

# Inhalt

## Einführung in C++

- namespaces
- neue header
- ios
- neue strings
- Referenzen
- dynamischer Speicher
- Klassen
- Dateistruktur
- Konstruktoren
- Destruktor

"C++ ist eine universelle Programmiersprache, die vorzugsweise zur Systemprogrammierung dient. Sie

- ist ein besseres C
- unterstützt Datenabstraktion
- unterstützt objektorientiertes Programmieren
- unterstützt generisches Programmieren"

(Bjarne Stroustrup)

## Entstehung:

- C (D.Ritchie, Bell Labs, 1969)
- C with Classes (Stroustrup, ab ~1979)
- C++ (1983)
- ANSI C++ (6/1998)
- C++11 (ISO-Std, 2011)
- C++14 (ISO-Std, Jan 2015)

## namespaces

- Um Namenskonflikte, die z.B. durch den Import von Bibliotheken etc. entstehen können zu vermeiden, wurden im ANSI C++ *namespaces* eingeführt.
- Im 'alten' C++ unterscheidet man zwischen der:
  - *Sichtbarkeit* der Namen auf der Ebene von Blöcken und Dateien
  - *Gültigkeit* (Lebensdauer) der Namen und der damit bezeichneten Objekte und Funktionen (von der Sichtbarkeit unabhängig).
- Die Konstruktion **namespace** erlaubt, zusätzlich *benannte Deklarationsregionen* mit separatem Namensraum anzulegen.

- namespaces können global oder innerhalb anderer namespaces, jedoch niemals in einem Block angelegt werden.
- Der bisherige globale Namensraum existiert weiterhin, aber als unbenannter namespace neben den neuen benannten namespaces.
- Existierende namespaces können erweitert werden, d.h. man kann Deklarationen hinzufügen.
- Der Zugriff auf Elemente eines namespaces erfolgt mit dem *Bereichsoperator (scope operator)* ::

**::Donald**

- Donald im globalen,  
unbenannten namespace

**meinBereich::Donald**

- Donald im namespace  
**meinBereich**

## Beispiel:

```
#include <iostream>
```

```
using namespace std;
```

```
namespace Bereich1
```

```
{
```

```
    float xyz = 1.0f;
```

```
    struct MeinStruct { char c; int i; };
```

```
    void abc();
```

```
}
```

benannter  
namespace

```
float xyz = 10.0f;
```

globale Variable  
im unbenannten  
namespace

```
void inc()
```

```
{
```

```
    Bereich1::xyz = Bereich1::xyz + 1.0f;
```

```
    xyz = xyz + 1.0f;
```

```
}
```

globale Funktion  
im unbenannten  
namespace

namespace

global, unbenannt

## Fortsetzung Beispiel:

```
void main()  
{
```

```
    float xyz = 100.0f;
```

lokale Variable  
verdeckt  
globale Variable

```
    Bereich1::xyz = Bereich1::xyz + 1.0f;
```

```
    cout << Bereich1::xyz << endl;
```

```
    cout << ::xyz << endl;
```

```
    cout << xyz << endl;
```

```
    inc();
```

```
    cout << Bereich1::xyz << endl;
```

```
    cout << ::xyz << endl;
```

```
    cout << xyz << endl;
```

```
}
```

xyz aus namespace

globales xyz

lokales xyz

## using Deklaration

- Die using Deklaration importiert einzelne Namen aus einem Namensbereich und redeklariert sie im aktuellen Bereich.
- Ein solcher Name ist dann unmittelbar verfügbar,   
 → ohne Bereichsoperator.
- Ein weiterer gleichnamiger, lokaler, verdeckender Name kann nicht gleichzeitig existieren.

## Beispiel:

```
void main()  
{
```

```
    using Bereich1::MeinStruct;
```

← using Deklaration

```
    struct MeinStruct  
    {  
        ...  
    };
```

← nicht möglich

```
    MeinStruct A, B;
```

← Zugriff auf Bezeichner  
im namespace  
ohne Bereichsoperator

```
    ...
```

```
}
```



## using Direktive:

- Die using Direktive importiert alle Namen aus einem Namensbereich.
- Diese Namen sind unmittelbar verfügbar,  
➔ ohne Bereichsoperator.
- Weitere gleichnamige, lokale, verdeckende Namen können gleichzeitig existieren.
- Verdeckungen werden mit dem Bereichsoperator aufgelöst.
- Die using Direktive allein führt nicht zum Konflikt, erst die tatsächliche Benutzung eines Namens.

- Die using Direktive widerspricht eigentlich dem Sinn der namespaces, da damit wieder alle Namen sichtbar sind.

→ Empfehlung für große Systeme:

- immer mit dem Bereichsoperator arbeiten:

```
std::cout << "Moin\n";  
std::cin >> xy;
```

- oder die tatsächlich benutzten Namen einzeln re deklarieren:

```
using std::cout;  
using std::cin;  
  
cout << "Moin\n";  
cin >> xy;
```

## Beispiel:

```
void main()  
{
```

```
    using namespace Bereich1;
```

using Direktive

```
    MeinStruct A, B;
```

Zugriff ohne  
Bereichsoperator

```
    A.c = 'X';  
    B.i = 42;
```

```
// xyz = 5;
```

nicht möglich,  
Konflikt globales xyz  
und namespace xyz

```
    Bereich1::xyz = 50;
```

```
    ::xyz = 500;
```

globales xyz

```
    abc();
```

function aus Bereich1

```
}
```

## Aliasnamen:

Bei geschachtelten Namensbereichen kann mit Aliasnamen eine sinnvolle Schreibweise herbeigeführt werden:

```
namespace Bereich1
{
    ...

    namespace Bereich2
    {
        ...

        namespace Bereich3
        {
            char c;
        }
    }
}

...

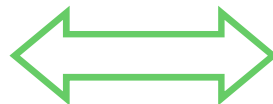
namespace innen = Bereich1::Bereich2::Bereich3;

innen::c = 'Y';
```

## neue Header:

- Die 'neuen Header' **ohne .h extension** (z.B. `<iostream>`) deklarieren alle Namen im namespace **std**.
- Durch die Direktive **using namespace std;** wird der gesamte Namensbereich **std** im globalen Namensbereich sichtbar gemacht.
- Bei Verwendung der 'alten' Header mit .h Extension (z.B. `<iostream.h>`) ist die using Direktive überflüssig, d.h. folgende Anweisungen sind äquivalent:


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



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

## alte Standard C header:

- Für alte Standard C header (z.B. `<stdio.h>` oder `<string.h>`) gibt es neue Entsprechungen, d.h. header, die alle Namen im **namespace std** deklarieren.
- Namenskonvention: alter Header ohne Extension plus **vorangestelltes c**, z.B.:

<code>&lt;stdio.h&gt;</code>		<code>&lt;cstdio&gt;</code>
<code>&lt;string.h&gt;</code>		<code>&lt;cstring&gt;</code>

Hinweis:

!

`<string>` ist neu im ANSI C++ , nicht mit `<string.h>` zu verwechseln !

## ios

- C++ enthält ein objektorientiertes I/O-stream System.
- Das ios besteht aus diversen Klassen und einigen vordefinierten Objekten:
  - `cin`, `wcin` → `stdin`
  - `cout`, `wcout` → `stdout`
  - `cerr`, `wcerr` → `stderr`, (unbuffered)
  - `clog`, `wclog` → `stderr`
- Für die Standarddatentypen – auch für strings – gibt es Überladungen der Stream-Operatoren `<<` und `>>`

```
cin >> x;
```

```
cout << "x = " << x << endl;
```

`<<` kann verkettet werden

**wxxx:**

wide character versions

## neue Strings:

- ANSI C++ definiert eine neue Standard String Bibliothek **<string>**
- Es handelt sich um eine *Template-Klassenbibliothek*, die einfach und sicher zu verwenden ist.
- Speicherbelegung und Speicherfreigabe erfolgen automatisch, d.h. die Länge wird dynamisch angepasst.
- Unzulässige Operationen (Zugriff auf Positionen außerhalb des strings) lösen eine exception aus.

Hinweis: dies gilt nicht, wenn man mit dem **[]**-Operator arbeitet



## **strings** kann man:

- zuweisen
- lexikografisch vergleichen
- einlesen / ausgeben mit **cin**, **cout** etc.
- verketten
- mit 'alten' Strings (NBTS, null-byte-terminated-strings) initialisieren
- nach ihrer Länge befragen
- und ... und ... und ... und ...

## Beispiel:

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

```
void main()
{
```

```
    string str1;
```

default-Konstruktor: leerer string

```
    str1 = "Hakuna";
```

NBTS zuweisen

```
    str1 = str1 + "matata";
```

mit NBTS verketteten

```
    string str2("Timon");
```

weiterer Konstruktor

```
    str2 = str2 + " und " + "Pumba: ";
```

```
    str1 = str1 + '!';
```

einzelnes Zeichen  
anhängen

```
    string str3( str2 + str1 );
```

noch ein Konstruktor

```
    cout << str3 << endl;
```

Ausgabe mit << Operator

## Fortsetzung Beispiel:

```
string str4(str3, 0, 5);
```

← Konstruktor: *quelle, start, anzahl*

```
str1 = str2 = str3 = str4;  
cout << str1 << endl;
```

← Mehrfachzuweisung

```
str1.assign( str2, 1, 4 );  
cout << str1 << endl;
```

← Methode: *quelle, start, anzahl*

```
if( !str1.empty() )  
    cout << "Laenge str1: "  
        << str1.length() << endl;
```

} weitere Methoden

```
}
```

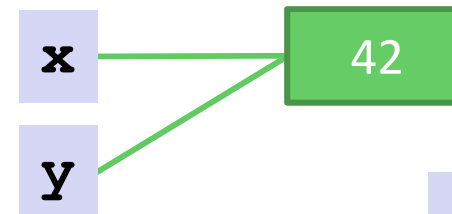
### AUSGABE:

```
Timon und Pumba: Hakunamatata!  
Timon  
imon  
Laenge str1: 4
```

# Referenzen (Lvalue-references\*)

Referenzen sind Aliasnamen für existierende Variablen/Objekte:

```
double x = 42;  
double &y = x;
```



„y referenziert x“

**Dies ist kein Adressoperator!**

Nun kann x oder y verwendet werden:

```
++++x;  
cout << "x = " << ----y << endl;
```

Ausgabe:  
**x = 42**

\*: C++11 definiert auch sog. Rvalue-references

# Referenzen als Parameter

```
#include <iostream>

using namespace std;

void tausche(char& a, char& b)
{
    char tmp = a;
    a = b;
    b = tmp;
}

// Überladung mit Zeigern:
void tausche(char* a, char* b)
{
    char tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Referenztyp



```
int main()
{
    char c1 = 'y';
    char c2 = 'x';

    tausche( c1, c2 );
    cout << c1 << c2 << endl; // -> xy

    char str[] =
        {'C', 'A', '/', 'C', 'D', '\\0'};

    tausche(str[0], str[1]);

    // Überladung:
    tausche( str + 3, &str[4] );
    cout << str << endl; // -> AC/DC

    return 0;
}
```

Adresse



## const Referenzen

- Objekte, die umfangreiche Daten enthalten sollten immer als Referenz übergeben werden
  - ➔ Kopieren vermeiden (keine by-value Übergabe)
- Sollen die Daten innerhalb der Funktion/Methode nicht verändert werden:
  - ➔ **const Referenzen** verwenden

```
void tuWatt(const BigIntClass& biggiObj)
{
    biggiObj.x = 42;
    biggiObj.setX(42);
    double x = biggiObj.getX();
    ...
}
```

← schreibende Zugriffe nicht möglich

## Dynamischer Speicher

Statt der Funktionen `malloc` und `free` werden in C++ die Operatoren `new`, `delete` und `delete[]` verwendet:

→ `new` liefert einen Zeiger zurück

```
double* zd1 = new double;  
double* zd2 = new double(21.0);
```

```
*zd1 = 2.0 * *zd2;  
cout << *zd1 << '\n';
```

Initialwert

```
int n;  
cin >> n;  
int* iFeld = new int[n];  
for(int i = 0; i < n; i++)  
    iFeld[i] = 12345 / 1000 * 3 + 6 + i;
```

dynamisches array

```
cout << iFeld[n-1] << endl;
```

```
delete zd1;  
delete zd2;  
delete[] iFeld;
```

Operator `delete[]`  
für dynamische arrays

```
return 0;
```

# Klassen

Die C++ Konstrukte **class** und **struct** unterscheiden sich nur bzgl. der default-Einstellung der access-modifier:

- **struct** → default **alles public**
- **class** → default **alles private**

```
struct Astruct
{
    int i;
    Astruct() { i = getMagic(); }
    void zeigDich()
    {
        cout << i << endl;
    }
}

private:
    int getMagic() { return 42; }
}aObj;
```

```
class Aclass
{
    private {
        int i;
        int getMagic() { return 42; }
    }

    public:
        Aclass(int i)
        {
            this->i = getMagic() + i;
        }

        int getI() { return i; }
}xObj(42);
```

Zeiger!



# Beispiel Fortsetzung:

**Typ**bezeichner, kein `struct` `Astruct` oder `typedef` erforderlich

```
Astruct bObj ;  
Astruct* pcObj = new Astruct ;  
Astruct feld1[42] ;
```

default-Konstruktoraufruf  
ohne Klammern!

```
Aclass yObj (84) ;  
Aclass* pzObj = new Aclass (168) ;  
Aclass feld2[42] ;
```

non-default-Konstruktoraufruf

kein default-Konstruktor verfügbar

```
aObj.zeigDich() ;  
bObj.zeigDich() ;  
pcObj->zeigDich() ;  
feld1[0].zeigDich() ;
```

```
cout << xObj.getI() / 2 << endl ;  
cout << yObj.getI() / 3 << endl ;  
cout << pzObj->getI() / 5 << endl ;
```

Hinweis:

`aObj`, `bObj`, `feld1`, `xObj`  
und `yObj` sind Stack-Objekte!

## Klassendeklaration:

Die oben gezeigte Klassen**deklaration**\* beinhaltet:

- die Deklaration aller Attribute

- ! – keine Initialisierungen (Werte sind objektspezifisch)

- die Prototypen aller Methoden

- ! – optional Methodenddefinitionen (d.h. mit body in { })

  - ➔ für diese Methoden wird **inline**-code erzeugt  
(im Beispiel oben sind alle Methoden inline)

- die Festlegung der Zugriffsrechte  
(private-, public-, protected-Bereiche)

\*: im Beispiel sind Klassendeklaration und –definition identisch - es gibt keinen extern definierten body

## Klassendefinition:

- Als Klassen**definition** bezeichnet man die Definition aller Methoden einer Klasse
  - ➔ die Definition aller bodies passend zu den definierten Prototypen der Methoden
- die Methodenbezeichner müssen dabei mit dem Vorsatz ***klassename* ::** eindeutig einer Klasse zugeordnet werden, da:
  - verschiedene Klassen identische Methodenbezeichner verwenden können, in einer Datei aber evtl. mehrere Klassen definiert sind
  - ohne den Vorsatz zusätzliche globale Funktionen angelegt würden, die keiner Klasse angehören

# Beispiel

- **Aclass** aufgeteilt in Deklaration und Definition
- jetzt mit default-Konstruktor

// **Deklaration** der Klasse:

```
class Aclass
{
    int i;
    int getMagic();
public:
    Aclass();
    Aclass(int i);
    int getI();
};
```

Instanziierung „Feld von Objekten“  
mit default-Konstruktor:

```
Aclass feld[42];
cout << feld[42-1].getI() + 42
    << endl;
```

// **Definition** der Klasse:

```
Aclass::Aclass()
{
    i = 0;
}
Aclass::Aclass(int i)
{
    this->i = getMagic() + i;
}
int Aclass::getMagic()
{
    return 42;
}
int Aclass::getI()
{
    return i;
}
```

## Dateistruktur

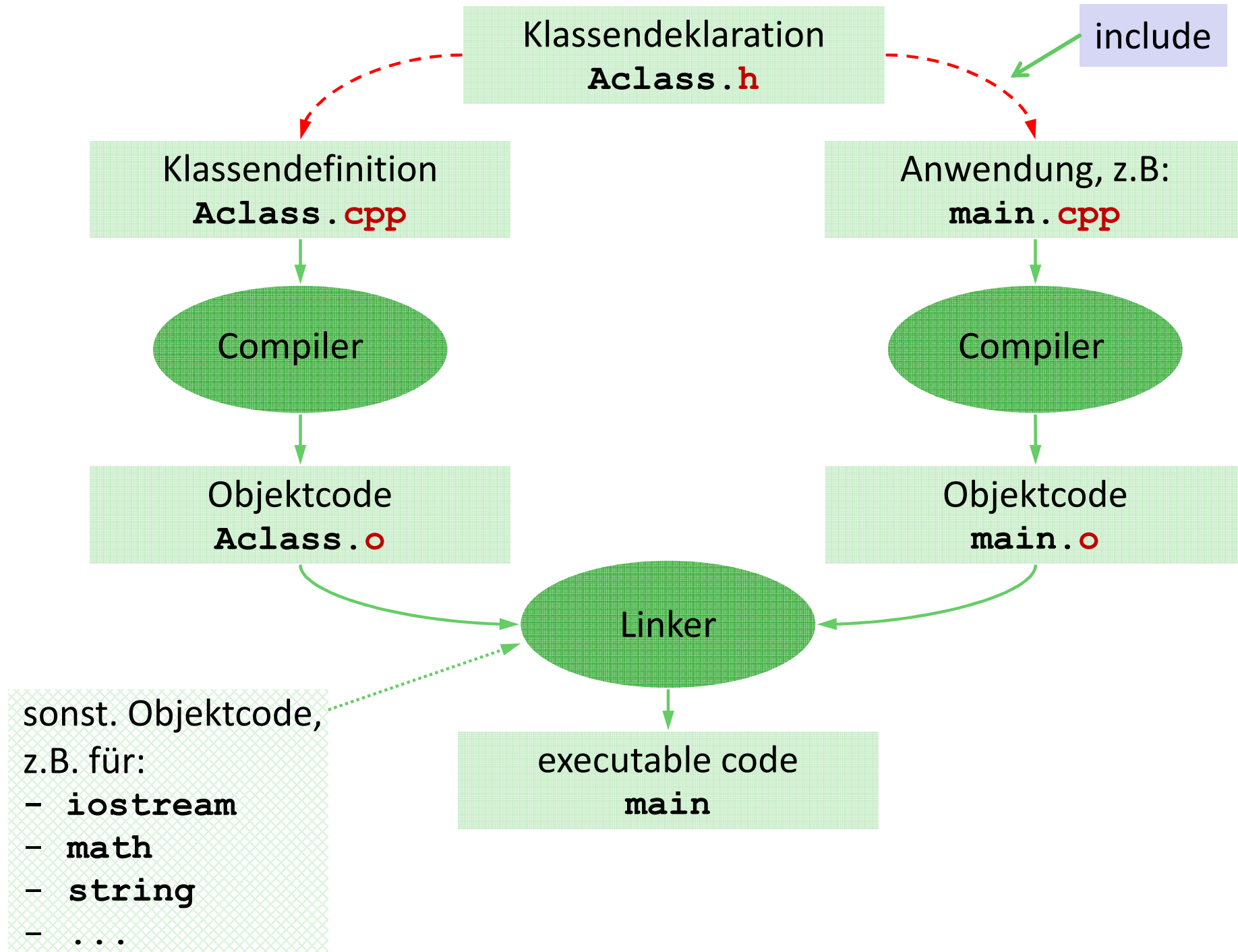
- Klassendeklarationen, Klassendefinitionen, globale Deklarationen, globale Funktionen und main *können* in einer Quelldatei realisiert werden.

Üblich ist:

- \* – pro Klasse **eine Deklarationsdatei** (z.B. *klassenname.h*)  
→ „öffentliche Schnittstelle“, Quelltext meist öffentlich
- pro Klasse **eine Definitionsdatei** (z.B. *klassenname.cpp*)  
→ „interne Definitionen“, oft separat kompiliert
- die Klassendefinitionsdatei und alle Dateien, in denen die Klasse verwendet wird importieren nur die Deklaration der Klasse:

```
#include "klassenname.h"
```

\*: Bei grösseren Projekten werden oft erst alle Klassendeklarationen erstellt und kompiliert.



## Konstrukturen

- **Konstrukturen** führen Instanziierung und Initialisierung in einer Operation aus.
- Der **Standard-** oder **default-Konstruktor**:
  - hat keine Argumente
  - wird vom Compiler nur dann als leere Methode generiert, wenn kein anderer Konstruktor definiert ist
  - wird implizit ausgeführt bei parameterloser Instanziierung (s.o. Feld von Objekten) und bei der Erstellung temporärer Objekte
- Alle Konstrukturen tragen den Namen ihrer Klasse und haben keinen Rückgabetyp (auch nicht `void`).

## – **Konstrukteure** können beliebig überladen werden:

- die Parameter können - wie bei allen Funktionen - *von rechts nach links* mit default-Werten versehen werden (nur im Prototyp angegeben)
  - werden für alle Parameter default-Werte angegeben, so handelt es sich auch um einen default-Konstruktor, der nicht gleichzeitig mit dem parameterlosen Konstruktor bestehen kann!
- durch Überladung und gleichzeitige Verwendung von Parametern mit default-Werten können **Konflikte** entstehen



# Beispiele

1. Konstruktor → `Mitarbeiter() ;`

2. Konstruktor → `Mitarbeiter( string name,  
string vorname, int alter ) ;`

3. Konstruktor → `Mitarbeiter( short schuhgr,  
string name = "Petersen",  
string vorname = "Peter" ) ;`

4. Konstruktor → `Mitarbeiter( int alter ) ;`

- Der default-Konstruktor muss hier explizit implementiert werden, die Existenz von non-default Konstruktoren unterbindet die implizite Generierung des default-Konstruktors durch den Compiler.
- Der 3. Konstruktor kann mit einem, zwei oder drei Parametern aufgerufen werden, d.h. er allein liefert eine dreifache Überladung.
- Für **schuhgr** darf im 3. Konstruktor kein Defaultwert angegeben werden (Konflikt mit Standard-Konstruktor).

## Mögliche Instanziierungen:

```
// 2. Konstruktor, explizite Aufrufform:
```

```
Mitarbeiter franz = Mitarbeiter( "Schulze", "Franz", 55 );
```

```
// 2. Konstruktor, implizite Aufrufform:
```

```
Mitarbeiter josef( "Meier", "Josef", 33 );
```

```
// 3. Konstruktor, implizite Aufrufform:
```

```
Mitarbeiter max( 47, "Riesig" );
```

```
// 4. Konstruktor, implizite Aufrufform:
```

```
Mitarbeiter bert( 40 );
```

```
// 4. Konstruktor, Sonderform des Aufrufs bei einem Parameter:
```

```
Mitarbeiter fritz = 65;
```

```
// Standard-Konstruktor:
```

```
Mitarbeiter* zm = new Mitarbeiter;
```



keine Klammern beim Standardkonstruktor

# Objekte als Feldelemente

Werden Felder mit Objekten angelegt, so erfordert dies meistens die Existenz des Standard-Konstruktors - es sei denn, alle Elemente werden explizit mit einem anderen Konstruktor angelegt:

```
Mitarbeiter team01[10];  
Mitarbeiter team02[4] =  
    { Mitarbeiter("Finn", "Huck", 18 ),  
      Mitarbeiter("Croft", "Lara", 19 ),  
      Mitarbeiter("Max", "Moritz", 20 ) };
```

→ team01 zehn Standard-Konstruktor-Aufrufe

→ team02 ein Standard-Konstruktor-Aufruf (für das 4.Element)

# Initialisierung

Die Initialisierung von Objekten (Attributen) im body von Konstruktoren (per Zuweisung) ist **nicht möglich** für:

- Konstanten
- Referenzen
- Aggregate ohne default-Konstruktor  
(Aggregate sind Objekte in Objekten. Der default-Konstruktor wird für Aggregate implizit ausgeführt – wenn er denn existiert.)

Konstruktor**definitionen** (nur bei Konstruktoren) können mit einer **Initialisierungsliste** versehen werden, die noch **vor Eintritt in den Rumpf ausgeführt** wird.

Diese Liste kann verwendet werden zur Initialisierung von:

- Attributen aus Standarddatentypen
- Aggregaten (Aufruf spezifischer Konstruktoren)
- Konstanten
- Referenzen

Die Syntax der Initialisierungsliste kann auch sonst verwendet werden:

```
int i = 42;           oder int i(42);  
double d = 12.34;    oder double d(12.34);
```

## Beispiel Initialisierungsliste

```
class A
{
    int i, j, ij;
public:
    A( int x, int y )
        : i(x), j(y), ij(x*y) { }
    int geti() { return i;}
    int getj() { return j;}
    int getij() { return ij;}
};
```

```
class B
{
    int k;
    int& rk;
    const float f;
    A agg; ← kein default-Konstruktor
public:
    B( float ff, int ii );
};
```

```
// Definition der Klasse B:
B::B( float ff, int ii )
    : agg( ii, 2*ii ),
      f( ff / 5.0f ), k(12345), rk(k)
{
    // Ausgabe von:
    // k, rk, f,
    // agg.i, agg.j, agg.ij
    cout << ...
}

int main()
{
    B obj( 42.0, 42 );

    return 0;
}
```

## Destruktor

Endet die Lebensdauer eines Objekts, so wird sein *Destruktor* aufgerufen:

- Der Destruktor :
  - hat keine Argumente
  - wird vom Compiler als leere Methode generiert, sofern er nicht explizit implementiert wird
  - kann nicht überladen werden
  - sein Name ist *~klassenname*
  - er hat keinen Rückgabetyt
- Der Prototyp des **Aclass**-Destruktors lautet somit:

***~Aclass* ( ) ;**

- Der Destruktor wird immer implizit ausgeführt:
  - bei **static** Objekten bei Prozessende
  - bei stack-Objekten, wenn der Block verlassen wird
  - bei mehreren stack-Objekten im selben Block
    - ➔ Reihenfolge reziprok zur Instanziierung
  - bei Objekten im heap, wenn **delete** für dieses Objekt ausgeführt wird

Destrukturen werden **hauptsächlich zur Speicherfreigabe** eingesetzt:

- Belegung und Freigabe können so vollständig in der Klasse verborgen werden, der Aufruf erfolgt automatisch.
- Bei geschachtelten Objektstrukturen (Objekte in Objekten, 'Aggregate') werden die Destrukturen der inneren Objekte auch implizit ausgeführt.

# Beispiel

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

```
class B
{
    A* zA;
    A a;
public:
    B() { zA = new A; }
    ~B() { cout << "Destruktor B\n"; delete zA; }
};
```

```
void main()
{
    {
        B b;
    }
    cout << "Ende main\n";
}
```

Ausgabe:

```
Destruktor B
Destruktor A
Destruktor A
Ende main
```

erstes Objekt der Klasse **A**,  
Destruktor wird automatisch ausgeführt

zweites Objekt der Klasse **A**,  
Destruktor wird durch **delete** im  
Destruktor der Klasse B ausgelöst

Objekt der Klasse B in einem inneren Block

Aufruf **~B()** bei Blockende