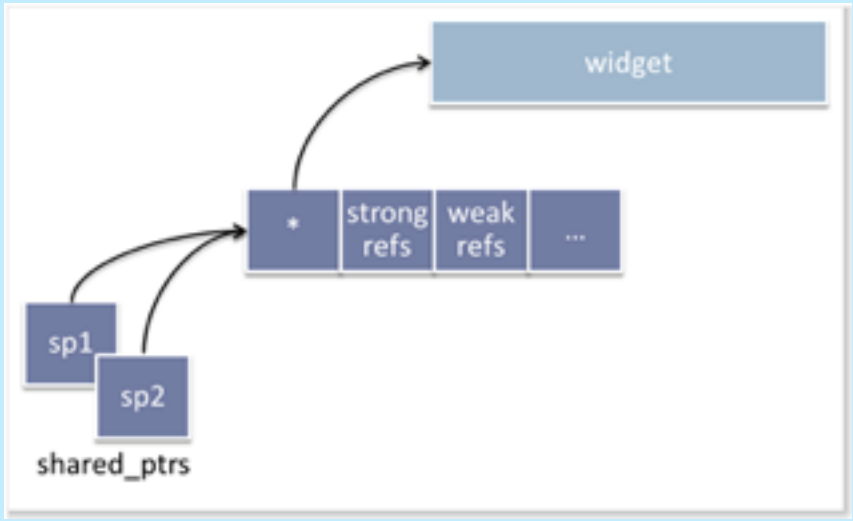
```
namespace std {
  template <class T> class shared_ptr;
  template <class T> class weak_ptr;
   }
```

smarte Zeiger mit geteiltem Besitz am aufbewahrten Zeiger
T& `operator*() const`, T* `operator->() const` [und `T* get() const`] sind definiert: **Benutzung wie *raw pointer***
der letzte räumt auf, aka *managed pointers*
kopier- und verschiebbar, in Containern erfassbar, *thread-safe !!!*
beim Kopieren wird ein interner Referenzzähler erhöht
verschwindet ein shared pointer wird (zunächst) nur der Referenzzähler reduziert, geht der auf 0 -> implizites delete
mehr als eine Klassenhülle um den Zeiger (teurer als `unique_ptr`)
Deleter kann hinterlegt werden (ist aber **kein** Template-Parameter)
nie wieder `delete` rufen! nie wieder `new` rufen mit Hilfe von `std::make_shared` ist sogar effizienter als Konstruktoraufruf

shared (ref-counted) pointers #include <memory>

http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/

```
// Example 2(a): separate allocation
auto sp1 =
  shared_ptr<widget>{ new widget{} };
auto sp2 = sp1;
```




```
// Example 2(b): single allocation
auto sp1 = make_shared<widget>();

auto sp2 = sp1;
```

reduces allocation overhead !!!
improves locality !!!

weak pointers #include <memory>

kann Objekte mit shared pointers teilen, ohne sie zu besitzen

zum Aufbrechen von Zyklen (die sonst nie freigegeben würden), kein direkter Zugang zum Objekt - Zugang nur durch Umwandlung in einen **shared_ptr** möglich

Problem: hat man eine Referenz, kann das Objekt schon weg sein :-(

some shared_ptr<T>

Lösungen:

```
std::weak_ptr<T> weak = ...;
// later:
try {
  std::shared_ptr<T> shrd {weak};
  // object alive ...
} catch (std::bad_weak_ptr)
{ // object not alive }
```

```
std::weak_ptr<T> weak = ...;
// later:
if (weak.expired()) // object not alive
else // object maybe alive
```

```
std::weak_ptr<T> weak = ...;
// later:
std::shared_ptr<T> shrd = weak.lock();
if (shrd) // object alive
else      // object not alive
```

ISO C++11: shared_ptr<T> weak_ptr<T> **lock** () const noexcept;
Returns: expired() ? shared_ptr<T>() : shared_ptr<T>(*this)

lambdas

neu in C++11

Generische Algorithmen brauchen oft Hilfsklassen für *functional objects*:

```
class between {
  double low, high;
public:
  between(double l, double u) : low(l), high(u) { }
  bool operator()(const employee& e) {
    return e.salary() >= low && e.salary() < high;
  }
};
double min_salary;

std::find_if(begin(employees), end(employees),
             between(min_salary, 1.1*min_salary));
```

lambdas

.... sind kleine Funktionen an Ort und Stelle ihrer Verwendung:

```
double min_salary = ....
double u_limit = 1.1 * min_salary;

std::find_if(begin(employees), end(employees),
 [&](const employee& e) {
      return e.salary()>=min_salary && e.salary()<u_limit;
   }
);
```

[&] ist eine sog. *capture list*, sie gibt an, dass alle lokalen Variablen per Referenz übergeben werden, Übergabe alle per Wert: **[=]** Einzelne Variablen können explizit benannt werden **[&x, y]**, Leere capture list **[]**, Der Rückgabetyp wird häufig aus dem *return statement* abgeleitet. Ohne return: **void**. Ansonsten ist (nur!) ***suffix return type syntax*** möglich.

3. Generische Programmierung in C++

lambdas

Lambdas können auch benannt und kombiniert werden. Als ‚rufbare' Objekte sind sie mit **`std::function`** 's kompatibel:

```
#include <functional>
#include <iostream>
std::function<int(int)> twice() {
 return [] (int n) { return 2*n; }; // lambda als Wert!
}
std::function<int(int)> addConst(int c) {
 return [c] (int n) { return n+c; };
}
template <typename OP>
int rechne_aus(int n, OP op) {
  return op(n);
}
```

something callable: int -> int

ohne capture c nicht definiert!

3. Generische Programmierung in C++

lambdas

```
•std::function<int(int)>
• operator* (std::function<int(int)> f, std::function<int(int)> g) {
•    return [f, g] (int n) { return f(g(n)); };
•}
•int main() {
•    auto add1 = addConst(1);
•    auto add4 = addConst(4);
•    auto dbl = twice();
•    int n = 6;
•    n = rechne_aus(n, add4);
•    n = rechne_aus(n, dbl);
•    n = rechne_aus(n, add1);
•    n = rechne_aus(n, dbl);
•    std::cout << n << std::endl;
•    auto h = dbl * add1 * dbl * add4;
•    std::cout << h(6) << std::endl;
•}
```

3. Generische Programmierung in C++

lambdas

Bei der capture list kann man zwischen ***capture by value*** und ***capture by reference*** wählen:

[x] --- x by value (Kopie) (x darf lokal NICHT verändert werden)

[&x] --- x by reference

Ersetzungsschema mit `operator () const` !

[=] capture all by value

(mit Vorsicht verwenden, u.U. muss viel Kontext kopiert werden)

[&] capture all by ref (dito)

Mischen ist möglich, aber nach **=** oder **&** ist nur noch die andere Variante aufzählbar, keine Variable darf doppelt vorkommen, fehlerhaft sind z.B.

```
[=, &x, y]
[&, x, y, &z]
[&, x, y, x]
```

lambdas

Lambdas in Memberfunktionen von Klassen haben keinen Zugang zu den Memberdaten. Sie können diesen aber durch explizites ***capture this*** erlangen:

```
struct X {
    int some_data;
    void foo(std::vector<int>& vec)
    {
        std::for_each(begin(vec), end(vec),
            [this](int i&) { i += some_data;} );
    }
};
```

Threads

Herb Sutter ("The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," Herb Sutter, Dr. Dobb's Journal, 30(3), March 2005.):

Applications will increasingly need to be concurrent if they want to fully exploit continuing exponential CPU throughput gains.

Efficiency and performance optimization will get more, not less, important.

Threads

BUT:

The 1998 C++ Standard doesn't acknowledge the existence of threads, and the operational effects of the various language elements are written in terms of a sequential abstract machine. Not only that, but the memory model isn't formally defined, so you can't write multithreaded applications without compiler-specific extensions to the 1998 C++ Standard. [Anthony Williams: C++ Concurrency in Action - Practical Multithreading, February, 2012,ISBN: 9781933988771]

C++11 behebt all diese Defizite:

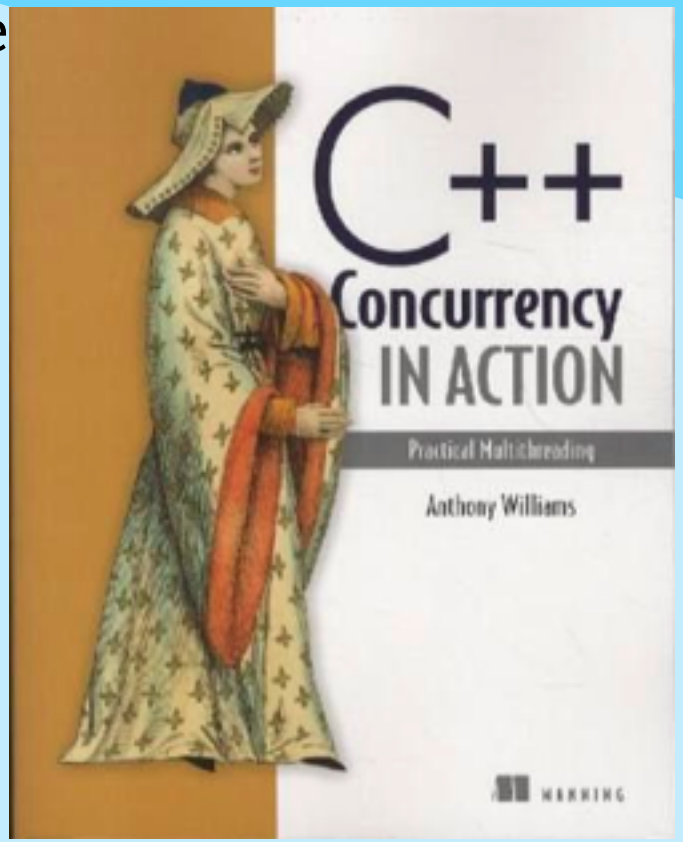
- `std::thread` (<thread>)
- Synchronisation und Ausschluss
- verschiedene Konsistenzmodelle des Speicherzugriffs
- Atomare Typen

4. Parallele Programmierung in C++

Threads

erst wenige Bücher :-(

ein sehr gute



folgende Codebeispiele
z.T. daraus übernommen

Threads

- nebenläufige (und damit potentiell parallele) Aktionen zum initialen `main`-Thread sind immer an ein `std::thread`-Objekt bei dessen Konstruktion zu knüpfen
- diese Aktion kann einen beliebige callable entity sein:
  - Funktion, Funktionszeiger
  - funktionale Objekte (`operator ()`)
  - `std::function`-Objekte (`<functional>`)
  - Memberfunktionen und Zeiger auf Memeberfunktionen
  - Lambdas (!)

Threads

- der Thread startet SOFORT (kein `run`() o. ä.) als (potentiell) parallele Aktion
- wenn das `std::thread`-Objekt verschwindet (nicht vor dem Ende des Threads selbst), muss entschieden sein, ob der startende Thread (initial `main`) auf die Beendigung des neuen Threads warten soll (`thread::join()`) oder von diesem entkoppelt weiterläuft (`thread::detach()`), andernfalls wird `std::terminate()` gerufen !

Threads

```
#include <iostream>
#include <thread>

void hello() {
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello); // very bad effort/benefit ratio

    t.join();
}
```

erreicht der `main`-Thread sein Ende, ist das Programm beendet (ggf. Abbruch noch laufender abgekoppelter Threads)

Threads

```
#include <iostream>
#include <thread>

class background_task {
public:
    void operator()() const {
        do_something();
        do_something_else();
    }
};

int main(){
    background_task f;
      std::thread t(f); // f will be copied into the storage of the new thread
    t.join();
}
```

```
std::thread((backgound_task())); OR
std::thread{backgound_task()};

// beware: std::thread(backgound_task()); ???
```

Threads

```
#include <iostream>
#include <thread>

int main(){
    std::thread t([]{    // Lambda: „inline action“
                         do_something();
                         do_something_else();
                     });
    t.join();
}
```

Threads

bei abgekoppelten Threads (`detach()`) können u.U. leicht (fehlerhaft) undefinierte Operationen auftreten:

```
struct func {
  int& i;
  func(int& i_):i(i_){}
  void operator()() {
    for(unsigned j=0;j<1000000;++j) {
      do_something(i);
    }
  }
};

void oops() {
  int some_local_state=0;
  func my_func(some_local_state);
  std::thread my_thread(my_func);
  my_thread.detach();
}
```

may(will)be a dangling reference

Threads

kann man Argumente an die Thread-Aktion übergeben?

JA: variadic templates machen das möglich:

```
void f(int i,std::string const& s);

std::thread t(f,3,"hello"); // std::string("hello")
```

Vorsicht bei impliziten Umwandlungen der Argumente: die Quelle muss noch garantiert existieren, wenn die Umwandlung (irgendwann vor dem Start des Threads) stattfindet

```
void f(int i,std::string const& s);
void oops(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3,buffer); // maybe undefined !
  t.detach();
}
```

Threads

kann man Argumente an die Thread-Aktion übergeben?

JA: variadic templates machen das möglich:

```
void f(int i,std::string const& s);

std::thread t(f,3,"hello"); // std::string("hello")
```

Vorsicht bei impliziten Umwandlungen der Argumente: die Quelle muss noch garantiert existieren, wenn die Umwandlung (irgendwann vor dem Start des Threads) stattfindet

```
void f(int i,std::string const& s);
void ok(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3,std::string(buffer)); // always defined !
  t.detach();
}
```

Threads

Argumente werden IMMER kopiert, selbst wenn die Thread-Funktion eigentlich eine Referenz erwartet:

Was, wenn der Thread am Original eine Änderung bewirken soll?

```
void update_data_for_widget(widget_id w, widget_data& data);

void oops_again(widget_id w) {
  widget_data data;
  std::thread t(update_data_for_widget, w, data);
  display_status();
  t.join();
  process_widget_data(data); // data unchanged !
}
```

Threads

Was, wenn der Thread am Original eine Änderung bewirken soll? Die Referenz explizit bereitstellen:

```
void update_data_for_widget(widget_id w, widget_data& data);

void ok_again(widget_id w) {
  widget_data data;
  std::thread t(update_data_for_widget, w, std::ref(data));
  display_status();
  t.join();
  process_widget_data(data); // data changed !
}
```

Threads

Memberfunktionen brauchen zum Aufruf (Memberfunktions-)Zeiger UND Objekt:

```
class X {
public:
    void do_lengthy_work(int i);
};

X x;

// call x.do_lengthy_work(); in a thread:

std::thread t( &X::do_lengthy_work, &x, 42 );
```

- `//` ... the function
- `//` ..... this
- `//` ..... more args

no hack !!!

Threads

`unique/shared_ptr` können an threads übergeben werden:

```
void process_big_object(std::unique_ptr<big_object>);

  std::thread t1(make_unique<big_object>(args)); // no leak


std::unique_ptr<big_object> b(new big_object(args));
b->prepare_data(42); // ... has a name - is an lvalue

std::thread t2(process_big_object,std::move(b));
```

4. Parallele Programmierung in C++

Threads

Threads sind identifizierbar:

- `// #include <thread>`
- `// ask a thread for its ID:`
- `std::thread::id std::thread::get_id();`

- `// ask this thread (the current) for its ID:`

```
std::thread::id std::this_thread::get_id();
```

`std::thread::id` ist vergleichbar und geordnet

`==` gleicher Thread oder beide ohne Thread

`<` ohne explizite Semantik

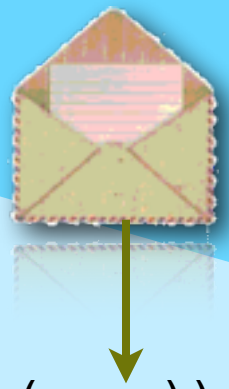
`std::hash<std::thread::id>` ist vordefiniert

`operator<<` definiert aber impl. defined

Threads std::promise und std::future

Wenn ein Thread Ergebnisse liefern soll: bisher nur void-Funktionen als Thread :-(

```
#include <future>

std::promise<std::string> prms;
// not copyable
```



```
std::thread th(&thFun, std::move(prms));
// now the thread owns it exclusively
```

Threads std::promise und std::future

Wenn ein Thread Ergebnisse liefern soll: bisher nur void-Funktionen als Thread :-(

```
#include <future>

std::promise<std::string> prms;

// before moving:
std::future<std::string> ftr = prms.get_future();

std::thread th(&thFun, std::move(prms));
```

```
std::string str = ftr.get();
// blocks until result ready
std::cout << str << std::endl;
th.join(); // still needed
```

```
th.join(); // still needed
std::string str = ftr.get();
// nonblocking
std::cout << str << std::endl;
```

Threads std::promise und std::future

Wenn ein Thread Ergebnisse liefern soll: bisher nur void-Funktionen als Thread :-(

```
void thFun(std::promise<std::string>&& prms) {
    std::string str("hello from future!");
    prms.set_value(str);
}
```

4. Parallele Programmierung in C++

Threads

Welche Parallelität kann maximal erreicht werden ?

```
// #include <thread>

[static] unsigned thread::hardware_concurrency();
```

Returns: The number of hardware thread contexts. [*Note:* This value should only be considered to be a hint. —*end note*] If this value is not computable or well defined an implementation should return 0.
Throws: Nothing.

Offenbar eine knappe Ressource:

- feste Zuordnung, überhaupt Zuordnung durch Nutzer besser vermeiden
- Parallelitätsbegriff ohne explizite Benutzung von Threads: `std::async`
- Thread-Pools (keine direkte Unterstützung durch C++11)

Threads doing things in parallel (or not): std::async

```
#include <iostream>
#include <string>
#include <thread>
#include <future>

std::string fun() { // normal function or the like
    return std::string ("hello from future!");
}

int main(int argc, const char * argv[])
{
    auto fut = std::async(&fun);     //  asyncronously: starts now or later
    std::string str = fut.get();     //  but will be finished here
    std::cout << str << std::endl;

    return 0;
}
```

sets up a suitable promise and returns its future

```
template< class Function, class... Args>
std::future<typename std::result_of<Function(Args...)>::type>
   async( Function&& f, Args&&... args );
```

Threads if parallel work can fail

```
#include <iostream>
#include <string>
#include <thread>
#include <future>
#include <exception>

std::string fun() {
    throw std::runtime_error("exception from future");
    return std::string ("hello from future!");
}

int main(int argc, const char * argv[])
{
    auto fut = std::async(&fun);     //  asyncronously: starts now or later
    try {
        std::string str = fut.get(); //  but will be finished here
        std::cout << str << std::endl;
    } catch(std::exception& ex) {
        std::cout << ex.what() << std::endl;

    }
    return 0;
}
```

Threads choosing the launch policy: std::launch::async vs. std::launch::deferred

```
#include <iostream>
#include <string>
#include <thread>
#include <future>

std::string fun() { // normal function or the like
    return std::string ("hello from future!");
}

int main(int argc, const char * argv[])
{
    auto f1 = std::async(std::launch::async, &fun); // starts now in a thread
      std::string str = f1.get();     // blocks until thread implicitly joined
      std::cout << str << std::endl;

      auto f2 = std::async(std::launch::deferred, &fun); // later in main
  thread
      std::string str = f2.get();      // call fun now
      std::cout << str << std::endl;

    return 0;
}
```

```
template < class Function, class... Args >
std::future<typename std::result_of<Function(Args...)>::type>
  async( std::launch policy, Function&& f, Args&&... args );
```

async (f, args...) == async(std::launch::async | std::launch::deferred, f, args...) let the system choose

Threads Wie schreibt man korrekte parallele Programme?

wenn parallele Threads auf den gleichen Daten operieren sind die Ergebnisse undefiniert :-(

selbst Annahmern über kausale Abläufe stimmen nicht mehr:

```
int x, r1;
int y, r2;
```

```
void t1f(){
    x = 1;
    r1 = y;
}
```

```
void t2f(){
    y = 1;
    r2 = x;
}
```

Ist es möglich, dass nach Abarbeitung beider Threads gilt ?

```
r1 == 0 && r2 == 0
```

LEIDER JA!

Threads Wie schreibt man korrekte parallele Programme?

Siehe auch Herb Sutter:
channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2
und http://sdrv.ms/NxDB6u

The Truth

- Q: Does your computer execute the program you wrote?

Threads Wie schreibt man korrekte parallele Programme?

The Truth

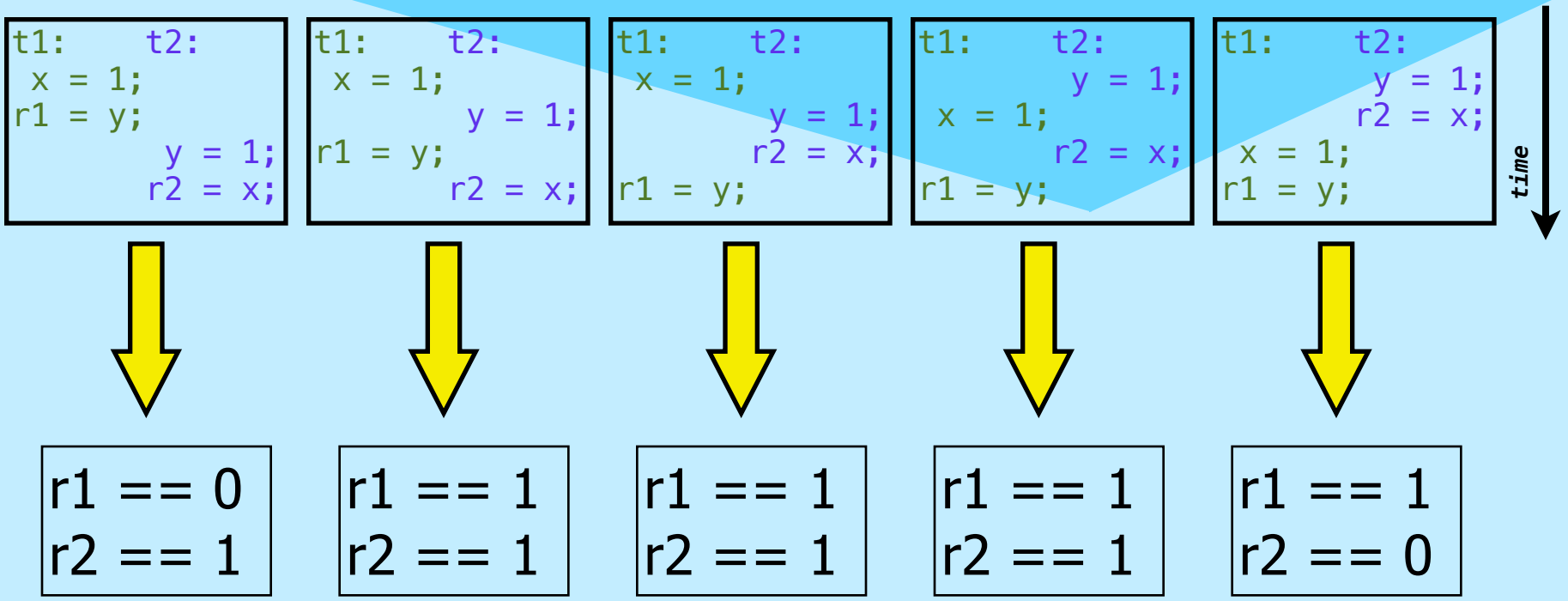
- Compiler/processor/cache says:

> "No, it's much better to **execute a different program**.
> Hey, don't complain. It's for your own good. You really wouldn't want to execute that *dreck* you actually wrote."

Leslie Lamport prägte (1979) den Begriff `Sequential Consistency` als die Eigenschaft bei der *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."*.

Threads Wie schreibt man korrekte parallele Programme?

anhand dieser Eigenschaft könnte man beweisbare Aussagen über das Verhalten von parallelen Programmen ableiten:



parallele Programme sind (heute) leider NICHT sequentiell konsistent :-(

Threads

leider kann aber auch folgendes passieren

```
t1:                              t2:
// x‘ lokaler cache zu x
x‘ = 1;
// in write queue; x‘ -> x
r1 = y;
...                               y = 1;
...                              r2 = x;
...
// write sync‘ed
```

time

r1 == 0
r2 == 0

nur wenn Einschränkungen der maximalen Entkopplung vorgenommen werden, gilt sequentielle Konsitstenz !!!

Threads

erforderlich dazu sind Synchronisationsmittel z.B. **`std::mutex`**

Sperren der gleichzeitigen Ausführung gewisser Programmabschnitte

```
#include <mutex>
...
std::mutex mutti;

// alle Kinder beschäftigen sich selbst
mutti.lock(); // frage an mutti: gibst du mir (exklusiv) ein
// Spielzeug? wenn grade nicht verfügbar: hinten anstellen,
// sonst:
//    ein Kind spielt damit unter muttis aufsicht :-)
mutti.unlock(); // der nächste kann danach fragen
  // alle Kinder beschäftigen sich selbst
```

Threads

wenn während der exklusiven Phase ein Ausnahme auftritt?

1. Sie muss im Thread selbst behandelt werden, ansonsten droht `terminate`.
2. In der Behandlung muss der Mutex wieder freigegeben werden.

```
std::mutex m;
try {
   m.lock();
   do_exclusive_work(); // may fail
   m.unlock(); // not reached if fails
}
catch(...) {
   // handle it and:
   m.unlock(); // ugly and does not scale
}
```

Threads

die Lösung: RAII - ein Mutex-Lock IST eine Ressource

```
std::mutex m;
try {
   std::lock_guard<std::mutex> lock(m); // calls m.lock()
   do_exclusive_work(); // may fail
}
catch(...) {
   // handle exception
}

// lock_guard - Konstruktor ruft m.lock();
// lock_guard - Destruktor ruft m.unlock();
```

alle anderen Threads sind quasi ausgesperrt und können nicht parallel weiterarbeiten, wenn sie die gleiche Ressource benötigen: diese Phase so kurz wie möglich halten, Deadlocks vermeiden (z.B. Anforderung von 2 Mutexes ....)

Threads

ein klassischer Fall von Deadlock:

```
#include all needed
std::mutex forks[5];

void philo(int i) {
    for (int n=0; n<10;++n) {
        forks[   i   ].lock();
        forks[(i+1)%5].lock();
        forks[(i+1)%5].unlock();
        forks[   i   ].unlock();
        std::this_thread::sleep_for(some time);
    }
}
std::vector<std::thread> philos;

int main(int argc, const char * argv[]) {
    for (int i=0; i<5; ++i) {
        std::thread th(philo, i);
        philos.push_back(std::move(th));
    }
    std::for_each(begin(philos), end(philos),[](std::thread& t) {t.join();});
    return 0;
}
```

Threads

mehr Freiheitsgrade hat **`std::unique_lock`**

ein Hüllobjekt, welches einen Mutex besitzen kann mit verzögertem, zeitlich begrenztem und rekursivem Blockieren

(constructor)	constructs a `unique_lock`, optionally locking the supplied mutex (public member function)
(destructor)	unlocks the associated mutex, if owned (public member function)
`operator=`	unlocks the mutex, if owned, and acquires ownership of another (public member function)
Locking	
`lock`	locks the associated mutex (public member function)
`try_lock`	tries to lock the associated mutex, returns if the mutex is not available (public member function)
`try_lock_for`	attempts to lock the associated `TimedLockable` mutex, returns if the mutex has been unavailable for the specified time duration (public member function)
`try_lock_until`	tries to lock the associated `TimedLockable` mutex, returns if the mutex has been unavailable until specified time point has been reached (public member function)
`unlock`	unlocks the associated mutex (public member function)

Threads

std::unique_lock

Observers	
mutex	returns a pointer to the associated mutex (public member function)
owns_lock	tests whether the lock owns its associated mutex (public member function)
operator bool	tests whether the lock owns its associated mutex (public member function)

```
#include <mutex>
#include <thread>
#include <chrono>

struct Box {
    explicit Box(int num) : num_things{num} {}
    int num_things;
    std::mutex m;
};

void transfer(Box &from, Box &to, int num) {
    // don't actually take the locks yet
    std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
    std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);
    // lock both unique_locks without deadlock
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    lock1.unlock();
    lock2.unlock();
}
```

Threads

std::unique_lock

```
int main()
{
    Box acc1(100);
    Box acc2(50);

    std::thread t1(transfer, std::ref(acc1), std::ref(acc2), 10);
    std::thread t2(transfer, std::ref(acc2), std::ref(acc1), 5);

    t1.join();
    t2.join();
}
```

spielt bei **condition_variable**'s eine wichtige Rolle

Threads

`std::condition_variable`

Warten auf eine Bedingung, die ein anderer Thread beeinflusst
z.B. Producer/Consumer

die `condition_variable` verkörpert die Verfügbarkeit der geteilten Daten der Anwendung (ist selbst natürlich auch geteilt)

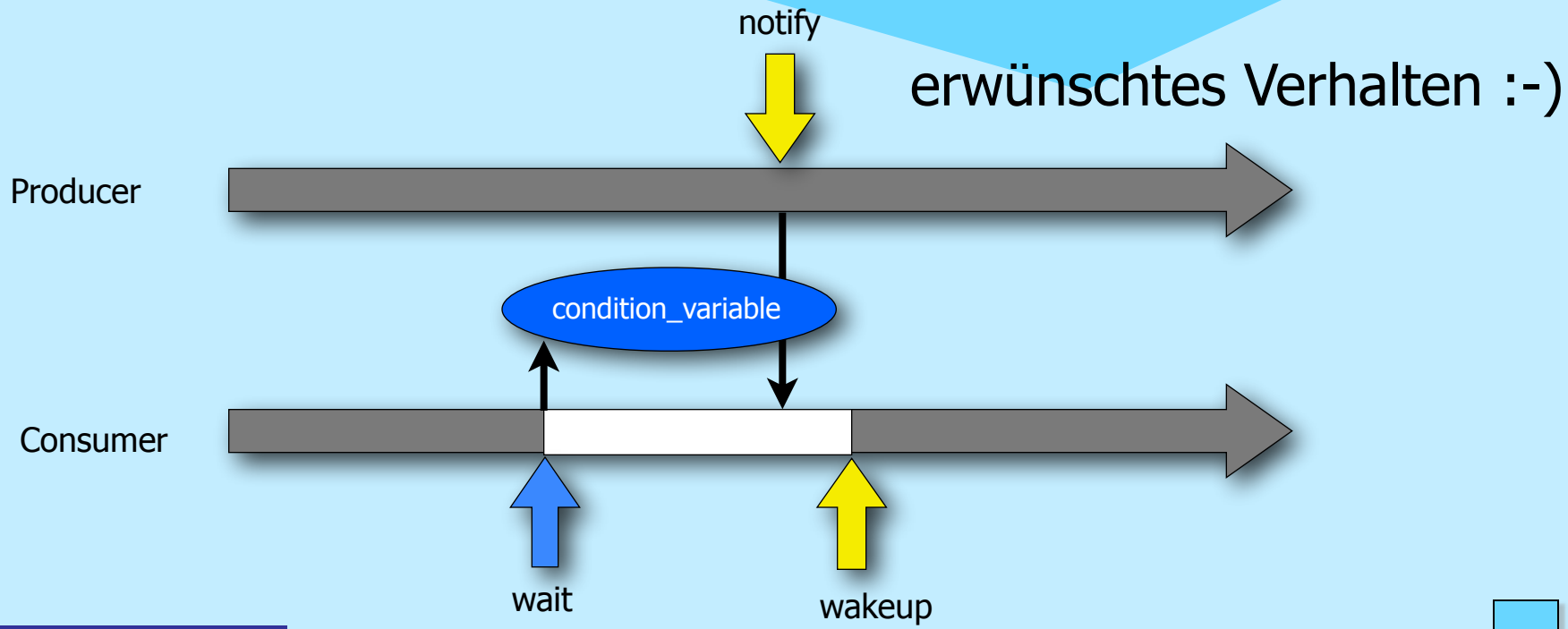
Notification	
`notify_one`	notifies one waiting thread (public member function)
`notify_all`	notifies all waiting threads (public member function)
Waiting	
`wait`	blocks the current thread until the condition variable is woken up (public member function)
`wait_for`	blocks the current thread until the condition variable is woken up or after the specified timeout duration (public member function)
`wait_until`	blocks the current thread until the condition variable is woken up or until specified time point has been reached (public member function)

Threads

`std::condition_variable`

effiziente Implementierbarkeit hat zwei (zunächst enttäuschende) Konsequenzen(*:

1. Notifications werden nicht aufgehoben: wenn kein Thread wartet, verfallen sie



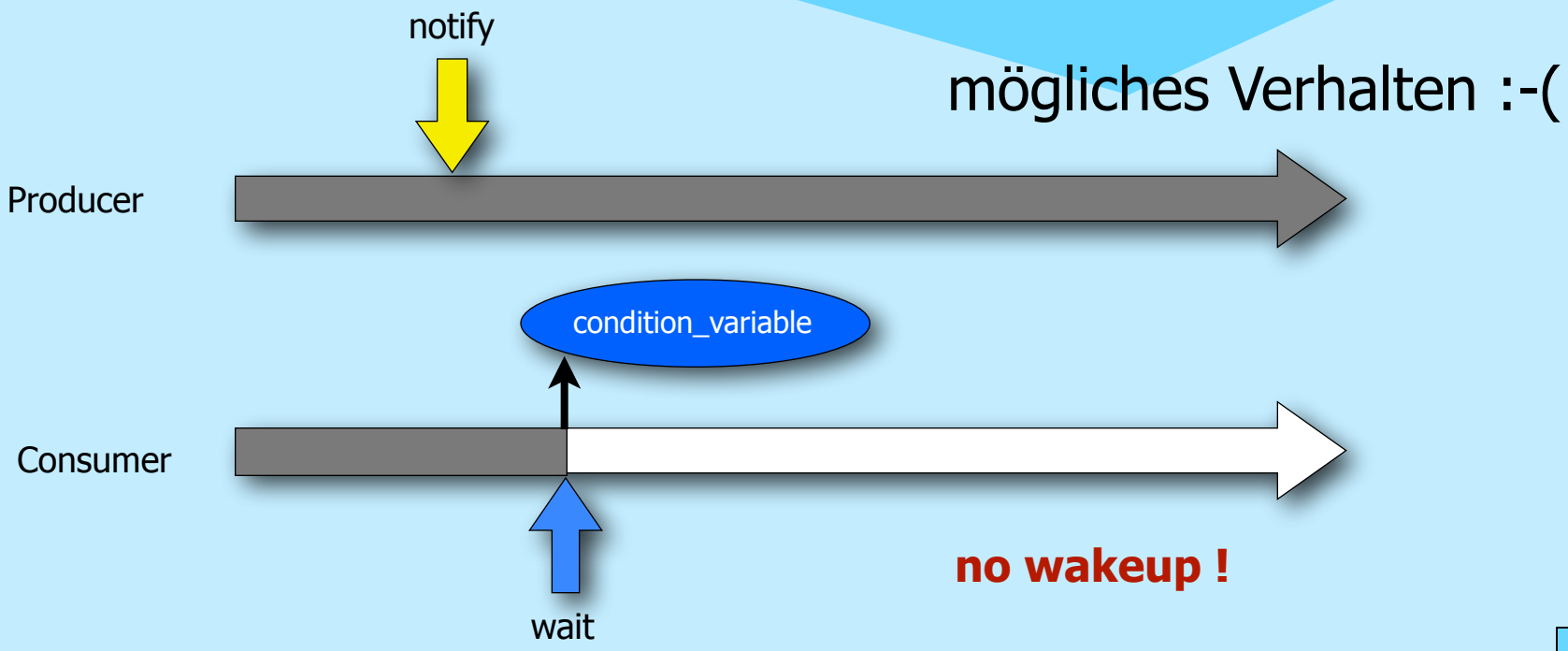
(* siehe auch [YouTube] Bartosz Milewski: C++11 Concurrency, Part 9

Threads

std::condition_variable

effiziente Implementierbarkeit hat zwei (zunächst enttäuschende) Konsequenzen(*:

1. Notifications werden nicht aufgehoben: wenn kein Thread wartet, verfallen sie



(* siehe auch [YouTube] Bartosz Milewski: C++11 Concurrency, Part 9

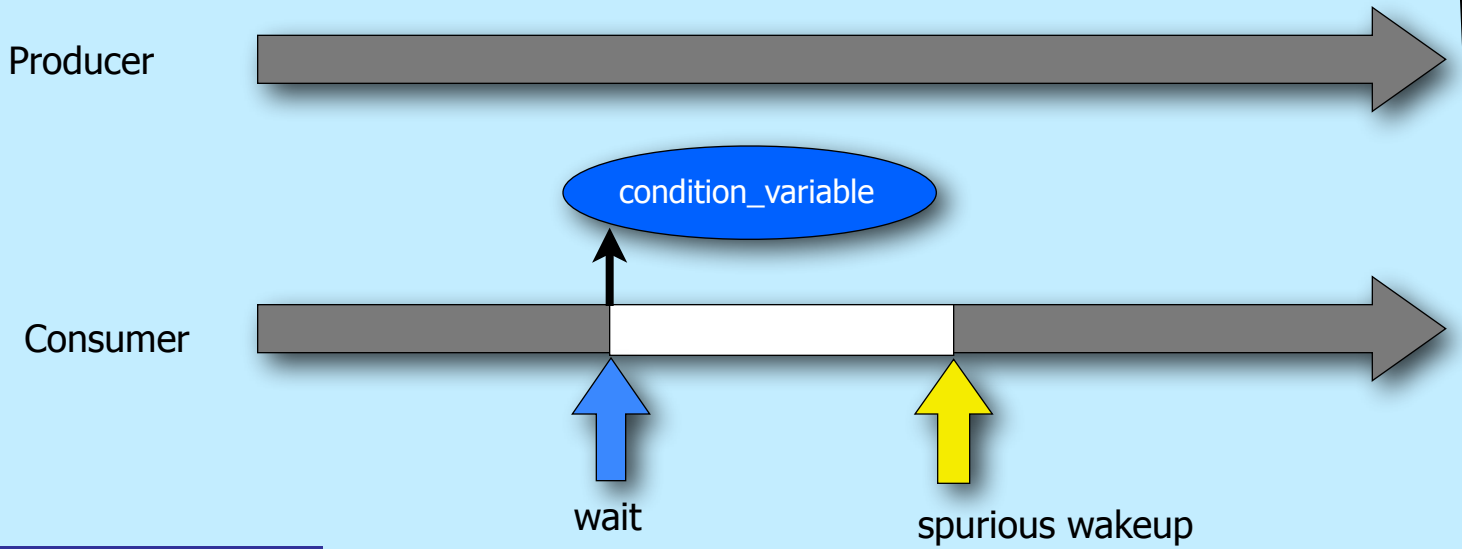
Threads

`std::condition_variable`

effiziente Implementierbarkeit hat zwei (zunächst enttäuschende) Konsequenzen(*:

2. Spurious Wakeup

BM: „A condition variable is not a very good communication channel. It can transmit only one bit and it doesn't do it reliably. ... So it's pretty amazing that this primitive condition variable can be used to create very reliable protocols of communication."



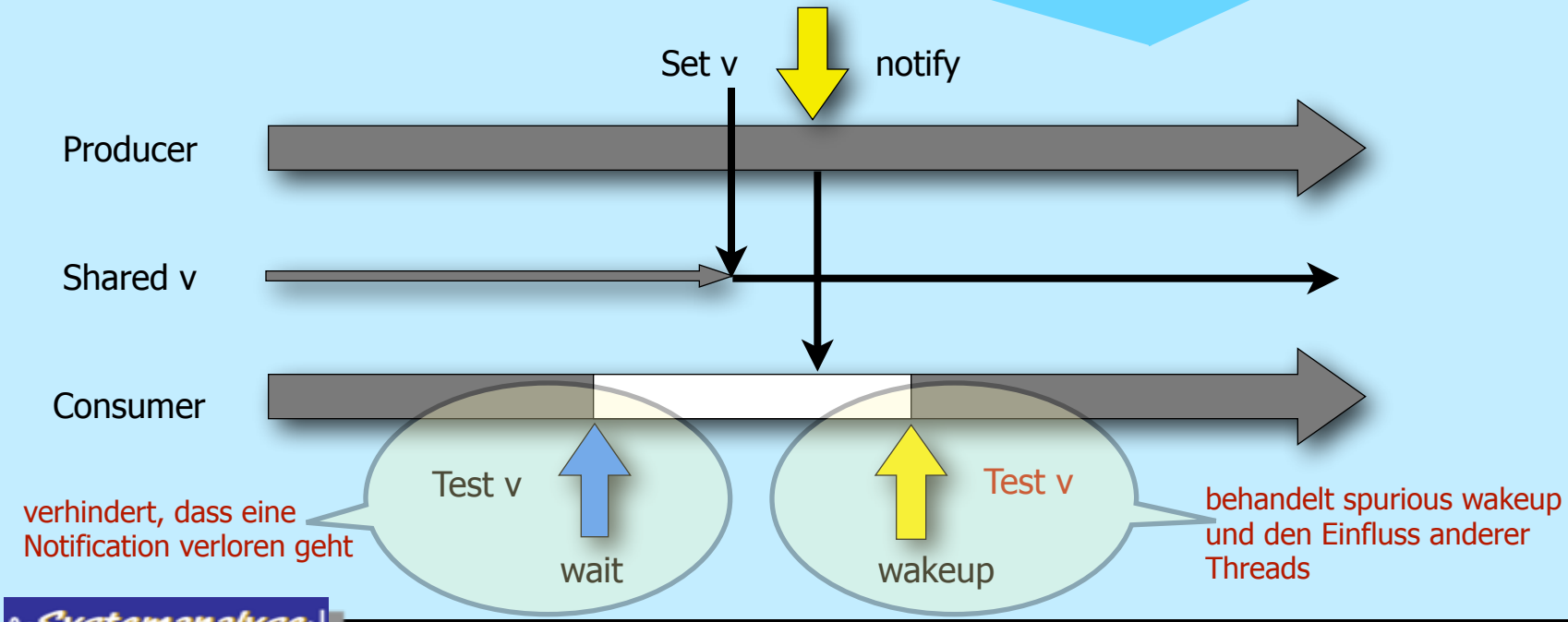
(* siehe auch [YouTube] Bartosz Milewski: C++11 Concurrency, Part 9

Threads

std::condition_variable

um dennoch ein zuverlässiges Kommunikationsprotokoll zu realisieren sind zusätzliche Vorkehrungen zu treffen

1. Einführung einer zusätzlichen geteilten (booleschen) Variable v



Threads

std::condition_variable

Data Race !

Producer Thread

```
v = true;
cond.notify_one();
```

Consumer Thread

```
while (!v) {
   cond.wait(?);
}
```

2. Zugriff zu dieser geteilten (booleschen) Variable v absichern

```
{
   lock_guard(mtx);
   v = true;
}
cond.notify_one();
```

```
unique_lock lck(mtx);
while (!v) {
   cond.wait(lck);
}
// v==true under lock
```

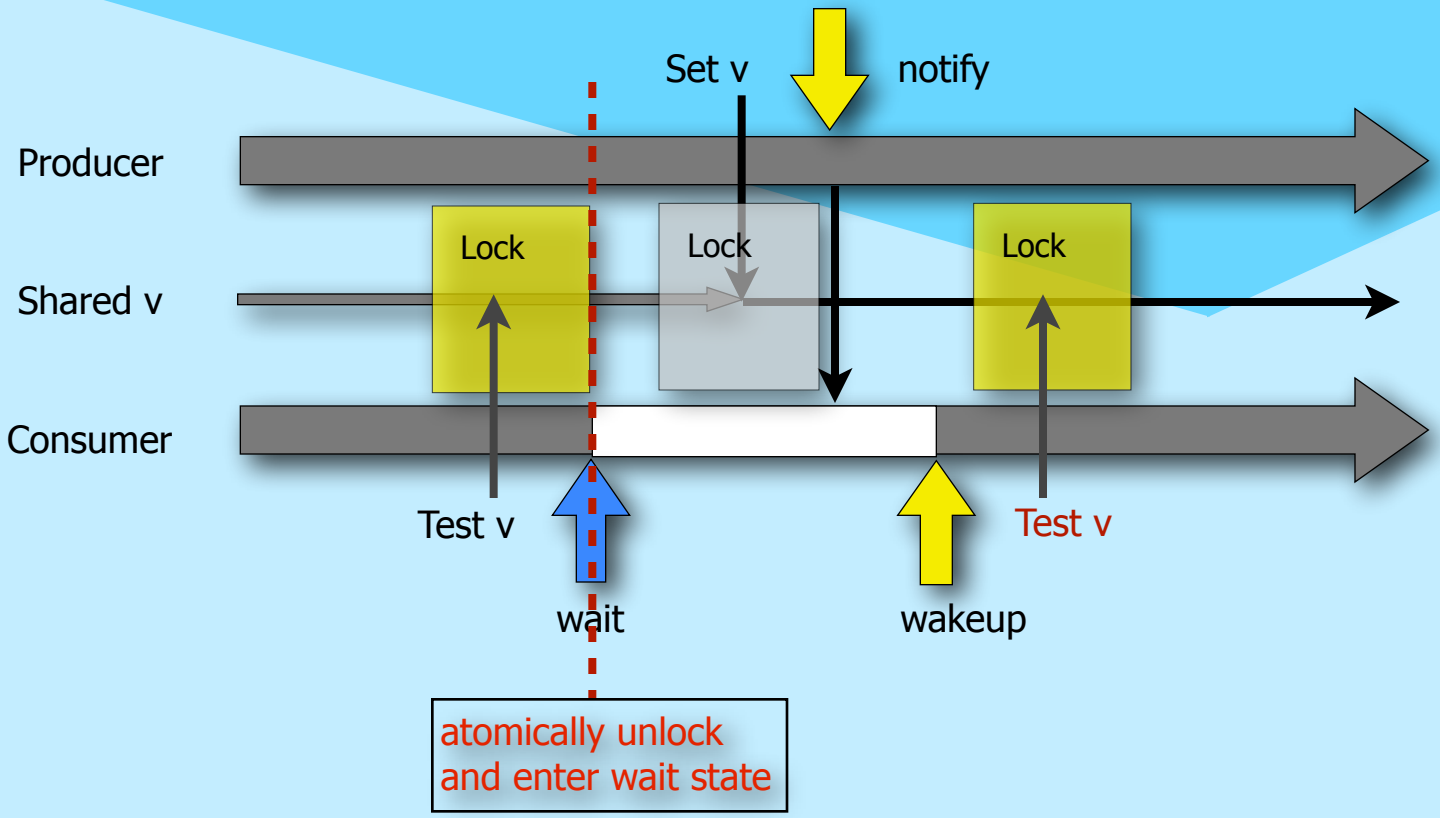
```
lck.unlock(); // !!!
// really wait:
// ...
// on wakeup:
lck.lock(); // !!!
```

equivalent und kompakter (looping within wait)

```
unique_lock lck(mtx);
cond.wait(lck, [&v] {return v;});
// v==true under lock
```

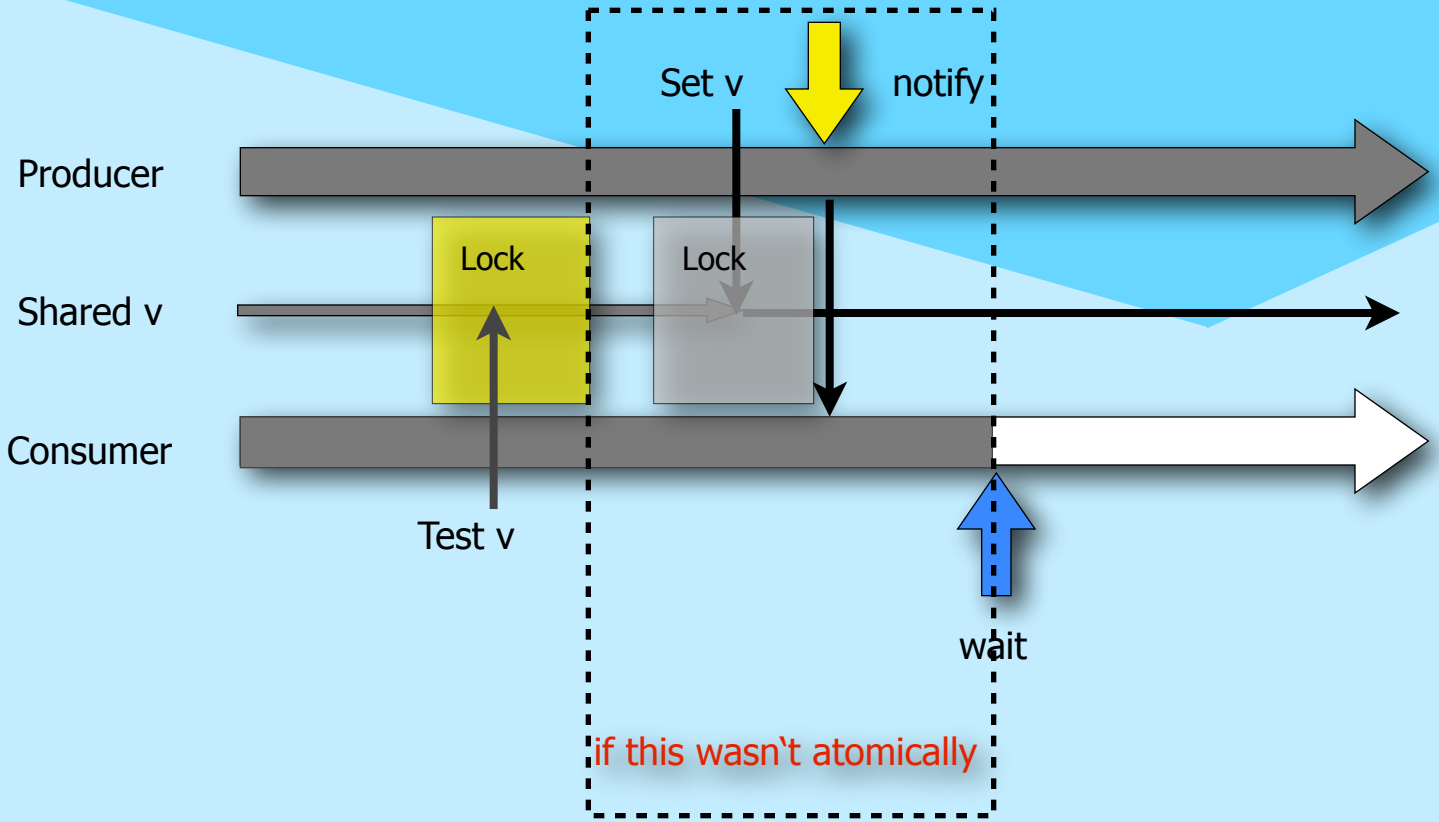
Threads

std::condition_variable



Threads

std::condition_variable



Threads

std::atomic_flag

Atomare Typen: unteilbare Operationen (möglichst!) ohne explizite (Mutex-)Locks

der einzig garantierte ***lock-free atomic type*** ist **std::atomic_flag**

Member functions

(constructor)	constructs an atomic_flag (public member function)	`atomic_flag();` Achtung: unspecified state `atomic_flag( const atomic_flag& ) = delete;`
operator=	the assignment operator (public member function)	
clear	atomically sets flag to `false` (public member function)	
test_and_set	atomically sets the flag to `true` and obtains its previous value (public member function)	

In addition, std::atomic_flag can be value-initialized to clear state with the expression ATOMIC_FLAG_INIT. For an atomic_flag with static storage duration, this guarantees that the values of this flag is known before any constructors are run for static objects.

Threads

std::atomic_flag

```
std::atomic_flag flag = ATOMIC_FLAG_INIT;


if (flag.test_and_set())
// I set it but it was set -> so it‘s used by someone else
// but how to precede ?
else
// I set it and it was unset -> so I‘m the one who owns it
exclusive
```

Threads

std::atomic_flag

```
std::atomic_flag flag = ATOMIC_FLAG_INIT;


while (flag.test_and_set())
// I set it but it was set -> so it‘s used by someone else
// ... loop !
;
// I set it and it was unset -> so I‘m the one who owns it
exclusive
```

Threads

`std::atomic<T>` weitere atomare Typen

u.U. nicht ***lock-free***

(constructor)	constructs an atomic object (public member function)
operator=	stores a value into an atomic object (public member function)
is_lock_free	checks if the atomic object is lock-free (public member function)
store (C++11)	atomically replaces the value of the atomic object with a non-atomic argument (public member function)
load (C++11)	atomically obtains the value of the atomic object (public member function)
operator T	loads a value from an atomic object (public member function)
fetch_add (C++11)	atomically adds the argument to the value stored in the atomic object and obtains the value held previously — dito fetch_* with sub, and, or, xor
operator++ operator++(int)	increments or decrements the atomic value by one — dito --, and +=, -=, &=, |=, ^=

.... und weitere

Spezialisierungen für: `bool, [[un]signed] char, [unsigned] short, [unsigned] int, [unsigned] long, [unsigned] long long, ...`

Threads

`std::memory_order` weitere atomare Typen

alle Operationen haben per *`default`* Argument die *`Memory Order`*
`std::memory_order_seq_cst` == Sequential Consistent

mehr Flexibilität (Parallelität) erlauben abgeschwächte Formen:

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

guru stuff: `read http://en.cppreference.com/w/cpp/atomic/memory_order`

Threads

Einfachheit vs. Performanz

Szenario: 100.000.000 * ++ / 100.000.000 * -- auf int

Lock-based	Lock-free			
`std::lock_guard`	`test_and_set (mo_seq_cst)`	`test_and_set (mo_acq/rel)`	`atomic<int>`	
234.774s	**18.397s**	**13.335s**	**2.887s**	clang 4.2 / 1.86GHz Intel Core 2 Duo
63.048s	**14.304s**	**11.874s**	**3.372s**	VS2013 / 2.54GHz Intel Core 2 Duo