

# Vorlesung 1

# Programmierung II

## Einführung in C++

Dozent: Prof. Dr. Peter Kelb  
Raumnummer: Z4030  
e-mail: [pkelb@hs-bremerhaven.de](mailto:pkelb@hs-bremerhaven.de)  
Sprechzeiten: nach Vereinbarung  
Sourcen: Elli

Ziel der Vorlesung:

- Einführung in C++

Parallel in Algorithmen:

- Sortiervverfahren
- dynamische Datenstrukturen (Vektoren und Listen)
- Suchverfahren (Bäume und Hashing)

Voraussetzung:

- Vorlesung: Programmierung I (inhaltlich, nicht formal)

Bücher:

Algorithmen:

„Algorithmen und Datenstrukturen“ (Pearson Studium - IT), Robert Sedgewick, Kevin Wayne, Pearson Studium, ISBN-13: 978-3868941845

C++:

„Die C++-Programmiersprache: Aktuell zu C++11“, Carl Hanser Verlag GmbH & Co. KG ISBN-13: 978-3446439610

„Der C++ Programmierer“ von Ulrich Breymann, Carl Hanser Verlag, ISBN-13 978-3-446-44346-4

## Organisation:

- 2 SWS Vorlesung C++ für Inf
- 2 SWS Vorlesung Algorithmen für WInf und Inf
- 1 x 2 SWS Übung

## Prüfung:

- Entwurf zum Thema C++ und Algorithmen
- nähere Informationen in den Übungsgruppen

## Grundlegendes zur Programmiersprache C++

- C++ ist eine standardisierte Sprache: ISO/IEC 14882, Standard for the C++ Programming Language
- C++ ist aus der Programmiersprache C hervorgegangen
- man muss nicht C können, um C++ zu lernen
- das Gegenteil ist der Fall: ohne C Kenntnisse lernt man leichter, gut C++ zu benutzen

## Historischer Abriss

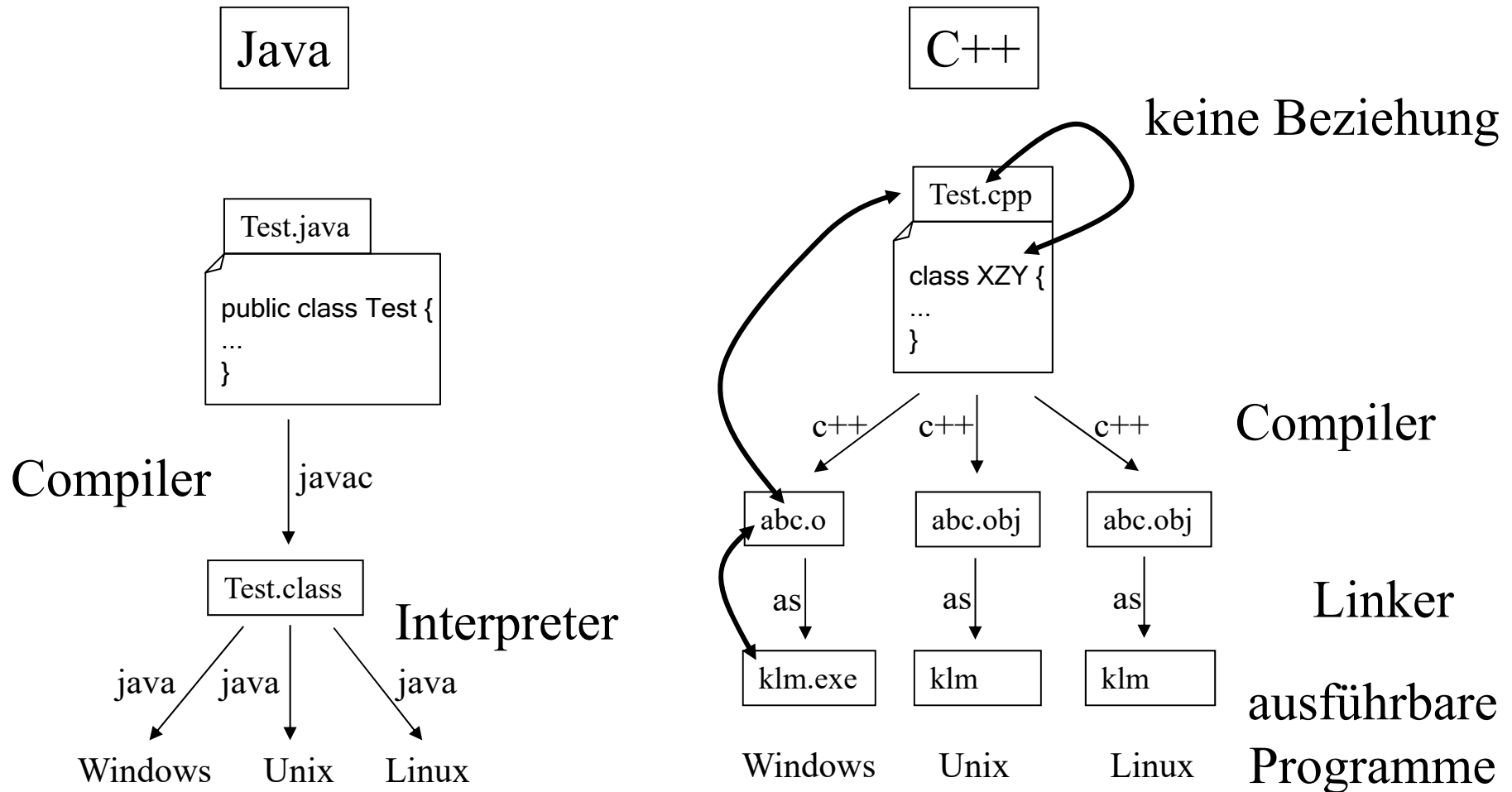
- 1977: Programmiersprache ‚C‘ für das UNIX Betriebssystem auf der DEC PDP-11 von Dennis Ritchie
- Zweck: weder hochabstrahierend, noch umfangreich, aber allgemein
- Anwendung: das komplette Betriebssystem, der C Übersetzer, damals fast alle UNIX Anwendungsprogramme mit C
- C++ ging aus C hervor
- C ist fast komplett in C++ eingebettet
- Das Klassenkonzept von C++ stammte aus Simula67
- Überladen von Operatoren, Platzierung von Deklarationen aus Algol68
- ...

## Historischer Abriss (Forts.)

- ...
- 1979: „C with Classes“ von B. Stroustrup
- 1984: der Name „C++“ wurde von Rick Mascetti geprägt
- 1985: erstes kommerzielles Release von C++
- 1991: Einführung von Templates und Exceptions
- 1997: Einführung von Namespaces, `dynamic_cast`, Standard Template Library
- 1998: ISO-C++ Standard
- 2002: Beginn der Arbeiten an C++ 11
- 2011: ISO-C++ 11 Standard
- 2012: erste vollständige Implementierungen von C++ 11



# Vom Sourcecode zum Programm



## Welcher Compiler ?

- es gibt eine Vielzahl von C++ Compilern
  - public domain Compiler vs. kommerziellen Compiler
  - public domain Compiler sind sehr gut
  - public domain Entwicklungsumgebungen arbeiten ok, aber
  - kommerzielle Entwicklungsumgebungen sind oft besser
  - in dieser Vorlesung:
    - public domain Entwicklungsumgebung CodeBlocks ([www.codeblocks.org](http://www.codeblocks.org)) mit
    - public domain C++ Compiler von GNU (MinGW)
  - in den Übungen und zu Hause:
    - GNU Compiler unter Linux zusammen mit valgrind
- WICHTIG !!!

Entwicklungsumgebung  $\neq$  Compiler

Go!

## Ein erstes Beispiel

`#include <iostream>` ← entspricht der `import` Anweisung

`int main() {` ← Programm fängt  
immer bei `main` an  
`std::cout << "Hello World!" << std::endl;`  
`return 0;`  
`}`

Der Rückgabewert  
des Programms an  
das Betriebssystem

Ausgabe auf dem  
Standardausgabemedium

Größter Unterschied zu Java: keine Klasse  
Dies gilt aber nur in diesem Programm !!!!

Go!

## Ein erstes Beispiel: Modifikation

```
#include <iostream>
#include <stdlib.h>
```

```
using namespace std;
```

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

der Namensraum „std“ wird  
für diese Datei geöffnet

- Aufruf des  
Systemprogramms „pause“
- kein Bestandteil von C++

kein std mehr

Go!

## Eine Rundreise durch C++

- C++ ist eine prozedurale Programmiersprache
- es gibt Funktionen und Methoden (wie in Java), aber auch außerhalb von Klassen gibt es Funktionen und Prozeduren (Funktionen, die vom Rückgabewert `void` sind)

```
#include <iostream>
```

```
using namespace std;
```

```
int quadrat(int i) { return i * i; }  
int malZwei(int i) { return i * 2; }  
int zweiMalQuadrat(int i) { return malZwei(quadrat(i)); }  
  
int main() {  
    cout << "4 * 4 = " << quadrat(4) << endl;  
    cout << "2 * 5 = " << malZwei(5) << endl;  
    cout << "2 * (6 * 6) = " << zweiMalQuadrat(6) << endl;  
    return 0;  
}
```

Funktionen, die einen `int`-Wert übergeben  
bekommen und einen `int`-Wert zurückliefern

Funktionen und  
Prozeduren können  
andere Funktionen und  
Prozeduren aufrufen

## Eine Rundreise durch C++ (Forts.)

- C++ hat Variablen und ist stark typisiert
- d.h. alle Variablen haben einen bestimmten Typen, der zur Compilezeit festliegt
- es gibt die Typen int, bool, char, double, und noch viele mehr
- im Gegensatz zu Java sind die Größen der Typen nicht durch die Sprache definiert, sondern können von Compiler zu Compiler unterschiedlich sein

- Beispiel:

- `int i;`
- `bool b, b2, b3;`
- `char c = 'a';`

Variablendeklaration  
ohne Initialisierung

Variablendeklaration mit Initialisierung

VORSICHT: anders als in Java werden Variablen  
nicht mit einem Standardwert belegt

Go!

## Eine Rundreise durch C++ (Forts.)

```
#include <iostream>
```

```
using namespace std;
```

```
int quadrat(int i) {  
    int result;  
    cout << "result = " << result << endl;  
    result = i * i;  
    return result;  
}
```

lokale Variable

lesender Zugriff ohne  
vorherige Initialisierung

lesender Zugriff  
nach Initialisierung

```
int main() {  
    int q = quadrat(4);  
    bool b = q > 15;  
    cout << "4 * 4 = " << q << endl;  
    cout << "4 * 4 > 15 = " << b << endl;  
    return 0;  
}
```

Variablendeklarationen  
mit Initialisierung

Go!

## Eine Rundreise durch C++ (Forts.)

- die Initialisierung einer Variablen bei ihrer Deklaration kann wie in Java erfolgen
  - `int i = 7;`
- oder mit einer sogenannten Initialisierungsliste
  - `int i {7};`
- beides kann auch kombiniert werden, entspricht aber der Initialisierungsliste
  - `int i = {7};`
- wesentlicher Unterschied ist, dass eine Typkonvertierung bei Verwendung der Initialisierungsliste eine Warnung hervorruft
  - `int i {7.2}; // Warnung, i wird mit 7 initialisiert`
  - `int j = 7.2; // ok, j wird mit 7 initialisiert`



Go!

## Eine Rundreise durch C++ (Forts.)

- C++ hat Abfragen und Schleifen
- Abfragen sind wie in Java mittels if-else
- Schleifen werden wie in Java mittels while und for gebildet

...

```
int quadrat(int i) { return i*i; }
```

```
int main() {  
    int q = quadrat(4);  
    if (q > 15) {  
        cout << "4 * 4 > 15" << endl;  
    } else {  
        cout << "4 * 4 <= 15" << endl;  
    }  
    return 0;  
}
```

Wenn die Bedingung wahr ist, wird die folgende Anweisung ausgeführt, ansonsten die else Anweisung (soweit vorhanden)

Go!

## Eine Rundreise durch C++ (Forts.)

```
#include <iostream>
```

```
using namespace std;
```

```
int quadrat(int i) {  
    return i*i;  
}
```

```
void ausgabe(int i,int j) {  
    int q = quadrat(i);  
    if (q > j)  
        cout << i << " * " << i << " > " << j << endl;  
    else  
        cout << i << " * " << i << " <= " << j << endl;  
}
```

```
int main() {  
    for(int i = 0;i < 7;++i)  
        ausgabe(i,15);  
    return 0;  
}
```

for-Schleifen in C++ werden  
analog zu for-Schleifen in Java  
gebildet und verwendet

## Eine Rundreise durch C++ (Forts.)

- C++ hat auch Arrays
- im Gegensatz zu Java merken sich Arrays aber nicht ihre Länge
- somit muss die Länge eines Arrays extra in einer Variablen gespeichert werden
- beim Zugriff auf ein Arrayelement (lesend / schreiben) werden die Arraygrenzen **nicht (!!!)** überprüft
- das Ergebnis ist undefiniert eventuell stürzt das Programm auch ab

Go!

## Eine Rundreise durch C++ (Forts.)

```
#include <iostream>
```

```
using namespace std;
```

Arrays können übergeben  
werden (analog zu Java)

```
void ausgabe(int a[],int len) {  
    for(int i = 0;i < len;++i)  
        cout << a[i] << endl;  
    a[0] = 43;  
}
```

Arrayzugriffe (lesend /  
schreiben) wie in Java

```
int main() {  
    const int len = 10;  
    int a[len];  
    for(int i = 0;i < len;++i)  
        a[i] = i;  
    ausgabe(a,len);  
    cout << endl;  
    ausgabe(a,len);  
    cout << endl;  
    ausgabe(a,len + 1);  
    return 0;  
}
```

Erzeugung sieht anders als  
in Java aus: hier statisches  
Array mit 10 Einträgen

erzeugt einen illegalen Lesezugriff  
außerhalb der Arraygrenzen

## Eine Rundreise durch C++ (Forts.)

- Außerdem gibt es in C++
  - Zeiger (nicht in Java)
  - dynamische Speicherverwaltung (kein Garbage Collection wie in Java)
  - Namensräume (ähnlich zu packages in Java)
  - Ausnahmebehandlung (Exceptions wie in Java)
  - (abstrakte) Klassen (ähnlich zu Java)
  - virtuelle Funktionen (ähnlich zu Java, aber präziser zu steuern)
  - Klassenhierarchien (mit Mehrfachvererbung)
  - generische Programmierung (viel mehr als Generics in Java)

# Vorlesung 2

## Typen in C++

- ähnlich wie in Java gibt es auch in C++ viele elementare Typen
- im Gegensatz zu Java werden die Größen aber nicht durch die Sprache oder den Standard festgelegt, sondern können von Compiler / Betriebssystem zu Compiler / Betriebssystem unterschiedlich sein
- folgende Typen gibt es:
  - ein boolescher Typ (**bool**) (in Java: **boolean**)
  - Zeichentypen (z.b. **char**) (in Java gibt es nur einen)
  - viele ganzzahlige Type (z.b. **int**)
  - Gleitkommatype (z.b. **double**)
  - selbstdefinierte Aufzählungstypen (**enum**)
  - ...

## Typen in C++ (Forts.)

- ...
- den Typ `void` (analog zu Java: keine Information)
- Feld- oder Arraytypen (z.B. `int[]`)
- Zeigertypen (z.B. `int*`) (gibt es nicht in Java)
- Referenztypen (z.B. `char&`) (gibt es auch nicht in Java)
- neben den elementaren Typen gibt es natürlich die selbstdefinierten
- Klassen (`class`) und Strukturen (`struct`)



## Der Typ bool

- analog zu Java gibt es den booleschen Typen
- er kann die beiden Wahrheitswerte **true** und **false**
- Unterschied zu Java: in C++ heißt er **bool** statt **boolean**
- Typkonvertierung: anders als in Java
  - **bool** kann in eine Ganzzahl konvertiert werden
    - **true** wird in **1** verwandelt
    - **false** wird in **0** verwandelt
  - eine Ganzzahl kann in ein **bool** verwandelt werden
    - **0** wird in **false** verwandelt
    - jede andere Zahl wird in **true** verwandelt

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```


```
bool gen() {  
    return 17;  
}
```

automatische  
Typkonvertierung



```
int main() {  
    bool b1 = true, b2 = false;  
    cout << b1 << endl << b2 << endl << gen() << endl << endl;  
    bool b3 = b1 == gen();  
    bool b4 = 17 == 18;  
    bool b5 = (bool)17 == (bool)18;  
    cout << b3 << endl << b4 << endl << b5 << endl << sizeof(b5) << endl;  
    return 0;  
}
```


boolesche  
Variablendeklaration  
mit Initialisierung



Zeilenumbruch



liefert die Größe  
der Variable b5 in  
Anzahl von Bytes



## Der Typ bool (Forts.)

- ähnlich wie in Java gibt es boolesche Operatoren zum Verknüpfen von booleschen Werten
- ! für die Negation
- && (logisches Und)
- || (logisches Oder)
- == (Gleichheit)
- != (Ungleichheit)
- & (bitweises Und)
- | (bitweises Oder)
- ^ (bitweises exklusives Oder)
- ? : (bedingter Ausdruck)

beide Argumente  
werden ausgewertet


Go!

## Beispiel

...

```
void print(bool b) {  
    cout << "Der Wert ist " << (b ? "true" : "false") << endl;  
}
```

bedingter  
Ausdruck



```
bool id(bool b) {  
    cout << "call of id" << endl;  
    return b;  
}
```

einfache Identitätsfunktion  
mit Seiteneffekt (Ausgabe)

```
int main() {  
    bool b1 = true, b2 = false;  
    print(b1 && b2);  
    print(b1 || b2);  
    print(b1 ^ b2);  
    print(id(true) | id(false));  
    print(id(true) || id(false));  
    print(b1 == b2);  
    print(b1 != b2);  
    return 0;  
}
```

hier sollte ein  
Unterschied existieren

## Der Typ bool (Forts.)

- da die booleschen Werte 0 und 1 entsprechen, können auch die arithmetischen Operatoren  $+$ ,  $-$ ,  $*$  und  $/$  angewendet werden
- $+$  entspricht  $||$
- $*$  entspricht  $\&\&$
- $-$  entspricht  $\wedge$
- auch sind Vergleiche somit möglich, weil `true` ( $\cong 1$ ) größer als `false` ( $\cong 0$ ) ist

+			
	0	0	0
	0	1	1
	1	0	1
	1	1	2

*			
	0	0	0
	0	1	0
	1	0	0
	1	1	1

-			
	0	0	0
	0	1	-1
	1	0	1
	1	1	0

Go!

## Beispiel

```
void print(bool b) {  
    cout << "der wert ist " << (b ? "true" : "false") << endl;  
}
```

```
int main() {  
    bool b1 = true, b2 = false;  
    bool b[] = {false, false, false, true, true};  
    print(b1 < b2);  
    print(b1 > b2);  
    b1 = b[0];  
    for(int i = 1; i < 3; ++i)  
        b1 = b1 + b[i];  
    print(b1);  
    b1 = b[0];  
    for(int i = 1; i < 4; ++i)  
        b1 = b1 + b[i];  
    print(b1);  
    b1 = b[0];  
    for(int i = 1; i < 5; ++i)  
        b1 = b1 + b[i];  
    print(b1);  
    return 0;  
}
```

Array von fünf  
booleschen Werte

arithmetischer  
Vergleich

Disjunktion der  
ersten 3 booleschen  
Werte

Disjunktion aller  
fünf booleschen  
Werte

## Der Typ char

- anders als in Java enthält der Typ `char` in C++ nur die ASCII Zeichen
- daher wird er i.A. auch nur ein Byte an Speicherplatz benötigen
- analog zu dem booleschen Typ kann auch ein `char` in ganze Zahlen (und umgekehrt) verwandelt werden
- somit sind auch auf `char` alle Vergleichs- und arithmetischen Operationen möglich

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
void print(bool b) {  
    cout << "der wert ist " << (b ? "true" : "false") << endl;  
}
```

Einlesen eines  
Zeichens über die  
Standardeingabe

```
int main() {  
    char c = 'a';  
    cout << c << endl << "Bitte einen Buchstaben eingeben: ";  
    cin >> c;
```

```
    cout << "Eingegeben wurde : " << c << endl;
```

Konvertierung eines  
char in ein bool

```
    print(c);
```

```
    for(int i = 0; i < 26; ++i) {
```

```
        char c = 'A' + i;
```

```
        cout << c << " " << (int)c << endl;
```

char plus int  
abgespeichert in  
einem char

```
    }  
    return 0;
```

```
}
```

Konvertierung eines  
int in ein char



## Die Typen der ganzen Zahlen

- ähnlich wie in Java gibt es in C++ verschiedene Typen, die ganze Zahlen repräsentieren
- wie in Java unterscheiden sich diese Typen daran, welcher Wertebereich durch einen Typen abgedeckt wird
- es gibt: `short`, `int`, `long` und `long long`
- es gibt kein `byte`, stattdessen wird `char` verwendet
- die Größe ist nicht durch den Standard absolut festgelegt
- so gilt:
  - $1 \leq \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$
  - `char` hat mindestens 8 Bits
  - `short` hat mindestens 16 Bits
  - `long` hat mindestens 32 Bits

Go!

## Beispiel

```
int main() {  
    char c;  
    short s;  
    int i;  
    long l;  
    long long ll;  
    cout << sizeof(c) << endl << sizeof(s) << endl << sizeof(i) << endl << sizeof(l)  
        << endl << sizeof(ll) << endl;  
    c = (1 << (sizeof(c)*8-1)) - 1;   
    cout << (int)c << endl;  ++c;  cout << (int)c << endl;  
    s = (1 << (sizeof(s)*8-1)) - 1;   
    cout << s << endl;  ++s;  cout << s << endl;  
    i = (1 << (sizeof(i)*8-1)) - 1;   
    cout << i << endl;  ++i;  cout << i << endl;  
    l = (1L << (sizeof(l)*8-1)) - 1;   
    cout << l << endl;  ++l;  cout << l << endl;  
    ll = (1LL << (sizeof(ll)*8-1)) - 1;   
    cout << ll << endl;  ++ll;  cout << ll << endl;  
    return 0;  
}
```

Ausgabe der  
Typgrößen in Byte

Berechnung des  
Maximalwerts

dies ist eine „große“ 1

## Die Typen der ganzen Zahlen (Forts.)

- sehr oft werden ganze Zahlen dazu verwendet, Arrayfelder zu indizieren
- hierzu würde es ausreichen, wenn die ganzen Zahlen nur positive Werte enthalten würden
- alle ganzen Zahlen Typen können durch den Präfix **unsigned** in einen rein positiven Ganzzahltypen verändert werden
- dieser enthält nur die positiven Werte (einschließlich der 0)
- da das Vorzeichenbit eingespart wird, können doppelt so viele positive Zahlen dargestellt werden wie bei dem normalen Ganzzahl Typen
- statt **unsigned** kann als Präfix auch **signed** angeführt werden
- da dies aber Standard ist, kann es auch weggelassen werden

Go!

## Beispiel

```
int main() {
    unsigned char c;
    unsigned short s;
    unsigned int i;
    unsigned long l;
    unsigned long long ll;

    cout << sizeof(c) << endl << sizeof(s) << endl << sizeof(i) << endl << sizeof(l)
        << endl << sizeof(ll) << endl;

    c = (1 << (sizeof(c)*8-1)) - 1;
    cout << (int)c << endl; ++c; cout << (int)c << endl;

    s = (1 << (sizeof(s)*8-1)) - 1;
    cout << s << endl; ++s; cout << s << endl;

    i = (1 << (sizeof(i)*8-1)) - 1;
    cout << i << endl; ++i; cout << i << endl;

    l = (1L << (sizeof(l)*8-1)) - 1;
    cout << l << endl; ++l; cout << l << endl;

    ll = (1LL << (sizeof(ll)*8-1)) - 1;
    cout << ll << endl; ++ll; cout << ll << endl;

    return 0;
}
```

Ausgabe der  
Typgrößen in Byte

Berechnung des  
Maximalwerts

dies ist eine „große“ 1

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int f[] = {23,17,-5,18,29};
```

```
    for(unsigned int ui = 0; ui < 5; ++ui)
```

```
        cout << f[ui] << endl;
```


```
    for(unsigned int ui = 4; ui >= 0; --ui)
```

```
        cout << f[ui] << endl;
```


```
    return 0;
```

```
}
```

gebe alle Werte von f aus,  
beginnend beim ersten Index



gebe alle Werte von f aus,  
beginnend beim letzten Index



## Die Typen der ganzen Zahlen (Forts.)

- das vorherige Programm endet nie (stützt vielleicht auch ab), weil  
    `for(unsigned int ui = 4; ui >= 0; --ui)`  
diese Schleife niemals terminiert
- die Bedingung `ui >= 0` ist trivialerweise immer erfüllt, weil `ui` vom Typ `unsigned int` ist
- die Abfrage muss derart geändert werden, dass die Schleife abbricht, wenn `ui` den Wert 0 annimmt
- da bei 0 noch ein Durchlauf stattfinden soll, muss das Testen vor dem Dekrementieren stattfinden
- Lösung:  
    `for(unsigned int ui = 5; ui-- > 0;)`
- übrigens: bei `unsigned int` kann das `int` weggelassen werden

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int f[] = {23,17,-5,18,29};  
    for(unsigned int ui = 0; ui < 5; ++ui)  
        cout << f[ui] << endl;
```

```
    for(unsigned ui = 5; ui-- > 0; )  
        cout << f[ui] << endl;  
    return 0;  
}
```

- erst Test auf  $> 0$
- dann Dekrementierung,  
**bevor** der Rumpf ausgeführt wird
- am Ende der Schleife **keine** neue Dekrementierung

## Die Gleitkommatypen

- analog zu Java gibt es in C++ **float's** und **double's**
- zusätzlich gibt es den Typen **long double**, der eine erweiterte Genauigkeit zu **double** darstellen soll
- alle Typen dienen dazu, Gleitkommazahlen darzustellen
- der Typ **double** verwendet u.U. mehr Bytes für die interne Darstellung als ein **float**, ein **long double** u.U. mehr Bytes für die Speicherung als ein **double**
- dies ist aber implementierungsabhängig



Go!

## Beispiel

```
#include <iostream>

using namespace std;

int main() {
    float f = 10213.32452e10;
    double d = 10213.32452e300;
    long double ld = 10213.32452e300;
    cout << sizeof(f) << endl << sizeof(d) << endl << sizeof(ld) << endl;
    cout << f << endl << d << endl << ld << endl;
    return 0;
}
```

## Die Typ void

- ähnlich wie in Java dient der Typ `void` zur Darstellung, dass kein Wert dargestellt werden soll
- somit macht es auch keinen Sinn, Variablen vom Typ `void` anzulegen

```
void i;    error: variable or field `i'  
           declared void
```

- der Typ wird genutzt, um zu zeigen, dass eine Methode keinen Wert zurückliefert
- in C++ wird `void` auch noch genutzt um unbekannte Zeiger (siehe später) zu definieren

## Aufzählungstypen

seit C++-11  
möglich

- analog zu Java gibt es in C++ Aufzählungstypen
- die Idee ist ähnlich, die Anwendung sieht anders aus
- die Deklaration eines Aufzählungstypen beginnt mit dem Schlüsselwort `enum`

`enum [class] { ROT, BLAU, GRUEN};`

- soll ein Enumerationstyp mehrfach verwendet werden, so muss ihm ein Namen gegeben werden

`enum class Farbe { ROT, BLAU, GRUEN};`

`Farbe x = Farbe::ROT;`

- intern werden die Aufzählungstypen als Subtypen von Ganzzahltypen verwaltet
- daher werden sie als Zahlen ausgegeben, man kann mit ihnen aber auch rechnen

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
enum Farbe {ROT, GRUEN, BLAU};
```

kein **class**: Elemente können ohne Prefix **Farbe::** verwendet werden, sind intern nur **int**-Werte

Aufzählungstyp mit Namen

```
int main() {
```

```
    Farbe f = ROT;
```

```
    enum {A_1, A_2, A_3, A_4} x;
```

```
    x = A_4;
```

```
    cout << f << endl << GRUEN << endl << x << endl;
```

```
    f = (Farbe)(f + 1);
```

```
    if (f == GRUEN)
```

```
        cout << "f ist jetzt GRUEN" << endl;
```

```
    return 0;
```

```
}
```

Aufzählungstyp ohne Namen

rechnen mit Aufzählungstypen.  
Vorsicht: es findet keine Bereichsüberprüfung statt

Go!

## Beispiel

```
enum class Farbe {ROT, GRUEN, BLAU};
```

```
ostream& operator<<(ostream& os, Farbe f) {  
    switch(f) {  
        case Farbe::ROT: os << "rot" << endl; break;  
        case Farbe::GRUEN: os << "gruen" << endl; break;  
        case Farbe::BLAU: os << "blau" << endl; break;  
    }  
    return os;  
}
```

Ausgabeoperator: siehe später

```
int main() {  
    Farbe f = Farbe::ROT;  
    enum {A_1, A_2, A_3, A_4} x;  
    x = A_4;  
    cout << f << endl << Farbe::GRUEN << endl << x << endl;  
    f = Farbe::GRUEN;  
    if (f == Farbe::GRUEN)  
        cout << "f ist jetzt GRUEN" << endl;  
    return 0;  
}
```

mit **class**: Elemente  
können nur mit Prefix  
**Farbe::** verwendet  
werden, man kann nicht  
mehr mit ihnen rechnen

Aufzählungstyp ohne Namen  
machen nur ohne class einen  
Sinn

## Aufzählungstypen (Forts.)

- die Elemente eines Aufzählungstyps werden von 0 aufsteigend bis  $n-1$  ( $n$  = Anzahl der Elemente) durchnummeriert
- da ein Aufzählungstyp intern als Subtyp eines Ganzzahltyps dargestellt wird, kann es sinnvoll sein, die Werte abweichend selber zu definieren
- so kann man (kleine) Mengen von Elementen eines Aufzählungstyps effizient in einem Ganzzahltyp speichern und verarbeiten
- hierzu werden Ganzzahltypen als Mengen und die bitweisen Operatoren als Mengenoperationen verwendet
  - $|$  ist die Mengenvereinigung
  - $\&$  ist der Mengendurchschnitt
  - die Zahl 0 repräsentiert die leere Menge

## Beispiel

Elemente haben  
selbstdefinierte Werte

```
enum Farbe {ROT = 1, GRUEN = 2, BLAU = 4, GELB = 8},
```

```
void testOne(int set, Farbe f) {  
    if ((set & f) != 0) { ← f ∈ set ?
```

```
        switch (f) {  
            case ROT: cout << " rot";break;  
            case GRUEN: cout << " gruen";break;  
            case BLAU: cout << " blau";break;  
            case GELB: cout << " gelb";break;
```

switch Anweisung  
analog zu Java

```
        }  
    }  
}
```

die erste Zeichenkette

```
void testAll(int set, const char s[]) {  
    cout << "Die Menge " << s << " enthaelt";  
    testOne(set, ROT);  
    testOne(set, GRUEN);  
    testOne(set, BLAU);  
    testOne(set, GELB);  
    cout << endl;  
}
```

testest alle Farben durch

...

Go!

## Beispiel

...

```
int main() {  
    int f1 = ROT | BLAU;  
    int f2 = GRUEN | GELB;  
    int f3 = f1 | f2;  
    int f4 = f1 & f2;  
    int f5 = f1 & ROT;  
    testAll(f1,"f1");  
    testAll(f2,"f2");  
    testAll(f3,"f3");  
    testAll(f4,"f4");  
    testAll(f5,"f5");  
    return 0;  
}
```

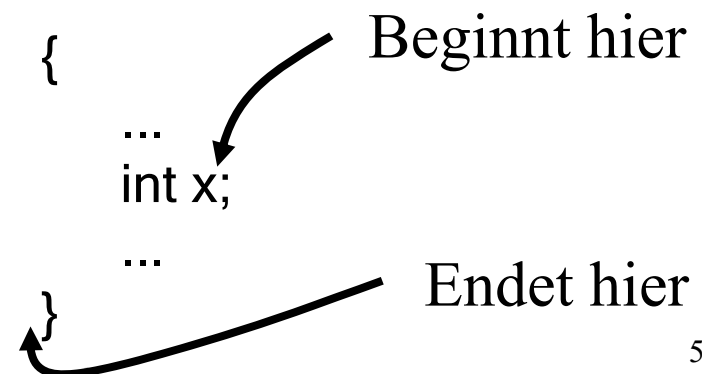
5 Mengen



# Vorlesung 3

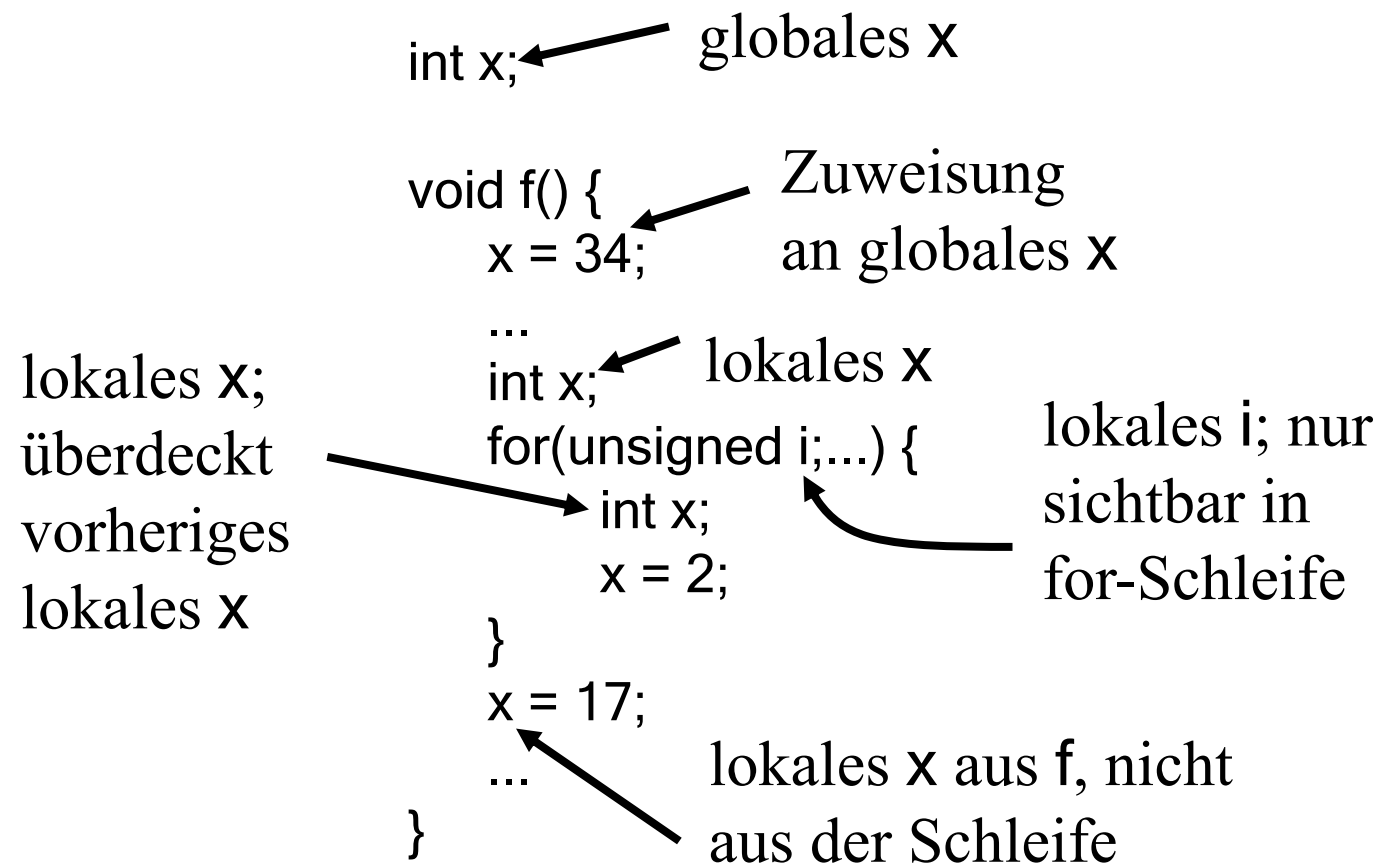
## Gültigkeitsbereiche

- wie in Java gibt es in C++ Regeln, die festlegen, ab wann eine Variable sichtbar ist und wie lange eine Variable sichtbar bleibt
- sind im C++ aber komplizierter, weil es
  - globale Variablen gibt
  - Variablen andere Variablen mit gleichem Namen überdecken können
- Gemeinsamkeit
  - eine Variable beginnt ihr Leben mit der Deklaration und endet mit dem Blockende



## Gültigkeitsbereiche (Forts.)

- Variablennamen können in geschachtelten Blöcken erneut definiert werden



## Gültigkeitsbereiche (Forts.)

- lokale Variablennamen in geschachtelten Blöcken zu überdecken ist schlechter Programmierstil
- globale Variablen sind schlechter Programmierstil
- dennoch ist es in Ordnung, globale Variablen durch lokale Variablen zu überdecken
- durch den Prefix Operator `::` kann zwischen lokalen und globalen Variablen unterschieden werden

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
int x; ← globales x
```

```
void f() {  
    int x; ← lokales x  
    x = 17; ← Zugriff auf lokales x  
    ::x = 23; ← Zugriff auf globales x  
    if (x != ::x)  
        cout << "die x'e sind ungleich\n";  
}
```


```
int main() {  
    f();  
    cout << x << endl; ← Zugriff auf globales x ohne ::  
    return 0;           weil es kein lokales x zum  
                        verwechseln gibt  
}
```

## Gültigkeitsbereiche (Forts.)

- der Variablenname ist sofort sichtbar
- folgendes Beispiel ist also unsinnig

```
int x;  
{  
    ...  
    int x = x;  
    ...  
}
```

Variable x wird durch sich selbst initialisiert: sinnlos



- Parameter können **nicht** überdeckt werden

```
5: void f(int x) {  
6:     int x;  
7: }
```

Sichtbarkeit2.cpp: In function `void f(int)':

Sichtbarkeit2.cpp:6: error: declaration of 'int x' shadows a parameter

# Typedefs

- oft werden Typen recht lang
- dadurch wird der Code recht schnell unleserlich
- mit Hilfe von **typedefs** können Typen andere Namen gegeben werden
- Beispiel:

```
typedef unsigned char uchar;  
uchar x;
```

- Wichtig: **uchar** ist kein neuer Typ, sondern nur ein Synonym für **unsigned char**
- Typedefs werden besonders in Verbindung mit Templates verwendet (später viel mehr)

```
typedef std::map<std::string,std::vector<unsigned int> > MyMap;  
MyMap x;
```

## Zeiger und Arrays

- Arrays sehen in C++ ähnlich wie in Java aus
- sie kennen jedoch nicht ihre Grenzen
- bei Zugriffen werden die Grenzen nicht überprüft
- wird außerhalb der Grenzen zugegriffen, ist das Verhalten undefiniert, das Programm kann abstürzen
- um Arrays in C++ verstehen zu können, muss man die Zeiger in C++ kennen
- Zeiger gibt es auch in Java, aber nicht als explizites Sprachkonstrukt



## Zeiger

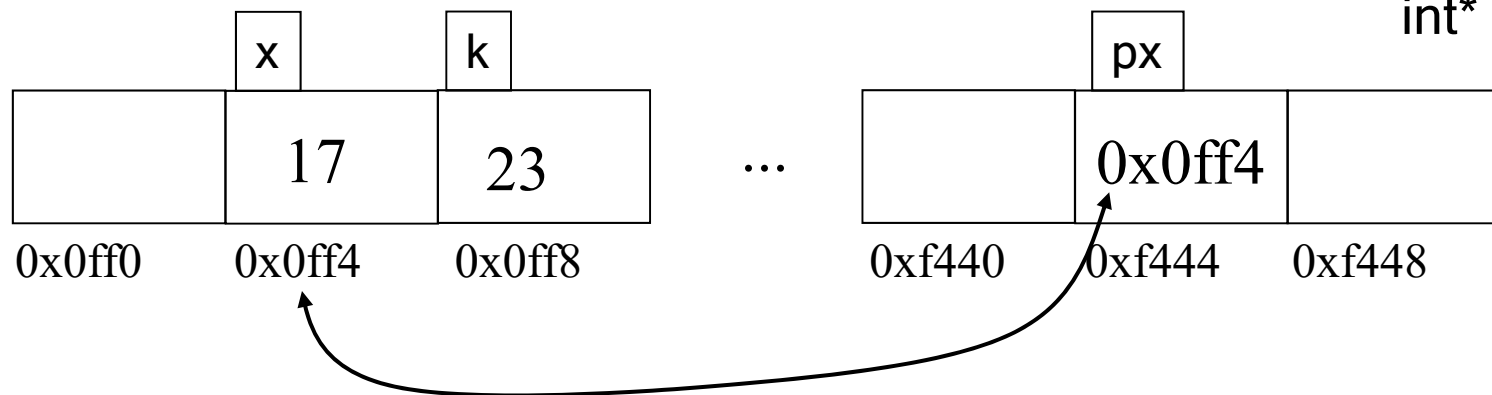
- jedes Objekt (oder auch Variable) lebt während des Programmablaufs an einer bestimmten Speicheradresse
- diese Speicheradresse kann man in C++ explizit in einer Variablen speichern
- dazu werden Zeiger (sogenannte Pointer) verwendet
- dabei kann eine Variable, die vom Typ Pointer ist, nicht auf alle Adresse zeigen
- ein Pointertyp muss genauer spezifizieren, auf was für Datenstrukturen er zeigt
- die Speicheradresse einer Variablen `x` erhält man durch den Präfixoperator `&`, d.h. `&x` ist die Adresse, in der `x` im Speicher liegt

Ohne Zeiger gut zu kennen, kann man nicht C++ (oder C) programmieren

## Zeiger (Forts.)

- `int x;`  
definiert ein Objekt mit Namen `x`, in dem ein `int`-Wert abgespeichert ist
- `int* y;`  
definiert ein Objekt mit Namen `y`, in dem die Speicheradresse eines anderen Objekts, in dem ein `int`-Wert abgespeichert ist, gespeichert wird

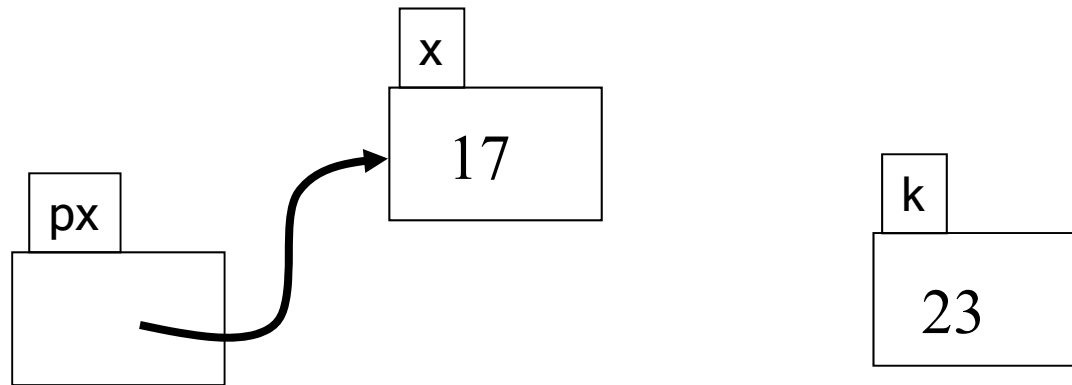
Speicher



```
int x = 17;  
int k = 23;  
int* px = &x;
```

## Zeiger (Forts.)

- eine andere Darstellungsart ist



```
int x = 17;  
int k = 23;  
int* px = &x;
```

- diese Darstellungsart zeigt deutlich, dass in Java alle Objekte und Arrays **immer** als Zeiger verwaltet werden
- in C++ kann hier unterschieden werden, ob eine Variable ein Objekt oder einen Zeiger auf ein Objekt speichert
- in Java kann eine Variable immer nur einen Zeiger auf ein Objekt, niemals das Objekt selber speichern

## Zeiger (Forts.)

- die Variable `px` bezeichnet die Adresse, die in `px` abgespeichert ist
- um das Objekt, auf das `px` zeigt, zu adressieren, muss `px` dereferenziert werden
- dies erfolgt durch den Präfixoperator `*`
- wenn `px` vom Typ `int*` (Zeiger auf ein `int`-Wert) ist, so ist `*px` vom Typ `int`

```
int x = 17;  
int k = 23;  
int* px = &x;
```

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int x = 17;  
    int y = 23;  
    int* p = &x;  
    cout << x << endl << y << endl << p << endl << *p << endl << endl;  

```

← zwei int Variablen

← eine Variable, die auf die int Variable x zeigt

```
    *p = 42;  
    p = &y;  
    *p = 45;
```

unterschiedliche  
Anwendungen von p

```
    cout << x << endl << y << endl << p << endl << *p << endl;  
    return 0;  
}
```

gebe die  
Adresse aus

gebe den Wert hinter  
der Adresse aus

## Zeiger (Forts.)

- in Java ist es **nicht möglich**, einer Methode (oder Funktion) eine int-Variable zu übergeben, so dass
  - die int-Variable in der Methode verändert wird
  - die Änderung außerhalb der Methode sichtbar ist
- in C++ kann dies erreicht werden, indem
  - nicht die int-Variable selber übergeben wird, sondern die Adresse, an der die int-Variable gespeichert ist
  - die Methode nicht eine int-Variable, sondern eine Adresse auf eine int-Variable erwartet
- diese Art der Übergabe nennt man **call-by-reference**
- werden stattdessen Werte übergeben, spricht man von **call-by-value**

Go!

## Beispiel

```
void f(int x,int y) {  
    if (x < y)  
        x = y;  
}
```

Funktion mit call-by-value

```
void g(int* x,int* y) {  
    if (*x < *y) // was passiert bei x < y  
        *x = *y;  
    /* der erste Kommentar und hier gleich  
       der zweite über mehrere Zeilen  
    */  
}
```

Funktion mit call-by-reference

```
int main() {  
    int a = 17;  
    int b = 23;  
    cout << a << " " << b << endl;  
    f(a,b);  
    cout << a << " " << b << endl;  
    g(&a,&b);  
    cout << a << " " << b << endl;  
    return 0;  
}
```

hier müssen die  
Adressen von x und  
y übergeben werden

## Zeiger (Forts.)

- versucht man einen Zeiger auf eine Variable falschen Typs zu setzen, gibt es einen Compilerfehler

```
6: int y = 23;
```

```
7: unsigned int* p;
```

```
8: p = &y;
```

Pointer3.cpp: In function `int main()':

Pointer3.cpp:8: error: invalid conversion from `int\*' to `unsigned int\*'

- Zeiger können nicht nur auf einfache Variablen zeigen, sondern auch auf
  - Funktionen und Methoden
  - Members
  - selber wieder auf Zeiger (Pointer to Pointer: wichtig in der Algorithmenvorlesung)



## Zeiger auf Funktionen

- Funktionen liegen im Speicher an einer bestimmten Adresse
- diese Adresse kann in einer Variablen gespeichert werden
- dazu muss der Variablen mitgeteilt werden, von welchem Typ (= Signatur) die Funktion ist
- Beispiel:

int f(char c, unsigned ui) hat die Signatur  
char×unsigned→int

- soll eine Variable x solche Funktionen speichern können, so muss sie wie folgt deklariert werden

int (\*x)(char,unsigned)  
x = &f;

- die Anwendung von x sieht dann wie folgt aus:

int y = (\*x)('a',34);

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
void f(int i) {  
    cout << "ich bin f und bekam " << i << endl;  
}
```

```
void g(int j) {  
    cout << "ich bin g und bekam " << j << endl;  
}
```

```
int main() {  
    void (*h)(int);  
    char c;  
    cin >> c;  
    h = (c == 'f') ? &f : &g;  
    (*h)(23);  
    return 0;  
}
```

Deklaration der  
Variablen h als Typ:  
Pointer auf Funktion  
vom Typ int→void

Aufruf der Funktion,  
die in h gespeichert ist

Go!

## Der nullptr-Zeiger

- möchte man explizit Variable vom Typ „Zeiger auf ...“ kennzeichnen, dass sie auf nichts zeigt, so verwendet man den 0-Zeiger bzw. nullptr
- der 0-Zeiger ist ein definierter Wert, der bedeutet: „kein gültiger Zeiger“

```
int main() {  
    int x = 45;  
    int* p;  
    cout << p << endl;  
    p = nullptr;  
    cout << p << endl;  
    // *p = 34; // hier gibt es einen Absturz  
    if (p == nullptr)  
        p = &x;  
    cout << p << endl << *p << endl;  
    return 0;  
}
```

nicht initialisierte Variable p

wohl-definiert: p hat keinen Wert

hat p eine gültige Adresse?

## Wilde Zeiger

- Zeiger können wieder auf Zeiger zeigen

```
int** x; // x zeigt auf einen Zeiger, der auf eine int-  
        // Variable zeigt
```

```
int* (**f)(char*) // f zeigt auf einen Zeiger, der auf eine  
                  // Funktion zeigt, die einen Zeiger auf  
                  // char (Zeichenfolge) übergeben  
                  // bekommt, und einen Zeiger auf einen  
                  // int-Wert zurückliefert
```

Einschätzung:

- Zeiger auf Zeiger kommen bei Algorithmen häufiger vor, ansonsten wenig
- Zeiger auf Funktionen oder auf Members kommen auch sehr selten vor

## Felder bzw. Arrays

- anders als in Java ist ein Array in C++ einfach ein Zeiger in den Speicher, ab dessen Adresse in sequentieller Abfolge Elemente des gleichen Typs gespeichert werden können
- anders als in Java merkt sich ein Array in C++ nicht seine Länge
- auch wird keine Bereichsüberprüfung beim Lese- bzw. Schreibzugriff durchgeführt

- Beispiele:

```
char x[15]; // Array von 15 char-Werten, indiziert von 0
           // bis 14
```

```
int* x[4]; // Array von 4 Pointer auf int-Werten,
           // indiziert von 0 bis 3
```

- wichtig: die Feldgrößen müssen zur Compilezeit konstant und bekannt sein

Go!

## Beispiel

```
int* f(int x[]) {  
    return &x[3];  
}
```

Funktion: bekommt ein int-Array übergeben,  
liefert einen Zeiger auf eine int-Variable zurück

```
void print(int x[]) {  
    for(unsigned ui = 0; ui < 15; ++ui)  
        cout << x[ui] << " ";  
    cout << endl;  
}
```

große Hoffnung: lass das Array  
15 Elemente enthalten

```
int main() {  
    int x[15];  
    for(unsigned ui = 0; ui < 15; ++ui)  
        x[ui] = 2 * ui;  
    int* y = f(x);  
    print(x);  
    *y = 34;  
    print(x);  
    return 0;  
}
```

Wichtig: die Größe muss zur  
Compilezeit bekannt sein

Seiteneffekt: hier wird x verändert

kein schöner Programmierstil

## Felder bzw. Arrays (Forts.)

- Mehrdimensionale Arrays sehen analog zu Java aus und werden definiert als

`int x[20][10];`      intern werden sie eindimensional !!!

- `x` ist ein Array mit 200 (=20x10) Einträgen von `int`-Werten
- der Zugriff erfolgt durch die hintereinander Selektion der verschiedenen Dimensionen

`x[5][3] = 35;`

- Arrays können bei der Deklaration direkt initialisiert werden

`int x[] = {3, 17, 42};`

- hier kann zusätzlich die Größe angegeben werden

`int x[3] = {3, 17, 42}; // ok`

`int y[5] = {3, 17, 42}; // ok, Rest wird 0 aufgefüllt`

`int z[2] = {3, 17, 42}; // Fehler`

Go!

## char Arrays

- Arrays von **chars** werden speziell behandelt
- **char** Arrays haben ein Extra Zeichen am Ende, dass mit `\0` gefüllt wird
- somit hat man die Möglichkeit, durch eine Zeichenkette zu laufen und das Ende zu erreichen, ohne die Länge der Zeichenkette zu kennen

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    bool x[5] = {true,true,false,true,true};  
    char c[] = "world";  
    cout << sizeof(x) << endl << sizeof(c) << endl;  
    return 0;  
}
```



Go!

## char Arrays (Forts.)

- Zeichenketten im Code der Form "world" sind vom Typ `const char[6]`
- Somit können sie Variablen vom Typ `char[]` zugewiesen werden
- dabei wird die Zeichenkette kopiert

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    char c[] = "world";  
    char d[] = "world";  
    cout << c << endl << d << endl;  
    c[2] = 'O';  
    cout << c << endl << d << endl;  
    return 0;  
}
```

## Zusammenhang: Arrays und Pointer

- Im Grunde genommen unterscheiden sich Pointer und Arrays in C++ nicht
- auch Arrays enthalten eine Speicheradresse, ab der in sequentieller Abfolge die Elemente des Arrays abgelegt sind
- somit können Variablen von Arrays und von Pointer untereinander zugewiesen werden

Go!

## Beispiel

```
void print(int h[]) {  
    for(unsigned ui = 0; ui < 3; ++ui)  
        cout << h[ui] << " ";  
    cout << endl;  
}
```

```
int main() {
```

```
    int x[] = {3,17,43};
```

```
    int* y = nullptr;
```

```
    cout << x << endl << y << endl;
```

```
    y = x;
```

```
    print(x);
```

```
    print(y);
```

```
    *x = 5;
```

```
    cout << x << endl << y << endl;
```

```
    print(x);
```

```
    print(y);
```

```
    y[2] = -66;
```

```
    print(x);
```

```
    print(y);
```

```
    return 0;
```

```
}
```

Array und Pointer:  
kein Unterschied

Array dereferenziert  
wie einen Pointer

Pointer indiziert  
wie ein Array

Go!

## Zusammenhang: Arrays und Pointer (Forts.)

- Vorsicht bei Zeichenketten
- der Compiler kann nicht erkennen, dass ein Zeiger in eine konstante Zeichenkette zeigt
- wird über solch einen Pointer eine konstante Zeichenkette verändert, ist das Ergebnis undefiniert

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    char* c = "world";
```

```
    char* d = c;
```

```
    cout << c << endl << d << endl;
```

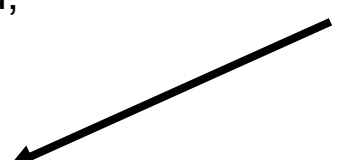
```
    c[2] = 'O';
```

```
    cout << c << endl << d << endl;
```

```
    return 0;
```

```
}
```

c und d sind Pointer  
auf die konstante  
Zeichenkette "world"



nichtdefiniertes Verhalten

## Navigation durch Arrays

- es gibt in C++ zwei Arten, wie durch Arrays navigiert werden kann
- der Standardweg: über den Index

```
int i[4];  
i[3] = 16;
```
- da Arrays aber auch als Pointer gesehen werden können, kann über den Dereferenzierungsoperator `*` auf den Inhalt des Arrayzellen zugegriffen werden
- für einen Pointer existieren auch die Post- und Preinkrement- bzw. –dekrementoperatoren
- mit Hilfe dieser kann ein Arrays Durchlaufen werden

Go!

## Beispiel

```
void print(int h[]) {  
    for(unsigned ui = 0; ui < 6; ++ui)  
        cout << h[ui] << " ";  
    cout << endl;  
}
```

```
int main() {  
    int x[] = {3,17,43,-34,1000,0};  
    int* p = x;  
    print(x);  
    print(p);  
    cout << x[3] << endl;  
    cout << *(p+3) << endl;  
    *(p+3) = 18;  
    cout << x[3] << endl;  
    cout << *(p+3) << endl;  
    print(x);  
    print(p);  
    return 0;  
}
```

Array und Pointer:  
kein Unterschied

kryptische Darstellung: auf die  
Adresse, die in **p** steht, addiere  
3 Einheiten; von dieser neuen  
Adresse nehme den Inhalt  
(lesend oder schreibend)

Diese Art der Anwendung ist wenig  
sinnvoll und soll eher abschrecken

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int x[] = {3,17,43,-34,1000,0};
```

```
    for(unsigned ui = 0; x[ui] != 0; ++ui)
```

```
        cout << x[ui] << " ";
```

```
    cout << endl;
```

```
    for(int* p = x; *p != 0; ++p)
```

```
        cout << *p << " ";
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

Spezialfall: im  
Array ist das eigene  
Ende markiert  
durch „0“

p merkt sich nacheinander die  
Adressen der aufeinander  
folgenden Zellen des Arrays x

Diese Art der Anwendung kann sinnvoll  
sein, **aber** es ist ein Spezialfall

Go!

## Beispiel

- der vorherige Spezialfall trifft bei Zeichenketten jedoch zu
- es sind alle Zeichenketten um ein Zeichen erweitert
- dieses zusätzliche Zeichen enthält den Wert \0

```
#include <iostream>
```

```
using namespace std;
```

```
void print(const char* v) {  
    for(const char* p = v; *p != '\0'; ++p)  
        cout << *p;  
}
```

```
int main() {  
    print("Hello ");  
    print("World\n\n");  
    return 0;  
}
```

Diese Art der Anwendung ist sinnvoll, weil **char** Arrays mit dem Zeichen \0 abschließen



# Vorlesung 4

# Referenzen

- anders als in Java gibt es in C++ Referenzen
- diese gab es noch nicht in C
- Referenzen sind alternative Namen für Objekte
- eine Referenz kann nur erzeugt werden, wenn bei ihrer Erzeugung das Objekt (oder Variable), für das es ein alternativer Name sein soll, schon existiert
- Referenzen werden in guten C++ Programmen sehr häufig gebraucht
- Referenzen ersetzen oft Zeiger (später mehr dazu)
- Referenzen können nicht ihren Wert ändern, d.h. sie zeigen immer auf das gleiche Objekt

## Referenzen (Forts.)

- Referenzen werden deklariert, indem nach dem Typ das & Zeichen gesetzt wird
- Beispiel:  
    `int x;`  
    `int& y = x;`
- die Variable y ist eine
  - Referenz auf x
  - ein Synonym für x
  - ein alternativer Name für x
- Referenzen müssen daher immer initialisiert sein, um sicherzustellen, dass sie alternative Namen für existierende Objekte sind

Go!

## Referenzen (Forts.)

- mit einer Referenz kann im folgenden ganz normal gearbeitet werden
- anders als bei Pointern muss kein unärer Operator vor die Variable gesetzt werden, um auf den eigentlichen Inhalt zuzugreifen, auf den die Referenz zeigt

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int x = 45;  
    int& y = x;  
    cout << x << " " << y << endl;  
    y = 34;  
    cout << x << " " << y << endl;  
    return 0;  
}
```

x und y sind das  
selbe Objekt

Zugriff auf y ohne  
zusätzliche Operatoren

Go!

## Referenzen (Forts.)


- der lesende Zugriff einer Referenz liefert den Wert, der in dem Objekt gespeichert ist, auf den die Referenz verweist
- es wird nicht die Adresse zurückgeliefert

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int x = 45;  
    int& y = x;  
    int z = y;  
    cout << x << " " << y << " " << z << endl;  
    y = 34;  
    cout << x << " " << y << " " << z << endl;  
    return 0;  
}
```

der jetzt aktuelle  
Wert von y (=x)  
wird in z kopiert



## Referenzen (Forts.)

- eine Referenz kann nicht auf eine neue Variable gesetzt werden
- sie ist immer mit dem bei der Initialisierung angegebenen Objekt assoziiert

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int x = 45;  
    int& y = x;  
    int z = 17;  
    y = z;
```

kopiert den  
Wert von z  
(=17) in y (=x)

```
    y = &z;  
    return 0;
```

Ref7.cpp: In function `int main()':  
Ref7.cpp:10: error: invalid conversion from `int\*' to `int'

```
}
```

Go!

## Referenzen (Forts.)

- Referenzen werden sehr oft für die Parameter an und von Funktionen / Methoden genutzt


```
void makeSame(int& a, int& b) {  
    if (a < b)  
        a = b;  
    else  
        b = a;  
}
```

nicht int-Werte,  
sondern int-Variablen  
werden übergeben



```
int main() {  
    int x = 45;  
    int y = 16;  
    cout << x << " " << y << endl;  
    makeSame(x, y);  
    cout << x << " " << y << endl;  
    return 0;  
}
```

Übergabe ganz  
normal, weil  
lesender Zugriff



## Referenzen (Forts.)

- diese Art der Parameterübergabe nennt man

**call-by-reference**

```
void print(int& r) {...}
```

- im Gegensatz zu der Parameterübergabe durch Werte, die man

**call-by-value**

nennt

```
void print(int i) {...}
```



Go!

## Referenzen (Forts.)

- einen ähnlichen Effekt hätte man auch mit Pointer erreichen können, jedoch sieht die Anwendung nicht schön aus

```
void makeSame(int* a, int* b) {  
    if (*a < *b)  
        *a = *b;  
    else  
        *b = *a;  
}
```

nicht int-Wert, sondern Pointer auf  
int-Variablen werden übergeben

überall muss dereferenziert werden

```
int main() {  
    int x = 45;  
    int y = 16;  
    cout << x << " " << y << endl;  
    makeSame(&x, &y);  
    cout << x << " " << y << endl;  
    return 0;  
}
```

Adressen müssen  
übergeben werden

Welche Gefahr besteht bei der  
Pointer Lösung, die nicht bei  
Referenzen lauert?

Go!

## Referenzen (Forts.)

```
void makeSame(int* a,int* b) {  
    // assert(a != nullptr);  
    // assert(b != nullptr);
```

1. Möglichkeit: gehe davon aus, dass niemals der nullptr-Pointer übergeben wird

```
    if (a != nullptr && b != nullptr) {  
        if (*a < *b)  
            *a = *b;  
        else  
            *b = *a;  
    }  
}
```

2. Möglichkeit: fange den Fall der nullptr-Pointer ab (wenn es geht)

```
int main() {  
    int x = 45;  
    int y = 16;  
    cout << x << " " << y << endl;  
    makeSame(nullptr,&y);  
    cout << x << " " << y << endl;  
    return 0;  
}
```

nullptr-Pointer

Beide Möglichkeiten sind keine sauberen Lösungen. Merke: ist der nullptr-Pointer kein gültiger Wert, ist der Pointer Typ falsch, die Referenz richtig

## Referenzen (Forts.)

- Referenzen können auch von Funktionen / Methoden zurückgeliefert werden
- Vorsichtig: niemals Referenzen von lokalen Variablen zurückgeben (warum?)
- damit kann das Ergebnis einer Funktion / Methode auf der linken Seite einer Zuweisung stehen
- dies ist in Java nur für Objekte und Arrays möglich, in C++ für jeden Typen
- Beispiel:

`doSomethingMagic(17,x,y) = 23;`

Go!

## Referenzen (Forts.)

```
#include <iostream>
```

```
using namespace std;
```

```
//int select(int& a,int& b,bool c) {  
//int& select(int a,int b,bool c) {  
int& select(int& a,int& b,bool c) {  
    return c ? a : b;  
}
```

Wichtig: Referenz  
zurückliefern

In Abhängigkeit von **b** wird die Variable  
**x** oder **y** (nicht die Werte) zurückgeliefert

```
int main() {  
    int x = 45;  
    int y = 16;  
    cout << x << " " << y << endl;  
    select(x,y,true) = 23;  
    cout << x << " " << y << endl;  
    select(x,y,false) = 17;  
    cout << x << " " << y << endl;  
    return 0;  
}
```

Methodenaufruf liefert Variable  
zurück, daher linke Seite einer  
Zuweisung möglich

## Referenzen (Forts.)

- analog zu der Parameterübergabe spricht man bei der Rückgabe von Funktionen mittels Referenzen von

**return-by-reference**

```
int& generate(...) {...}
```

- im Gegensatz zu der Rückgabe von Werten, die man

**return-by-value**

```
int generate(...) {...}
```

nennt

## Referenzen: Schlussbemerkung

- in fast allen Funktionen / Methoden werden die Parameter als Referenzen übergeben
- in Java passiert dies (zwingend) nur für Objekte und Arrays
- in C++ **können** Objekte als Referenz übergeben werden, **müssen** aber **nicht**
- die Rückgabe von Referenzen und damit die Möglichkeit, auf der linken Seite einer Zuweisung zu stehen, wird sehr oft bei selbstdefinierten Operatoren verwendet (siehe später)

Beispiel: `a[45] = 67;`

- wenn möglich, verwende Referenzen statt Pointer

## Struct's und Union's

- in C gab es keine Klassen
- um verschiedene Datenelemente zu einem neuen Datenelement zusammenzufassen gibt es sogenannte Strukturen (Schlüsselwort: **struct**)
- diese führen selbstdefinierte Typen ein, die später für Variablendeklarationen verwendet werden können
- Beispiel:

```
struct Adresse {  
    char* name;  
    char* vorname;  
    unsigned matrikelnummer;  
};
```

```
Adresse a;
```

# Strukturen

- um auf die verschiedenen Elemente einer Struktur zuzugreifen (lesend/schreiben) bedient man sich der „.“ Notation (analog zu Klassen)

- Beispiel:

```
cout << a.name;
```

```
a.matrikelnummer = 23467;
```

- im Gegensatz zu Java legt die Anweisung

```
Adresse a;
```

bereits ein **Adresse** Objekt an

```
struct Adresse {  
    char* name;  
    char* vorname;  
    unsigned matrikelnummer;  
};
```

```
Adresse a;
```

!!!

- hier muss nicht mehr durch new ein Objekt erzeugt werden



Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
struct Adresse {  
    const char* name;  
    unsigned matrikelnummer;  
};
```

neuer,  
selbstdefinierter Typ

```
int main() {
```

```
    Adresse a;
```

```
    a.name = "otto";
```

```
    a.matrikelnummer = 32564;
```

```
    cout << a.name << " " << a.matrikelnummer << endl;
```

```
    return 0;
```

```
}
```

erzeugt schon gleich ein Objekt von  
Adresse; kein **new** notwendig;  
sichtbar bis zum Ende des Blocks

Go!

## Strukturen (Forts.)

```
struct Adresse {  
    const char* name;  
    unsigned matrikelnummer;  
};  
  
void print(Adresse* p) {  
    if (p != nullptr)  
        cout << "mit pointer " << (*p).name << " " << (*p).matrikelnummer << endl;  
}  
  
void print(Adresse& r) {  
    cout << "mit referenz " << r.name << " " << r.matrikelnummer << endl;  
}  
  
int main() {  
    Adresse a;  
    a.name = "otto";  
    a.matrikelnummer = 32564;  
    print(&a);  
    print(a);  
    return 0;  
}
```

wie auf alle anderen Typen kann  
man natürlich auch auf Strukturen  
Pointer und Referenzen setzen

Überladung  
einer Funktion

anhand des Argumenttyps  
wird die richtige Funktion  
identifiziert

## Strukturen (Forts.)

- Strukturen sind ein Überbleibsel aus C
- dort gab es keine Referenzen
- daher hat man oft Pointer auf Strukturen
- der Zugriff auf die einzelnen Elemente über einen Pointer mittels der Syntax

*(\*pointer\_name).element\_name*

ist umständlich

- daher gibt es die syntaktische Möglichkeit

*pointer\_name -> element\_name*

```
Adresse a;  
Adresse* p = &a;  
a.name = "otto";  
a.matrikelnummer = 32564;  
cout << p->name << " " << p->matrikelnummer << endl;
```

## Strukturen (Forts.)

- Strukturen haben analog zu Klassen Konstruktoren und Destrukturen
- sie können dazu genutzt werden, die Strukturen zu initialisieren
- anders als in Java gibt es in C++ keinen Garbage Collector
- der Destruktur wird aufgerufen, wenn das Objekt verschwindet (am Ende des Blocks, in dem es deklariert ist)
  - in Java sind Destrukturen unwichtig
  - in C++ sind sie eine der wichtigsten Methoden überhaupt
  - sie sollten den Speicher aufräumen (später viel mehr dazu)

Go!

## Beispiel

```
struct Adresse {  
    Adresse(const char* p,unsigned nr) {  
        name = p;  
        matrikelnummer = nr;  
    }  
    ~Adresse() {  
        cout << "ich bin tot\n";  
    }  
    const char* name;  
    unsigned matrikelnummer;  
};  
  
void print(Adresse& r) {  
    cout << r.name << " " << r.matrikelnummer << endl;  
}  
  
int main() {  
    Adresse a("otto",32564);  
    print(a);  
    return 0;  
}
```

Konstruktor führt Initialisierung durch

Destruktor wird am Lebensende des Objekts aufgerufen

a ist lokal deklariert

hier stirbt a

Go!

## Strukturen (Forts.)

- vorsichtig bei der Deklaration von Strukturen
- es ist nicht egal, in welcher Reihenfolge die Elements deklariert werden

```
struct S1 {  
    bool b1;  
    int i;  
    bool b2;  
};
```

```
struct S2 {  
    bool b1;  
    bool b2;  
    int i;  
};
```

```
int main() {  
    cout << sizeof(S1) << " " << sizeof(S2) << endl;  
    return 0;  
}
```

beide Strukturen können  
das gleiche speichern,  
doch die eine ist viel  
speicherplatzeffizienter

## Vereinigungen

- aus C stammen (zu Zeiten, in denen Speicher sehr knapp war) stammt noch die Möglichkeit, Datenelemente zu überlagern
- gibt es in einer Struktur zwei (oder mehr) Datenelemente, die niemals gleichzeitig benutzt werden, so kann man sie mittels eines `union` zusammenfassen
- der benötigte Speicherplatz ist nicht gleich der Summe der Elemente sondern das Maximum der Elemente
- Problem: beim Programmieren muss man wissen, was in dem Union gespeichert ist, um auf das richtige Element zuzugreifen
- ein Fehlzugriff führt garantiert zu falschen Ergebnissen, oft zum Absturz
- werden in C++ wegen Klassen und Polymorphie so gut wie nicht mehr verwendet

Go!

## Beispiel

```
enum TreeType {Leaf, Node};
```

```
struct TreeNode;
```

```
struct InternalNode {
```

```
    TreeNode* left;
```

```
    TreeNode* right;
```

```
};
```

```
struct TreeNode {
```

```
    TreeType type;
```

```
    union {
```

```
        InternalNode node;
```

```
        /*long long*/ unsigned content;
```

```
    };
```

```
};
```

```
int main() {
```

```
    cout << sizeof(TreeNode) << endl;
```

```
    TreeNode t;
```

```
    cout << t.node.left << " " << t.node.right << " " << t.content << endl;
```

```
    t.content = 0;
```

```
    cout << t.node.left << " " << t.node.right << " " << t.content << endl;
```

forward Deklaration: ist in C und C++ anders als in Java für rekursive Datenstruktur notwendig

ein Baumknoten hat immer einen **type**, aber entweder einen **content** oder ist ein **InternalNode** (hat also einen **node**)



# Vorlesung 5

# Konstanten

- in Java drückt das Schlüsselwort **final** aus, dass gewisse Elemente „konstant“ sind, nicht mehr verändert werden können
- in C++ wird dies durch das Schlüsselwort **const** ausgedrückt
- jedoch gibt es einige Unterschiede zwischen den Konstanz Konzepten in Java und C++
- das Schlüsselwort wird bei der Deklaration von Variablen, Membern, Parametern und Methoden verwendet
- anders als in Java kann eine Klasse **nicht** als konstant deklariert werden (Verhindern einer weiteren Ableitung)

Go!

## Konstanten (Forts.)

- anders als in Java muss eine konstante Variable bei der Deklaration bereits initialisiert werden

`const int i = 17;`

- dies bedeutet, dass `i` eine Konstanten ist, die niemals mehr verändert werden kann
- `const` in Verbindung mit Referenzen bedeutet ebenfalls, dass die Variable (bzw. das Objekt), auf die die Referenz verweist, nicht verändert werden kann

`j` kann nicht  
mehr verändert  
werden, `i` schon

```
int main() {  
    int i;  
    const int& j = i;  
    cout << j << endl;  
    i = 17;  
    cout << j << endl;  
    // j = 23;  
    return 0;  
}
```

wäre ein  
(statischer)  
Semantik-  
fehler

Go!

## Konstanten (Forts.)

- ähnlich wie bei Referenzen verhält es sich mit `const` und Pointern

```
int i;
```

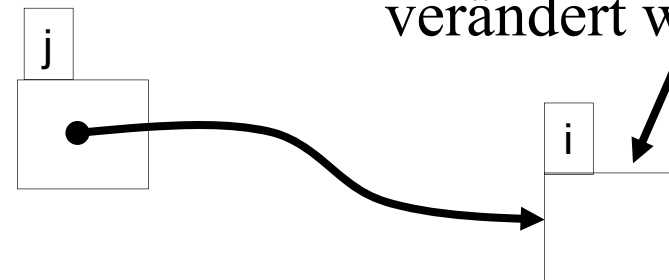
```
const int* j = &i;
```

- auch hier bedeutet das `const`, dass der Inhalt, auf den der Pointer `j` zeigt, nicht verändert werden kann
- aber `j` selber kann verändert werden

```
int main() {  
    int i,z;  
    const int* j = &i;  
    cout << *j << endl;  
    i = 17;  
    cout << *j << endl;  
    // *j = 43;  
    j = &z;  
    cout << *j << endl;  
    return 0;  
}
```

`j` selber kann verändert  
werden, der Inhalt, auf  
den `j` zeigt aber nicht

kann über `j` nicht  
verändert werden



Go!

## Konstanten (Forts.)

- es kann aber auch spezifiziert werden, dass der Pointer selber nicht verändert werden darf

```
int i;
```

```
int* const j = &i;
```

- der Inhalt, auf den der Pointer j zeigt, kann jetzt verändert werden

j selber kann *nicht*  
verändert werden, der  
Inhalt, auf den j zeigt  
aber schon

```
int main() {  
    int i,z;  
    int* const j = &i;  
    cout << *j << " " << i << endl;  
    i = 17;  
    cout << *j << endl;  
    *j = 43;  
    // j = &z;  
    cout << *j << " " << i << endl;  
    return 0;  
}
```



kann nicht verändert werden

Go!

## Konstanten (Forts.)

- zusammengesetzt bewirkt es, dass weder der Pointer noch der Inhalt des Pointers verändert werden kann

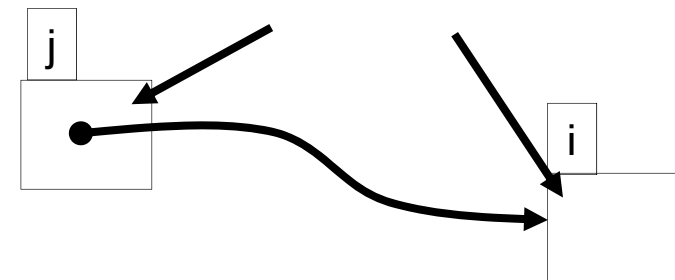
```
int i;
```

```
const int* const j = &i;
```

```
int main() {  
    int i,z;  
    const int* const j = &i;  
    cout << *j << " " << i << endl;  
    i = 17;  
    cout << *j << endl;  
    // *j = 43;  
    // j = &z;  
    return 0;  
}
```

weder j noch der  
Inhalt, auf den j zeigt  
kann verändert werden

können nicht  
verändert werden



## Konstanten (Forts.)

- Wie kann man sich das merken?

`const int* j = &i;`    `const` ist dichter an `int`  
als an `*`, daher  
konstante `int` Variable

`int* const j = &i;`    `const` ist dichter an `*`  
als an `int`, daher  
konstanter Pointer

Eselsbrücke: wo `const` näher ist, darauf bezieht es sich

## Konstanten (Forts.)

- da Arrays auch nur Pointer sind, kann man sehr genau steuern, ob ein übergebenes Array verändert werden darf oder nicht

<code>void f(const int a[]) {...}</code>	a darf verändert werden, Arrayinhalte
<code>void g(const int* a) {...}</code>	aber nicht

- das `const` bezieht sich auf das `int`, d.h. der Inhalt von `a` kann *nicht* verändert werden

- dies ist anders als in Java !!!

<code>void f(final int[] a) {...}</code>	a darf nicht verändert werden, Arrayinhalte aber schon
--	--

- dies ist deutlich besser als in Java !!!



Go!

## Beispiel

```
void f(const int a[]) {  
    cout << a[0] << endl;  
    // a[0] = 23; ← illegale  
    int j[] = {3, 5, 89, 34, 1};  
    a = j;           Schreibzugriffe  
}                  auf das Array a;  
void g(const int* a) {  
    cout << a[0] << endl;  
    // a[0] = 23; ← illegale  
    int j[] = {3, 5, 89, 34, 1};  
    a = j;           der Parameter a  
}                  darf aber  
                  verändert werden  
  
int main() {  
    int i[] = {13, -9, 12};  
    f(i);  
    g(i);  
    return 0;  
}
```

Go!

## Konstanten (Forts.)

- das Verhalten von Java (konstanter Parameter, veränderbares Array)

`void f(final int[] a) {...}`

kann in C++ nur mit Pointerschreibweise, nicht mit Arrayschreibweise erreicht werden

```
//void f(int a[] const) {}  
//void f(int a const []) {}
```

beides sind  
Syntaxfehler

entspricht dem  
Java **final int[] a**

```
void g(int* const a) {  
    cout << a[0] << endl;  
    a[0] = 23;  
    int j[] = {3, 5, 89, 34, 1};  
    // a = j;  
}
```

Inhalt des Arrays kann  
verändert werden

der Parameter nicht

## Konstanten und Strukturen / Klassen

- anders als in Java erzeugt die Deklaration

`A x; // A ist eine Klasse oder Struktur`

bereits ein Objekt der Klasse (oder Struktur) `A` in `C++`

- die Funktionsdeklaration

`void f(A p) { ... }`

`f(x);`

legt eine lokale Kopie von `x` beim Funktionsaufruf `f` in `p` an

- dies kostet Zeit und Speicherplatz
- daher wird oft das Objekt als Referenz übergeben

`void f(A& p) {...}`

`f(x);`

## Konstanten und Strukturen / Klassen (Forts.)

- um sicherzustellen, dass der Funktionsaufruf `f` das Objekt `x` nicht verändern kann, wird die Referenz als `const` deklariert

```
void f(const A& p) {...}  
f(x);
```


- es sind somit nur lesende Zugriffe auf `p` möglich
- um sicherzustellen, dass ein Methodenaufruf von `p` das übergebene `x` Objekt nicht verändert, dürfen nur konstante Methoden aufgerufen werden

```
void f(const A& p) {... p.doit() ...}
```

- konstante Methoden werden mit `const` deklariert

```
class A {  
    void doit() const {...}  
}
```

nur lesende Zugriffe auf  
Member, nur Aufrufe von  
`const` Methoden



## Dynamische Speicherverwaltung

- analog zu Java wird in C++ mittels des **new** Kommandos Speicherplatz auf dem Heap alloziert
- anders als in Java gibt es in C++ kein Garbage Collector, d.h. jeder Speicherplatz, der mit **new** alloziert worden ist, muss mittels des Kommandos **delete** wieder freigegeben werden
- mit **new** kann Platz für einzelne Objekte aber auch für Arrays angelegt werden
- in jedem Fall liefert **new** einen Pointer auf den allozierten Speicherplatz zurück

Klingt einfach, ist aber in  
der Praxis recht kompliziert

Go!

## Beispiel

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int* p = nullptr;  
    int i,j;  
    cout << p << endl;  
    p = &i;  
    cout << p << endl;  
    p = &j;  
    cout << p << endl;  
    p = new int;  
    cout << p << endl;  
    p = new int;  
    cout << p << endl;  
    return 0;  
}
```

← p kann auf lokale  
Variablen zeigen ...

... oder auf allozierten  
Speicher vom Heap

Go!

## Dynamische Speicherverwaltung (Forts.)


- der allozierte Speicher ist nicht initialisiert
- vor dem ersten lesenden Zugriff muss er mit Werten gefüllt werden, ansonsten ist das Ergebnis undefiniert
- Vorsicht: nicht Pointer auf lokale Variablen aus Funktionen zurückliefern, diese sind nach dem Funktionsaufruf ungültig

```
int* gen(int j) {  
    int i = j;  
    return &i;  
}
```

ganz schlechte Idee  
— wäre &j besser?

```
int main() {  
    int* p = gen(17);  
    cout << p << " " << *p << endl;  
    p = new int;  
    cout << p << " " << *p << endl;  
    p = new int;  
    cout << p << " " << *p << endl;  
    ..  
}
```

p wird mit einer  
illegalen Adresse  
initialisiert



Go!

## Dynamische Speicherverwaltung (Forts.)

- Speicher, der mit **new** alloziert worden ist, muss mit **delete** freigegeben werden
- Schwierigkeit: freigeben, wenn die Adresse noch bekannt ist, aber der Speicherplatz nicht mehr gebraucht wird

```
int main() {  
    int* p = new int;  
    int* q = nullptr;  
    cout << q << " " << p << " " << *p << endl;  
    *p = 16;  
    cout << q << " " << p << " " << *p << endl;  
    delete p;  
    q = new int;  
    cout << q << " " << p << " " << *p << endl;  
    return 0;  
}
```

korrekte Zugriffe

nicht gut: **\*p** ist uninitialisiert

alles korrekte Zugriffe

ganz schlecht: **\*p** zeigt auf ungültigen Speicher



## Klassisches Problem der Dynamischen Speicherverwaltung

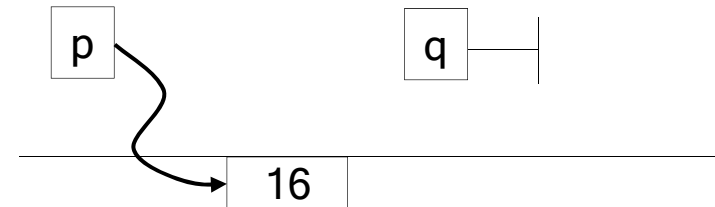
- Speicher (S1) wird mit new alloziert und beschrieben und gelesen
- Speicher (S1) wird freigegeben
- für andere Variable wird Speicher (S2) alloziert
- dieser Speicher (S2) überdeckt teilweise den alten Speicher (S1)
- der neue Speicher (S2) wird beschrieben
- über die alte Variable wird der alte Speicher (S1) gelesen
- man erhält einen undefinierten, seltsamen Wert, aber keinen Absturz, weil man auf allozierten Speicher zugreift
- das Programm läuft weiter, rechnet aber falsch
- es stürzt u.U. sehr viel später ab

Sehr schwer zu findender Fehler

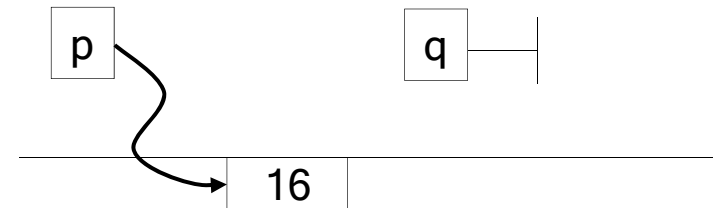
Go!

## Beispiel

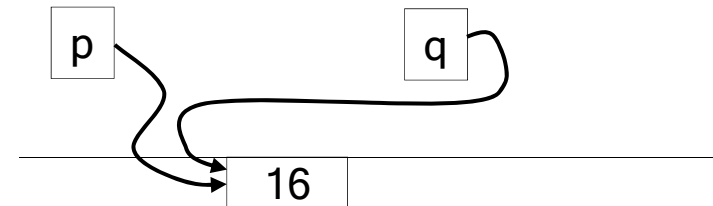
```
int main() {  
    int* p = new int;  
    float* q = nullptr;  
    cout << q << " " << p << " " << *p << endl;  
    *p = 16;  
    cout << q << " " << p << " " << *p << endl;  
}
```



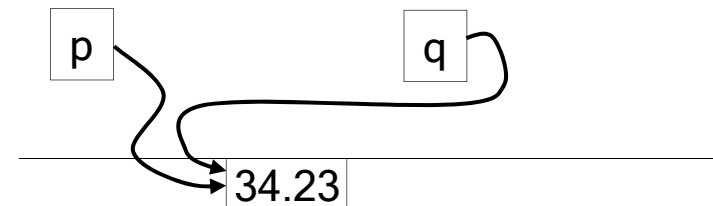
```
delete p;
```



```
q = new float;
```



```
*q = 34.23;  
cout << q << " " << p << " " << *p << endl;  
return 0;
```




Go!

## Klassisches Problem der Dyn. Speicherverwaltung (Forts.)

- Um diesen Fehler zu verhindern, sollte ein Pointer nach dem **delete** immer auf 0 gesetzt werden
- dies behebt den Fehler nicht, aber führt zu einem Absturz
- dieser ist leichter in den Debug Tools nachzuvollziehen

```
int main() {  
    int* p = new int;  
    float* q = nullptr;  
    cout << q << " " << p << " " << *p << endl;  
    *p = 16;  
    cout << q << " " << p << " " << *p << endl;  
    delete p;  
    p = nullptr;  
    q = new float;  
    *q = 34.23;  
    cout << q << " " << p << " " << *p << endl;  
    return 0;  
}
```

führt zu einem  
Programmabsturz



## Allokation von Arrays

- analog zu Java werden Arrays dynamisch dadurch erzeugt, dass nach dem **new** die Anzahl der Elemente angegeben werden

`int* p = new int[20];`

- da man dem Typen `int*` nicht ansehen kann, ob hinter dem Pointer ein einfaches Objekt oder ein Array von Objekten gespeichert ist, muss dem `delete` Operator dies mitgeteilt werden

`delete[] p;`

- es ist ein Fehler, `delete[] p` aufzurufen, wenn `p` nicht auf ein Array zeigt

Go!

## Beispiel

```
void print(int a[], unsigned len) {  
    for(unsigned ui = 0; ui < len; ++ui)  
        cout << a[ui] << " ";  
    cout << endl;  
}
```

```
int main() {  
    unsigned len = 0;  
    cout << "Laenge eingeben: ";  
    cin >> len;  
    int* p = new int[len];  
    print(p, len);  
    for(unsigned ui = 0; ui < len; ++ui)  
        p[ui] = 17 - ui * 2;  
    print(p, len);  
    delete[] p;  
    return 0;  
}
```

die Größe des Arrays  
muss zur Laufzeit nicht  
konstant sein

## Allokation von Arrays (Forts.)

- das vorherige Beispiel hat gezeigt, dass die Elemente des `int`-Arrays bei der Erzeugung nicht initialisiert werden
- aber  
genau wie bei Java werden auch in C++ die Arrayelemente bei der Erzeugung initialisiert  
nur  
`int`-Variablen werden (wie alle elementaren Typen) nicht initialisiert, also auch nicht, wenn sie Arrayelemente sind
- bei der Zerstörung von Arrays werden auch alle Elemente zerstört, sprich, es werden die Destrukturen aufgerufen
- um dies zu überprüfen, soll ein Array von Struktur Objekten angelegt werden, die einen Konstruktor und Destruktor mit Ausgaben besitzen

Go!

## Beispiel

```
struct Test {  
    Test() {    cout << this << " +" << endl; }  
    ~Test() {   cout << this << " -" << endl; }  
};
```

```
void f() {  
    cout << "begin f" << endl;  
    Test b[3];  
    cout << "end f" << endl;  
}
```

```
int main() {  
    Test* p = new Test[5];  
    char c;  
    {  
        cout << "begin sub" << endl;  
        Test a[3];  
        cout << "end sub" << endl;  
    }
```

```
    f();  
    cin >> c;  
    if (c == 'c')  
        delete p;  
    else  
        delete[] p;
```

..

lokale Arrays mit  
konstanter Größe

Array, das  
dynamisch auf  
dem Heap  
erzeugt wird

# Vorlesung 6



## Auswertungsreihenfolge

- die Reihenfolge der Auswertung von Teilausdrücken in einem Ausdruck ist in C++ (und auch in C) **nicht** definiert

$f(3) + g(4)$  // undefiniert, ob erst  $f$  oder  $g$   
// ausgewertet wird

- Grund ist, dass der Compiler besser optimieren kann, wenn die Reihenfolge nicht festgelegt ist
- dies führt zu Problemen, wenn  $f$  und  $g$  Seiteneffekte haben

- Beispiel:

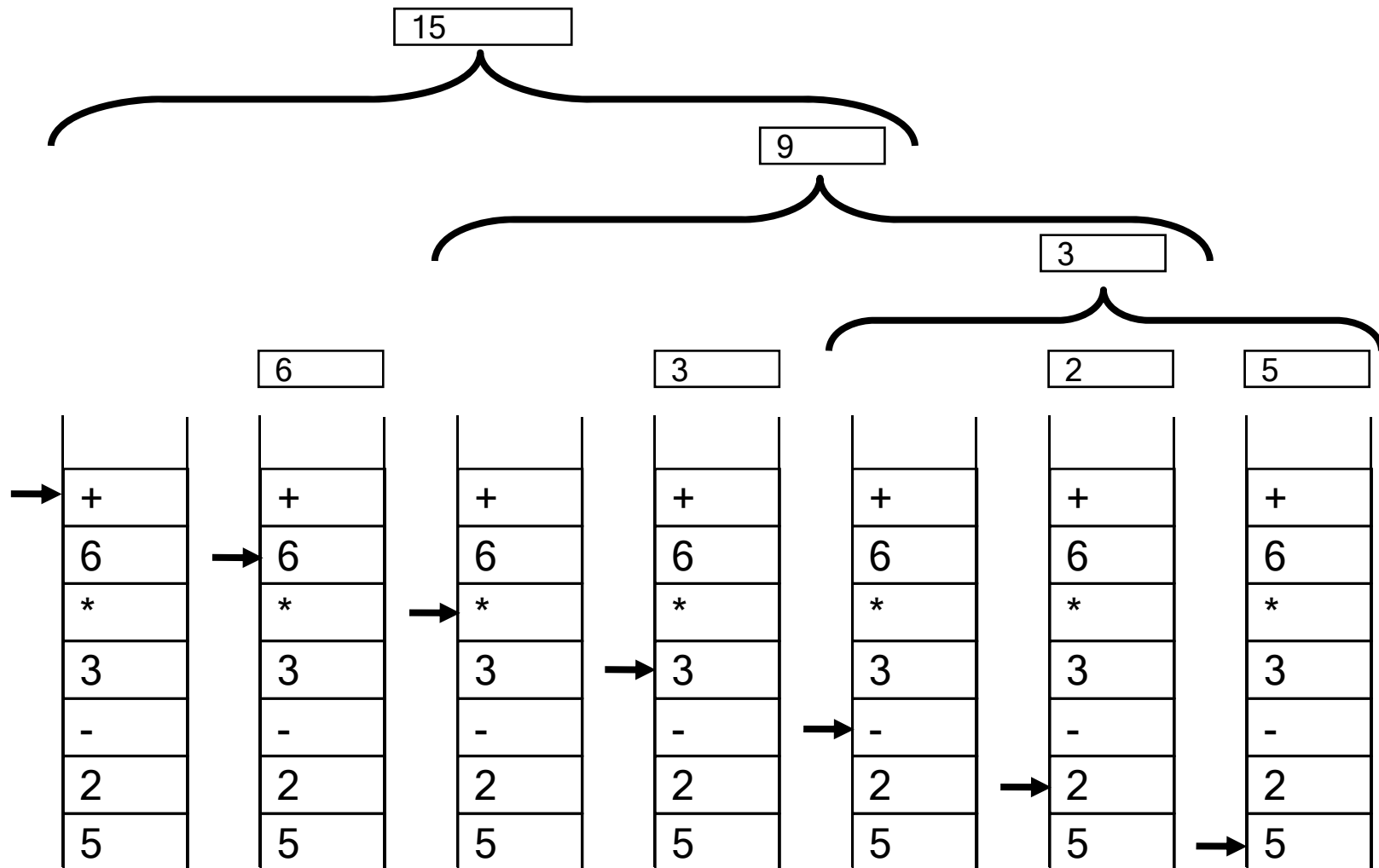
Taschenrechner, dessen Operatoren und Operanden auf einem Stack gemäß der ***umgekehrten polnischen Notation*** (= postfix Notation) abgelegt sind

## Auswertungsreihenfolge (Forts.)

1. sowohl die Operanden als auch die Operatoren liegen auf einem Stack: unten die beiden Operanden, oben der Operator
2. Beispiel: aus  $3 - 4$  wird  $3\ 4\ -$
3. ist der oberste Stackeintrag ein Operand, ist er das Ergebnis der (Teil-)Berechnung
4. ist der oberste Stackeintrag ein Operator,
  1. wird er entfernt,
  2. die Evaluierung rekursiv für die beiden Operanden fortgesetzt und
  3. die beiden Ergebnisse gemäß des Operanden verknüpft und zurückgeliefert

## Auswertungsreihenfolge (Forts.)

- Auswertung von  $(5-2)*3+6$  als Postfix:  $5\ 2\ -\ 3\ *\ 6\ +$



## Beispiel

```
enum Op {PLUS, MINUS, MULT, DIV};
```

Operatoren werden als  
Aufzählungstyp definiert

```
struct Entry {  
    Entry(Op op) {  
        m_blsOp = true;  
        m_op = op;  
    }  
  
    Entry(int iNum) {  
        m_blsOp = false;  
        m_iNum = iNum;  
    }  
  
    bool m_blsOp;  
    union {  
        Op m_op;  
        int m_iNum;  
    };  
};  
  
...
```

Konstruktoren  
für Stackeinträge

ein Stackeintrag muss  
selber sagen können, ob er  
ein Operand oder ein  
Operator ist

## Beispiel (Forts.)

...

```
int eval(Entry* pStack,unsigned& uiLen) {
    assert(pStack != nullptr);
    if (pStack[uiLen].m_bIsOp) {
        const Op cOp = pStack[uiLen].m_op;
        --uiLen;
        const int ciArg2 = eval(pStack,uiLen); // oben nach dem Operator liegt der
                                                // rechte Operand
        const int ciArg1 = eval(pStack,uiLen); // dann kommt der linke Operand
        switch (cOp) {
            case PLUS: return ciArg1+ciArg2;
            case MINUS: return ciArg1-ciArg2;
            case MULT: return ciArg1*ciArg2;
            case DIV: return ciArg1/ciArg2;
        }
    } else {
        --uiLen;
        return pStack[uiLen+1].m_iNum;
    }
}
```

der Zeiger auf den  
Stackeintrag wird in  
jedem Fall verringert

...

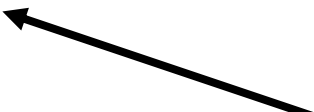
Go!

## Beispiel (Forts.)

...

```
int main() {  
    Entry f[] = {Entry(5),  
                 Entry(2),  
                 Entry(MINUS),  
                 Entry(3),  
                 Entry(MULT),  
                 Entry(6),  
                 Entry(PLUS)}; // rechnet (5-2)*3+6  
    unsigned uiLen = 6;  
    cout << eval(f, uiLen) << endl;  
    return 0;  
}
```

Warum kann hier die  
Zahl 6 nicht direkt  
übergeben werden?



## Auswertungsreihenfolge (Forts.)

- da die Reihenfolge der Auswertungen der Argumente nicht vorgegeben ist, ist die folgende Optimierung nicht semantikerhaltend (also schlecht)
- warum ???

```
int eval(Entry* pStack,unsigned& uiLen) {  
    assert(pStack != nullptr);  
    if (pStack[uiLen].m_bIsOp) {  
        switch (pStack[uiLen--].m_op) {  
            case PLUS: return eval(pStack,uiLen)+eval(pStack,uiLen);  
            case MINUS: return eval(pStack,uiLen)-eval(pStack,uiLen);  
            case MULT: return eval(pStack,uiLen)*eval(pStack,uiLen);  
            case DIV: return eval(pStack,uiLen)/eval(pStack,uiLen);  
        }  
    } else {  
        return pStack[uiLen--].m_iNum;  
    }  
}
```

## Auswertungsreihenfolge (Forts.)

- das Komma `,` dient in C++ auch dazu, Ausdrücke zu einer Sequenz zusammenzufassen

- Beispiel

`a++,b++,c++,34`

inkrementiert die Variablen `a`, `b` und `c` und liefert als Ergebnis den Wert `34` zurück

- im Zusammenhang mit Funktionsaufrufen, Funktionsüberladung und falscher Klammersetzung kann es Fehlern kommen



Go!

## Beispiel (Forts.)

```
#include <iostream>
```

```
using namespace std;
```

```
void f(int i,int j) {  
    cout << "ich bin f(" << i << "," << j << ")" << endl;  
}
```

überladene Funktion f

```
void f(int i) {  
    cout << "ich bin f(" << i << ")" << endl;  
}
```

```
int main() {  
    f(13,42);  
    f((13,42));  
    return 0;  
}
```

← kleine syntaktische  
Ungenauigkeit mit  
großer Auswirkung

Go!

## Auswertungsreihenfolge (Forts.)

- der Kommaoperator wird sehr oft in for-Schleifen eingesetzt
- er dient dazu, sowohl im Initialteil mehrere Variablen zu initialisieren, als auch
- im Schlussteil mehrere Variablen zu verändern

for(<Initialteil> ; <Bedingung> ; <Schlussteil> )

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    for(int i = 1, j = 15; i < j; i+=i, j+=1000)
```

```
        cout << "i = " << i << " j = " << j << endl;
```

```
    return 0;
```

```
}
```

2 neue lokale  
int-Variablen

beide werden am  
Ende der Schleife  
verändert

## Operatorenpriorität

- die Operatoren werden gemäß der Prioritäten ausgeführt
- dabei sind die Prioritäten derart gewählt, dass die meisten Benutzungen ohne Klammerung auskommen

- Beispiel:

`if (i <= 0 && max<i) ⇔ if ((i <= 0) && (max<i))`

- es gibt aber auch Ausnahmen, im Zweifelsfall sollte man Klammern setzen

`if (i & mask == 0)  $\nRightarrow$  if ((i & mask) == 0)`

sondern

`if (i & mask == 0) ⇔ if (i & (mask == 0))`

- andere „Klassiker“ von C++ Neulingen sind

`if (0 <= x <= 99)`

`if (x = 7)`

Go!

## Beispiel (Forts.)

```
int main() {  
    int x = 200;                Sieht gut aus, ist aber falsch  
    if (0 <= x <= 100)  
        cout << "x liegt zwischen 0 und 100 und hat den Wert: " << x << endl;  
    else  
        cout << "x liegt außerhalb der Grenzen" << endl;  
    if (x == 7)                 Der Klassiker schlechthin  
        cout << "x ist 7: " << x << endl;  
    if (3 & 4 == 0)  
        cout << "3 und 4 haben alles unterschiedliche bits" << endl;  
    else  
        cout << "3 und 4 haben anscheinend gemeinsame bits" << endl;  
    if (2 & 4 == 0)  
        cout << "2 und 4 haben alles unterschiedliche bits" << endl;  
    else  
        cout << "2 und 4 haben anscheinend gemeinsame bits" << endl;  
    return 0;  
}
```

## Inkrement- und Dekrementoperatoren

- analog zu Java gibt es in C++ In- und Dekrementoperatoren
- diese liegen als Pre- und Postoperatoren vor
- anders als in Java kann man in C++ diese nicht nur auf Ganzzahlen anwenden, sondern auch auf Pointer
- die Anwendung auf Pointer macht im Kontext von Arrays Sinn
- durch das Inkrement wird auf das nächste Element in dem Array gezeigt, durch das Dekrement auf das vorherige

Go!

## Beispiel (Forts.)

```
#include <iostream>
```

```
using namespace std;
```

pTrg und pSrc zeigen  
auf char Arrays

Zeichen  
kopieren

```
void cpy(char* pTrg,const char* pSrc) {  
    while (*pSrc != '\0') {  
        *pTrg = *pSrc;  
        ++pTrg;  
        ++pSrc;  
    }  
}
```

Ist das Ende  
Zeichen erreicht?

an der nächsten  
Position  
wetermachen

```
int main() {  
    char a[] = "juhu";  
    char b[] = "toll";  
    cout << "a = " << a << "; b = " << b << endl;  
    cpy(a,b);  
    cout << "a = " << a << "; b = " << b << endl;  
    return 0;  
}
```

## Inkrement- und Dekrementoperatoren (Forts.)

- diese einfache Kopierfunktion wird oft anders geschrieben
- die Schleife

```
while (*pSrc != '\0')
```

kann ersetzt werden durch

```
while (*pSrc)
```

weil jedes Zeichen ungleich '\0', zu dem booleschen Wert **true** umgewandelt wird

- die Post- und Prede- und –inkrementoperatoren binden stärker als der Dereferenzierungsoperator, daher kann

```
*pTrg = *pSrc;
```

```
++pTrg;
```

```
++pSrc;
```

- ersetzt werden durch: `*pTrg++ = *pSrc++;`

Go!

## Inkrement- und Dekrementoperatoren (Forts.)

- das Ergebnis sieht so aus:

```
while (*pSrc)
    *pTrg++ = *pSrc++;
```

- analog zu Java ist das Ergebnis einer Zuweisung der Wert der rechten Seite der Zuweisung
- somit kann in diesem Fall die Zuweisung auch in die Bedingung eingebaut werden

```
while (*pTrg++ = *pSrc++) ;
```

← die leere Anweisung

- die gesamte Kopierfunktion lautet:

```
void cpy(char* pTrg, const char* pSrc) {
    while (*pTrg++ = *pSrc++);
}
```

nicht schön, passiert  
aber oft in C Code



## Funktionen: statische Variablen

- neben lokalen Variablen können Funktionen auch statische Variablen besitzen (Schlüsselwort: **static**)
- diese Variablen werden beim ersten Aufruf der Funktion initialisiert und existieren über den Funktionsaufruf hinweg bis zum Ende des Programms

```
void f(int a) {  
    for(int x = 0; a--;) {  
        static int n = 0;  
        int y = 0;  
        cout << "a: " << a << " n: " << n++ << " y: " << y++ << " x: " << x++ << endl;  
    }  
}  
  
int main() {  
    f(3);  
    return 0;  
}
```

Go!

## Funktionen: statische Variablen (Forts.)

- wenn die Funktion nicht aufgerufen wird, wird die Variable auch gar nicht angelegt

```
struct A {  
    A(const char* p) {  
        cout << "A+ " << p << endl;  
    }  
};  
  
static A a("globales A");  
  
void f(int a) {  
    static A b("fast globales A");  
}  
  
int main() {  
    // f(3);  
    // f(4);  
    return 0;  
}
```

Konstruktor  
mit Ausgabe

a ist global zum  
gesamten Programm

b ist global  
zur Funktion f

## Funktionen: überladen

- analog zu Java können in C++ Funktionen/Methoden überladen werden (gleicher Name, unterschiedliche Anzahl von Parametern und/oder unterschiedliche Parametertypen)
- die Auflösung, welche Funktion gemeint ist, erfolgt zur Compilezeit
- Mehrdeutigkeiten werden zur Compilezeit als Fehler ausgegeben
- Vorsicht: die Ganzzahlkonstanten sind vom Typ `int`, auch werden `char` Konstanten nach `int` konvertiert, wenn nicht eine Funktion für `char` zur Verfügung steht

Go!

## Beispiel

```
//void f(unsigned int a) {  
void f(int a) {  
    cout << "int " << a << endl;  
}  
  
void f(double a) {  
    cout << "double " << a << endl;  
}  
  
void f(bool a) {  
    cout << "bool " << a << endl;  
}  
  
int main() {  
    f(1.0);  
    f(true);  
    f(1);  
    f('c');  
    return 0;  
}
```

## Funktionen: überladen (Forts.)

- wie in Java reicht es nicht aus, dass sich die Funktionen nur in ihrem Rückgabetyp unterscheiden

```
1: bool f(int a) {  
2:     return true;  
3: }  
4:  
5: char f(int a) {  
6:     return 'c';  
7: }  
8:  
9: int main() {  
10:     bool b = f(3);  
11:     char c = f(3);  
12:     return 0;  
13: }
```

```
FuncOver2.cpp: In function `char f(int)':  
FuncOver2.cpp:5: error: new declaration `char f(int)'  
FuncOver2.cpp:1: error: ambiguates old declaration `bool f(int)'
```

Go!

## Funktionen: überladen (Forts.)

- man kann die Mehrdeutigkeit in unterschiedlichen Scopes eliminieren, indem die Funktionen dort nochmals deklariert werden
- die Definition muss aber woanders erfolgen

```
void f(int a) {      cout << "f(int) " << a << endl; }
```

```
void f(double a) {  cout << "f(double) " << a << endl; }
```

```
int main() {  
    f(3);  
    {  
        void f(double);  
        f(1);  
    }  
    f(1);  
    return 0;  
}
```

2 Definitionen +  
2 Deklarationen:  
f wird überladen

1 Deklarationen:  
f(int a) wird  
ausgeblendet

## Funktionen: Standardargumente

- anders als in Java kann man in C++ den Funktionsparametern bereits bei der Deklaration Werte mitgeben
- beim Aufruf können diese Parameter ausgelassen werden
- in diesem Fall werden die Standardwerte der Deklaration beim Aufruf verwendet
- wenn ein Parameter einen Standardwert hat, müssen alle nachfolgenden Parameter ebenfalls Standardwerte haben

```
void f(int a, bool b = true, double d = 1.0) {  
    cout << a << " " << (b ? "true" : "false") << " " << d << endl;  
}  
int main() {  
    f(3);  
    f(17,false);  
    f(42,true,3.234);  
    return 0;  
}
```

## Funktionen: Standardargumente (Forts.)

- durch die Verwendung von Standardwerten kann es passieren, dass das Überladen von Funktionen trotz unterschiedlicher Parameterlisten nicht mehr eindeutig ist

```
5: void f(int a, bool b = true, double d = 1.0) { ... }
```

```
...
```

```
9: void f(int a) { ... }
```

```
...
```

```
13: f(3);
```

```
FuncDef2.cpp:13: error: call of overloaded `f(int)' is ambiguous
FuncDef2.cpp:5: note: candidates are: void f(int, bool, double)
FuncDef2.cpp:9: note: void f(int)
```

- beim Überladen muss darauf geachtet werden, dass die Funktionen bzgl. der Parameter, die keine Standardwerte haben, eindeutig sind



# Vorlesung 7

## Klassen und Strukturen

- in C gab es Strukturen (**struct**), um mehrere Datenelemente zu einem neuen Datenelement zusammenzufassen
- das gleiche machen Klassen
- daher — und um alten C Code in C++ Code einbinden zu können — wurden Strukturen in C++ übernommen
- sie wurden auch erweitert, so dass Strukturen alles können, was Klassen auch können, d.h.
  - sie haben Kon- und Destrukturen, (virtuelle) Methoden, Members
  - man kann Hierarchien bilden (Ableitung)
- sie unterscheiden sich nur in ihrer Zugriffskontrolle
  - in Strukturen ist standardmäßig alles **public**
  - in Klassen ist standardmäßig alles **private**

## Klassen und Strukturen (Forts.)

- ein gutes Design zeichnet sich dadurch aus, dass die Daten nicht direkt zugreifbar sind, sondern
- durch eine wohldefinierte Schnittstelle dem Benutzer *kontrolliert* zur Verfügung stehen
- daher sollten Daten (Members) einer Klasse (oder Struktur) niemals **public** sein
- daher sollte man Klassen anstatt Strukturen verwenden

In Zukunft werden (fast) ausschließlich Klassen betrachtet (jedoch gilt alles auch für Strukturen)

## Beispiel

```
struct Datum {
```

```
    Datum() {
```

```
        m_uiTag = 1;
```

```
        m_uiMonat = 1;
```

```
        m_uiJahr = 1970;
```

```
    }
```

Standardkonstruktor

```
void print() {
```

```
    cout << m_uiTag << "." << m_uiMonat << "." << m_uiJahr << endl;
```

```
}
```

Ausgaberoutine

```
bool setMonat(unsigned uiMonat) {  
    if (uiMonat >= 1 && uiMonat <= 12) {  
        m_uiMonat = uiMonat;  
        return true;  
    } else  
        return false;  
}
```

kontrollierte  
Veränderung des  
Monats mit  
Sicherheitsabfrage

```
    unsigned m_uiTag;
```

```
    unsigned m_uiMonat;
```

```
    unsigned m_uiJahr;
```

```
};
```

Go!

## Beispiel



...

```
int main() {  
    Datum d;  
    d.print();  
    d.setMonat(14);  
    d.print();  
    d.m_uiMonat = 14;  
    d.print();  
    return 0;  
}
```

kontrollierte Veränderung des  
Objekts: **Autor** der Klasse ist  
für Fehler verantwortlich

chaotische Veränderung des  
Objekts: **Anwender** der Klasse  
ist für Fehler verantwortlich



Merke: Klassen statt  
Strukturen verwenden !!!

- Beispiel für die Deklaration einer Klasse:

```
class List {  
    public:  
        List() { ... }  
        void print() { ... }  
    private:  
        ListElem* m_pHead;  
};
```

Klassen

Schlüsselwort **class** wie in Java

Sichtbarkeitskontrolle gilt bis zur nächsten Regel für Sichtbarkeit

ganz wichtig: das ; am Ende nicht vergessen

## Klassen (Forts.)

- analog zu Java werden Klassen mit dem Schlüsselwort **class** gebildet
- Wichtig: am Ende einer Klassendeklaration nicht das ; vergessen
- anders als in Java wird eine Klasse nicht automatisch von einer bestimmten Basisklasse abgeleitet (**Object** in Java)
- anders als in Java sind alle Elemente als **private** deklariert
- anders als in Java müssen nicht die Elemente (Members oder Methoden) als **public** oder **private** deklariert werden, sondern Sektionen in der Klasse (siehe nächste Seite)

## Klassen: Zugriffsrechte

- ähnlich wie in Java gibt es die Zugriffsrechte
  - `public`: jeder darf darauf zugreifen
  - `private`: nur die eigene Klasse darf darauf zugreifen
  - `protected`: die eigene Klasse und die abgeleiteten Klassen dürfen darauf zugreifen
- anders als in Java gibt es nicht das Zugriffsrecht, das man standardmäßig ohne eines dieser 3 Rechte hat
- in Klassen ist standardmäßig alles `private`
- in Strukturen ist standardmäßig alles `public`
- die Schlüsselworte können mehrmals in einer Klassen (Struktur) verwendet werden, um die Rechte der nachfolgenden Elemente (Members / Methoden) zu ändern



# Beispiel

```
class Datum {  
public:
```

beide Konstruktoren  
sind public

```
Datum() { ... }
```

```
Datum(unsigned uiTag,unsigned uiMonat,unsigned uiJahr) { ... }
```

```
protected:
```

```
void print() { ... }
```

die print Methode ist nach  
außen nicht sichtbar (außer  
abgeleitete Klassen)

```
public:
```

```
bool setMonat(unsigned uiMonat) { ... }
```

Zugriffsrechte können  
wiederholt werden

```
private:
```

```
unsigned m_uiTag;
```

```
unsigned m_uiMonat;
```

```
unsigned m_uiJahr;
```

```
};
```

```
...
```

gutes Design: Members sind privat,  
können nur von der Klasse aus  
gelesen und beschrieben werden

Go!

## Beispiel (Forts.)

...

ok: print ist  
protected, also  
sichtbar in  
abgeleiteter Klasse

```
class Geburtstag : Datum {  
public:  
    void drucken() {  
        print();  
        unsigned uiMonat = m_uiMonat;  
    }  
};
```

Fehler : m\_uiMonat  
ist **private**, auch in  
abgeleiteter Klasse

Fehler: print ist  
protected

```
int main() {  
    Datum d;  
    d.print();  
    d.setMonat(14);  
    d.print();  
    d.m_uiMonat = 14;  
    d.print();  
    return 0;  
}
```

ok: setMonat ist  
public

Fehler : m\_uiMonat  
ist **private**

## Klassen: Konstruktoren



- analog zu Java dienen Konstruktoren zur korrekten Initialisierung von Objekten
- wie in Java können Konstruktoren überladen werden
- wie Funktionen und Methoden können auch die Argumente von Konstruktoren Standardwerte enthalten
- damit ist es möglich, weniger Konstruktoren als in Java implementieren zu müssen


Go!

## Beispiel

ein Konstruktor weiß  
nicht, ob das Objekt auf  
dem Stack oder Heap  
erzeugt wird

```
class A {  
public:  
  
    A(unsigned ui = 0) {  
        cout << "Konstruktor A " << ui << endl;  
    }  
  
};
```

```
int main() {  
    A a1;  kein Unterschied  
    // A a5(); // Funktionsdeklaration vom Typ void->A mit Namen a5  
    A a2(0);   
    A a3(34);  
    A* p = nullptr;  
    cout << "gleich werden vier A's erzeugt" << endl;  
    p = new A;  
    p = new A();  
    p = new A(0);  
    p = new A(42);  
    return 0;  
}
```

 Objekte auf  
dem Heap

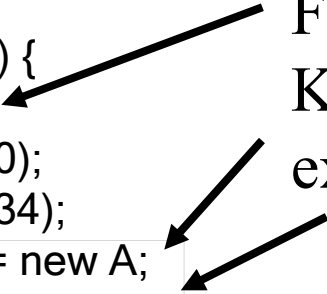
Go!

## Beispiel

- wird ein Konstruktor definiert, der Argumente erwartet, so wird der Default-Konstruktor ausgeschaltet

```
class A {  
public:  
  
    A(unsigned ui) {  
        cout << "Konstruktor A " << ui << endl;  
    }  
};  
  
int main() {  
    A a1;  
    A a2(0);  
    A a3(34);  
    A* p = new A;  
    p = new A();  
    p = new A(42);  
    return 0;  
}
```

Fehler: Default  
Konstruktor  
existiert nicht



Go!

## Beispiel

- durch die Deklaration des Standardkonstruktors mit dem Schlüsselwort **default** danach wird der Default Konstruktor wieder definiert

```
class A {  
public:
```

```
    A() = default;  
    A(unsigned ui) {  
        cout << "Konstruktor A " << ui << endl;  
    }
```

```
};
```

Default Konstruktor  
wird durch Compiler  
generiert



```
int main() {
```

```
    A a1;  
    A a2(0);  
    A a3(34);  
    A* p = new A;  
    p = new A();  
    p = new A(42);  
    return 0;
```

```
}
```

ok




Go!

## Delegierender Konstruktor

- Ähnlich wie in Java kann ein Konstruktor einen anderen Konstruktor aufrufen
- dies nennt man delegierenden Konstruktor

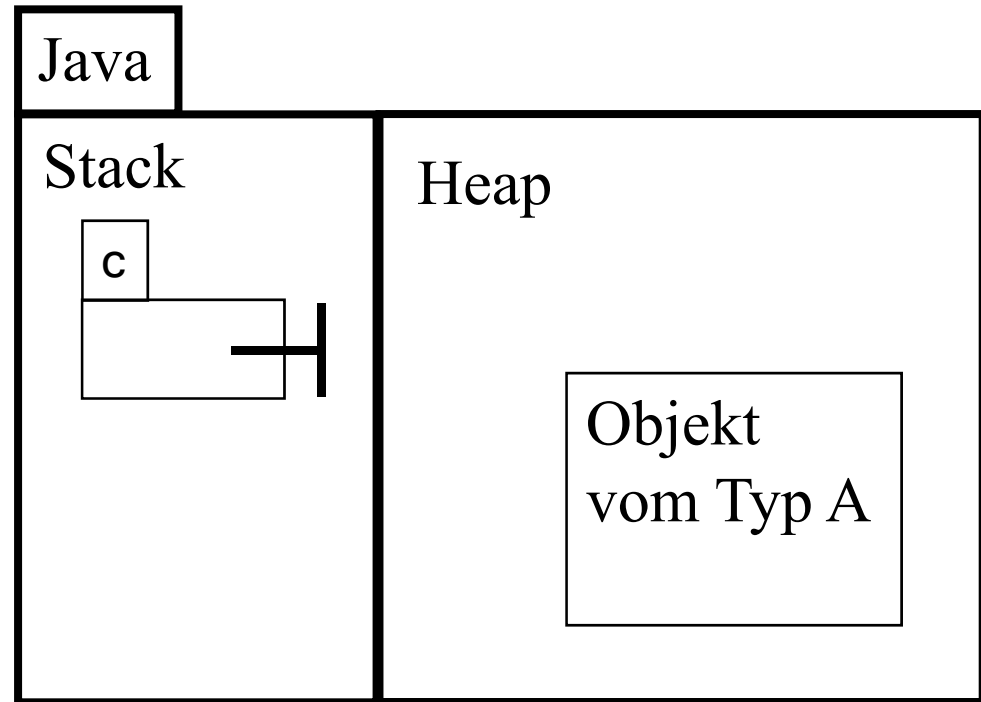
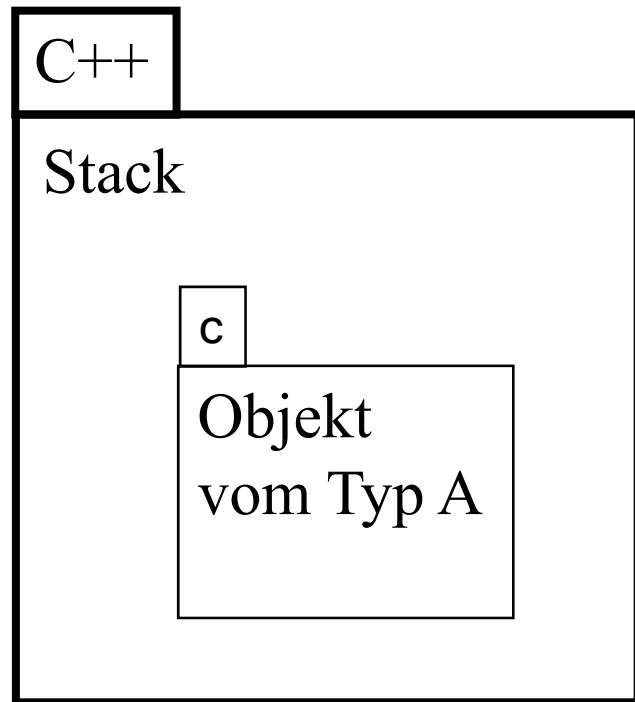
```
class A {  
public:  
    A() : A(23) {  
        cout << "ich kann noch mehr" << endl;  
    }  
  
    A(unsigned ui) {  
        cout << "Konstruktor A " << ui << endl;  
    }  
};  
  
int main() {  
    A a1;  
    ...  
    return 0;  
}
```

führt erst den  
anderen Konstruktor  
aus, dann seinen  
eigenen Rumpf



## Klassen: Konstruktoren (Forts.)

- größter Unterschied zwischen Java und C++:  
Objektdeklaration: `A c;`
- kann man in Java und in C++ hinschreiben, sieht gleich aus, bedeutet aber etwas vollkommen Unterschiedliches





## Klassen: Konstruktoren (Forts.)

- der Java Ausdruck

`A c;`

entspricht in C++

`A* c = nullptr;`

- der Java Ausdruck

`A c = new A();`

entspricht in C++

`A* c = new A();`

- für den C++ Ausdruck

`A c;`

gibt es keinen Java Ausdruck, weil

- in Java keine Objekte auf dem Stack liegen können

## Klassen: Copykonstruktor

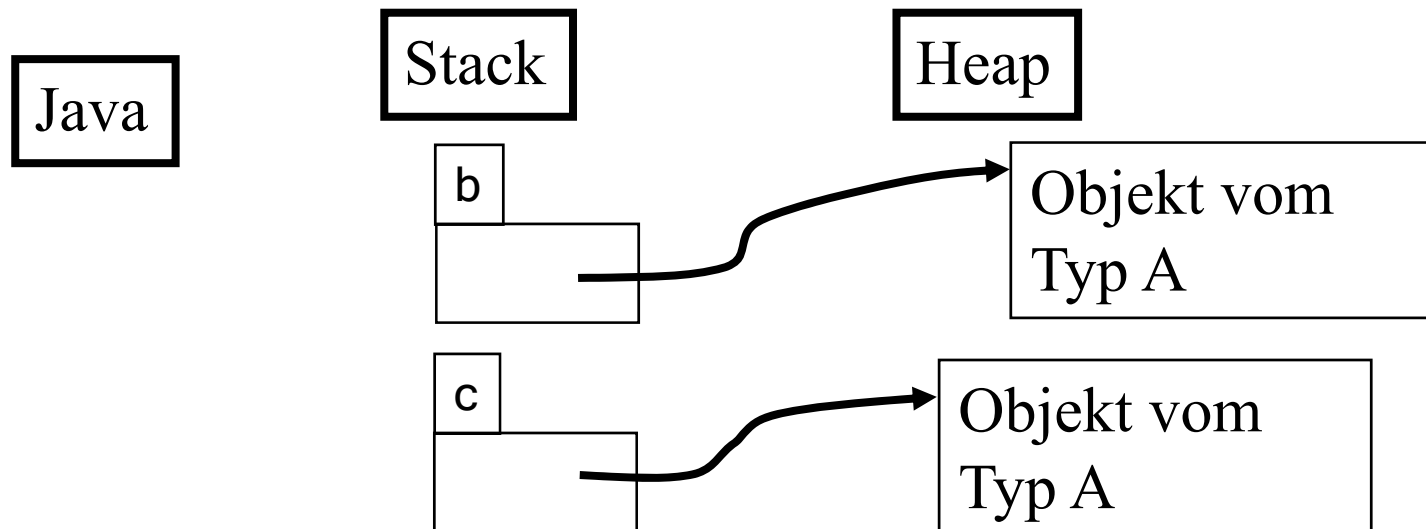
- aus diesem gravierenden Unterschied ergibt sich auch eine andere Bedeutung für die Zuweisung von Objekten

A b,c;

...

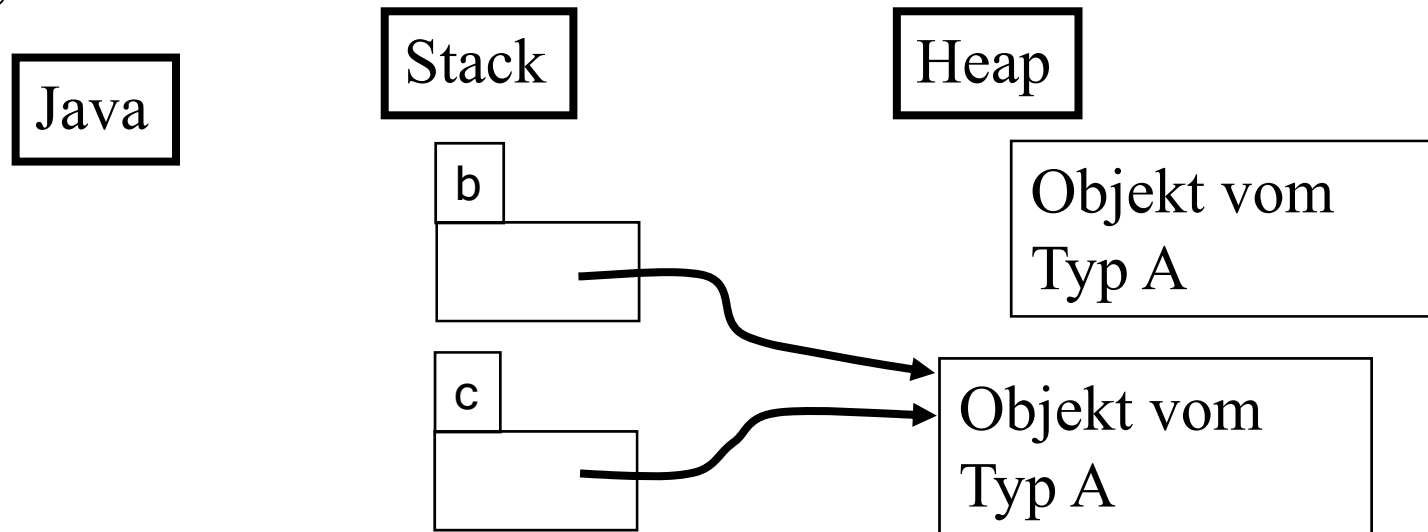
b = c;

- in Java bedeutet dies, dass nach der Zuweisung b auf das gleiche Objekt zeigt wie c



## Klassen: Copykonstruktor (Forts.)

- nach der Zuweisung `b = c;` präsentiert sich der Speicher wie folgt:



- in C++ gilt das gleiche Verhalten bei Pointern auf Objekten

```
A* b, c;  
...  
b = c;
```

!!!

## Klassen: Copykonstruktor (Forts.)

- in C++ bedeutet aber

`A b,c;`

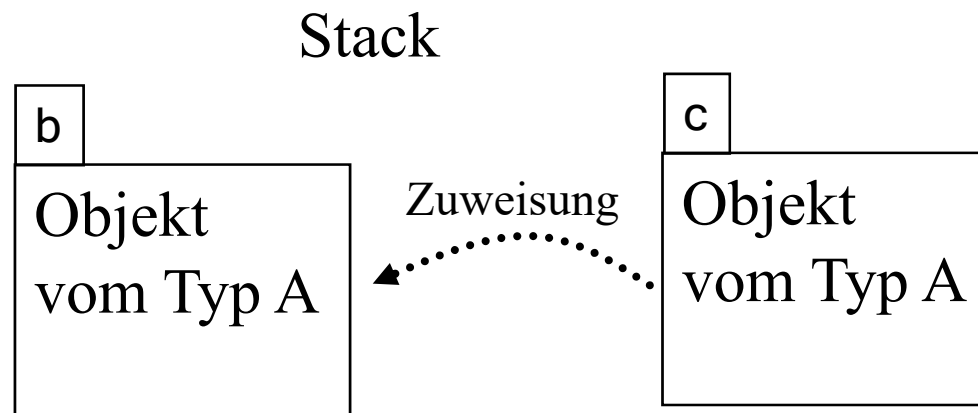
...

`b = c;`

das dem Objekt b das Objekt c zugewiesen wird

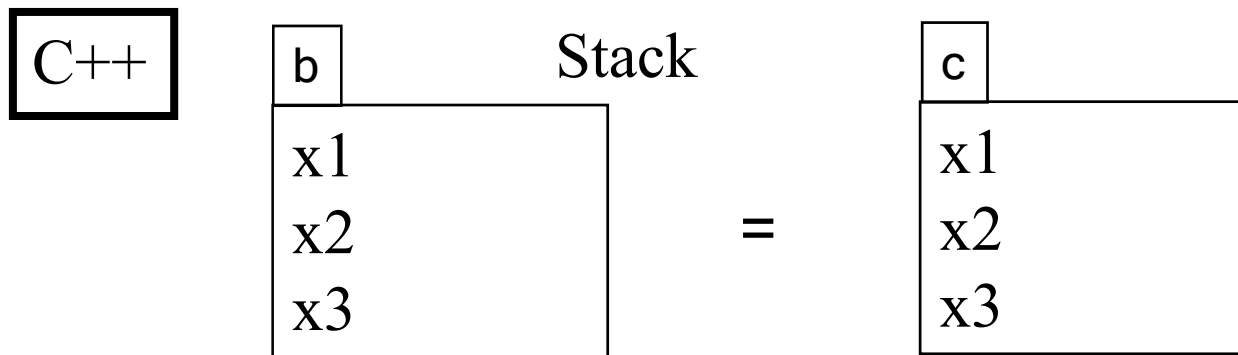
- eine solche Zuweisung ist in Java gar nicht möglich

**C++**

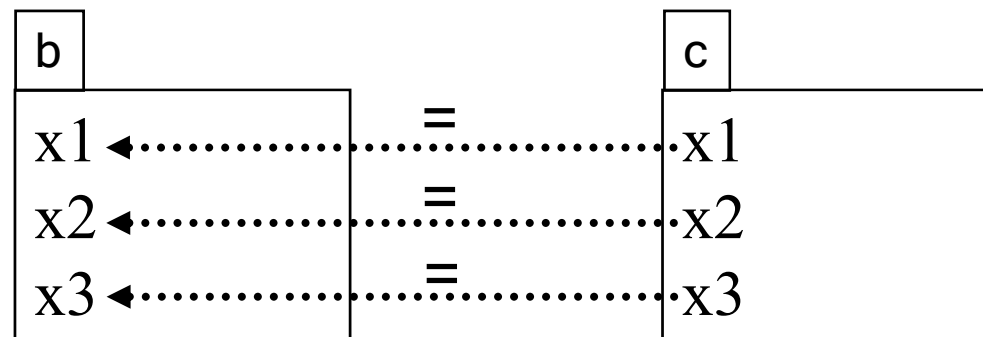


## Klassen: Copykonstruktor (Forts.)

- wird ein Objekt **c** einem anderen Objekt **b** zugewiesen, so werden standardmäßig alle Members von **c** den entsprechenden Members von **b** zugewiesen



- wird standardmäßig implementiert durch



# Beispiel

```
class A {  
public:
```

```
    A(int i, char c) {  
        m_i = i;  
        m_c = c;  
    }
```

```
    void print(const char* pMsg) {  
        cout << pMsg << m_i << " " << m_c << endl;  
    }
```

```
    void change(int i, char c) {  
        m_i = i;  
        m_c = c;  
    }
```

Objektvariablen  
können explizit  
gesetzt werden

```
private:
```

```
    int m_i;  
    char m_c;  
};
```

```
...
```

Go!

## Beispiel

...

```
int main() {  
    A b(17,'b'), c(42,'c');  
    b.print("b: ");  
    c.print("c: ");  
    b = c;  
    b.print("b: ");  
    c.print("c: ");  
    c.change(23,'#');  
    b.print("b: ");  
    c.print("c: ");  
    return 0;  
}
```


Objektzuweisung:  
danach hat b die  
gleichen Werte wie c

b sollte sich nicht  
geändert haben

## Klassen: Copykonstruktor (Forts.)

- u.U. kann dieses Standardverhalten nicht das erforderte Verhalten sein
- in diesem Fall kann das Verhalten verändert werden (siehe später: Assignmentoperator)
- diese Art der Memberkopie erfolgt aber auch noch an zwei anderen Stellen
  - Initialisierung von Objekten durch Objekte

`A b;`  
`A c(b);`    oder    `A c = b;`



syntaktisch  
unterschiedlich, aber  
semantisch identisch

- Parameterübergabe  
`void test(A c) {...}`  
`test(b);`



Go!

## Beispiel

```
void test(A b) {  
    b.print("test b: ");  
    b.change(-1, '?');  
    b.print("test b: ");  
}
```

call-by-value: Änderungen  
sieht man nicht außerhalb

```
int main() {  
    A b(17, 'b');  
    A c(b);  
    A d = b;  
    b.print("b: ");  
    c.print("c: ");  
    d.print("d: ");  
    test(d);  
    b.print("b: ");  
    c.print("c: ");  
    d.print("d: ");  
    c.change(23, '#');  
    b.print("b: ");  
    c.print("c: ");  
    d.print("d: ");  
    return 0;  
}
```

Objekte c und d gehen  
durch Kopien von b  
hervor

nur c verändert sich

Go!

## Klassen: Copykonstruktor (Forts.)

- dieses Standardverhalten bei der Initialisierung von Objekten und bei der Parameterübergabe von Objekten mittels call-by-value kann durch den Copykonstruktor beeinflusst werden
- er hat die Form:

```
class A {  
    A(const A& crArg) {...}  
};
```

- hier kann beliebiger Code erfolgen
- Wichtig: es wird kein Konstruktor, sondern nur der Copykonstruktor ausgeführt

```
A(const A& crArg) {  
    cout << "juhu" << endl;  
}
```

## Klassen: Copykonstruktor (Forts.)

- wird der Copykonstruktor implementiert, so muss das Standardverhalten (soweit gewünscht) selber implementiert werden

Go!

```
A(const A& crArg) {  
    cout << "juhu" << endl;  
    m_i = crArg.m_i;  
    m_c = crArg.m_c;  
}
```

Standardverhalten  
nachgebildet

- doch Vorsicht vor dem falschen Freund des Copykonstruktors

Go!

```
A(A& rArg) {  
    cout << "juhu" << endl;  
    m_i = rArg.m_i;  
    m_c = rArg.m_c;  
    rArg.m_i = 1024;  
}
```

kein const:  
Argument kann  
verändert werden

# Vorlesung 8

## Klassen: konstante Members

- Members können mittels **const** als konstant markiert werden
- sie müssen bei der Konstruktion einen Wert bekommen, der nicht mehr veränderbar ist

```
5: class A {  
6: public:  
7:  
8:     A(int i = 0) {  
9:         m_ciVal = i;  
10:    }  
11:  
12: private:  
13:  
14:     const int m_ciVal;  
15: };
```

so macht man es in Java;  
**m\_ciVal** kann nie wieder  
erneut beschrieben werden

in C++ ein Fehler:  
schreibender Zugriff  
auf ein **const** Member

```
Konstruktor2.cpp: In constructor `A::A(int)':  
Konstruktor2.cpp:8: error: uninitialized member `A::m_ciVal' with `const' type `const int'  
Konstruktor2.cpp:9: error: assignment of read-only data-member `A::m_ciVal'
```

## Klassen: konstante Members (Forts.)

- Members können (konstante Members müssen) in der Initialisierungsliste gefüllt werden
- diese Initialisierungsliste wird noch vor dem Konstruktorcode ausgewertet

```
class A {  
public:
```

```
    A(int i = 0) : m_ciVal(i) {  
    }
```

```
private:
```

```
    const int m_ciVal;  
};
```

hier können durch Komma  
getrennt alle Members stehen



Go!

## Beispiel

```
class B {  
public:  
    B(int i) {    cout << "B " << i << endl;    }  
};
```

Members am besten immer  
in der Initialisierungsliste  
mit Werten füllen

```
class A {  
public:  
    A(int i = 0) : m_ciVal(i), m_b(17) {  
        cout << "A " << m_ciVal << endl;  
    }  
private:  
    const int m_ciVal;  
    B m_b;  
};
```

```
int main() {  
    A a1;  
    A a2(3);  
    return 0;  
}
```

Go!

## Klassen: konstante Members (Forts.)

- die Reihenfolge der Initialisierungsliste muss identisch zu der Reihenfolge der Memberdeklaration sein
- weicht sie davon ab, wird trotzdem gemäß der Deklarationsreihenfolge initialisiert
- gute Compiler geben hier eine Warnung aus

```
class B {  
public:    B(int i) {    cout << "B " << i << endl;    }  
};
```

```
class C {  
public:    C(int i) {    cout << "C " << i << endl;    }  
};
```

```
class A {  
public:  
    A(int i = 0) : m_c(i),m_b(2*i) {    cout << "A " << endl;    }  
private:  
    B m_b;  
    C m_c;  
};
```

erst C, dann B

erst B, dann C

Reihenfolge der  
Initialisierung ist  
*nicht* identisch mit  
Deklarations-  
reihenfolge



Go!

## Klassen: konstante Members (Forts.)

- wie bei der „normalen“ Initialisierung, kann statt () auch {} für das Argument verwendet werden
- bei einer Typkonvertierung wird dann eine Warnung ausgegeben

ok, da keine  
Konvertierung

```
struct A {  
    A(int i) : m_{i} {}  
    int m_i;  
};
```

Warnung

```
struct B {  
    B(float i) : m_{i} {}  
    int m_i;  
};
```

keine Warnung,  
trotz Konvertierung

```
struct C {  
    C(float i) : m_i(i) {}  
    int m_i;  
};
```

Warnung wegen  
Typkonvertierung

```
int main() {  
    A a1(7.2);  
    A a2 = {7.2};  
    A a3 {7.2};  
    B b1(7.2);  
    B b2 = {7.2};  
    B b3 {7.2};  
    C c1(7.2);  
    C c2 = {7.2};  
    C c3 {7.2};  
    return 0;  
}
```

## Klassen: konstante Methoden

- in C++ können (anders als in Java) Methoden als konstant deklariert werden
- Beispiel:
  - eigene String Klasse mit einer Längenmethode
  - diese Methode liest den Member nur

```
class String {  
public:
```

```
    unsigned length() {  
        return strlen(m_pContent);  
    }
```

```
private:
```

```
    char* m_pContent = nullptr;  
};
```

lesender Zugriff; Objekt  
wird nicht verändert



## Klassen: konstante Methoden (Forts.)

- wird ein String an eine Methode oder Funktion übergeben, stehen dafür drei (fünf) unterschiedliche Möglichkeiten zur Verfügung
  - `void doit(String s) {...}` call-by-value
  - `void doit(String& s) {...}`
  - `void doit(const String& s) {...}` call-by-reference
  - `void doit(String* s) {...}`
  - `void doit(const String* s) {...}`
- die Pointer Varianten sind schlechter als die Referenzvarianten, wenn nicht wirklich auch der nullptr Pointer übergeben werden kann

## Klassen: konstante Methoden (Forts.)

- der call-by-value Übergabemechanismus ist ineffizient
- bei der Übergabe wird der Copykonstruktor aufgerufen
- ist dieser nichttrivial (siehe Übung: Copykonstruktor für Listen), wird hier unnötig Zeit verbraucht
- die Übergabe call-by-reference ist dem immer vorzuziehen
- meistens wird das übergebene Objekt aber nur in der Methode/Funktion gelesen
- dies wird dokumentiert durch das `const`
- also:

```
void doit(const String& s) {...}
```

so übergibt man Objekte

## Klassen: konstante Methoden (Forts.)

- Problem: die Methode `length` kann nicht mehr aufgerufen werden

```
27: void doit(const String& crArg) {  
28:     cout << "laenge: " << crArg.length() << endl;  
29: }
```

String.cpp: In function `void doit(const String&':  
String.cpp:28: error: passing `const String' as `this' argument of `unsigned int String::length()' discards qualifiers

- die Methode `length` der Klasse `String` muss als konstant deklariert werden
- dies erfolgt durch das Anhängen von `const` nach dem Methodenkopf

```
class String {  
...
```

```
    unsigned length() const { ... }
```

↙ signalisiert: `length` hat nur  
lesenden Zugriff auf Members

# Beispiel

```
class String {  
public:
```

```
    String(const char* p = nullptr) {  
        if (p == nullptr) {  
            m_pContent = new char[1];  
            m_pContent[0] = '\0';  
        } else {  
            m_pContent = new char[strlen(p)+1];  
            strcpy(m_pContent,p);  
        }  
    }  
}
```

```
    unsigned length() const {  
        return strlen(m_pContent);  
    }  
}
```

```
private:
```

```
    char* m_pContent = nullptr;  
};
```

```
...
```


nur lesende Zugriffe  
erlaubt, da **const**

Go!

## Beispiel (Forts.)

Aufruf von `length` von  
konstantem Objekt `crArg`  
erlaubt, weil `length` `const` ist

```
...  
void doit(const String& crArg) {  
    cout << "laenge: " << crArg.length() << endl;  
}  
  
int main() {  
    String s1;  
    String s2("juhu");  
    doit(s1);  
    doit(s2);  
    return 0;  
}
```



## Klassen: mutable Members

- das vorherige Beispiel soll optimiert werden
- Annahme: die Methode `length` wird sehr oft aufgerufen
- sie berechnet die Länge jedes mal neu
- Idee: die Länge wird zusätzlich gespeichert

```
class String {  
public:  
  
    String(const char* p = nullptr) { ... }  
  
    unsigned length() const {  
        if (m_iLength == -1)  
            m_iLength = strlen(m_pContent);  
        return m_iLength;  
    }  
private:  
    char* m_pContent = nullptr;  
    int m_iLength = -1;  
};
```

Problem: `length` hat  
einen schreibenden  
Zugriff auf `m_iLength`

```
String2.cpp: In member function 'unsigned int  
String::length() const':  
String2.cpp:20: error: assignment of data-member  
'String::m_iLength' in read-only structure
```



Go!

## Klassen: mutable Members (Forts.)

- bei Konstanz unterscheidet man in C++ zwischen physikalischer Konstanz (Standardfall) und
- logischer Konstanz (physikalische Konstanz impliziert logische Konstanz)
- im vorliegenden Fall verändert sich das String Objekt durch den `length` Aufruf physikalisch aber nicht logisch, da
- logisch nur der Stringinhalt von Bedeutung ist
- `m_iLength` spielt für die Logik keine Rolle und soll auch in konstanten Methoden verändert werden können
- dies erreicht man durch das Schlüsselwort `mutable` bei der Deklaration

```
class String { ...  
private:  
    char* m_pContent;  
    mutable int m_iLength;  
};
```

`m_iLength` gehört  
nicht zur logischen  
Semantik des Objekts

## Klassen: Destruktoren

- Destruktoren haben in C++ eine sehr wichtige Rolle
- anders als in Java weiß man in C++ genau, wann der Destruktor einer Klasse aufgerufen wird:
  - bei einem Objekt auf dem Heap (durch new erzeugt) wenn er durch delete zerstört wird
  - bei einem Objekt auf dem Stack (normal in einem Scope deklariert) wenn der Scope zu Ende ist
- eine Typische Aufgabe im Destruktor ist die Freigabe des Speichers, den dieses Objekt erzeugt hat

## Beispiel

```
class String {  
public:
```

```
    String(const char* p = nullptr) {  
        if (p == nullptr) {  
            m_pContent = new char[1];  
            m_pContent[0] = '\0';  
        } else {  
            m_pContent = new char[strlen(p)+1];  
            strcpy(m_pContent,p);  
        }  
    }  
}
```

kopiert den  
übergebenen  
Pointerinhalt

```
    ~String() {  
        delete [] m_pContent;  
    }
```

gibt den Speicher  
wieder frei



```
    void print() const {  
        cout << m_pContent << endl;  
    }
```

```
private:
```

```
    char* m_pContent = nullptr;  
};
```

eigener Speicher, um  
die Buchstaben zu  
speichern

Go!

## Beispiel (Forts.)

...

```
int main() {  
    String s1;  
    String s2("juhu");  
    s1.print();  
    s2.print();  
    for(;;) {  
        String s3("Hello World");  
    }  
    return 0;  
}
```

Ohne Destruktor wird  
der Speicher schnell voll

Go!

## Private Kon- und Destruktoren

- Was passiert, wenn alle Konstruktoren einer Klasse **private** oder **protected** ist?
- von der Klasse kann nur noch aus der Klasse heraus oder von abgeleiteten Klassen Objekte erzeugt werden
- analoges gilt für Destruktoren

```
class A {  
private:  
    A() {}  
public:  
    static A* gen() {  
        return new A;  
    }  
};  
int main() {  
    // A a;  
    A* p = A::gen();  
    return 0;  
}
```

ok: aus der Klasse heraus  
kann der Konstruktor  
aufgerufen werden

Fehler: außerhalb der  
Klasse ist der Konstruktor  
nicht sichtbar

Wann macht so  
etwas Sinn?

## Klassenvariablen und -methoden

- analog zu Java gibt es auch in C++ Klassenvariablen und –methoden
- auch sie werden durch das Schlüsselwort **static** markiert
- analog zu Java gibt es die Klassenvariablen auch nur einmal für die Klasse und nicht für jede Instanz
- Klassenvariablen und –methoden können verwendet werden, ohne dass ein Objekt vorhanden ist
- Klassenmethoden können auf Klassenvariablen zugreifen, aber nicht auf Objektvariablen

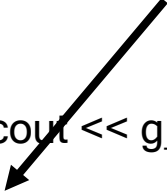
Go!

## Klassenvariablen und –methoden (Forts.)

- anders als in Java reicht die Deklaration einer Klassenvariablen noch nicht aus
- sie muss zusätzlich noch definiert werden

```
class A {  
public:  
    void print() { cout << g_uiCnt << endl; }  
private:  
    static unsigned g_uiCnt;  
};  
  
int main() {  
    A a;  
    a.print();  
    return 0;  
}
```

deklariert aber noch nicht  
definiert (es gibt noch  
keinen Speicherplatz)



kein Compilerfehler,  
sondern ein Linkerfehler

```
... \Temp/cc6Jbaaa.o(.text+0xbb): In function `main':  
... /c++/3.4.2/ostream:204: undefined reference to `A::g_uiCnt'  
collect2: ld returned 1 exit status  
  
Execution terminated
```

Go!

## Klassenvariablen und –methoden (Forts.)

- die Definition der Klassenvariablen erfolgt separat
- in einer Compileeinheit muss die globale Variable definiert werden

```
class A {  
public:  
    void print() {  
        cout << g_uiCnt << endl;  
    }  
private:  
    static unsigned g_uiCnt;  
};
```

↙ Deklaration in der Klasse

```
unsigned A::g_uiCnt = 17;
```

```
int main() {  
    ...  
}
```

↖ Definition (mit  
Initialisierung) in  
einer Compileeinheit



Go!

## Beispiel

```
class A {  
public:  
    A(int i) : m_iCnt(i) {}  
    void print() { cout << m_iCnt++ << " " << g_uiCnt++ << endl; }  
    static void globalPrint() { cout << "globalPrint: " << g_uiCnt << endl; }  
  
private:  
    int m_iCnt;  
    static unsigned g_uiCnt;  
};
```

Objektmethode

Objektvariable

Klassenmethode

Klassenvariable

```
unsigned A::g_uiCnt = 0;
```

```
int main() {  
    A::globalPrint();  
    A a1(-17);  
    A a2(-42);  
    a1.print();  
    a2.print();  
    a1.print();  
    a2.globalPrint();  
    cout << "sizeof A: " << sizeof(a1) << " bytes" << endl;
```

Klassenmethodenzugriff  
über Klassennamen

Klassenmethodenzugriff  
über Objektnamen

```
...  
}
```

## Größen von Klassen

- die Größe einer Klasse richtet sich i.A. nach den Größen der Member
- im allg. Fall ist die Größe einer Klasse gleich der Summe der Größen der Member
- Ausnahme: hat eine Klasse keine Member, hat sie dennoch die Größe 1 (= 1 Byte)
- jedoch können Variablen nicht beliebig im Speicher platziert werden
- i.d.R. müssen Wortgrenzen eingehalten werden, so dass auch kleine Variablen (1 oder 2 Byte groß) auf ein ganzes Wort abgebildet werden
- werden nun die Members in einer ungeschickten Reihenfolge deklariert, wird eine Klasse unnötig groß

## Größen von Klassen (Forts.)

- das vorherige Beispiel verschwendet aber immer noch viel Speicherplatz
- für einen booleschen Wert reicht ein Bit
- in C++ kann man hinter den Member angeben, wie viele Bits zu ihrer Repräsentation im Speicher verwendet werden soll

```
bool m_b1 : 1;
```

```
int m_i1 : 10;
```

- doch Vorsicht: es wird weder vom Compiler, noch zur Laufzeit überprüft, ob ein Überlauf stattfindet
- so kann in `m_i1` nur noch Werte zwischen  $-512$  und  $511$  gespeichert werden

Go!

## Beispiel


```
class B1 {  
    bool m_b1;  
    int m_i1;  
    bool m_b2;  
    int m_i2;  
    char m_c;  
};
```

```
class B2 {  
    bool m_b1 : 1;  
    bool m_b2 : 1;  
    char m_c;  
    int m_i1 : 30;  
    int m_i2 : 24;  
};
```

```
class B3 {  
    bool m_b1 : 1;  
    bool m_b2 : 1;  
    int m_i1 : 30;  
    char m_c;  
    int m_i2 : 24;  
};
```

```
int main() {  
    cout << sizeof(B1) << " " << sizeof(B2) << " " << sizeof(B3) << endl;  
    return 0;  
}
```

Vorsicht: hier können  
nicht mehr alle int-Werte  
abgespeichert werden



auch hier gilt: die richtige  
Anordnung bestimmt die  
Speichergröße

# Vorlesung 9

## Ein Beispiel: komplexe Zahlen

- sei eine Klasse zur Darstellung von komplexen Zahlen gegeben

```
class Complex {  
public:  
  
    Complex(double dReal = 0.0, double dImag = 0.0);  
  
    Complex add(const Complex& crArg) const;  
    Complex sub(const Complex& crArg) const;  
    Complex mult(const Complex& crArg) const;  
    Complex div(const Complex& crArg) const;  
    void print(std::ostream&) const;  
  
private:  
  
    double m_dReal;  
    double m_dImag;  
};
```

Go!

## Ein Beispiel: komplexe Zahlen (Forts.)

- dann könnte eine Anwendung wie folgt aussehen:

```
int main() {  
    Complex c1(2.0, 5.0);    // (2 + 5i)  
    Complex c2(3.0, 7.0);    // (3 + 7i)  
    Complex c3;              // (0 + 0i)  
  
    c3 = c1.add(c2);          addiere zu c1 c2 und  
    c3.print(cout);           speichere in c3; gib c3 aus  
    cout << endl;  
  
    c3 = c1.mult(c2);          multipliziere c1 mit c2 und  
    c3.print(cout);           speichere in c3; gib c3 aus  
    cout << endl;  
  
    c3 = c1.div(c2);          dividiere c1 durch c2 und  
    c3.print(cout);           speichere in c3; gib c3 aus  
    cout << endl;  
    ...  
}
```

## Ein Beispiel: komplexe Zahlen (Forts.)

- sehr unschöne Anwendung, weil schwer zu lesen
- es wird noch schwerer, wenn für die 4 komplexen Zahlen  $c_1$ ,  $c_2$ ,  $c_3$  und  $c_4$  der Ausdruck  $c_1 + c_2 * c_3 / c_4$  berechnet werden sollte
- Ergebnis: `c1.add(c2.mult(c3).div(c4))`
- 2 Probleme:
  - sehr schwer zu lesen
  - beim Programmieren müssen die Operatorprioritäten selber festgelegt werden (erst `mult`, dann `div`, zum Schluss `add`)
- somit ist es insgesamt sehr fehleranfällig



# Operatoren

- wären es keine komplexen Zahlen, sondern ein Standardtyp (z.B. int), könnte man die „normalen“ Operatoren verwenden
- wünschenswert: auch für selbstdefinierte Typen (Klassen und Strukturen) möchte man Operatoren implementieren können
- damit wird erreicht, dass selbstdefinierte Typen (z.B. Klasse **Complex**) sich wie Standardtypen verhalten
- in C++ können alle folgende Operatoren mit neuer Funktion versehen werden

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

- dies ist ein wesentlicher Vorteil gegenüber Java im Bereich der technischen und naturwissenschaftlichen Anwendungen

## Operatoren (Forts.)

- mit den Operatoren lässt sich das Beispiel der komplexen Zahlen umschreiben
- der + Operator wird deklariert durch  
`Complex operator+(const Complex& crArg) const;`
- und implementiert (definiert) durch  
`Complex`  
`Complex::operator+(const Complex& crArg) const {...}`
- die Prioritäten lassen sich nicht verändern, d.h. auch für selbstgeschriebenen Operatoren gilt z.B. Punkt- vor Strichrechnung

Go!

## Ein Beispiel: komplexe Zahlen (Forts.)

Complex.h

```
class Complex {  
public:  
...  
    Complex operator+(const Complex& crArg) const;  
    Complex operator-(const Complex& crArg) const;  
    Complex operator*(const Complex& crArg) const;  
    Complex operator/(const Complex& crArg) const;  
...  
};
```

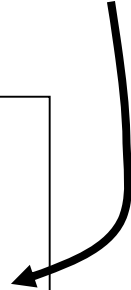
Complex.cpp

```
...  
Complex  
Complex::operator+(const Complex& crArg) const {...}  
  
Complex  
Complex::operator-(const Complex& crArg) const {...}
```

Anwendung sieht  
schon besser aus

main.cpp

```
...  
int main() {  
...  
    c3 = c1 + c2;  
    c3.print(cout);  
    cout << endl;  
...  
}
```



## Operatoren: ein- oder zweistellig?

- bisher wurden die Operatoren als einstellige Elementfunktionen (Methoden) der Klasse implementiert
- die Interpretation (und auch alternative Schreibweise) in C++ lautet dabei:  
 $c1 + c2$  ist identisch zu  $c1.operator+(c2)$
- man kann  $c1 + c2$  aber auch als zweistelligen Operator verstehen:  $operator+(c1,c2)$
- in diesem Fall wäre es keine Elementfunktion, sprich keine Methode der Klasse sondern eine globale Funktion
- diese müssten dann definiert als:

## Operatoren: ein- oder zweistellig? (Forts.)

```
class Complex {  
    ...  
private:  
    double m_bReal;  
    double m_bImag;  
};  
...  
Complex operator+(const Complex& crArg1, const Complex& crArg2) {  
    return Complex(crArg1.m_dReal + crArg2.m_dReal,  
                   crArg1.m_dImag + crArg2.m_dImag);  
}
```

- dies hätte folgende Fehlermeldung zur Folge

```
Complex.h: In function `Complex operator+(const Complex&, const Complex&)':  
Complex.h:17: error: `double Complex::m_dImag' is private  
Complex.cpp:13: error: within this context  
Complex.h:16: error: `double Complex::m_dReal' is private  
Complex.cpp:13: error: within this context
```

## Operatoren: ein- oder zweistellig? (Forts.)

- die globalen Operatoren dürfen (genau wie andere Funktionen) **nicht** auf die private Elemente der Klasse zugreifen
- mögliche Lösungen:
  - Elemente **public** machen (ganz schlechtes Design)
  - lesende Zugriffsfunktionen implementieren (zu aufwendiges Interface)
  - speziell diesen Operatoren den Zugriff auf die private Elemente gestatten (gutes Design)
- einer globalen Funktion (oder Operator) *f* kann für eine Klasse *A* der Zugriff auf die privaten Elemente wie folgt gewährt werden:

```
class A {  
    ...  
    friend void f(...);  
};
```

*f* ist global (keine Element-funktion von *A*!!!!), aber ein Freund von *A*

Go!

## Ein Beispiel: komplexe Zahlen (Forts.)

- damit lässt sich das Beispiel wie folgt umschreiben:

Complex.h

```
class Complex {  
public:
```

```
...
```

```
    friend Complex operator+(const Complex&,const Complex&);  
    friend Complex operator-(const Complex&,const Complex&);  
    friend Complex operator*(const Complex&,const Complex&);  
    friend Complex operator/(const Complex&,const Complex&);
```

```
...
```

kein **const** mehr: globale Funktionen  
können natürlich nicht **const** sein

Complex.cpp

```
...
```

```
Complex
```

```
operator+(const Complex& crArg1,const Complex& crArg2) {...}
```

```
Complex
```

```
operator-(const Complex& crArg1,const Complex& crArg2) {...}
```

kein Klassenname mehr: sind keine  
Elementfunktionen, sondern global

## Operatoren: ein- oder zweistellig? (Forts.)

- Was ist der Vorteil von zwei- gegenüber einstelligen Operatoren?
- Antwort: ein Beispiel

Complex.h

```
class Complex {  
public:  
    Complex(double dReal = 0.0, double dImag = 0.0);  
  
    Complex operator+(const Complex& crArg) const;  
    ...  
};
```

main.cpp

```
...  
int main() {  
    ...  
    c3 = c1 + 5;  
    c3 = 5 + c1;  
    ...  
}
```

Warum ist das ok?

main.cpp: In function `int main()':  
main.cpp:24: error: no match for 'operator+' in '5 + c1'



## Operatoren: ein- oder zweistellig? (Forts.)

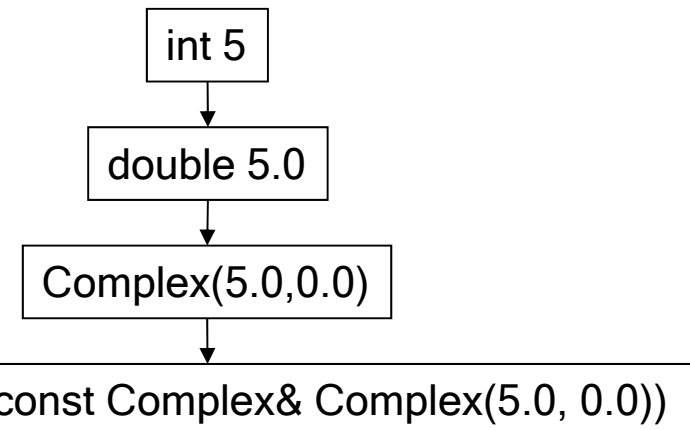
- `c1 + 5` wird interpretiert als `c1.operator+(5)`, wobei 5 ein `int`-Wert ist
- `c1` ist vom Typ `Complex`, es gibt `operator+` für `Complex`, der aber eine konstante Referenz auf ein `Complex` Objekt erwartet
- ein `int`-Wert kann in ein `double`-Wert konvertiert werden
- es gibt einen Konstruktor für `Complex`, der mit einem `double`-Wert auskommt

```
Complex(double dReal = 0.0, double dImag = 0.0);
```

- aus dem `int`-Wert wird ein `double` gemacht
- mit dem `double` wird ein temporäres `Complex` Objekt erzeugt
- dieses temporäre `Complex` Objekt wird dem Operator `operator+` übergeben

## Operatoren: ein- oder zweistellig? (Forts.)

- die automatische Typkonvertierung graphisch dargestellt:
- Frage: warum funktioniert die Typkonvertierung nicht bei `5 + c1`?



- Antwort: hier wird geprüft, ob es für einen `int`-Wert einen Operator `+` gibt, der als zweiten Wert einen `Complex` Typen erwartet
- dies gibt es nicht, daher die Fehlermeldung

## Operatoren: ein- oder zweistellig? (Forts.)

- ist der `+` Operator nicht als Elementfunktion, sondern als globale Funktion definiert, funktionieren beide Aufrufe

`c3 = c1 + 5`

`c3 = 5 + c1`

- hier werden die Aufrufe von `+` interpretiert als `operator+(Complex, Complex)`
- in beiden Fällen wird jeweils ein `Complex` Argument gefunden
- das andere Argument (der `int`-Wert 5) kann in ein `double` und dann in ein `Complex` Objekt umgewandelt werden

Merke: zweistellige Operatoren  
lieber global (friend) definieren  
als lokal als Elementfunktion

## Der Ausgabeoperator

- manche Operatoren lassen sich nur als globale Funktionen definieren
- Grund: die Klasse, für die der Operator definiert werden soll, gibt es schon und kann nicht verändert werden
- Beispiel: der Aus- und Eingabeoperator (<< bzw. >>)  
`std::cout << 17 << std::endl;`
- wird interpretiert als:  
`operator<<(operator<<(std::cout, 17), std::endl);`
- um ein Objekt einer eigenen Klasse Complex mittels des Operators << auszugeben, muss eine neue globale Funktion geschrieben werden

`operator<<(std::ostream&, const Complex&)`

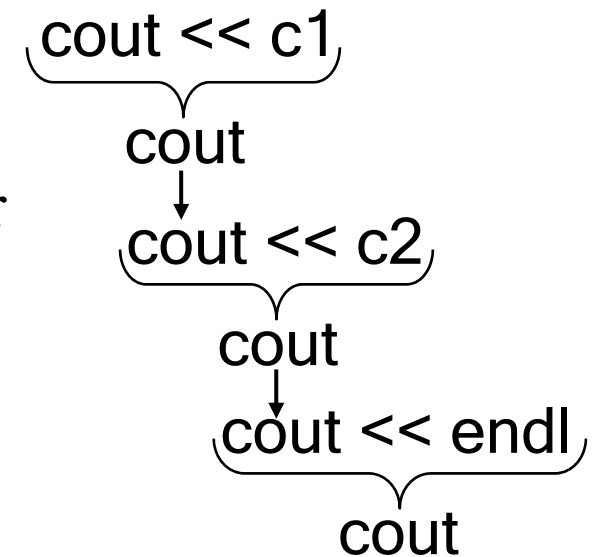
## Der Ausgabeoperator (Forts.)

- um eine Hintereinanderschaltung von << Aufrufen zu erreichen

```
cout << c1 << c2 << endl;
```

muss das Ergebnis von `cout << c1` wieder `cout` sein

- daher muss der `operator<<` das erste Argument (den Ausgabestream) als Ergebnis zurückliefern



```
std::ostream& operator<<(std::ostream&  
                        const Complex&);
```

Go!

## Ein Beispiel: komplexe Zahlen (Forts.)

Complex.h

```
class Complex {  
public:
```

```
...
```

```
    friend std::ostream& operator<<(std::ostream&  
                                   const Complex&);
```

```
...
```

kein **print** mehr, nur noch  
der Ausgabeoperator **<<**

main.cpp

```
...
```

```
int main() {
```

```
    Complex c1(2.0,5.0);
```

```
    Complex c2(3.0,7.0);
```

```
    cout << c1+c2 << endl;
```

```
    cout << c1*c2 << endl;
```

```
    cout << c1/c2 << endl;
```

Anwendung ist  
viel eleganter

Complex.cpp

```
...
```

```
ostream&
```

```
operator<<(ostream& os,const Complex& crArg) {
```

```
    os << "(" << crArg.m_dReal << "+" << crArg.m_dImag << "i";
```

```
    return os;
```

```
}
```

Implementierung (fast) identisch zu **print**  
(Ausnahme: Rückgabe des Streams)

## Vordefinierte Operatoren

- für eine selbstgeschriebene Klasse gibt es normalerweise keine Operatoren
- (fast) alle Operatoren müssen selber geschrieben werden, wenn sie benötigt werden
- 3 Operatoren sind aber vordefiniert, es gibt sie immer zu allen Klassen
  - der Zuweisungsoperator =
  - der Adressoperator &
  - der Sequenzoperator ,
- davon ist der Zuweisungsoperator der kritischste
- oft ist die Standardimplementierung nicht die gewünschte

## Vordefinierte Operatoren (Forts.)

- sollen die vordefinierten Operatoren ausgeschaltet werden, so können sie

- als `private` deklariert werden, und
- nicht implementiert werden

Oldschool, so haben wir das früher vor dem Krieg gemacht

```
class A {  
public:  
    void doit(A& rA) {  
        rA = *this;  
    }  
private:  
    A& operator=(const A&);  
    A* operator&();  
    A& operator,(const A&);  
};
```

kein Compilefehler,  
aber ... Linkerfehler

als `private` deklariert,  
aber nicht implementiert }

```
int main() {  
    A a,b;  
    A* p;  
    a.doit(b);  
    a = b;  
    p = &a;  
    a,b;  
}
```

Compilefehler: Zugriff  
auf `private` Operatoren



## Vordefinierte Operatoren (Forts.)

- ab C++-11 kann mit dem Schlüsselwort
  - **delete** die Standardimplementierung aus-, mit
  - **default** die Standardimplementierung eingeschaltet werden

```
class A {  
public:  
    void doit(A& rA) {  
        rA = *this;  
    }  
    A& operator=(const A&) = delete;  
    A* operator&() = delete;  
    A& operator,(const A&) = delete;  
};
```

Compilefehler

```
int main() {  
    A a,b;  
    A* p;  
    a.doit(b);  
    a = b;  
    p = &a;  
    a,b;  
}
```

so macht man  
es heutzutage

## Der Zuweisungsoperator

- soll die Zuweisung erlaubt sein, muss oft der Zuweisungsoperator implementiert werden
- sein Standardverhalten ist die elementweise Zuordnung der Members
- im Fall der **Complex** Klasse ist dieses Verhalten richtig
- es ist aber nicht im Fall der folgenden **String** Klasse

## Beispiel (Forts.)

```
class String {
public:
    String(const char* p = nullptr) {
        if (p == nullptr) {
            m_pContent = new char[1];
            m_pContent[0] = '\0';
        } else {
            m_pContent = new char[strlen(p)+1];
            strcpy(m_pContent,p);
        }
    }
    ~String() {
        delete [] m_pContent;
    }
    friend ostream& operator<<(ostream& os,const String& crArg) {
        return os << crArg.m_pContent;
    }
    void change(char c,unsigned uiIndex) {
        m_pContent[uiIndex] = c;
    }
private:
    char* m_pContent;
};
```

gebe Speicher am Ende wieder frei

Ausgabeoperator

schreibende Zugriffsfunktion

...

Go!

## Beispiel

```
...  
int main() {  
    String s1(" Hello ");  
    String s2(" World ");  
    cout << s1 << s2 << endl;  
  
    s1 = s2;  
  
    cout << s1 << s2 << endl;  
  
    s1.change('X',3);  
  
    cout << s1 << s2 << endl;  
    return 0;  
}
```

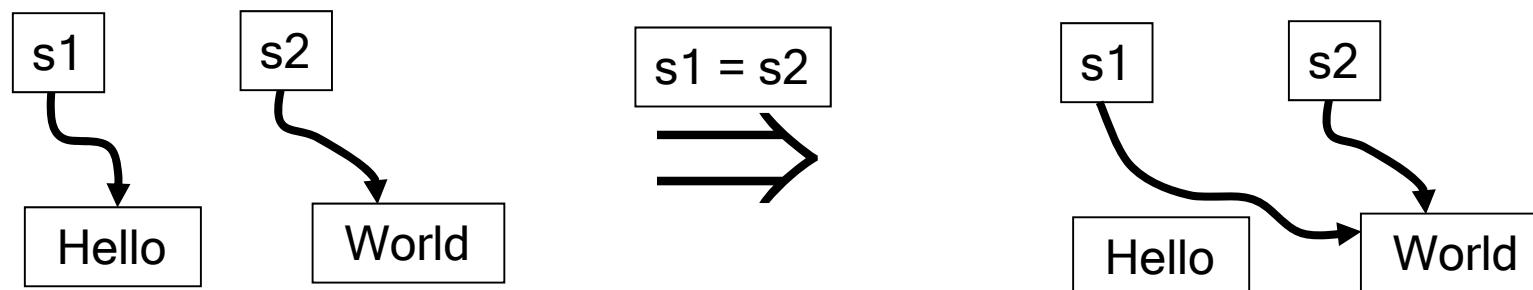
2 String Objekte mit  
jeweils eigenem Text

hiernach sollten s1 und s2  
den gleichen Text haben

hiernach sollten s1 und  
s2 unterschiedlichen  
Text haben, aber ...

## Der Zuweisungsoperator (Forts.)

- Problem: das Standardverhalten des Zuweisungsoperator hat den `char` Pointer (`m_pContent`) einfach kopiert
- danach zeigten `s1.m_pContent` und `s2.m_pContent` auf den gleichen Speicherbereich
- weiteres Problem: der Destruktor gibt zweimal den gleichen Speicher frei
- hier droht die Gefahr eines Programmabsturzes



Go!

## Der Zuweisungsoperator (Forts.)

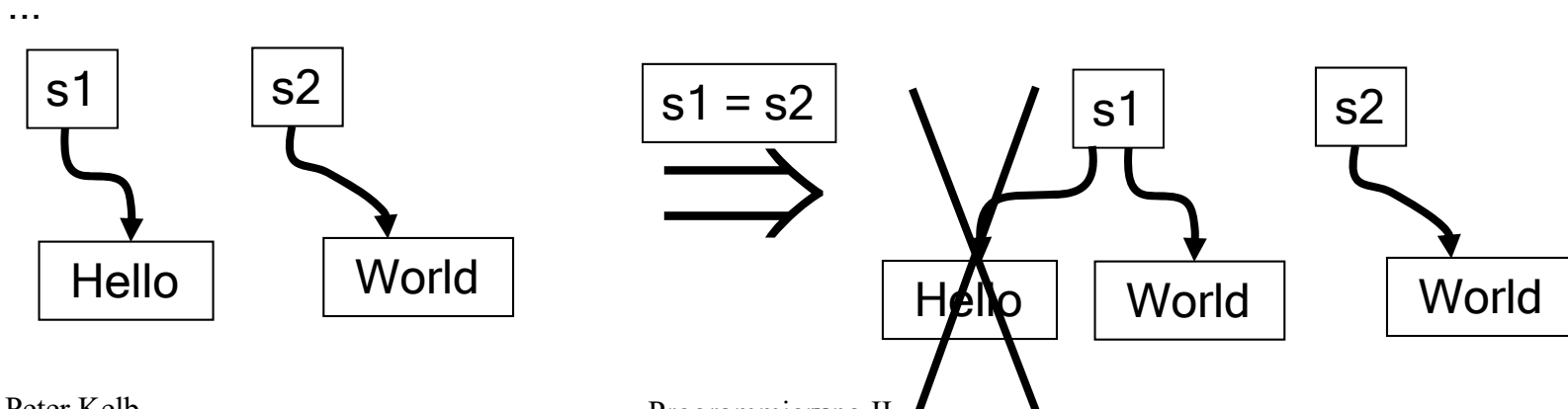
- Lösung: ein eigener Zuweisungsoperator, der den Inhalt des Strings kopiert

```
class String {  
public:
```

bei Selbstzuweisungen  
(a=a) passiert nichts

```
String& operator=(const String& crArg) {  
    if (&crArg != this) {  
        delete [] m_pContent;  
        m_pContent = new char[strlen(crArg.m_pContent)+1];  
        strcpy(m_pContent,crArg.m_pContent);  
    }  
    return *this;  
}
```

1. geben Speicher frei,
2. schaffe neuen Speicher
3. kopiere Inhalt




Go!

## Der Zugriffsoperator

- die `change` Methode erfüllt im wesentlichen die Aufgabe eines Zugriffs auf den String
- anstelle dessen kann man auch den Zugriffsoperator implementieren

```
class String {  
public:  
    char& operator[](unsigned uiIndex) {  
        return m_pContent[uiIndex];  
    }  
...  
}
```

liefert eine Referenz auf einen  
Buchstaben (**char**) zurück



```
int main() {  
...  
    s1[3] = 'X';  
...  
}
```

Anwendung sieht  
aus, als wäre `s1`  
ein **char-Array**

## Der Zugriffsoperator (Forts.)

- Problem: für ein konstantes String Objekt kann der Operator [] nicht für lesende Zugriffe verwendet werden

```
void doit(const String& crArg) {  
    cout << crArg[0] << endl;  
}
```

```
String6.cpp: In function `void doit(const String&':  
String6.cpp:44: error: passing `const String' as `this' argument of `char&  
String::operator[](unsigned int)' discards qualifiers
```

- Lösung: auch noch die konstante Version von Operator [] implementieren

```
class String {  
public:  
    char& operator[](unsigned uiIndex) {  
        return m_pContent[uiIndex];  
    }  
  
    const char& operator[](unsigned uiIndex) const {  
        return m_pContent[uiIndex];  
    }  
}
```

...



# Vorlesung 10

## Der Ink- und Dekrementoperator

- in der Standard Template Library (STL, siehe später) kommen sehr viele sogenannte Iteratoren vor
- diese Iteratoren kann man immer fortschalten mittels des Inkrementoperators ++
- manchmal kann man die Iteratoren auch einen Schritt zurückschalten mittels des Dekrementoperators --
- beide Operatoren kann man für selbstgeschriebene Klassen (z.B. Iteratoren Klassen) selber definieren
- jedoch gibt es von beiden Ink- und Dekrementoperatoren jeweils 2 Versionen
  - Postoperator: i++ bzw i--
  - Preoperator: ++i bzw --i

## Der Ink- und Dekrementoperator (Forts.)

- um jeweils diese beiden Versionen zu unterscheiden gibt es die Operatoren
  - `operator++()` bzw. `operator--()` (Preoperatoren)
  - `operator++(int)` bzw. `operator--(int)` (Postoperatoren)
- der übergebene int-Wert hat keine Bedeutung und sollte in der Operatorimplementierung auch nicht verwendet werden
- er dient ausschließlich dazu, die beiden Versionen voneinander zu unterscheiden
- alle vier Operatoren haben 2 Aufgaben
  - sie verändern das Objekt  $\Rightarrow$  Objektmethoden, nicht const
  - sie liefern ein Objekt zurück:
    - das Veränderte (Preoperatoren)
    - das Unveränderte (Postoperatoren)

Go!

## Beispiel

```
class A {  
public:  
    A() : m_i(0) {}
```

### Ausgabeoperator

```
    friend ostream& operator<<(ostream& os,const A& crArg) {  
        return os << crArg.m_i;  
    }  
}
```

```
A& operator++(){    Preinkrement  
    ++m_i;  
    return *this;  
}
```

```
A operator++(int){  
    A c = *this;  
    ++m_i;  
    return c;    Postinkrement  
}
```

```
private:  
    int m_i;  
};
```

a erst nach  
Ausgabe verändern

```
int main() {  
    A a;  
    cout << a << endl;  
    cout << a++ << endl;  
    cout << a << endl;  
    cout << ++a << endl;  
    cout << a << endl;  
    return 0;  
}
```

a schon vor  
Ausgabe verändern

## Der Ink- und Dekrementoperator (Forts.)

```
A operator++(int){  
    A c = *this;  
    ++m_i;  
    return c;  
}
```

- es ist immer so, dass die Postoperatoren aufwendiger sind als die Preoperator
- bei den Preoperatoren kann das Objekt verändert werden, dann wird das Objekt zurückgeliefert
- bei den Postoperatoren muss erst eine Kopie des Objekts angelegt werden, weil nach der Veränderung des Objekts der ursprüngliche Zustand zurückgeliefert wird
- das kostet i.d.R. deutlich mehr Rechenzeit als im Preoperator Fall

## Operatoren für Typkonvertierung

- in C++ werden viele Typen automatisch in andere Typen konvertiert
- in Java passiert das deutlich seltener (und das ist auch gut so)
- Objekte eigener Klassen werden normalerweise nicht in andere Typen konvertiert (eine Liste kann nicht einem Vektor zugewiesen werden)
- möchte man dennoch, dass die eigene Klasse in andere Typen konvertiert wird, kann man Konvertierungsoperatoren implementieren
- dies kann auf zwei unterschiedliche Arten erfolgen:
  - durch einen Konstruktor des Zieltyps der ein Argument des Ausgangstyps erwartet
  - durch einen Konvertierungsoperator im Ausgangstyp

## Operatoren für Typkonvertierung (Forts.)

- Aufgabe:  
ein Objekt vom Typ **A** soll in ein Objekt vom Typ **B** verwandelt werden
- Möglichkeit 1: in der Klasse **B** gibt es einen Konstruktor, der ein **A** Objekt erwartet

```
class B {  
    public:  
        B(const A& ...);  
};
```
- wird ein **B** Objekt erwartet, aber ein **A** Objekt verwendet, so wird ein temporäres **B** Objekt mit diesem Konstruktor erzeugt und stattdessen verwendet

```
class B;
```

```
class A {  
    friend class B;
```

```
public:
```

```
    A() : m_i(13) {}
```

```
    friend ostream& operator<<(ostream& os,const A& crArg) {  
        return os << "A(" << crArg.m_i << " )";  
    }
```

```
private:
```

```
    int m_i;
```

```
};
```

```
class B {
```

```
public:
```

```
    B() : m_i(0) {}
```

```
    B(const A& crArg) : m_i(crArg.m_i) {}
```

```
    friend ostream& operator<<(ostream& os,const B& crArg) {  
        return os << "B(" << crArg.m_i << " )";  
    }
```

```
private:
```

```
    int m_i;
```

```
};
```

Beispiel

B ist ein Freund von  
A, darf auch private  
Elemente nutzen

Forward Deklaration: man  
muss nur wissen, dass es  
ein B gibt, aber nicht, wie  
es aussieht

Standardkonstruktor

Typkonvertierungs-  
konstruktor  $A \rightarrow B$



Go!

## Beispiel (Forts.)

```
...  
void testB(const B& crArg) {  
    cout << "B: " << crArg << endl;  
}
```

erwartet B Objekt

```
void testA(const A& crArg) {  
    cout << "A: " << crArg << endl;  
}
```

erwartet A Objekt

```
int main() {  
    A a;  
    B b;  
    testB(b);  
    testA(a);
```

keine Konvertierung  
notwendig

```
    testB(a);
```

Konvertierung  $A \rightarrow B$  notwendig

```
// testA(b);  
    return 0;  
}
```

Konvertierung  $B \rightarrow A$   
notwendig, aber...

```
TypeConv.cpp:52: error: invalid  
initialization of reference of type 'const  
A&' from expression of type 'B'  
TypeConv.cpp:42: error: in passing  
argument 1 of `void testA(const A&)'
```

## Operatoren für Typkonvertierung (Forts.)

- in großen Programmen überblickt man nicht mehr, an welchen Stellen eine automatische Typkonvertierung durchgeführt wird
- die Konvertierung mittels Konvertierungskonstruktor kann viel Zeit in Anspruch nehmen
- jedes mal wird ein temporäres Objekt erzeugt, an einer Stelle, von der man glaubte, dass das Objekt direkt mittels Referenz übergeben wird
- um das automatische Konvertieren auszuschalten, aber dennoch den Konvertierungskonstruktor beizubehalten, kann der Konstruktor durch das Schlüsselwort **explicit** gekennzeichnet werden
- ...

## Operatoren für Typkonvertierung (Forts.)

- ...
- dies führt dann zu einer gewünschten Fehlermeldung

...

```
class B {  
public:
```

```
    B() : m_i(0) {}
```

```
    explicit B(const A& crArg) : m_i(crArg.m_i) {}
```

...

```
int main() {
```

...

```
    testB(a);
```

```
TypeConv2.cpp: In function `int main()':  
TypeConv2.cpp:51: error: invalid initialization of reference  
                of type 'const B&' from expression of type 'A'  
TypeConv2.cpp:38: error: in passing argument 1 of `void  
                testB(const B&)'
```

Go!

## Operatoren für Typkonvertierung (Forts.)

- um dennoch die Konvertierung durchführen zu können, muss der Konstruktor explizit (daher das Schlüsselwort) aufgerufen werden

```
...  
class B {  
public:  
  
    B() : m_i(0) {}  
  
    explicit B(const A& crArg) : m_i(crArg.m_i) {}  
}
```

```
...  
int main() {  
    ...  
    testB(B(a));  
}
```

dokumentiert den  
Ort der temporären  
Objekterzeugung

## Operatoren für Typkonvertierung (Forts.)

- Möglichkeit 2: in der Klasse A gibt es einen Operator, der als Ergebnis ein B Objekt zurückliefert

```
class A {  
    public:  
        operator B() const;  
};
```

seltsame Syntax: der  
Rückgabotyp ist der  
Operatorname

- wird ein B Objekt erwartet, aber ein A Objekt verwendet, so wird ein temporäres B Objekt mit diesem Operator erzeugt und stattdessen verwendet

Go!

## Beispiel

```
class B {  
public:  
    B(int i = 0) : m_i(i) {}
```

Konstruktor  
von **B** erweitert

```
...  
};
```

```
class A {  
public:
```

da **B** nicht mehr auf private Elemente  
von **A** zugreift: keine friend Deklaration  
mehr, keine Forward Deklaration mehr

```
    A() : m_i(13) {}
```

```
...
```

```
    operator B() const {  
        return B(m_i);  
    }
```

Typkonvertierungs-  
operator  $A \rightarrow B$

```
private:
```

```
    int m_i;  
};
```

## Typkonvertierung: Mehrdeutigkeiten

- i.d.R. gibt es in großen Programmen viele Konvertierungskonstruktoren, die nicht zur Konvertierung konzipiert worden sind
- durch die Vielzahl solcher Konstrukturen kann es zu Mehrdeutigkeiten kommen, die der Compiler als Fehler anzeigt

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

```
class B {  
public:    B(int i) {}  
};
```

```
void test(const B& crArg) { cout << "test(B&)" << endl;}  
void test(const A& crArg) { cout << "test(A&)" << endl;}
```

```
int main() {  
    test(12);  
}
```

Möglichkeiten: **test(A(12))** oder **test(B(12))**

```
TypeConv4.cpp: In function `int main()':  
TypeConv4.cpp:36: error: call of overloaded `test(int)' is  
                    ambiguous  
TypeConv4.cpp:27: note: candidates are:  
                    void test(const B&)  
TypeConv4.cpp:31: note: void test(const A&)
```

# Vorlesung 11



# Ableitungen

- auch in C++ können Klassenhierarchien durch Ableitungen aufgebaut werden
- es gibt aber mehr Unterschiede als Gemeinsamkeiten zu Java
- man kann:
  - mehrfach vererben (in Java nicht möglich)
  - die Zugriffsrechte der ererbten Elemente kontrollieren (in Java nicht möglich)
  - die virtuellen Methoden kontrollieren (in Java nicht möglich)
- man kann nicht
  - Interfaces definieren, nur abstrakte Klassen (eh eine Hilfskonstruktion in Java)
  - Klassen als konstant deklarieren (in Java mittels final möglich, um weitere Ableitungen zu verhindern)

## Ableitungen (Forts.)

- Ableitungen werden gemäß der folgenden Syntax gebildet

class <KlassenName<sub>1</sub>> : [<Zugriffsrecht>] <KlassenName<sub>2</sub>>

- es wird eine neue Klasse KlassenName<sub>1</sub> deklariert, die von der bereits bestehenden Klasse KlassenName<sub>2</sub> erbt
- die Art, wie die ererbten Elemente von KlassenName<sub>2</sub> in KlassenName<sub>1</sub> sichtbar werden, regelt das optionale <Zugriffsrecht>
- wird es nicht angegeben, so ist es **private** (siehe später)

Go!

## Basisklasse Beispiel

```
class Base {  
public:  
  
    Base() {    cout << "ich bin Base" << endl; }  
  
    void doit() { cout << "doit von Base" << endl;}  
};
```

```
class Derived : Base {  
public:
```

abgeleitetet  
Klasse

```
    Derived() {    cout << "ich bin Derived" << endl;}  
  
    void magic() { cout << "magic von Derived" << endl; }  
};
```

```
int main() {  
    Derived d;  
    d.magic();  
    Base b;  
    b.doit();  
    return 0;  
}
```

## Initialisierung von Ableitungen

- wie bei Java werden die zunächst die Basisklasse initialisiert, dann die abgeleitete Klasse
- da in C++ die Klasse als Member Objekte haben kann (in Java sind es nur Verweise auf Member), können die Member nichttriviale Initialisierungen haben
- die Member werden nach der Basisklasse initialisiert, aber noch vor dem Konstruktor
- Initialisierungsreihenfolge
  1. Basisklasse
  2. Members
  3. Konstruktor

Go!

## Beispiel

```
class A {
public:
    A() {      cout << "ich bin A" << endl; }
};

class B {
public:
    B() {      cout << "ich bin B" << endl; }
};

class Base {
public:
    Base() {    cout << "ich bin Base" << endl; }
private:
    A m_a;
};

class Derived : Base {
public:
    Derived() { cout << "ich bin Derived" << endl; }
private:
    B m_b;
};
...
```

abgeleitetet Klasse:  
erst **Base**, dann **B**

Go!

## Destruktion von Ableitungen

- die Destruktion von Klassen erfolgt natürlich in umgekehrter Reihenfolge:

1. Destruktorcode der abgeleiteten Klasse
2. Destrukturen der Members
3. Destruktor der Basisklasse

```
class A {  
public:  
    A() {    cout << "ich bin A" << endl; }  
    ~A() {   cout << "ich war A" << endl; }  
};  
...  
int main() {  
    Derived d;  
    return 0;  
}
```

neben Konstruktor  
auch einen Destruktor

## Initialisierung der Basisklasse

- wenn die Basisklasse einen nichttrivialen Konstruktor besitzt, oder
- ein nichttrivialer Konstruktor bei der Initialisierung verwendet werden soll, dann
- muss (müssen) der (die) Basiskonstruktor(en) in der Initialisierungsliste aufgeführt werden
- diese werden noch vor den Konstruktoren aufgeführt

```
class A : B {
```

```
    A(int i) : B(i), m_c(2*i) {
```

```
        ...
```

```
    }
```

```
    C m_c;
```

```
};
```

z.B. in Java erfolgte dies  
durch den `super(...)`  
Aufruf zum Anfang des  
Konstruktors

## Sichtbarkeit und Rechte

- bei der Ableitung muss die Spezifikation des Rechts erfolgen, wie die ererbten Elemente (Member und Methods) in der abgeleiteten Klasse sichtbar sind
  - es wird unterschieden zwischen:
    - `public`
    - `private` (Standard, wenn nichts spezifiziert wird)
    - `protected`
- in Java werden die ererbten Rechte 1-zu-1 übernommen
- bei `public` werden die Rechte übernommen
  - bei `private` sind die ererbten Elemente nur in der abgeleiteten Klasse sichtbar; außerhalb und auch in weiter abgeleiteten Klassen sind sie unsichtbar
  - bei `protected` sind die ererbten Elemente in den direkt und indirekt abgeleiteten Klassen sichtbar, außerhalb unsichtbar



## Sichtbarkeit und Rechte (Forts.)

```
class Base {  
public:    void doitPublic() {}  
protected: void doitProtected() {}  
private:  void doitPrivate() {}  
};
```

← immer sichtbar

← in Hierarchie sichtbar

← nie sichtbar

```
class DerivedPublic : public Base {  
public:  
    void doit() {  
        doitPublic();  
        doitProtected();  
        doitPrivate(); // illegal  
    }  
};
```

```
class DDerivedPublic: public DerivedPublic {  
public:  
    void doit() {  
        doitPublic();  
        doitProtected();  
        doitPrivate(); // illegal  
    }  
};
```

```
...  
int main() {  
    DerivedPublic dPub;  
  
    dPub.doitPublic();  
    dPub.doitProtected(); // illegal  
    dPub.doitPrivate(); // illegal  
  
    ...  
}
```

hier sichtbar, hier nicht

## Sichtbarkeit und Rechte (Forts.)

```
class DerivedProtected : protected Base {  
public:
```

wie bei  
public

```
    void doit() {  
        doitPublic();  
        doitProtected();  
        doitPrivate(); // illegal  
    }  
};
```

protected Ableitungen:  
kein Unterschied zu  
public in der Hierarchie

...

```
class DDerivedProtected: public DerivedProtected {  
public:
```

wie bei  
public

```
    void doit() {  
        doitPublic();  
        doitProtected();  
        doitPrivate(); // illegal  
    }  
};
```

...

...

... aber in der  
Anwendung

```
int main() {  
    DerivedProtected dPro;  
  
    dPro.doitPublic(); // illegal  
    dPro.doitProtected(); // illegal  
    dPro.doitPrivate(); // illegal
```

...

```
}
```

## Sichtbarkeit und Rechte (Forts.)

```
class DerivedPrivate : private Base {  
public:
```

```
    void doit() {  
        doitPublic();  
        doitProtected();  
        doitPrivate(); // illegal  
    }  
};
```

**private** Ableitungen:  
Unterschied (!!!) zu  
**public** und **protected**  
in der Hierarchie ...

```
class DDerivedPrivate: public DerivedPrivate {  
public:
```

```
    void doit() {  
        doitPublic(); // illegal  
        doitProtected(); // illegal  
        doitPrivate(); // illegal  
    }  
};
```

Unterschied zu **public**  
und **protected** in der  
nächsten Hierarchieebene

... aber ***nicht*** in  
der Anwendung

```
int main() {  
    DerivedPrivate dPri;  
  
    dPri.doitPublic(); // illegal  
    dPri.doitProtected(); // illegal  
    dPri.doitPrivate(); // illegal  
}
```

## virtuelle Methoden

- in Java ist jede Methode virtuell
- um dies zu verhindern, muss eine Methode als konstant (**final**) deklariert werden
- in C++ ist es genau umgekehrt:
  - standardmäßig ist keine Methode virtuell
  - um sie als virtuell zu kennzeichnen, muss das Schlüsselwort **virtual** vor die Methode gesetzt werden
- Warum ist dies so?

Virtualität (virtual dispatcher)  
gibt es nicht umsonst, der kostet  
viel Speicherplatz und Laufzeit

Go!

## Beispiel

eine, oder mehrer  
virtuelle Methoden  
machen keinen  
Unterschied

```
class A {  
public:  
    void doit() const {}  
private:  
    bool m_b;  
    char m_c;  
};
```

```
class B {  
public:  
    void doit1() const {}  
    virtual void doit2() const {}  
private:  
    bool m_b;  
    char m_c;  
};  
...
```

```
...  
class C {  
public:  
    void doit1() const {}  
    virtual void doit2() const {}  
    virtual void doit3() const {}  
private:  
    bool m_b;  
    char m_c;  
};
```

```
int main() {  
    cout << "size of A " << sizeof(A) << endl;  
    cout << "size of B " << sizeof(B) << endl;  
    cout << "size of C " << sizeof(C) << endl;  
    return 0;  
}
```

## virtuelle Methoden (Forts.)

- die Kosten für virtuelle Methoden in einer Klasse A:  
ein Verweis pro A Objekt auf die virtual Table
- d.h.
  - auf einer 32 Bit Architektur: 4 Byte
  - auf einer 64 Bit Architektur: 8 Byte
- bei 1 Million Objekten (z.B. Knoten in einem Graphen) = 4 (bzw.) 8 MByte für Virtualität
- Was passiert, wenn Methoden nicht virtuell deklariert sind?

Die Auflösung, welche Methoden ausgeführt werden, erfolgt zur Compilezeit.

Genau bedenken, ob von einer Klasse viele Objekte erzeugt werden und ob von dieser Klasse weitere Klassen abgeleitet werden.

Go!

## Beispiel

```
class A {  
public:  
    void doit() const { cout << "A::doit()" << endl; }  
};  
  
class B : public A {  
public:  
    void doit() const { cout << "B::doit()" << endl; }  
};  
  
int main() {  
    A a;  
    B b;  
    A& rA1 = a;  
    A& rA2 = b;  
    B& rB = b;  
    a.doit();  
    b.doit();  
    rA1.doit();  
    rA2.doit();  
    rB.doit();  
    return 0;  
}
```

doit() ist  
nicht virtuell

kritischer Fall:  
rA2 ist vom Typ eine Referenz  
auf ein A Objekt, zeigt aber auf  
ein B Objekt

Go!

## Beispiel

- es reicht, wenn die Methode in der Basisklasse als virtuell deklariert ist

```
class A {  
public:  
    virtual void doit() const {...}  
};
```

```
class B : public A {  
public:  
    void doit() const {  
// virtual void doit() const {...}  
};
```

**virtual** muss **nicht**  
wiederholt werden

```
int main() {
```

```
...  
    B b;  
    A& rA2 = b;
```

```
...  
    rA2.doit();
```

```
...  
}
```

da **doit()** in der Basisklasse virtuell ist,  
wird zur Laufzeit entschieden, welche  
**doit** Methode aufgerufen wird (virtual  
dispatcher)




## virtuelle Destruktoren

- den Destrukturen kommt eine viel wichtigere Aufgabe in C++ zu als in Java
- typischerweise wird in Destrukturen der Speicherplatz freigegeben, den das zugehörige Objekt angefordert hat
- es passiert oft, dass Objekte von Klassenhierarchien über einen Verweis auf die Basisklasse freigegeben werden

```
class A { ... };  
class B : public A { ... };  
...  
A* p = new B();  
...  
delete p;
```

hier muss zur Laufzeit entschieden werden (virtual dispatcher), welcher Destruktor aufzurufen ist



- in diesem Fall muss der Destruktor virtuell sein
- gute Compiler warnen für Klassen mit virtuellen Methoden ohne virtuelle Destruktoren

```
class A {  
public:
```

```
// ~A() {  
    virtual ~A() {  
        cout << "A ist weg" << endl;  
    }  
};
```

```
class B : public A {  
public:
```

```
    ~B() {  
        cout << "B ist weg" << endl;  
    }  
};
```

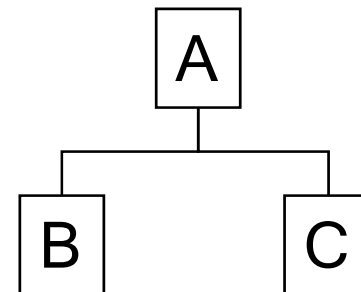
```
class C : public A {  
public:
```

```
    virtual ~C() {  
        cout << "C ist weg" << endl;  
    }  
};  
...
```

## Beispiel

in Ableitungshierarchien  
sind die Destruktoren  
meistens virtuell

es reicht, wenn der  
Basisdestruktor als  
virtuell deklariert wurde



Go!

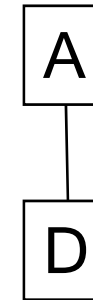
## Beispiel (Forts.)

```
class D : public A {  
public:  
  
    virtual ~D() {  
        cout << "D ist weg" << endl;  
    }  
};
```

```
void test(A* p) {  
    delete p;  
    cout << endl;  
}
```

```
int main() {  
    test(new A);  
    test(new B);  
    test(new C);  
    test(new D);  
    return 0;  
}
```

zweite Ableitungs-  
hierarchie



der Compiler weiß nicht, ob hier ein A,  
B, C oder D Objekt abkommt, dass kann  
erst zur Laufzeit entschieden werden

## abstrakte Klassen

- analog zu Java gibt es auch in C++ abstrakte Klasse
- diese müssen aber **nicht** mit dem Schlüsselwort **abstract** deklariert werden
- eine Klasse ist automatisch abstrakt, sobald mindestens eine Methode abstrakt ist, oder

```
class A {  
    virtual void doit() = 0;    2 abstrakte  
    virtual void makelt() = 0; Methoden  
};
```

- eine Klasse von einer abstrakten Klasse abgeleitet ist, und nicht alle abstrakten Methoden implementiert

```
class B : public A {  
    virtual void doit() { ... }  
};
```

doit ist nicht mehr abstrakt, aber  
makelt ist nicht implementiert  
⇒ B ist abstrakte Klasse

Go!

## abstrakte Klassen (Forts.)

- wie in Java können von abstrakten Klassen keine Objekte erzeugt werden

```
class A {  
public:  
    virtual void doit() = 0;  
};
```

abstrakte Klasse, da  
abstrakte Methode



```
class B : public A {  
public:  
    virtual void doit() { cout << "B::doit()" << endl; }  
};
```

konkrete Klasse, da  
abstrakte Methode  
implementiert ist

```
int main() {  
    B b;  
    // A a; ← würde Fehler erzeugen  
    A* p = &b;  
    b.doit();  
    p->doit();  
    return 0;  
}
```

## Das Slicing Problem

- werden Objekte abgeleiteter Klassen an Methoden übergeben, ist die Parameterübergabe **call-by-value** oder **call-by-reference** nicht nur eine Frage der Effizienz
- bei **call-by-reference** wird das Objekt selber übergeben
- bei **call-by-value** wird von dem Objekt eine Kopie erzeugt
- diese Kopie ist natürlich von dem Typ des Parameters
- der Typ des Parameters muss nicht mit dem Typ des Objekts übereinstimmen
- bei Ableitungshierarchien ist der Parametertyp typischerweise von einem niederen Typen
- damit wird auch nur der Objektanteil bzgl. des niederen Typs kopiert
- damit gehen wichtige Teile verloren

Go!

## Beispiel (Forts.)

```
class A {  
public:  
    virtual void doit() const {  
        cout << "ich bin A::doit" << endl;  
    }  
};
```

```
class B : public A {  
public:  
    virtual void doit() const {  
        cout << "ich bin B::doit" << endl;  
    }  
};
```

```
void test1(A a) {          a.doit(); }  
void test2(const A& a) {    a.doit(); }
```

```
int main() {  
    B b;  
    test1(b);  
    test2(b);  
    ...  
}
```

call-by-value

call-by-reference



**Merke:** Objekte  
abgeleiteter Klassen  
werden immer mittels  
**by-reference** über-  
bzw. zurückgegeben

## Mehrfachvererbung

- im Gegensatz zu Java gibt es in C++ *keine* Interfaces
- dies ist auch nicht notwendig, da Interfaces in Java nur eine Hilfskonstruktion sind, um von mehreren abstrakten Klassen ohne Members ableiten zu können
- in C++ kann eh eine Klasse von mehreren Klassen abgeleitet werden (somit ist die Hilfskonstruktion über Interfaces nicht notwendig)
- Mehrfachvererbung erfolgt durch die Aufzählung der Klassen

```
class A : public B, protected C, public D {  
    ...  
};
```



Go!

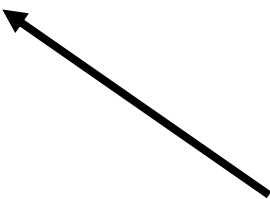
## Mehrfachvererbung (Forts.)

- der Aufruf der Konstruktoren erfolgt in der Initialisierungsliste vor den Members
- der Aufruf erfolgt in der Reihenfolge, in der die Ableitungsdeklaration erfolgt

```
..
A(int i) : m_i(i) {    cout << "ich bin A(" << i << ")" << endl; }
..
B(int i) : m_i(i) {    cout << "ich bin B(" << i << ")" << endl; }
..
class C : public A, public B {
public:
    C(int i) : A(2*i), B(i+13), m_i(i) { }
private:
    int m_i;
};

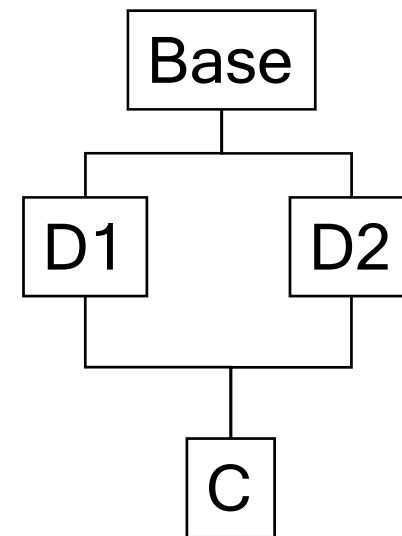
int main() {
    C c(12);
    cout << sizeof(c) << endl;
..
```

Initialisierung: erst die  
Basisklassen, dann die  
Members



## Mehrfachvererbung (Forts.)

- bei der Mehrfachvererbung kann es passieren, dass man von einer Klasse mehrfach erbt
- C erbt Base einmal über D1 und einmal über D2
- dies führt zu Mehrdeutigkeiten
  - beim Aufruf einer Base Methode
  - beim Zugriff auf ein Base Member



Go!

## Beispiel

```
class Base {  
public:  
    Base(int i) : m_i(i) {}  
    void doit() {      cout << m_i << endl;    }  
    int m_i;  
};
```

```
class D1 : public Base {...  
};
```

```
class D2 : public Base {...  
};
```

```
class C : public D1, public D2 {  
public:  
    C(int i) : D1(i), D2(i) {}  
};
```

```
int main() {  
    C c(12);  
    c.m_i = 13;  
    c.doit();  
    return 0;  
}
```

Mehrdeutigkeit: welches  
m\_i ist gemeint

Mehrdeutigkeit: welches  
doit() ist gemeint

Go!

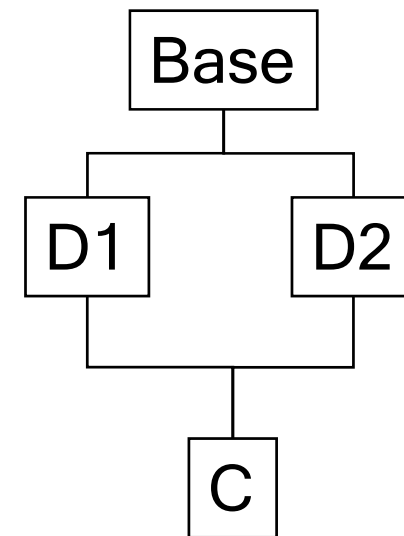
## Mehrfachvererbung (Forts.)

- die Mehrdeutigkeit lässt sich auflösen, indem dem mehrdeutigen Element der Klassenname vorangestellt wird

```
...  
int main() {  
    C c(12);  
    c.D1::m_i = 13;  
    c.D2::doit();  
    return 0;  
}
```

das m\_i, dass über D1 vererbt wurde

das doit, dass über D2 vererbt wurde



Go!

## Mehrfachvererbung (Forts.)

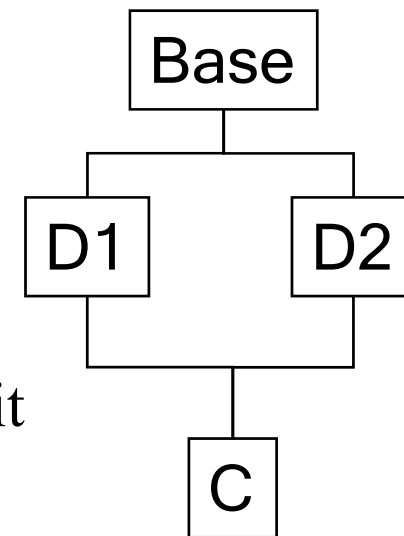
- sollen eventuelle Basisklassen, die durch Mehrfachvererbung mehrfach in einer Klasse enthalten wären nur einmal enthalten sein, so muss von diesen Klassen virtuell abgeleitet werden
- in diesem Fall gibt es auch keine Mehrdeutigkeit (Elemente werden nur einmal vererbt)

```
...  
class D1 : virtual public Base {...};  
class D2 : virtual public Base {...};  
  
class C : public D1, public D2 {  
public:  
    C(int i) : Base(i), D1(i), D2(i) {}  
};  
  
int main() {  
    C c(12);  
    c.m_i = 13;  
    c.doit();  
}
```

Klassen werden  
virtuell vererbt

für die Eindeutigkeit  
muss Base direkt  
initialisiert werden

keine Mehrdeutigkeit mehr



# Vorlesung 12

## Ausnahmebehandlung

- analog zu Java gibt es auch in C++ eine Ausnahmebehandlung
- wie in Java ist in try-catch Blöcken Code enthalten, der normalerweise bzw. im Ausnahmefall ausgeführt werden soll
- anders als in Java kann in C++ fast alles als Ausnahme geworfen werden
- zur Erinnerung: in Java musste jede Ausnahme von der Klasse `Exception` abgeleitet sein
- ansonsten gibt es auch in C++ die Elemente
  - `try { ... }` gefolgt von (mehreren)
  - `catch (<Ausnahmetyp>) { ... }`
  - `throw <Ausnahme>`

Go!

## Beispiel

```
int test(int i) {  
    if (i != 0)  
        return 17 / i;  
    else  
        throw 12;  
}
```


als Ausnahme  
kann fast alles  
geworfen werden



```
int main() {  
    try {  
        cout << test(13) << endl;  
        cout << test(0) << endl;  
        cout << "Ende" << endl;  
    } catch (int j) {  
        cout << "Kein Fehler; Fehlercode: " << j << endl;  
    }  
    return 0;  
}
```

test Aufrufe stehen im try  
Block, da sie prinzipiell eine  
Ausnahme auslösen könnten

fange den Integer  
Fehlercode ab und  
gib ihn aus





Go!

## Ausnahmebehandlung (Forts.)

- die Verwendung eines int Wertes als Ausnahme ist eine schlechte Art, Ausnahmen anzuzeigen
- oft vergisst man, dass Konstanten wie 0 oder 17 vom Typ int und nicht vom Typ unsigned int sind

```
int test(int i) {  
..  
    throw 12;  
}  
  
int main() {  
..  
    } catch (unsigned int j) {  
        cout << "ein Fehler; Fehlercode: " << j << endl;  
    }  
..  
}
```

es wird ein int Wert geworfen,  
aber ...

... nur ein unsigned int  
Wert kann gefangen werden

Ergebnis: der Fehler wird nicht erkannt, das  
Programm terminiert mit einer Fehlermeldung

## Ausnahmebehandlung (Forts.)

- eine deutlich bessere Art der Fehlerbehandlung ist das Werfen von Objekten einer Klasse
- diese Klasse sollte (muss nicht) direkt oder indirekt von der Klasse `std::exception` abgeleitet sein      aus der Standard Template Library
- somit kann immer im Programm mindestens diese Klasse abgefangen werden, ohne dass das Programm unkontrolliert terminiert wird

```
namespace std {  
  
    class exception {  
    public:  
        virtual const char* what() const throw();  
        ...  
    };  
    ...  
}
```

Go!

## Beispiel

```
class MyException : public std::exception {  
public:  
    virtual const char* what() const throw() {  
        return "meine erste Ausnahme";  
    }  
};
```

eine eigene  
Ausnahmeklasse


Methode **what**  
wird überlagert

```
int test(int i) {  
    if (i != 0)  
        return 17 / i;  
    else  
        throw MyException();  
}
```

erzeuge neues Objekt und  
werfe dieses als Ausnahme

```
int main() {  
    try {  
        cout << test(13) << endl;  
        cout << test(0) << endl;  
    } catch (std::exception& e) {  
        cout << "ein Fehler: " << e.what() << endl;  
    }  
}
```

**Wichtig:** fange Referenz auf das  
Objekt und nicht das Objekt selber  
(Slicingproblem)



...

## Ausnahmebehandlung (Forts.)

- hinter der Deklaration von `what` stand `throw()`
- mittels `throw` kann spezifiziert werden, welche Exceptions eine Methode prinzipiell werfen kann (ähnlich zu Java)
- anders als in Java ist es aber kein Compilefehler, falls eine Methode eine andere Exception wirft als spezifiziert ist
- in diesem Fall wird jedoch die Methode `std::unexpected()` aufgerufen
- das Standardverhalten von `std::unexpected()` ist `std::terminate()`, das normalerweise `abort()` aufruft

Ab C++-11 ist diese Art der Spezifikation „deprecated“, sprich sie wird nicht mehr verwendet und wird in späteren C++ Versionen nicht mehr unterstützt

## Ausnahmebehandlung (Forts.)

- kann eine Methode mehrere Exceptions auslösen, so können diese per Komma getrennt in der **throw**-Liste aufgeführt werden
- eine leere **throw**-Liste bedeutet, dass diese Methode keine Exception werfen soll (tut sie es trotzdem, wird **std::unexpected()** aufgerufen)
- gibt es gar keine **throw**-Liste, kann eine Methode jede Exception werfen

Lange Rede, kurzer Sinn, ab C++-11 macht nur noch **throw()** einen Sinn, um zu spezifizieren, dass keine Exception geworfen wird. Besser noch ab C++-11: **noexcept()**

## Ausnahmebehandlung (Forts.)

- wie in Java können mehrere Exception in nachfolgenden **catch**-Blöcken behandelt werden
- dabei gilt (analog zu Java): die Exceptions müssen vom Speziellen zum Allgemeinen aufgeführt werden
- wird diese Reihenfolge **nicht** eingehalten, werden Exception gemäß der allgemeinen Regel verarbeitet
- mit **catch (...)** werden alle Exceptions gefangen
- wenn überhaupt, muss dieser **catch** Block am Ende stehen

## Beispiel

eine Exceptionhierarchie

```
class MyException : public std::exception {  
public:  
  
    virtual const char* what() const throw() {  
        return "meine erste Ausnahme";  
    }  
};
```

```
class MySpecialException : public MyException {  
public:
```

```
    MySpecialException(int i) : m_ci(i) {}
```

```
    virtual const char* what() const throw() {  
        return "spezielle Ausnahme";  
    }  
}
```

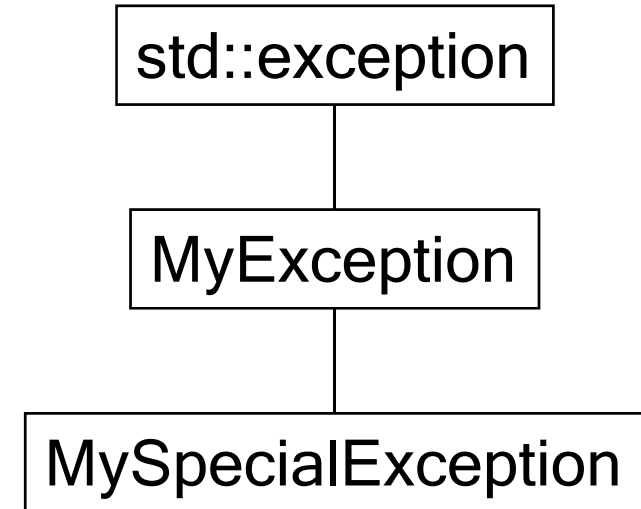
```
    int val() const { return m_ci; }
```

```
private:
```

```
    const int m_ci;
```

```
};
```

```
...
```



passiert häufig:  
Fehlerobjekt merkt sich  
noch Informationen

Go!

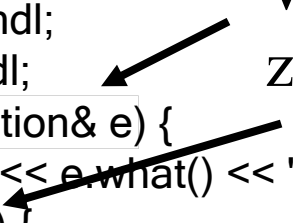
## Beispiel (Forts.)

...

```
int test(int i) {  
    if (i > 0)  
        return 17 / i;  
    else if (i == 0)  
        throw MySpecialException(23);  
    else  
        throw MyException();  
}
```

```
int main() {  
    try {  
        cout << test(13) << endl;  
        cout << test(0) << endl;  
    } catch (MySpecialException& e) {  
        cout << "ein Fehler: " << e.what() << " " << e.val() << endl;  
    } catch (MyException& e) {  
        cout << "ein Fehler: " << e.what() << endl;  
    }  
}
```

**Wichtig:** immer vom Speziellen  
zum Allgemeinen abfragen





## Ausnahmebehandlung (Forts.)

- wie in Java können Exceptions weitergeworfen werden
- dies erfolgt in einem **catch**-Block durch die **throw** Anweisung

```
class MySpecialException : public std::exception {
public:
    MySpecialException(int i) : m_ci(i) {}
    virtual const char* what() const throw() {    return "spezielle Ausnahme";    }
    int val() const {    return m_ci;    }
private:
    const int m_ci;
};

int test(int i) {
    if (i > 0)
        return 17 / i;
    else
        throw MySpecialException(23);
}

...
```

Go!

## Ausnahmebehandlung (Forts.)

```
...
void doit() {
    try {
        cout << test(13) << endl;
        cout << test(0) << endl;
    } catch (MySpecialException& e) {
        if (e.val() == 22)
            cout << "ein Fehler in test: " << e.what() << " " << e.val() << endl;
        else
            throw;
    }
}

int main() {
    try {
        doit();
    } catch (MySpecialException& e) {
        cout << "ein Fehler in doit: " << e.what() << " " << e.val() << endl;
    }
    return 0;
}
```

wenn **e** nicht die richtige  
Exception ist (**==22**), dann  
wird sie weitergeworfen

hier werden alle  
MyException  
Objekte behandelt

## Ausnahmebehandlung (Forts.)

- in Java gibt es die **finally** Anweisung
- Idee: der Code der **finally** Anweisung wird in jedem Fall ausgeführt, entweder am Ende des **try**-Blocks oder am Ende des zugehörigen **catch**-Blocks
- eine solche **finally** Anweisung gibt es in C++ nicht
- dies ist auch nicht notwendig, da mittels Klassen und Objekte ein ähnlicher, besser zu durchschauender Effekt zu erzielen ist
- Frage: was passiert mit den Objekten in einem **try**-Block, die vor einer geworfenen Exception auf dem Stack erzeugt wurden?

```
try {  
    ...  
} catch (<Ausnahmetyp> x) {  
    ...  
} finally {  
    ...  
}
```

Antwort: sie werden destruiert!

Go!

## Beispiel

```
class A {  
public:  
    A(const char* cpMsg) : m_cpMsg(cpMsg) {cout << "+A " << m_cpMsg << endl;}  
    ~A() {cout << "~A " << m_cpMsg << endl;}  
private:  
    const char* m_cpMsg;  
};
```

eine Klasse, die eine Meldung  
im Kon- und Destruktor ausgibt

```
int test(int i) {  
    if (i > 0) return 17 / i;  
    else      throw exception();  
}
```

```
int main() {  
    try {  
        A a1("a1");  
        cout << test(2) << endl;  
        A a2("a2");  
        cout << test(0) << endl;  
        A a3("a3");  
    } catch (exception& e) {  
        cout << "ein Fehler: " << e.what() << endl;  
    }  
}
```

hier tritt der Fehler  
ein, dann sollte erst  
aufgeräumt werden ...

... sprich **a1** und **a2**  
werden destruiert ...

... und dann wird der  
**catch-Block** ausgeführt

...

## Ausnahmebehandlung (Forts.)

- dieses Verhalten kann analog zu **finally** Regel in Java genutzt werden
- folgendes Szenario:
  - eine Methode **eval** legt einen Wert auf einen übergebenen Stack ab
  - ruft eine Methode **evalInternal** mit diesem Stack auf
  - und entfernt danach das Element wieder vom Stack
  - als Invariante soll gelten, dass der Stack vor und nach dem Aufruf von **eval** die gleiche Größe hat
  - **evalInternal** kann eine Exception werfen

## Beispiel

```
#include <iostream>
```

```
#include <stack>
```

← für den Stack

(siehe später STL)

```
using namespace std;
```

```
void evalInternal(const stack<int>& crStack) {
```

```
    int iVal = crStack.top();
```

```
    if (iVal == 0)
```

```
        throw exception();
```

```
    else
```

```
        cout << "evalInternal " << iVal << endl;
```

```
}
```

kann eine

Exception auslösen

```
void eval(stack<int>& rStack,int iVal) {
```

```
    try {
```

```
        rStack.push(iVal);
```

```
        evalInternal(rStack);
```

```
        rStack.pop();
```

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

```
        cout << "ein Fehler ist aufgetreten" << endl;
```

```
    }
```

```
}
```

im Normalfall gilt die Invariante:  
vorher Stack == nachher Stack

Go!

## Beispiel (Forts.)

...

```
void test(stack<int>& rStack,int iVal) {  
    const unsigned cuiSize = rStack.size();  
    eval(rStack,iVal);  
    if (cuiSize != rStack.size())  
        cout << "illegaler Stack" << endl;  
    cout << endl;  
}
```

```
int main() {  
    stack<int> st;  
    test(st,23);  
    test(st,0);  
    return 0;  
}
```

Gilt noch die Invariante?  
Wenn nicht, gib eine  
Fehlermeldung aus

## Ausnahmebehandlung (Forts.)

- dieses Beispiel ist typisch für eine ganze Reihe von Programmen, bei denen durch das Werfen von Exceptions die Invarianten nicht mehr gelten
- oft können die Aufgaben in zwei Teile geteilt werden
  - Initialisierungsteil (vor dem Methodenaufruf)
  - Aufräumteil (nach dem Methodenaufruf)
- das vorgestellte Problem kann mittels eines einfachen Design Patterns gelöst werden
- eine spezielle Klasse übernimmt im Konstruktor den Initialisierungsteil
- im Destruktor werden die Aufräumanteile implementiert
- es wird ein lokales Objekt von dieser Klasse im try-Block angelegt



Go!

## Beispiel (Forts.)

```
...  
void eval(stack<int>& rStack,int iVal) {  
    try {
```

```
        struct Cleaner {  
            Cleaner(stack<int>& rStack,int iVal) : m_rStack(rStack) {  
                m_rStack.push(iVal);  
            }  
            ~Cleaner() {  
                m_rStack.pop();  
            }  
            stack<int>& m_rStack;  
        };  
        Cleaner c(rStack,iVal);
```

lokale Klasse (Struktur), die  
im Konstruktor initialisiert,  
im Destruktor aufräumt

```
        evalInternal(rStack);  
    } catch (...) {  
        cout << "ein Fehler ist aufgetreten" << endl;  
    }  
}
```

Destruktor von c wird  
garantiert aufgerufen

```
...  
}
```

```
void eval(stack<int>& rStack,int iVal) {  
    try {  
        rStack.push(iVal);  
        evalInternal(rStack);  
        rStack.pop();  
    } catch (...) {  
        cout << "ein Fehler ist aufgetreten" << endl;  
    }  
}
```

## Aufbau von großen Programmen

- bisher waren alle Programme in einer Datei gespeichert
- genau wie bei Java werden aber komplexe Programme über mehrere (oft viele) Dateien verstreut implementiert
- in C++ wird oft für jede Klasse zwei Dateien angelegt
  - eine Headerdatei (Endung .h oder .H)
  - eine Implementationsdatei (Endung .cpp oder .cc oder .C)
- Headerdatei: enthält die Klassendeklaration
- Implementationsdatei: enthält die Klassendefinition
- die Headerdateien werden nicht kompiliert
- die Implementationsdateien werden einzeln kompiliert
- das Ergebnis sind Objektdaten (Endung .obj), die zu einem Programm zusammengelinkt werden

## Aufbau von großen Programmen (Forts.)

- wird in einer Klasse **B** die Klasse **A** benötigt, so wird die Headerdatei für die Klasse **A** inkludiert (**include** Anweisung)
- hier muss unterschieden werden, ob **A** für **B** schon bei der Deklaration notwendig ist oder erst bei der Definition
- im Fall der Deklaration muss die Headerdatei von **A** in der Headerdatei von **B** inkludiert werden
- im Fall der Definition muss die Headerdatei von **A** nur in der Implementationsdatei von **B** inkludiert werden

Go!

## Beispiel

### A.cpp

```
#include <iostream>
#include "A.h"

using namespace std;

A::A(int i) : m_i(i) {
    cout << "A(" << m_i << ")" << endl;
}

A::~~A() {
    cout << "~A" << endl;
}

void A::doit() const {
    cout << "A::doit " << m_i << endl;
}
```

### Implementations- datei

### A.h

### Headerdatei

```
class A {
public:
    A(int i);
    ~A();
    void doit() const;
private:
    int m_i;
};
```

### main.cpp

```
#include "A.h"

int main() {
    A a(17);
    a.doit();
    return 0;
}
```

### Implementations- datei

## Aufbau von großen Programmen (Forts.)

- durch den Include Mechanismus kann es passieren, dass eine Headerdatei mehrfach eingeladen wird
- in diesem Fall würde der Compiler einen Fehler melden, dass Elemente mehrfach deklariert sind

Mehrfach-  
inkludierung



```
main.cpp
#include <iostream>
#include "A.h"
#include "A.h"

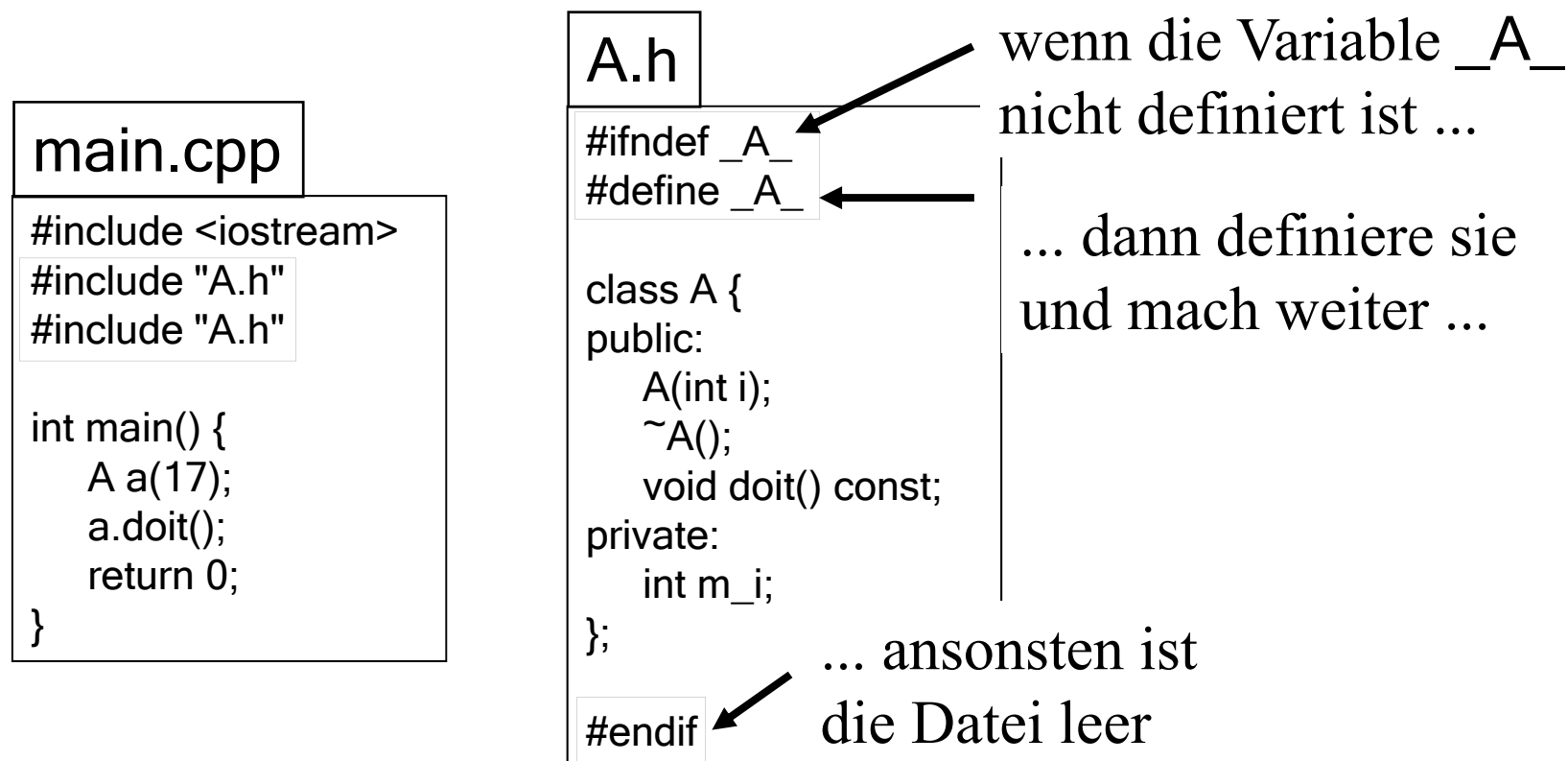
int main() {
    A a(17);
    a.doit();
    return 0;
}
```

```
In file included from main.cpp:3:
A.h:1: error: redefinition of `class A'
A.h:1: error: previous definition of `class A'
```

Go!

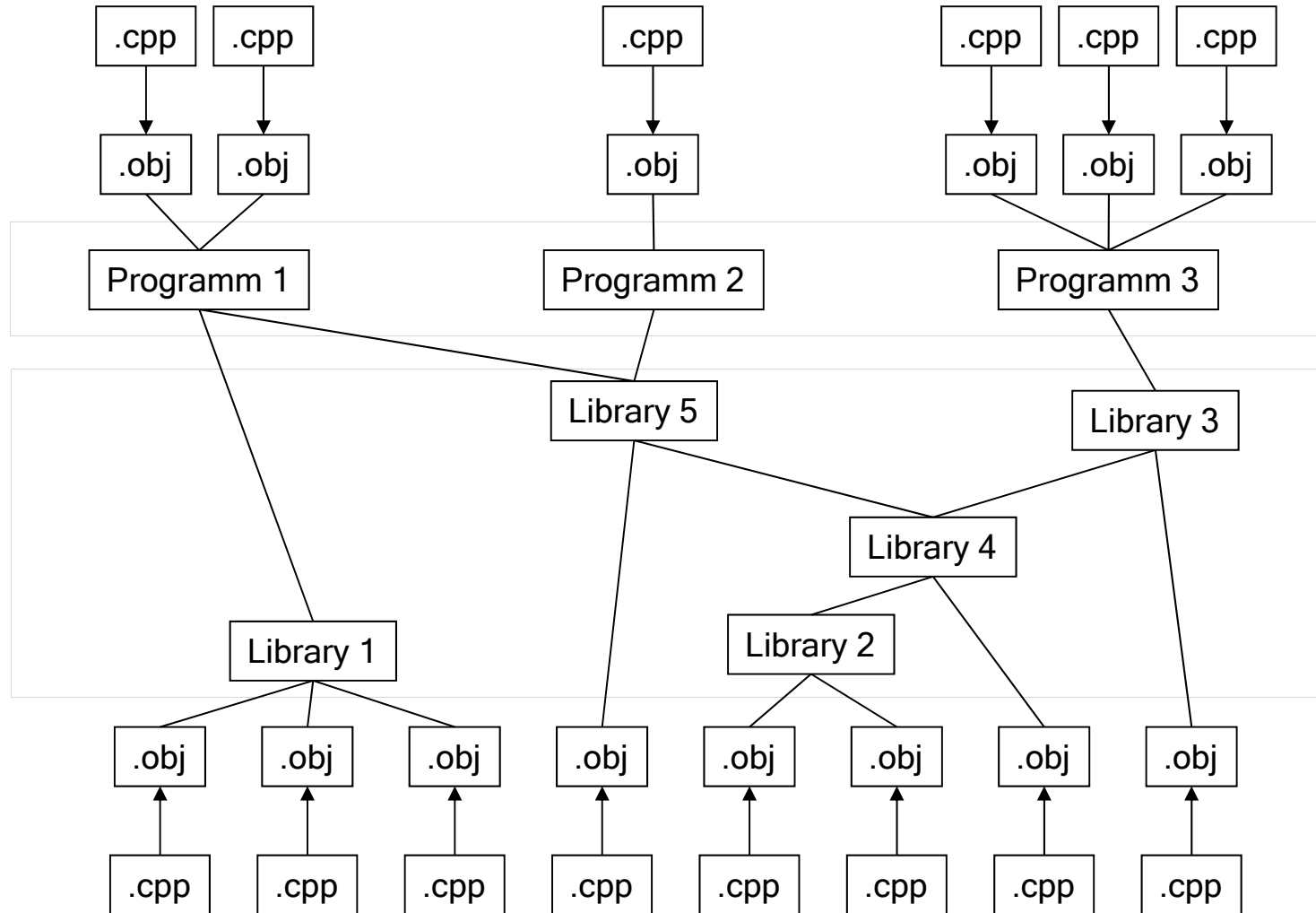
## Aufbau von großen Programmen (Forts.)

- um dies zu verhindern, wird jede Header Datei mit einem Mechanismus versehen, um Mehrfachinkludierungen zu vermeiden
- dies erfolgt über eine bedingte Kompilierung



## Aufbau von großen Programmen (Forts.)

- im allgemeinen werden große Projekte wie folgt strukturiert:

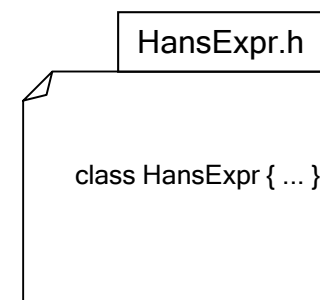
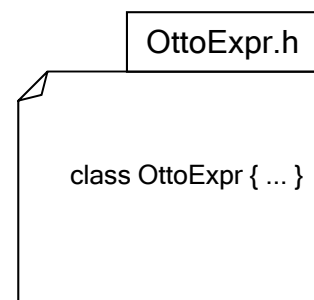


# Vorlesung 13



# Namespaces

- Namespaces dienen dazu, Namenskonflikte in großen Programmen/Projekten zu vermeiden
- übliches Problem in großen Projekten: unterschiedliche Klassen und/oder Methoden haben irgendwann einmal gleiche Namen
- frühere „Lösung“: vor einer Klasse wird der Name (Kürzel) des Entwicklers geschrieben
- sehr unübersichtlich, Namen nicht leserlich



## Namespaces (Forts.)

- um diese Namenskonflikte zu lösen, gibt es in C++ Namensräume, sogenannte Namespaces
- die Syntax ist `namespace <name> { ... }`
- alles was in den geschweiften Klammern steht, ist in dem Namensraum `<name>` deklariert und unterscheidet sich von allem,
  - was in einem anderen Namenraum, oder
  - in dem globalen Namensraum (ohne Namespace)definiert ist
- der Zugriff auf die Elemente erfolgt durch das Voransetzen von `<name>::`

Go!

```
#include <iostream>
```

## Beispiel

```
namespace TollerName {
```

Namensraum TollerName

```
class A {  
public:  
    A() { std::cout << "A+" << std::endl; }  
    void print() const;  
};
```

```
    A* gen() { return new A(); }  
}
```

cout und endl stehen in  
dem Namensraum std

```
void
```

```
TollerName::A::print() const {  
    std::cout << "A::print" << std::endl;  
}
```

```
int main() {  
    TollerName::A* p = TollerName::gen();  
    p->print();  
    delete p;  
    return 0;  
}
```

außerhalb des Namensraum  
muss dieser angegeben werden

## Namespaces (Forts.)

- der Zugriff auf Elemente aus einem Namensraum ist durch die Voranstellung des Namens oft sehr umständlich
- daher kann man Namensräume global öffnen
- `using namespace <name>` öffnet den Namensraum `<name>` bis zum Ende des Blocks, in dem die Anweisung steht
- steht die Anweisung nicht in einem Block, wird der Namensraum bis zum Ende der Datei geöffnet
- passiert dies in einer Header Datei, wird der Namensraum in der Datei geöffnet, die diese Header Datei inkludiert

Niemals Namensräume in Header Dateien öffnen



Go!

```
namespace TollerName {
```

```
    class A {
```

```
        ...
```

```
    };
```

```
    A* gen() {...}
```

```
}
```

Beispiel

Namensraum TollerName

```
void
```

```
TollerName::A::print() const {
```

```
    std::cout << "A::print" << std::endl;
```

```
}
```

```
int main() {
```

```
{
```

```
    using namespace TollerName;
```

```
    A* p = gen();
```

```
    p->print();
```

```
    delete p;
```

```
}
```

```
// A a;
```

```
return 0;
```

```
}
```

öffnet Namensraum  
TollerName bis zum  
Ende des Blocks

Name A ist hier  
nicht mehr bekannt

Go!

## Namespaces: Mehrdeutigkeit

- Elemente mit gleichem Namen können in unterschiedlichen Namensräumen stehen, oder auch im globalen Namensraum
- um auf die Elemente im globalen Namensraum zuzugreifen, muss dem Namen ein `::` vorangestellt werden

```
using namespace std;
```

```
namespace Toll {  
    void doit() {...}  
    void juhu() { ... }  
};
```

Namespace  
Toll

```
void doit() { ... }
```

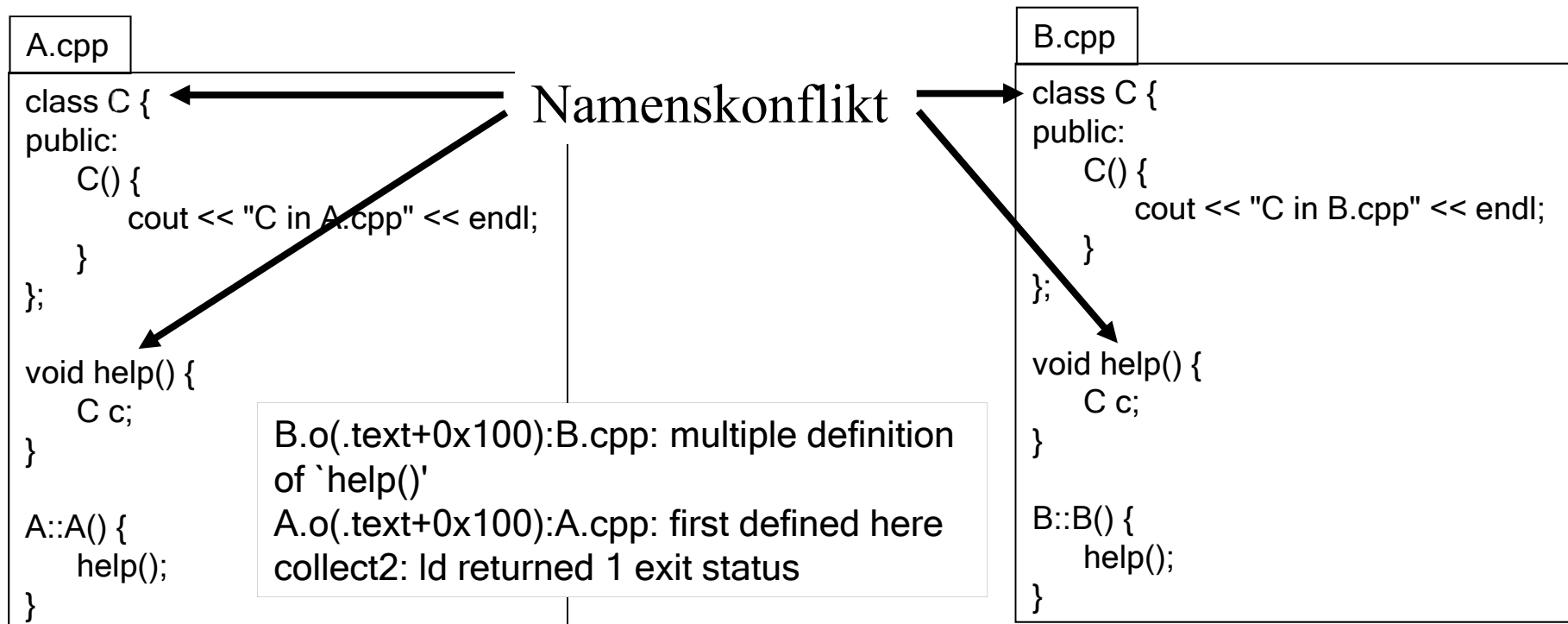
... globaler  
Namens-  
raum

```
...  
int main() {  
    doit();  
    Toll::juhu();  
    {  
        using namespace Toll;  
        Toll::doit();  
        juhu();  
        ::doit();  
    }  
    Toll::doit();  
    doit();  
    ::doit();  
}
```

Mehrdeutigkeit  
auflösen

## Namespaces: Anonym

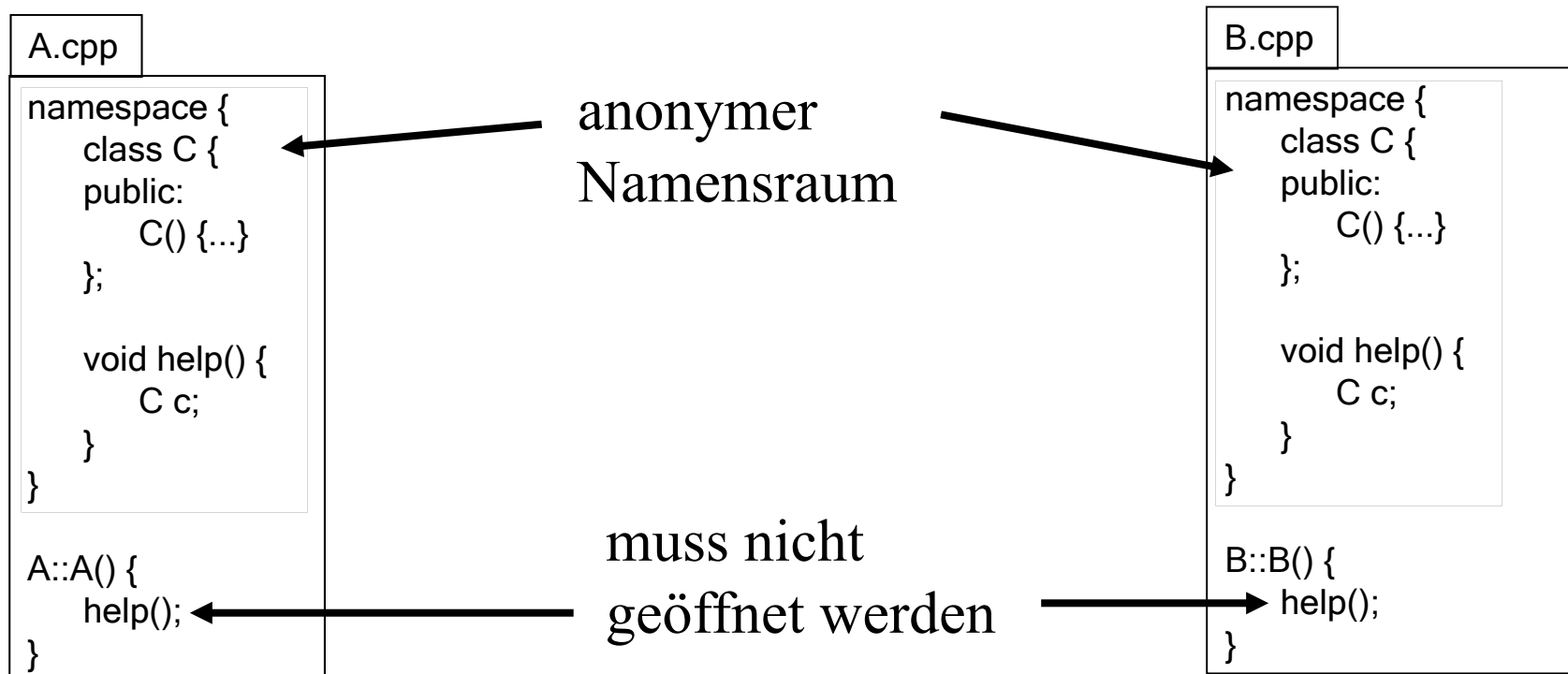
- oft werden in Implementierungsdateien Hilfsfunktionen und Hilfsklassen deklariert und definiert
- in großen Projekten kann es passieren, dass 2 oder mehrerer Hilfsklassen dann den gleichen Namen haben
- dies führt zu einem Linkerfehler



Go!

## Namespaces: Anonym (Forts.)

- Lösung: lokale Hilfsklassen und Funktionen in anonyme Namensräume setzen namespace { ... }
- diese müssen in der Datei nicht geöffnet werden, sind global (in der Datei bekannt)
- sie sind alle unterschiedlich in dem Programm





Go!

## Macros

- Macros dienen dazu, Textmuster durch andere Textmuster zu ersetzen
- dazu muss man zunächst das Macro definieren
- Beispiel: `#define juhu 17`
- im folgenden wird *überall* juhu durch 17 ersetzt

```
#include <iostream>
```

```
using namespace std;
```

```
#define juhu 17
```

ab jetzt wird überall für

```
int main() {
```

juhu 17 geschrieben

```
    cout << juhu + 2 << endl;
```

```
    return 0;
```

```
}
```

Go!

## Macros (Forts.)

- die Ersetzung erfolgt bereits im sogenannten Präprozessor-schritt des C++ Compilers (siehe Compilerbau)
- durch den Einsatz von Macros kann es zu schwer zu durchschauenden Anwendungen kommen

```
#include <iostream>
```

```
using namespace std;
```

```
void juhu() { cout << "juhu" << endl; }
```

```
void toll() { cout << "toll" << endl; }
```

```
#define juhu toll
```

```
int main() {  
    juhu();  
    return 0;  
}
```

ab jetzt wird überall für  
juhu toll geschrieben

Go!

## Macros (Forts.)

- mit Macros hat man auch gute Chancen, *The International Obfuscated C Code Contest* zu gewinnen (1988 wesley.c)

```
#define _ F-->00||-F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
```

Go!

## Macros (Forts.)

- Macros können auch parametrisiert sein
- `#define juhu(X) toll(X)` ersetzt jedes Vorkommen von `juhu(17)` durch `toll(17)`
- Beispiel: Berechnung eines maximalen Wertes

```
#include <iostream>
```

```
using namespace std;
```

```
#define MAX(x,y) (x > y ? x : y)
```

```
int main() {  
    cout << MAX(3,2) << endl;  
    return 0;  
}
```

Macro MAX bekommt 2  
Argumente übergeben

Go!

## Macros (Forts.)

- jedoch ist diese Art mit Vorsicht zu genießen
- Macros werden textuell ersetzt
- dadurch kann es zu nicht überschaubaren Problemen kommen

```
#include <iostream>
```

```
using namespace std;
```

```
#define MAX(x,y) (x > y ? x : y)
```

```
int main() {  
    int a = 6;  
    int b = 7;  
    cout << MAX(a,++b) << endl;  
    return 0;  
}
```

Vorsicht: hier erfolgt  
eine textuelle Expansion



Ergebnis?

## Macros: Beispiel

- die Dereferenzierung von nullptr Pointern führt zu einem Absturz
- man kann natürlich vor jeder Dereferenzierung eine Überprüfung durchführen

```
if (p == nullptr) {  
    cout << „Fehler in Datei ... Zeile ...“ << endl;  
    exit(0);  
} else  
    p->...
```

- um nicht an allen Stellen im Programm (oft viele Tausende) diesen Code hinschreiben zu müssen, kann man ein Macro definieren

## Macros: Beispiel (Forts.)

- Aufgabe des Macros ist es, einen übergebenen Pointer
  - auf nullptr zu überprüfen
  - einen Fehler auszugeben, wenn er nullptr ist
  - ihn zu dereferenzieren, wenn er nicht nullptr ist
- im Fehlerfall
  - soll das Programm beenden werden
  - es soll die Stelle im Programm ausgegeben werden
- dazu kann man die Macros `__LINE__` und `__FILE__` verwenden
- sie werden in jeder Datei bzw. in jeder Zeile neu definiert

Go!

## Macros: Beispiel (Forts.)

Template  
übernimmt die  
eigentliche  
Aufgabe

```
template<class T>
T& Deref_MACRO(T* pPtr,const char* cpFile,int iLineNo) {
    if (pPtr == nullptr) {
        std::cerr << cpFile << ", line " << iLineNo << ": 0-pointer dereferenced" << std::endl;
        std::abort();
    }
    return *pPtr;
}
```

das Dereferenzierungs-  
macro

```
#define Deref(p) Deref_MACRO(p,__FILE__,__LINE__)
```

```
int main() {
    int* p = new int;
    Deref(p) = 34;
    cout << Deref(p) << endl;
    delete p;
    p = nullptr;
    Deref(p) = 34;
    return 0;
}
```

hier entsteht  
ein Fehler

Frage: Warum ist noch ein  
Macro notwendig, warum  
reicht das Template nicht?



## Bedingte Übersetzung

- Macros werden (wenn überhaupt) oft mit bedingten Übersetzungen zusammen angewendet
- Mit bedingten Übersetzungen kann man Code ein- und ausschalten
- Bedingte Übersetzungen werden durch `#ifdef`, `#if`, `#else`, `#elseif` und `#endif` gesteuert
- bisher wurde die bedingte Übersetzung für die Steuerung von Header Dateien verwendet
- oft kann man die bedingte Übersetzung dazu verwendet, Entwicklungsversionen anders als Releaseversionen zu übersetzen

Go!

## Bedingte Übersetzung (Forts.)

- so kostet das Macro zur Überprüfung von 0-Pointern bei der Dereferenzierung viel Rechenzeit
- diese Rechenzeit möchte man nur im Testfall, dem sogenannten Debug-Mode eingeschaltet haben
- in der Release Version, die dem Kunden zugeht, soll die Überprüfung nicht stattfinden

ist das Macro NDEBUB  
*nicht* gesetzt (Standard)

`#ifndef NDEBUB`

`template<class T>`

`T& DEREF_MACRO(T* pPtr,const char* cpFile,int iLineNo) {`

`if (pPtr == nullptr) {`

`std::cerr << cpFile << ", line " << iLineNo << ": 0-pointer dereferenced,, << std::endl;`

`std::abort();`

`}`

`return *pPtr;`

`}`

`#define DEREF(p) DEREF_MACRO(p,__FILE__,__LINE__)`

`#else`

`#define DEREF(p) (*p)`

`#endif`

Testversion

Kundenversion

# Vorlesung 14

# Templates

- oft sind Algorithmen und Datenstrukturen unabhängig von den zu behandelnden Daten
- z.B. spielt es für eine Liste oder einen Vektor keine Rolle, ob Integer oder boolsche Werte, oder komplexe Objekte gespeichert werden
- betrachtet man sich die beispielhafte Implementierung einer einfach verketteten Liste, die `int`-Werte speichert, so fällt auf, dass
  1. nur an wenigen Stellen der Typ `int` benötigt wird
  2. niemals der `int`-Wert an sich benötigt wird

# Beispiel

class List {

Substruktur der  
eigentlichen Liste

```
struct ListElem {
```

```
ListElem(int iElem, ListElem* pNext) :  
    m_pNext(pNext),  
    m_iCont(iElem) {}
```

```
friend ostream& operator<<(ostream& os, const ListElem& crArg) {  
    os << " " << crArg.m_iCont;  
    if (crArg.m_pNext)  
        os << *crArg.m_pNext;  
    return os;  
}
```

```
ListElem* m_pNext;  
int m_iCont;  
};
```

public:

...

einziges  
Vorkommen des  
Speichertyps in  
der Substruktur

Go!

## Beispiel (Forts.)

```
List() : m_pRoot(nullptr) {}
```

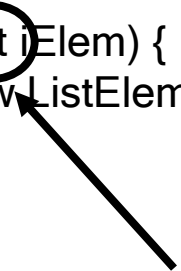
```
friend ostream& operator<<(ostream& os,const List& crArg) {  
    if (crArg.m_pRoot)  
        os << *crArg.m_pRoot;  
    return os;  
}
```

```
void push_front(int iElem) {  
    m_pRoot = new ListElem(iElem,m_pRoot);  
}
```

```
private:  
    ListElem* m_pRoot;  
};
```

```
int main() {  
    List l;  
    l.push_front(12);  
    l.push_front(23);  
    l.push_front(42);  
    cout << l << endl;  
    ...  
}
```

einziges  
Vorkommen des  
Speichertyps in  
der Substruktur



## Templates (Forts.)

- würde man an diesen drei Stelle den Typ `int` durch den Typ `float` ersetzen, hätte man eine Liste für `float`-Werte
- die Liste würde für `float`-Werte genauso wie für `int`-Werte arbeiten
- würde man den `int`-Typ durch den Klassennamen einer selbstgeschriebenen Klasse ersetzen, würde es auch funktionieren, vorausgesetzt, der Copykonstruktor ist definiert (sprich, nicht verboten, er ist ja standardmäßig definiert)
- um die Liste nur einmal implementieren zu müssen, gibt es in C++ sogenannte ***Templates***
- dies sind (zunächst) in Typen parametrisierte Klassen !!!

## Beispiel

```
template<class T>  
class List {
```

```
    struct ListElem {
```

```
        ListElem(T iElem, ListElem* pNext) :  
            m_pNext(pNext),  
            m_iCont(iElem) {}
```

```
        friend ostream& operator<<(ostream& os, const ListElem& crArg) {  
            os << " " << crArg.m_iCont;  
            if (crArg.m_pNext)  
                os << *crArg.m_pNext;  
            return os;  
        }
```

```
        ListElem* m_pNext;  
        T m_iCont;  
    };
```

```
public:
```

```
    ...
```

Einführung eines „Variablen“, die sich nicht Werte, sondern Typen merken kann

kann dort verwendet werden, wo auch sonst Typen stehen



Go!

## Beispiel (Forts.)

```
List() : m_pRoot(nullptr) {}
```

```
friend ostream& operator<<(ostream& os,const List& crArg) {  
    if (crArg.m_pRoot)  
        os << *crArg.m_pRoot;  
    return os;  
}
```

```
void push_front(T iElem) {  
    m_pRoot = new ListElem(iElem,m_pRoot);  
}
```

einziges Vorkommen  
des Speichertyps in der  
Hauptstruktur; auch hier:  
Templateparameter

```
private:  
    ListElem* m_pRoot;  
};
```

```
int main() {  
    List<int> l;  
    l.push_front(12);  
    l.push_front(23);  
    l.push_front(42);  
    cout << l << endl;
```

...

erst bei der Instanziierung muss  
der Typ festgelegt werden

Go!

## Templates (Forts.)

- die Verwendung von Templates schafft eine deutlich bessere Typsicherheit
- wird eine Liste von `unsigned int`-Werten angelegt und ein `int`-Wert abgespeichert, so findet eine Typkonvertierung statt
- nicht der `int`-Wert, sondern der verwandelte Wert wird gespeichert

...

```
int main() {  
    List<unsigned int> l;  
    l.push_front(12);  
    l.push_front(-23);  
    l.push_front(42);  
    cout << l << endl;  
    return 0;  
}
```

hier sollte der  
Compiler warnen



Go!

## Templates (Forts.)

- wie bereits erwähnt, spricht nichts dagegen, auch eigene Klassen als Templateparameter zu verwenden
- jedoch gibt es manchmal Probleme und die Fehlermeldungen mit Templates sind sehr schwer zu verstehen

...

```
class A {  
};
```

```
int main() {  
    A a1,a2,a3;  
    List<A> l;  
    l.push_front(a1);  
    l.push_front(a2);  
    l.push_front(a3);  
    cout << l << endl;  
    return 0;  
}
```

```
List4.cpp: In function `std::basic_ostream<char, std::char_traits<char>  
>& operator<<(std::basic_ostream<char, std::char_traits<char> >&, const  
List<A>::ListElem&):
```

```
List4.cpp:30: instantiated from `std::basic_ostream<char,  
std::char_traits<char> >& operator<<(std::basic_ostream<char,  
std::char_traits<char> >&, const List<A>&)`
```

```
List4.cpp:52: instantiated from here
```

```
List4.cpp:14: error: no match for 'operator<<' in 'std::operator<< [with  
_Traits = std::char_traits<char>](((std::basic_ostream<char,  
std::char_traits<char> >&)(+os)), ((const char*)" ")) << crArg-  
>List<A>::ListElem::m_iCont'
```

...

Go!

## Templates (Forts.)

- Grund für das vorangegangene Problem ist, dass im Ausgabeoperator der Liste der Ausgabeoperator des Listenelements aufgerufen wird, der den Ausgabeoperator des Elements aufruft
- das Element ist vom Typ A
- A hat keinen Ausgabeoperator
- Lösung: Ausgabeoperator für A definieren

```
...
class A {
public:
    friend ostream& operator<<(ostream& os,const A& crArg) {
        return os << &crArg;
    }
};
int main() {
    A a1,a2,a3;
    List<A> l;
```

Go!

## Templates: mehrere Parameter

- Templates können beliebig viele Templateparameter besitzen
- somit ist es möglich, z.B. eine Hashtabelle zu schreiben, die Schlüssel auf Werte abbildet
- diese Hashtabelle kann dann sowohl im Schlüsseltyp als auch im Wertetyp parametrisiert sein
- weiteres Beispiel kann ein Template zur Darstellung von Paaren von Werten sein

2 Parameter

```
template<class T1,class T2>
struct Pair {
    Pair(T1 t1,T2 t2) : m_t1(t1),m_t2(t2) {}
    Pair() : m_t1(),m_t2() {}
    T1 m_t1;
    T2 m_t2;
};
...
```

```
int main() {
    Pair<int,char> p1,p2(17,'c');
    cout << p1.m_t1 << p1.m_t2 << endl;
    cout << p2.m_t1 << p2.m_t2 << endl;
    ...
}
```

## Templates: mehrere Parameter (Forts.)

- die eingesetzten Templatetypen werden direkt mit allen Rechten und Möglichkeiten eingesetzt
- hat der eingesetzte Typ keinen Defaultkonstruktor, kann der Defaultkonstruktor von Pair nicht verwendet werden

```
struct A {  
    A(int) {}  
    friend ostream& operator<<(ostream& os, const A& crArg) {  
        return os << &crArg;  
    }  
};  
  
int main() {  
    Pair<A,char> p1,p2(A(17),'c');  
    cout << p2.m_t1 << p2.m_t2 << endl;  
    return 0;  
}
```

verwendet Default-  
konstruktor: Problem

*Pair2.cpp: In instantiation of 'Pair<T1, T2>::Pair()' [with T1 = A; T2 = char]:*  
*Pair2.cpp:23:15: required from here*  
*Pair2.cpp:9:23: error: no matching function for call to 'A::A()'*  
*Pair() : m\_t1(),m\_t2() {}*

Go!

## Templates: mehrere Parameter (Forts.)

- entfernt man die Deklaration von p1, lässt sich das Programm kompilieren und auch korrekt ausführen

```
int main() {  
    Pair<A,char> p2(A(17),'c');  
    cout << p2.m_t1 << p2.m_t2 << endl;  
    return 0;  
}
```

- das ist insofern sehr verwunderlich, weil der fehlerhafte Konstruktor in dem Template immer noch vorhanden ist

```
template<class T1,class T2>  
struct Pair {  
    Pair(T1 t1,T2 t2) : m_t1(t1),m_t2(t2) {}  
    Pair() : m_t1(),m_t2() {}  
    T1 m_t1;  
    T2 m_t2;  
};  
...
```

fehlerhaft, falls T1 oder T2:

- keine Defaultkonstruktoren haben
- eine Referenz sind

## Templates: you get what you use

- das vorherige Beispiel hat deutlich gezeigt: Templates sind anders als andere Klassen !!!
- im Gegensatz zu anderen Klassen können Templates Code enthalten, der nicht vom Compiler übersetzt werden kann und dennoch kann das Template übersetzt werden und es funktioniert
- Voraussetzung ist, dass der fehlerhafte Code nicht verwendet wird !!!
- bei Klassen ist dies anders, unabhängig von ihrer Verwendung müssen sie fehlerfrei sein



## Templates: you get what you use (Forts.)

- hierbei muss jedoch zwischen unterschiedlichen Fehlern unterschieden werden
  - hat das Template Syntaxfehler, so kann es auch trotz Nichtverwendung nicht kompiliert werden
  - statische Semantikprobleme (Aufruf nichtdefinierter Funktionen, Typfehler, usw.) in nichtverwendeten Templatecode stellt kein Problem da
- dieses Vorgehen hat enorme Vorteile: Templates können Annahmen über ihre Parameter machen und darauf basierend Funktionalität zur Verfügung stellen
- sind die Annahmen korrekt, kann die Funktionalität verwendet werden
- ...

## Templates: you get what you use (Forts.)

- ...
- sind die Annahmen nicht korrekt, wird die Funktionalität aber auch nicht verwendet, sollte auch kein Problem auftreten
- dies ist in Java mit Generics nicht ansatzweise möglich
- Beispiel:
  - das **Pair** Template nimmt an, dass für beide Argumente ein Ausgabeoperator existiert
  - unter dieser Annahme kann ein Ausgabeoperator für ein **Pair** Objekt angeboten werden

Go!

## Beispiel

```
template<class T1,class T2>
struct Pair {
    Pair(T1 t1,T2 t2) : m_t1(t1),m_t2(t2) {}

    Pair() : m_t1(),m_t2() {}

    friend ostream& operator<<(ostream& os,const Pair& crArg) {
        os << "(" << crArg.m_t1 << "," << crArg.m_t2 << ")";
        return os;
    }

    T1 m_t1;
    T2 m_t2;
};

int main() {
    Pair<A,char> p2(A(17),'c');
    cout << p2 << endl;
    return 0;
}
```

Annahme: für T1 und  
T2 existiert jeweils der  
Ausgabeoperator <<

Verwendung des  
Ausgabeoperators  
<< von Pair



## Templates: you get what you use (Forts.)

- wird nun diese Annahme verletzt, sprich **Pair** wird instanziiert mit einer Klasse ohne Deklaration von <<, führt dies bei der Anwendung zu einem Fehler

```
class A {  
};  
  
int main() {  
    Pair<A,char> p;  
    cout << p << endl;  
    return 0;  
}
```

```
Pair4.cpp: In function 'std::basic_ostream<char, std::char_traits<char>  
>& operator<<(std::basic_ostream<char, std::char_traits<char> >&, const  
Pair<A, char>&):'
```

```
Pair4.cpp:25: instantiated from here
```

```
Pair4.cpp:12: error: no match for 'operator<<' in 'std::operator<< [with  
_Traits = std::char_traits<char>](((std::basic_ostream<char,  
std::char_traits<char> >&)(+os)), ((const char*)"(") << crArg->Pair<A,  
char>::m_t1'
```

```
.../ostream.tcc:63: note: candidates are: std::basic_ostream<_CharT,  
_Traits>& std::basic_ostream<_CharT,  
_Traits>::operator<<(std::basic_ostream<_CharT,  
_Traits>&(*) (std::basic_ostream<_CharT, _Traits>&)) [with _CharT =  
char, _Traits = std::char_traits<char>]
```

```
.../ostream.tcc:74: note:          std::basic_ostream<_CharT, _Traits>&  
std::basic_ostream<_CharT,  
_Traits>::operator<<(std::basic_ios<_CharT,  
_Traits>&(*) (std::basic_ios<_CharT, _Traits>&)) [with _CharT = char,  
_Traits = std::char_traits<char>]
```

Go!

Templates: you get what you use (Forts.)

- wird nun diese Annahme verletzt, sprich **Pair** wird instanziiert mit einer Klasse ohne Deklaration von `<<`, wird aber der Ausgabeoperator von **Pair** gar nicht verwendet, so gibt es auch kein Problem

keine Verwendung mehr  
von **Pair::operator<<**

```
class A {  
};  
  
int main() {  
    Pair<A,char> p;  
    // cout << p << endl;  
    return 0;  
}
```

- Syntaxfehler sind aber immer ein Problem

```
friend ostream& operator<<(ostream& os,const Pair& crArg) {  
    os << "(" << crArg.m_t1 << "," << crArg.m_t2 << ")"  
    return os;  
}
```

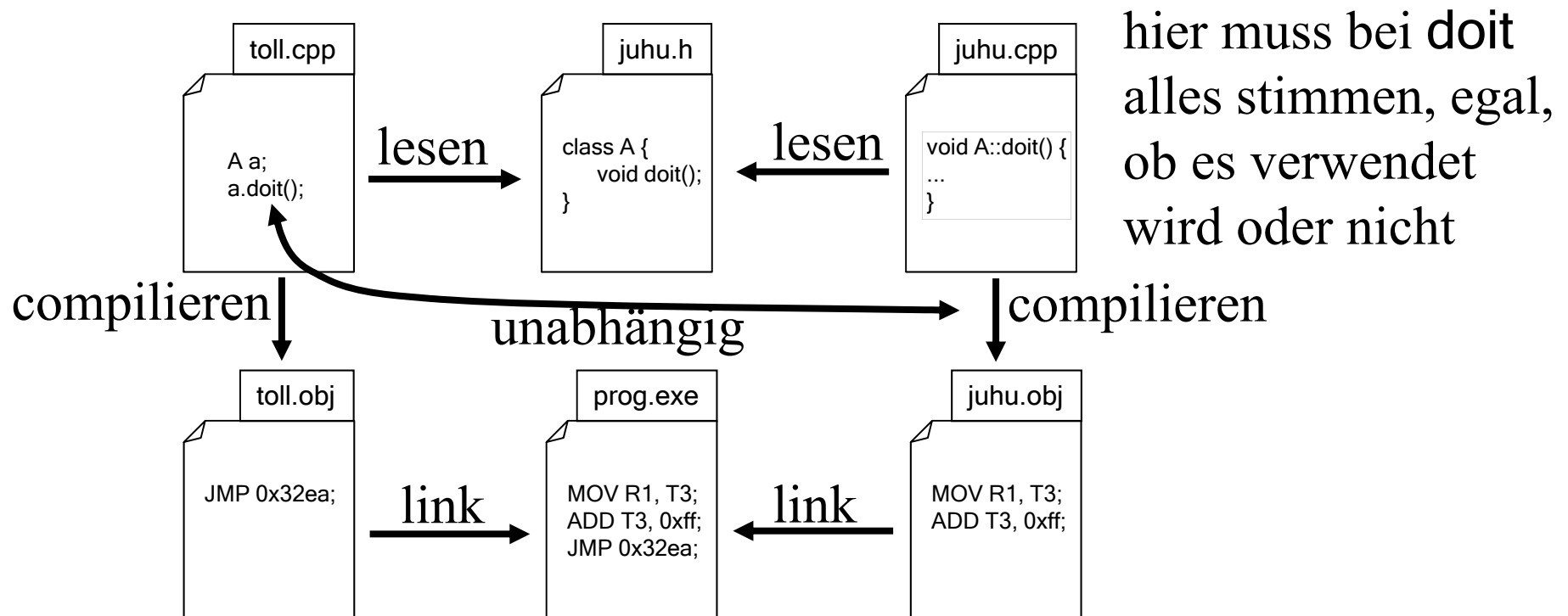
„;“ fehlt

```
Pair6.cpp: In function `std::ostream&  
operator<<(std::ostream&, const Pair<T1, T2>&):  
Pair6.cpp:13: error: expected `;' before "return"
```

## Normalfall

## Templates: Wie werden sie übersetzt?

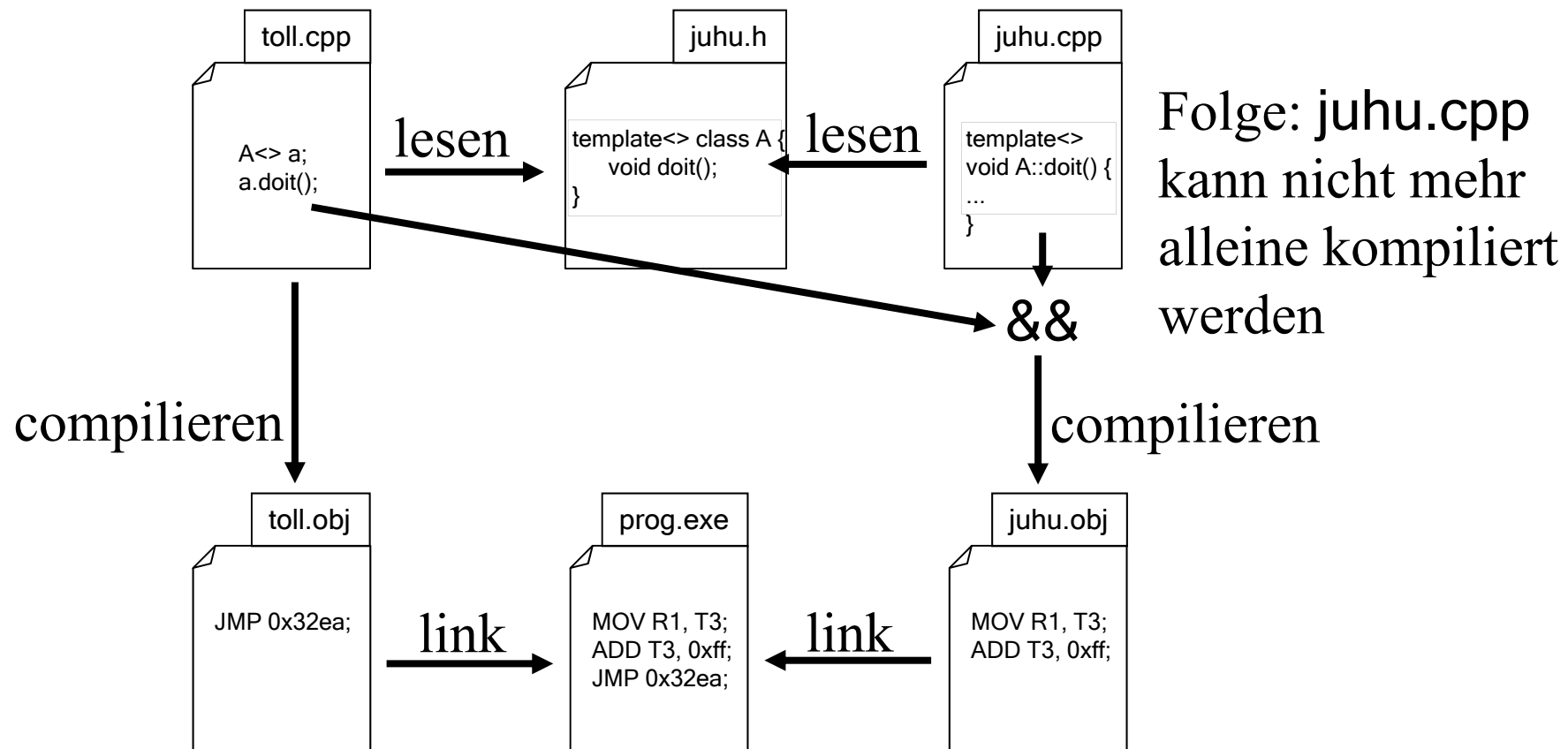
- diese Art der Templatebehandlung (nur benutzter Code muss eine korrekte statische Semantik haben) hat einen großen Einfluss auf die Übersetzung von Templates
- dies führt zu einer restriktiven Verwendung von Templates, die man kennen sollte



# Templatefall

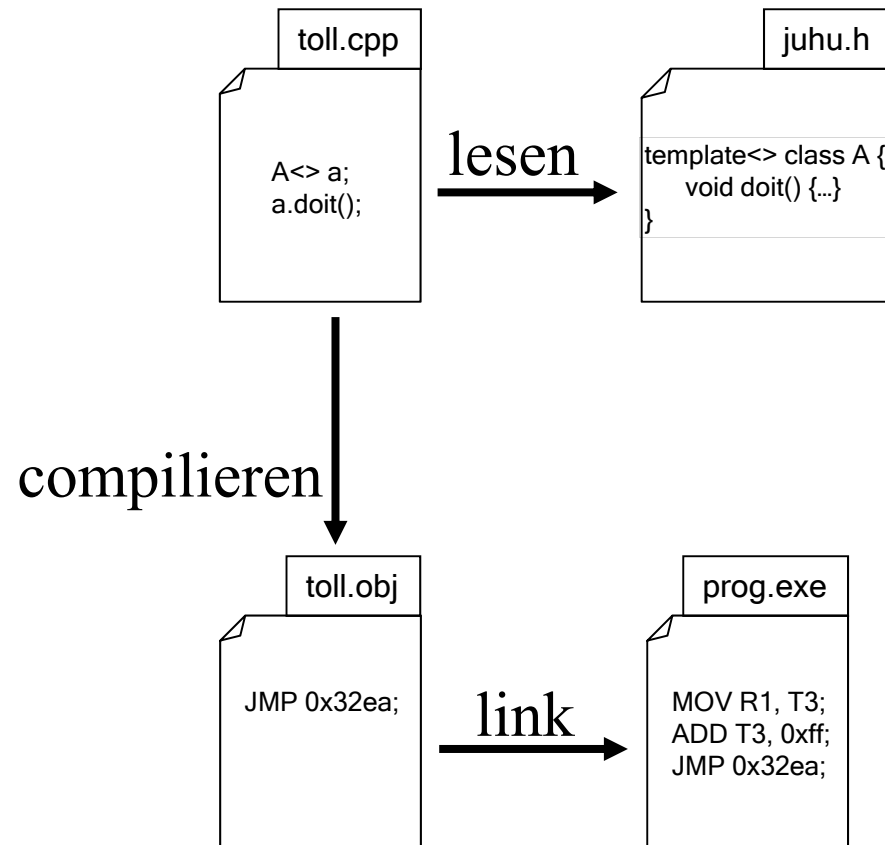
## Templates: Wie werden sie übersetzt? (Forts.)

- im Fall von Templates werden die Methoden und der entsprechende Code erst dann erzeugt und kompiliert, wenn die Methode verwendet wird



## Templates: Wie werden sie übersetzt? (Forts.)

- die Folge ist, dass Templates nicht wie normale Klassen in eine Header und eine Cpp Datei aufgeteilt werden
- die werden sowohl deklariert als auch implementiert in der Headerdatei
- dabei kann die Definition auch getrennt von der Deklaration erfolgen
- sie muss jedoch in der Headerdatei stehen, die von der verwendenden Datei inkludiert wird





Go!

## Beispiel und Schreibweise

```
#ifndef __PAIR_H__  
#define __PAIR_H__  
#include <iostream>
```

### Headerdatei

```
template<class T1,class T2>  
struct Pair {  
    Pair();  
    Pair(T1,T2);  
    T1 m_t1;  
    T2 m_t2;  
};
```

### Template- deklaration

```
template<class T1,class T2>  
Pair<T1,T2>::Pair() : m_t1(),m_t2() {}
```

### Template- definition

```
template<class T1,class T2>  
Pair<T1,T2>::Pair(T1 t1,T2 t2) : m_t1(t1),m_t2(t2) {}
```

```
template<class T1,class T2>  
std::ostream& operator<<(std::ostream& os,const Pair<T1,T2>& crArg) {  
    os << "(" << crArg.m_t1 << "," << crArg.m_t2 << ")";  
}  
#endif // __PAIR_H__
```

```
#include <iostream>  
#include "Pair.h"  
  
using namespace std;  
  
int main(int argc, char *argv[]) {  
    Pair<int,char> p1;  
    Pair<int,bool> p2(17,false);  
  
    cout << p1 << p2 << endl;  
  
    return 0;  
}
```

### Templatefunktion (siehe später)

## Templates: Parametertypen

- bisher waren die Parametertypen ausschließlich Klassen
- neben Klassen können auch Werte elementarer Datentypen Templateparameter sein
- hierdurch ist es möglich, sich konstante Werte zu merken
- Beispiel: ein Vektor von konstanter Größe

```
template<class A, unsigned int uiLength>
class Array {
public:
```

unsigned int als  
Templateparameter

```
    Array() : m_pField(new A[uiLength]) {}
```

```
    ~Array() {
        delete [] m_pField;
    }
```

...

```
    Array<int, 34> v;
```

muss konstant zur  
Compilezeit sein

# Beispiel

```
template<class A,unsigned int uiLength>
class Array {
public:
    Array() : m_pField(new A[uiLength]) {}
    ~Array() {
        delete [] m_pField;
    }
    A& operator[](unsigned int uiIndex) {
        assert(uiIndex < uiLength);
        return m_pField[uiIndex];
    }
    const A& operator[](unsigned int uiIndex) const {
        assert(uiIndex < uiLength);
        return m_pField[uiIndex];
    }
    unsigned int size() const {
        return uiLength;
    }
private:
    A* m_pField;
};
...
```

wird direkt im  
Konstruktor verwendet

überprüft zur Laufzeit  
die Indexgrenzen

Größe muss nicht  
explizit gemerkt werden

Go!

## Beispiel (Forts.)

```
...  
int main() {  
    Array<int,34> v;  
    for(unsigned ui = 0; ui < v.size(); ++ui)  
        v[ui] = ui*ui;  
    for(unsigned ui = 0; ui < v.size(); ++ui)  
        cout << v[ui] << endl;  
    // v[34] = 34;  
    return 0;  
}
```

muss konstant sein

würde zu einem  
Laufzeitfehler führen

## Templates: Parametertypen (Forts.)

- die Konstante wird sich im Code (im Compiler) gemerkt, es wird kein Speicherplatz im Objekt benötigt
- für jeden unterschiedlich *vorkommenden* unsigned int Wert wird eine eigene Klasseninstanz gebildet
- dies kostet Speicherplatz für den generierten Code, der umfangreicher wird
- werden die Klassen jedoch sehr oft instanziiert, so wird in den Objekten Speicherplatz wieder eingespart, da dann die Konstante nicht explizit gemerkt werden muss
- eine andere Lösung wäre gewesen, sich den unsigned int Wert in einer Objektvariablen zu merken
- hier würde sich jedes Objekt die Größe merken müssen
- ...

## Templates: Parametertypen (Forts.)

- ...
- weiterhin gibt es eine weitere Typsicherheit
- ein `Array<int,12>` ist etwas anderes als ein `Array<int,13>`
- dies sind beides unterschiedliche Typen und nicht unterschiedliche Werte wie bei `Array<int>(12)` und `Array<int>(13)`
- der Compiler hat hier die Möglichkeit, Typfehler zur Compilezeit zu erkennen
- Beispiel: Matrizenmultiplikation (siehe Übung)
  - der Compiler kann zur Compilezeit überprüfen, ob die Anzahl der Spalten des 1. Arguments identisch zu der Anzahl der Zeilen des 2. Arguments ist ( $M_{i \times k} \times M_{k \times j} = M_{i \times j}$ )

## Templates: Parametertypen (Forts.)

- es können nicht alle elementaren Datentypen als Template-parameter verwendet werden
- sie müssen in gewissen Sinne abzählbar sein
- dazu zählen
  - alle Ganzzahlen (int, unsigned int, short, ...)
  - bool
  - char
- es funktioniert *nicht* mit
  - float, double
  - Pointern
- da ein String (= char\*) ein Pointer ist, kann einem Template kein String als Parameter übergeben werden (schade, ist aber so)

## Templates: Standardwerte

- Analog zu Funktionsparameter können auch den Templateparameter Standardwerte bekommen
- diese werden dann verwendet, wenn bei der Instanziierung keine Werte oder Klassen angegeben werden
- dennoch muss bei der Syntax der Templateinstanziierung die  $\langle \rangle$  Klammern angegeben werden, selbst wenn alle Parameter durch die Standardwerte besetzt und auch ausgewählt werden
- Beispiel:

```
template<class T = float, unsigned int length = 17>
```

```
class Array {...};
```

```
Array<int, 13> v1;    int Array der Länge 13
```

```
Array<double> v2;    double Array der Länge 17
```

```
Array<> v3;    float Array der Länge 17
```



Go!

## Templates: Standardwerte (Forts.)

- selbstverständlich können auch selbstdefinierte Klassen als Standardwert vorgegeben werden

```
struct A {  
    static void print() { cout << "ich bin print der Klasse A" << endl; }  
};
```

```
template<unsigned int uiLength, class T = A>  
class Magic {  
public:  
    Magic() {  
        for(unsigned ui = 0; ui < uiLength; ++ui)  
            T::print();  
    }  
};
```

nimm Klasse A, falls  
nicht anders definiert

Annahme: Parameterklasse T hat  
eine Klassenmethode print

```
struct B {  
    static void print() { cout << "B***" << endl; }  
};  
...
```

```
int main() {  
    Magic<3> m1;  
    Magic<5,B> m2;  
    return 0;  
}
```

## Templates: Funktionen

- bisher wurden nur Klassen bzgl. Typen parametrisiert
- wünschenswert ist aber auch eine Parametrisierung von Funktionen, um Algorithmen zu verallgemeinern
- so ist ein Sortiervorgehen oft unabhängig davon, was für Elemente sortiert werden sollen
  - sie müssen nur effizient vergleichbar sein
  - sie müssen vertauschbar sein (Assignment Operator muss implementiert sein)
- die Syntax sieht vor, einfach `template<...>` vor einer Funktion zu setzen
- ansonsten gilt (fast) alles, was auch für Klassentemplates gilt
- als Beispiel soll der Bubblesort dienen

## Beispiel

```
template<class T,unsigned uiSize>
void bubbleSort(T* pField) {
    for(unsigned ui = 1;ui < uiSize;++ui) {
        for(unsigned ui2 = 0;ui2 < uiSize-ui;++ui2) {
            if (pField[ui2+1] < pField[ui2]) {
                T tmp = pField[ui2+1];
                pField[ui2+1] = pField[ui2];
                pField[ui2] = tmp;
            }
        }
    }
}
```

Bubblesort: parametrisiert  
im Typ der Elemente und  
in der Anzahl der Elemente

Annahmen:

- T hat Operator < definiert
- T hat Operator = definiert
- T hat Copykonstruktor

```
template<class T,unsigned uiSize>
void print(T* pField) {
    cout << "[";
    for(unsigned ui = 0;ui < uiSize;++ui) {
        if (ui != 0)
            cout << ",";
        cout << pField[ui];
    }
    cout << "]" << endl;
}
```

print: parametrisiert im  
Typ der Elemente und in  
der Anzahl der Elemente

Annahmen:

- T hat Operator << definiert

...

Go!

## Beispiel (Forts.)

```
...
int main() {
    int p1[] = {23,16,-67,123,0};
    float p2[] = {23.23,16.1567,-67.121,123.2,0.45,567.89};
    print<int,5>(p1);
    print<float,6>(p2);

    bubbleSort<int,5>(p1);
    bubbleSort<float,6>(p2);

    print<int,5>(p1);
    print<float,6>(p2);
    return 0;
}
```

sehr umständliche  
Anwendung

## Templates: Funktionen (Forts.)

- würde die Größe der übergebene Arrays nicht als Template-parameter sondern als Funktionsparameter übergeben werden, würde
  - die Anwendung immer noch funktionieren
  - beim Aufruf der Funktion *keine Template*-Parameter angegeben werden müssen
- Grund: bei Templatefunktionsanwendung sucht der Compiler (anders als bei Templateklassen) die richtige Template-instanziierung heraus

Go!

## Beispiel

```
template<class T>
void bubbleSort(T* pField, unsigned uiSize) {
```

```
...
}
```

jetzt Funktionsparameter, nicht  
mehr Templateparameter

```
template<class T>
void print(T* pField, unsigned uiSize) {
```

```
...
}
```

```
int main() {
    int p1[] = {23,16,-67,123,0};
    float p2[] = {23.23,16.1567,-67.121,123.2,0.45,567.89};
```

```
    print(p1,5);
    print(p2,6);
```

```
    bubbleSort(p1,5);
    bubbleSort(p2,6);
```

```
    print(p1,5);
    print(p2,6);
```

```
...
```

deutlich elegantere  
Anwendung