



University of Applied Sciences

HOCHSCHULE Networked  
EMDEN·LEER Systems



# Programmieren 3

## C++

### Das Semester im Schnelldurchlauf

Prof. Dr. Dirk Kutscher  
Dr. Olaf Bergmann

# Terminplan

14.12.2023: Das Semester im Schnelldurchlauf

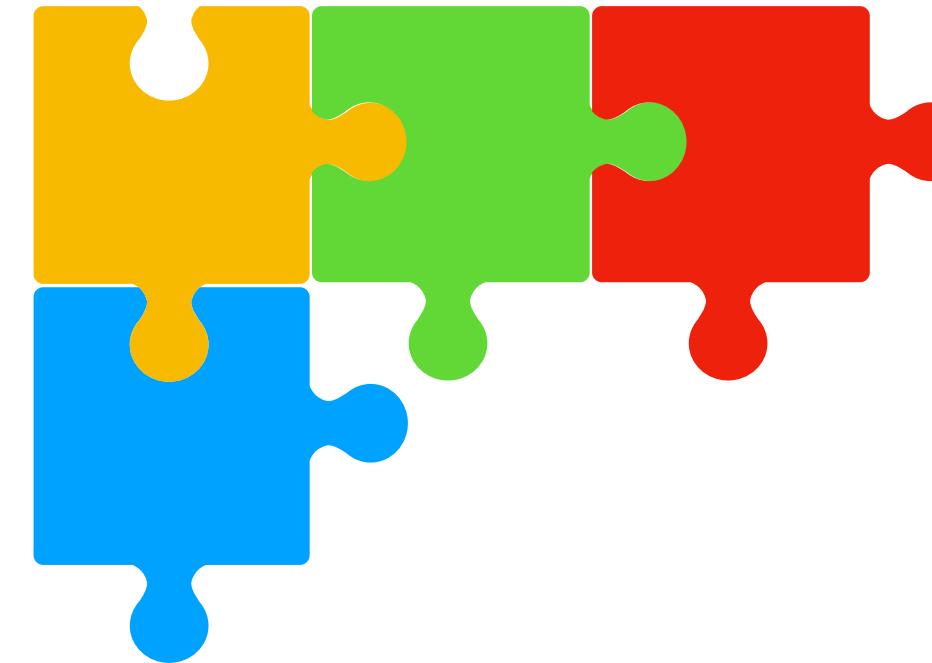
**19.12.2023: Abgabe Übung 4**

21.12.2023: Probeklausur

04.01.2024: Fragen und Antworten

**12.01.2024, 13:00: Klausur (T-Foyer)**

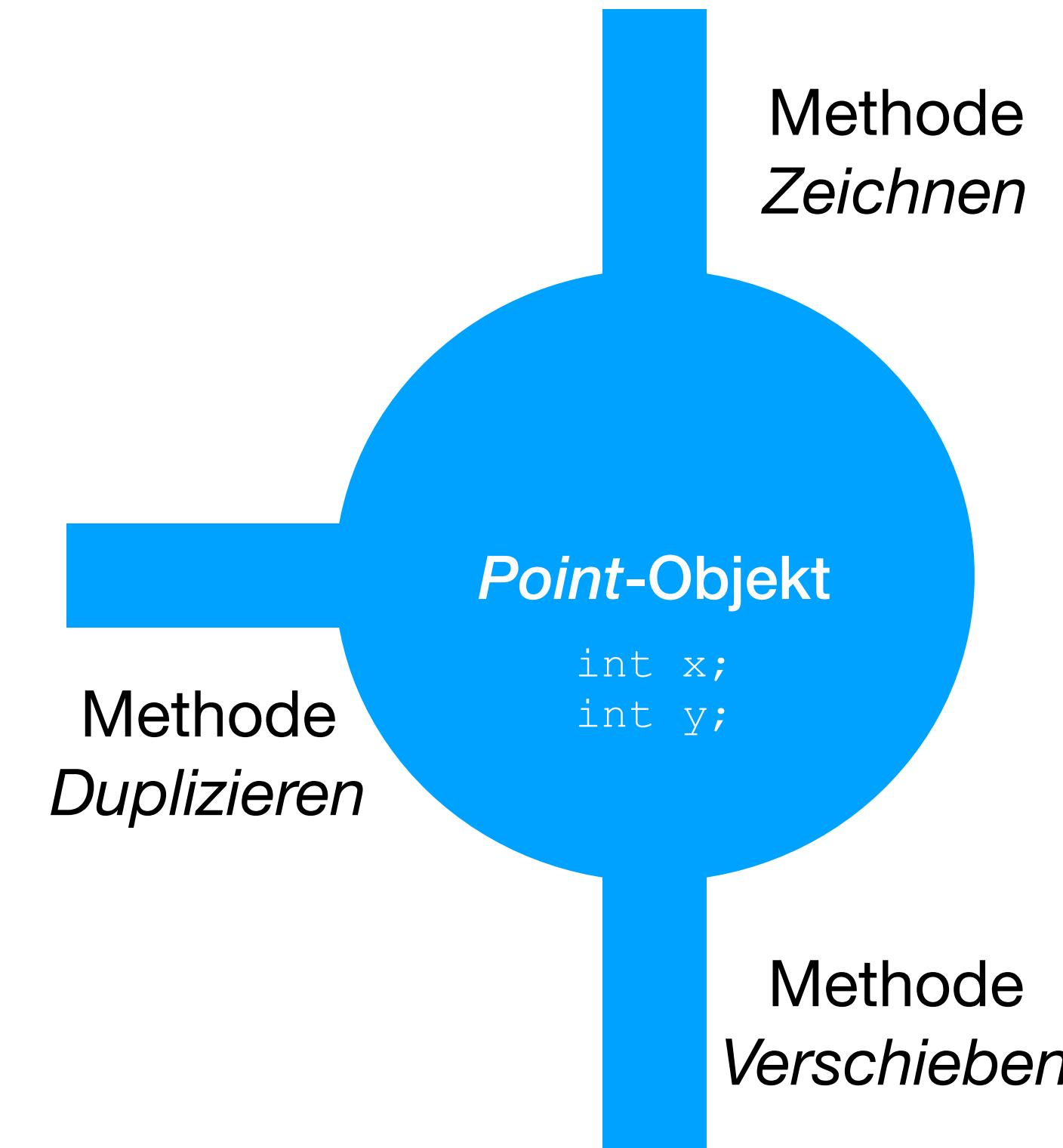
# Large-Scale Software-Entwicklung



- Modularisierung
  - Module wiederverwenden
  - Module erweitern
- Unabhängige Entwicklung
- Flexible Komposition

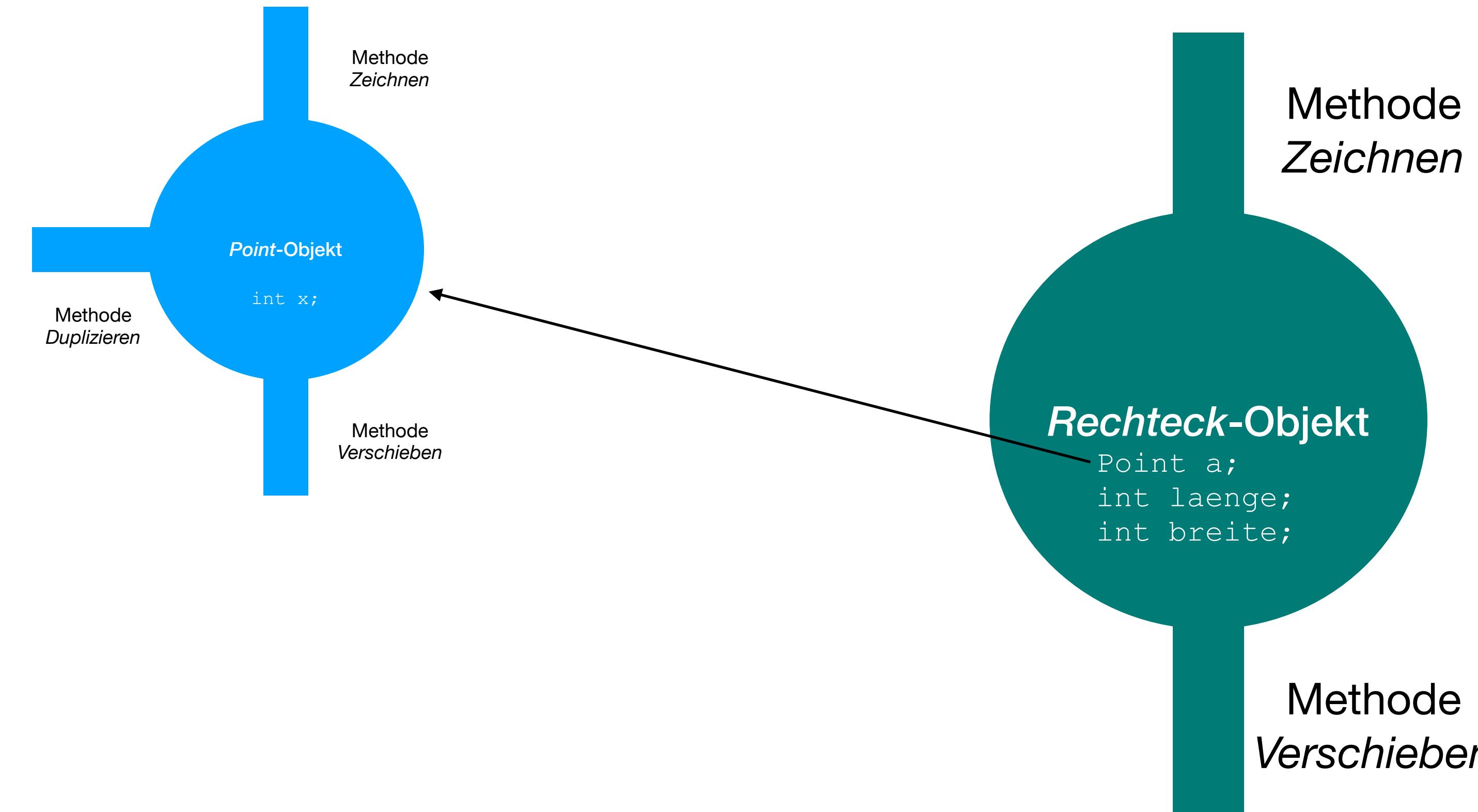
# Objektorientierte Programmierung

- Vollwertige Abstrakte Datentypen
- Die sich wie “eingebaute” Datentypen verhalten
- Interne Datenstruktur:  
**Kapselung**
- Extern aufrufbare Funktionen (Methoden)



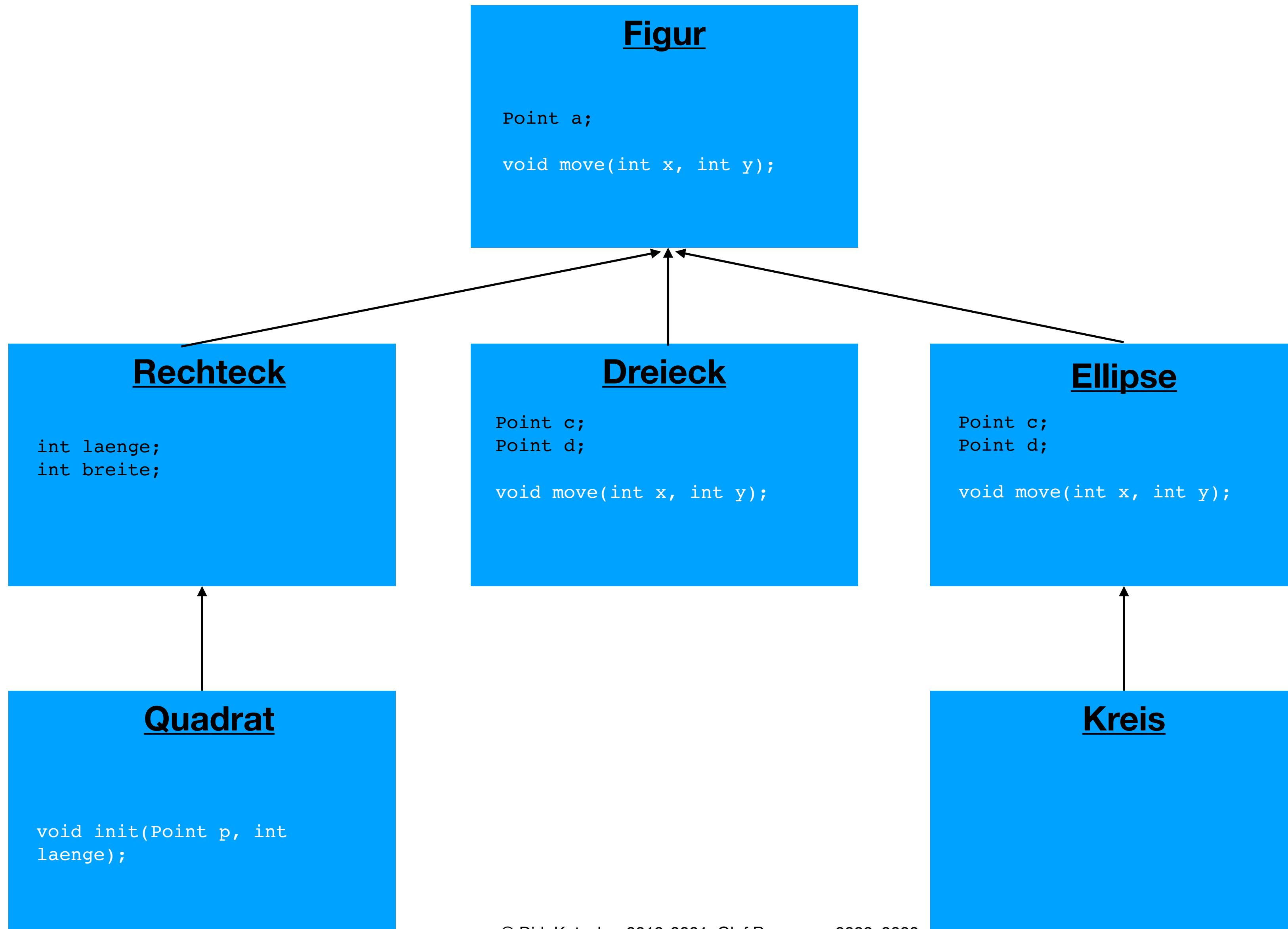
# Objektorientierte Programmierung

## Wiederverwendung durch *Komposition*

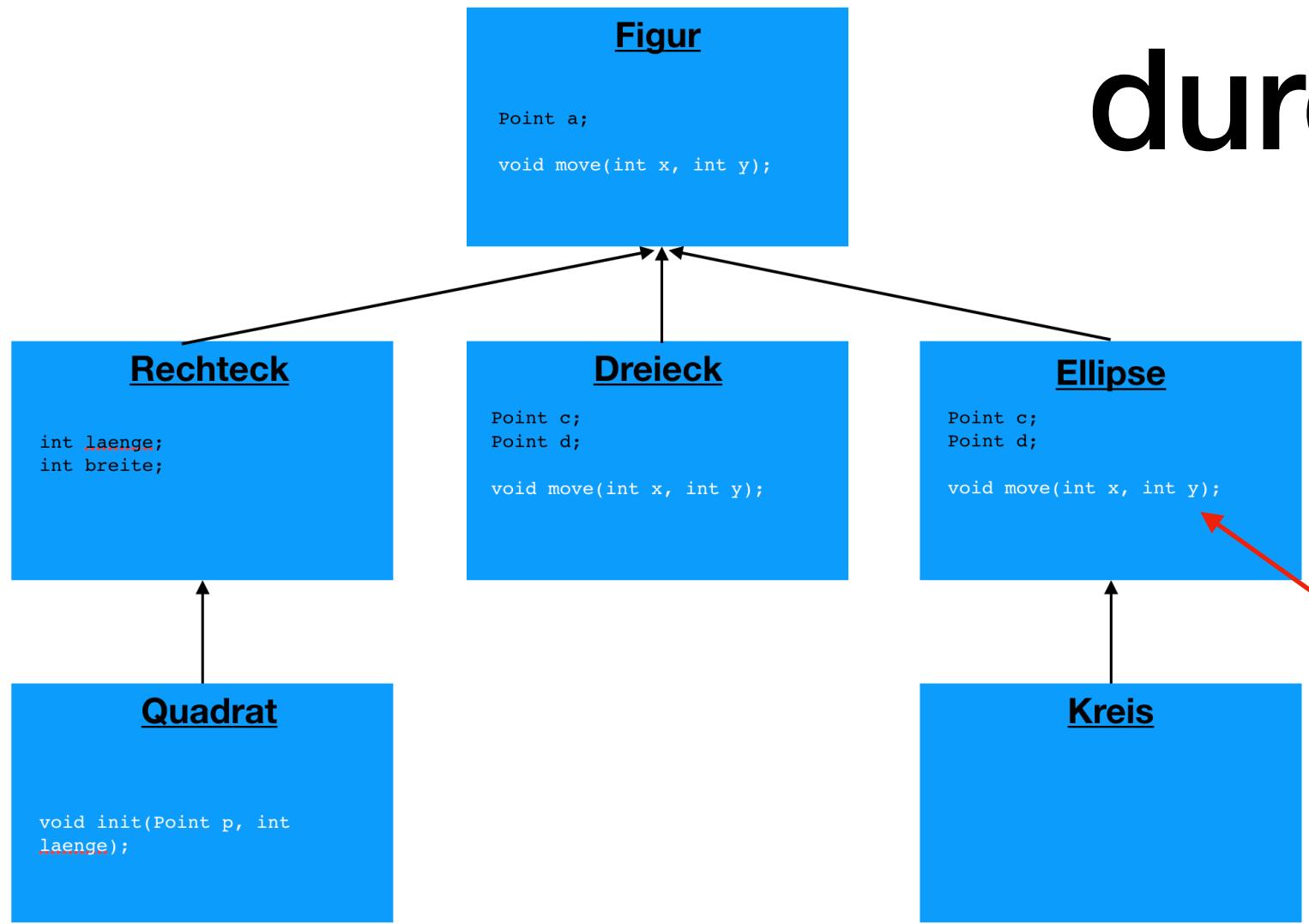


# Objektorientierte Programmierung

## Wiederverwendung durch *Vererbung*



# Wiederverwendung durch Vererbung



```
class Figur {  
    Point a;  
public:  
    void move(int x, int y);  
};  
  
class Rechteck: public Figur {  
    int laenge;  
    int breite;  
};  
  
void moveAndDraw(Figur &f) {  
    f.move(10,10);  
    // draw...  
}  
  
int main() {  
    Figur fi;  
    Rechteck re;  
  
    moveAndDraw(fi);  
    moveAndDraw(re);  
}
```

1. Funktionen von Basis-Klassen wiederverwenden
  - z. B. `Ellipse::move` in `Kreis`
2. Funktionen, die auf Basisklassen arbeiten, können auch auf abgeleitete Klassen angewendet werden → *Referenzparameter*

# Wiederverwendung durch Polymorphie

```
void moveAndDraw( Figure& f, int x, int y ) {  
    f.move( x, y );  
    f.draw( );  
}
```

```
Quadrat q;  
Kreis k;  
  
moveAndDraw( q, 10, 5 );  
moveAndDraw( k, 5, 1 );
```

- Polymorphie: dieselbe Aufrufsschnittstelle für eigentlich unterschiedliche Funktionen
  - Streng genommen, kein Kern-Feature von OOP
  - Wird aber bei C++ und anderen Sprachen so aufgefasst – ist auch ganz praktisch

# Generisches Programmieren in C++

## *Templates*

Ein generischer Listen-Typ

Elementtyp als Template-Parameter

- Templates: Funktionen und Klassen als „Schablonen“ definieren

- Für beliebige Typen nach Bedarf instanziieren

```
#include <algorithm>
#include <iostream>
#include <list>

int main()
{
    // Create a list containing integers
    std::list<int> l = { 7, 5, 16, 8 };

    // Add an integer to the front of the list
    l.push_front(25);
    // Add an integer to the back of the list
    l.push_back(13);

    // Insert an integer before 16 by searching
    auto it = std::find(l.begin(), l.end(), 16);
    if (it != l.end()) {
        l.insert(it, 42);
    }

    // Iterate and print values of the list
    for (int n : l) {
        std::cout << n << '\n';
    }
}
```

# Namensräume

## Library A

```
namespace A {  
    class Circle {  
        ...  
    }  
}
```

## Library B

```
namespace B {  
    class Circle {  
        ...  
    }  
}
```

```
#include "liba.h"  
#include "libb.h"  
  
int main (int argc, char** argv) {  
    A::Circle kreis1;  
    B::Circle kreis2;  
    // ...  
    return 0;  
}
```

- Plus weitere Mechanismen (z. B. um Definitionen in Namensräumen ohne Präfix zu verwenden)

# Fehlersituationen

- Zur Laufzeit können immer Fehler auftreten, auf die man reagieren muss
- Bei kleineren Programmen: eigene Konventionen
- Bei Large-Scale
  - Verwendete Bibliotheken sollten besser standardisierte Schnittstellen verwenden

# Exceptions in C++

```
int dividiere(int dividend, int divisor) {
    if(divisor==0)
        throw(-1);
    else
        return (dividend/divisor);
}

int main(int argc, char** argv) {
    int result;
    try {
        result=dividiere(10/0);
    }
    catch (int) {cout << "oops";}
    return 0;
}
```

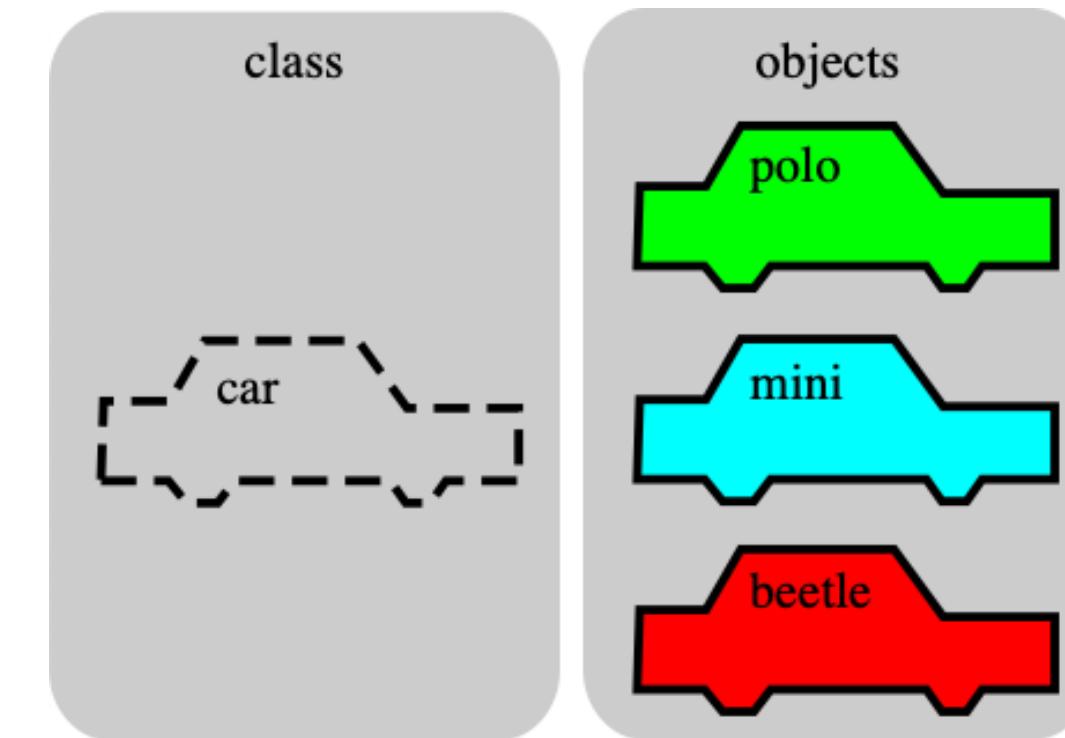
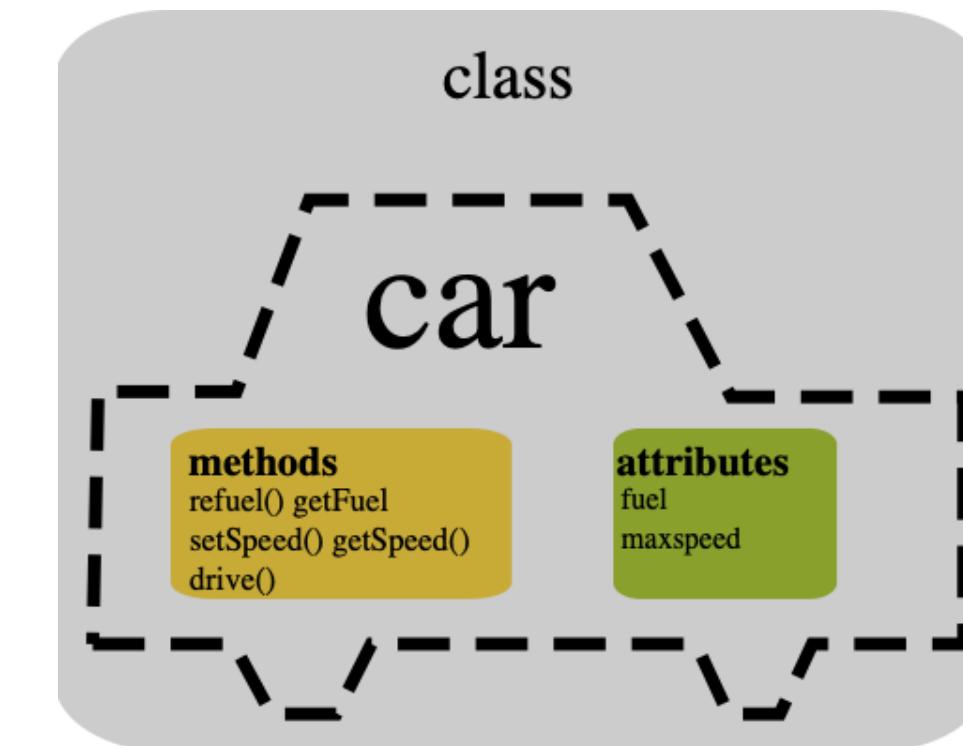
- Strukturierte Fehlerbehandlung für Laufzeitfehler
- Auch über mehrere Aufrufebenen hinweg
- Module, Libraries usw. können eigene Exception-Typen definieren (und dokumentieren)

# C++ für Embedded Systems

- C++ unterstützt alle C-Mechanismen, z. B.,
  - Bit-Manipulationen
  - Werte per Referenz übergeben (Zeiger in C)
  - Eigene Speicherverwaltung (wenn Standard-Bibliothek nicht benutzt wird)
  - Speicher-Management-Operatoren (`new`, `delete` usw.) können überladen werden (*Overloading*)

# Objektorientierte Analyse und Design

- Die Welt als Klassen und Objekte modellieren
- Über Beziehungen zwischen Klassen und Objekten nachdenken
- Was sind Eigenschaften / Member?
- Was sind Vererbungsbeziehungen?
- Passt nicht immer, aber man kommt damit recht weit
- Braucht etwas Erfahrung



# Standardbibliothek

- Tools & Mechanismen zur Interaktion mit Betriebssystem
  - Ein-/Ausgabe
  - Strings
  - Container (Listen, Vektoren usw.)
  - ... und vieles mehr
- Wie bei C (`stdio.h` usw.) – nur viel umfangreicher

## C++ Standard Library

- Input/output
- Strings

## Standard Template Library

- `algorithm`
- `functional`
- Sequence containers
- Associative containers
- Unordered associative containers

## C standard library

- Data types
- Character classification
- Strings
- Mathematics
- File input/output
- Date/time
- Localization
- Memory allocation
- Process control
- Signals
- Alternative tokens

## Miscellaneous headers:

- `<assert.h>`
- `<errno.h>`
- `<setjmp.h>`
- `<stdarg.h>`

V • T • E

# Container in C++ (Stdlib)

## Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

<b>array</b> (C++11)	static contiguous array (class template)
<b>vector</b>	dynamic contiguous array (class template)
<b>deque</b>	double-ended queue (class template)
<b>forward_list</b> (C++11)	singly-linked list (class template)
<b>list</b>	doubly-linked list (class template)

## Associative containers

Associative containers implement sorted data structures that can be quickly searched ( $O(\log n)$  complexity).

<b>set</b>	collection of unique keys, sorted by keys (class template)
<b>map</b>	collection of key-value pairs, sorted by keys, keys are unique (class template)
<b>multiset</b>	collection of keys, sorted by keys (class template)
<b>multimap</b>	collection of key-value pairs, sorted by keys (class template)

# Hello World

Deklarationen aus der Datei `iostream` einbinden

Compiler kennt den Suchpfad für System-Header-Dateien

```
// my first program in C++  
#include <iostream>  
  
int main () {  
    std::cout << "Hello World!";  
}
```

Objekt für die Default-Ausgabe (z. B. Konsole)

Ausgabe-Operator

# Eingabe

Wir verwenden alle Deklarationen aus dem Namespace std ohne Präfix

```
// i/o example

#include <iostream>
using namespace std;

int main ()
{
    int i;
    cout << "Please enter an integer value: ";
    cin >> i;
    cout << "The value you entered is " << i;
    cout << " and its double is " << i*2 << ".\n";
    return 0;
}
```

Objekt für die Default-Eingabe (z.B. Konsole)

Eingabe-Operator

# Dateien

Klasse für schreibbare Dateien

Neues Objekt – lokale Variable

```
#include <fstream>
int main()
{
    std::ofstream file("file.txt");
    file << "Hello, world!" << std::endl;
}
```

# Arrays in C

```
// Program to take 5 values from the user and store them in an array  
// Print the elements stored in the array  
#include <stdio.h>  
int main() {  
    int values[5];  
    const size_t n_values = sizeof(values)/sizeof(values[0]);  
    printf("Enter 5 integers: ");  
    // taking input and storing it in an array  
    for(int i = 0; i < n_values; ++i) {  
        scanf("%d", &values[i]);  
    }  
    printf("Displaying integers: ");  
    // printing elements of an array  
    for(int i = 0; i < n_values; ++i) {  
        printf("%d\n", values[i]);  
    }  
    return 0;  
}
```

Anzahl der Elemente  
in values berechnen

Pre-increment operator:  
i wird erst erhöht, und der  
Ausdruck liefert den **neuen Wert**  
von i

# Vectors in C++

```
#include <iostream>
#include <vector>

Container-Typ vector
Element-Typ int           Initiale Größe: 5 Elemente

using namespace std;

int main() {
    vector<int> values(5);

    cout << "Enter 5 integers: ";
    // taking input and storing it in a vector
    for(int i = 0; i < values.size(); ++i) {
        cin >> values[i];
    }
    cout << "Displaying integers: ";
    // printing elements of an array
    for(int i = 0; i < values.size(); ++i) {
        cout << values[i] << endl;
    }
    return 0;
}
```

# Arrays kopieren: korrekte Fasung

```
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>
int main() {
    int values[5];
    const size_t n_values = sizeof(values)/sizeof(values[0]);
    int copyOfValues[n_values];
    printf("Enter 5 integers: ");
    // taking input and storing it in an array
    for(int i = 0; i < n_values; ++i) {
        scanf("%d", &values[i]);
    }
    for(int i = 0; i < n_values; ++i) {
        copyOfValues[i]=values[i];
    }
    printf("Displaying integers: ");
    // printing elements of an array
    for(int i = 0; i < n_values; ++i) {
        printf("%d\n", copyOfValues[i]);
    }
    return 0;
}
```

Arrays in C müssen  
elementweise kopiert werden

# Einen Vector kopieren in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> values(5);
    vector<int> copyOfValues;

    cout << "Enter 5 integers: ";
    // taking input and storing it in a vector
    for(int i = 0; i < values.size(); ++i) {
        cin >> values[i];
    }

    copyOfValues=values; // Elementweises Kopieren

    cout << "Displaying integers: ";
    // printing elements of an array
    for(int i = 0; i < copyOfValues.size(); ++i) {
        cout << copyOfValues[i] << endl;
    }
    return 0;
}
```

Zuweisungsoperator erledigt das  
elementweise Kopieren

# Zeichenketten in C

```
#include <stdio.h>

int main () {
    const char moin[]="Moin!";
    const char *greeting=moin;
    printf("Greeting message: %s\n", greeting);
    return 0;
}
```

Index	0	1	2	3	4	5
Variable	M	o	i	n	!	\0
Adresse	0x23450	0x23451	0x23452	0x23453	0x23454	0x23455

# Strings in C++

- In C++: Strings werden auch in Arrays gespeichert
- Aber Speichermanagement passiert automatisch – und dynamisch
- Zuweisungsoperator (=) bewirkt Kopieren der Zeichenkette

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

using namespace std;

int main () {
    string str1 = "Hello";
    string str2 = "World";
    string str3;

    // copy str1 into str3
    str3 = str1;
    cout << "str3: " << str3 << endl;

    // concatenates str1 and str2
    str3 = str1 + str2;
    cout << "str1 + str2: " << str3 << endl;

    return 0;
}
```

# Strings in C++

## Element access

`at`  
`operator[]`  
`front` (C++11)  
`back` (C++11)  
`data`  
`c_str`  
`operator basic_string_view` (C++17)

## Iterators

`begin`  
`cbegin` (C++11)  
`end`  
`cend` (C++11)  
`rbegin`  
`crbegin` (C++11)  
`rend`  
`crend` (C++11)

## Capacity

`empty`  
`size`  
`length`  
`max_size`  
`reserve`  
`capacity`  
`shrink_to_fit` (C++11)

## Operations

`clear`  
`insert`  
`erase`  
`push_back`  
`pop_back` (C++11)  
`append`  
`operator+=`  
`compare`  
`starts_with` (C++20)  
`ends_with` (C++20)  
`replace`  
`substr`  
`copy`  
`resize`  
`swap`

## Search

`find`  
`rfind`  
`find_first_of`  
`find_first_not_of`  
`find_last_of`  
`find_last_not_of`

## Non-member functions

`operator+`  
`operator==`  
`operator!=`  
`operator<`  
`operator>`  
`operator<=`  
`operator>=`  
`std::swap` (`std::basic_string`)  
`erase` (`std::basic_string`)  
`erase_if` (`std::basic_string`) (C++20)

## Input/output

`operator<<`  
`operator>>`  
`getline`

# Pointer in C

Wozu benötigt man Pointer in C?

1. Dynamische Speicherverwaltung
2. Effiziente Parameterübergabe (“Call by Reference”)
3. Übergabe von Ergebnisparametern (“Call by Reference”)

Probleme dabei?

# Referenzen in C++

1. Dynamische Speicherverwaltung
2. Effiziente Parameterübergabe (“Call by Reference”)
3. Übergabe von Ergebnisparametern (“Call by Reference”)

# Default (C/C++): Call-by-Value

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

using namespace std;
// call by value

void printString(string printString) {
    cout << printString << endl;
}

int main() {
    string moin("Moin");
    printString(moin);
}
```

**Wertparameter,**  
string-Objekt  
wird **kopiert**



# Referenzen in C++

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

using namespace std;

// call by reference

void printString(const string &printString) {
    cout << printString << endl;
}

int main() {
    string moin("Moin");
    printString(moin);
}
```

**Referenz** auf  
string-Objekt moin  
wird übergeben



→ Effiziente Parameter-Übergabe

# Referenzen in C++

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

using namespace std;

// call by reference

void readString(string &eingabe) {
    cin >> eingabe;
}

int main() {
    string mystring;
    readString(mystring);
}
```

**Referenz** auf  
string-Objekt mystring  
wird übergeben  
(hier nicht const)



→ Übergabe von Ergebnis-Parametern

# Constness beachten!

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

using namespace std;

// call by reference

void readString(string &eingabe) {
    cin >> eingabe;
}

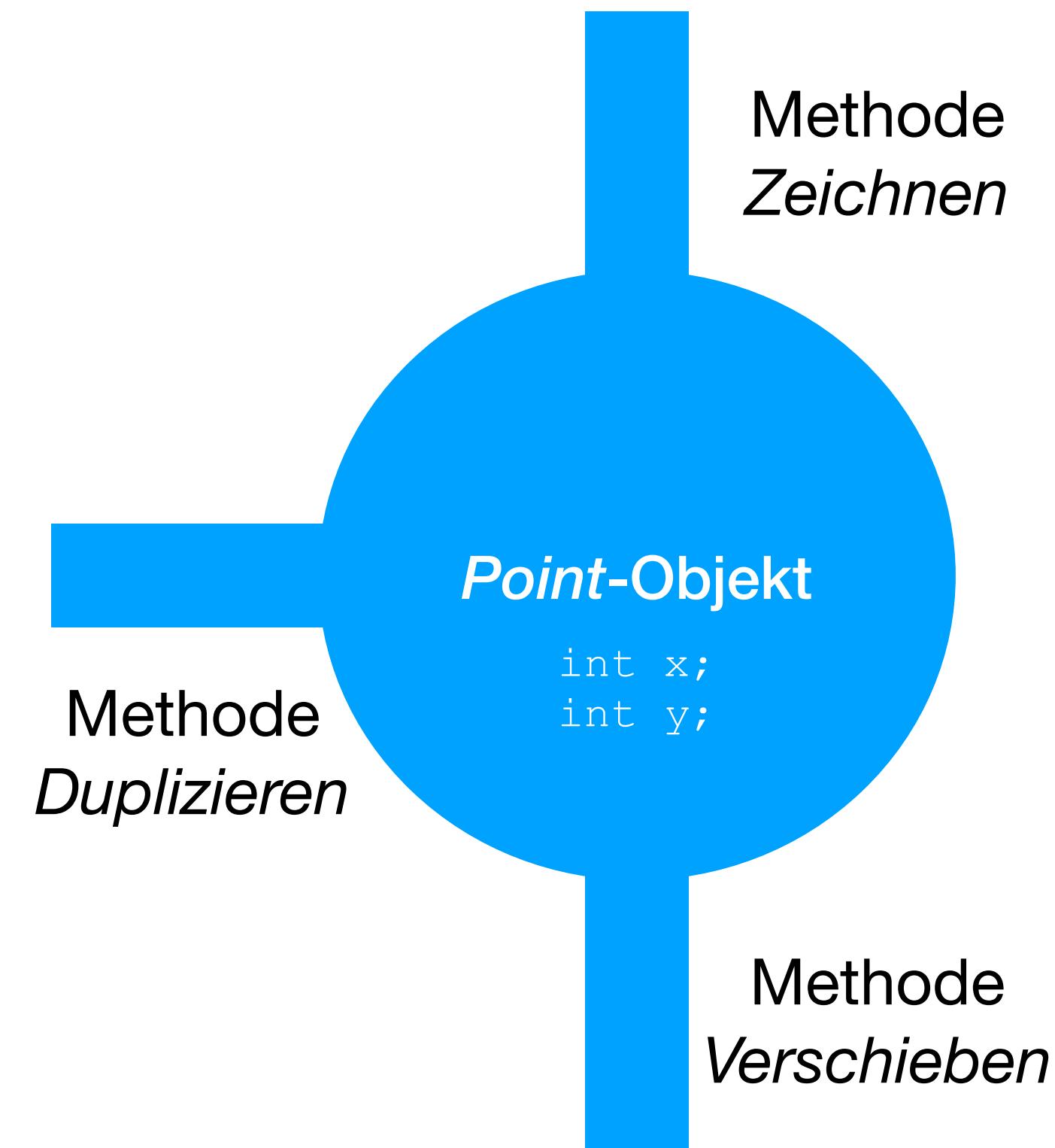
int main() {
    const string mystring;
    readString(mystring);
}
```

**Fehler:** mystring  
als const deklariert



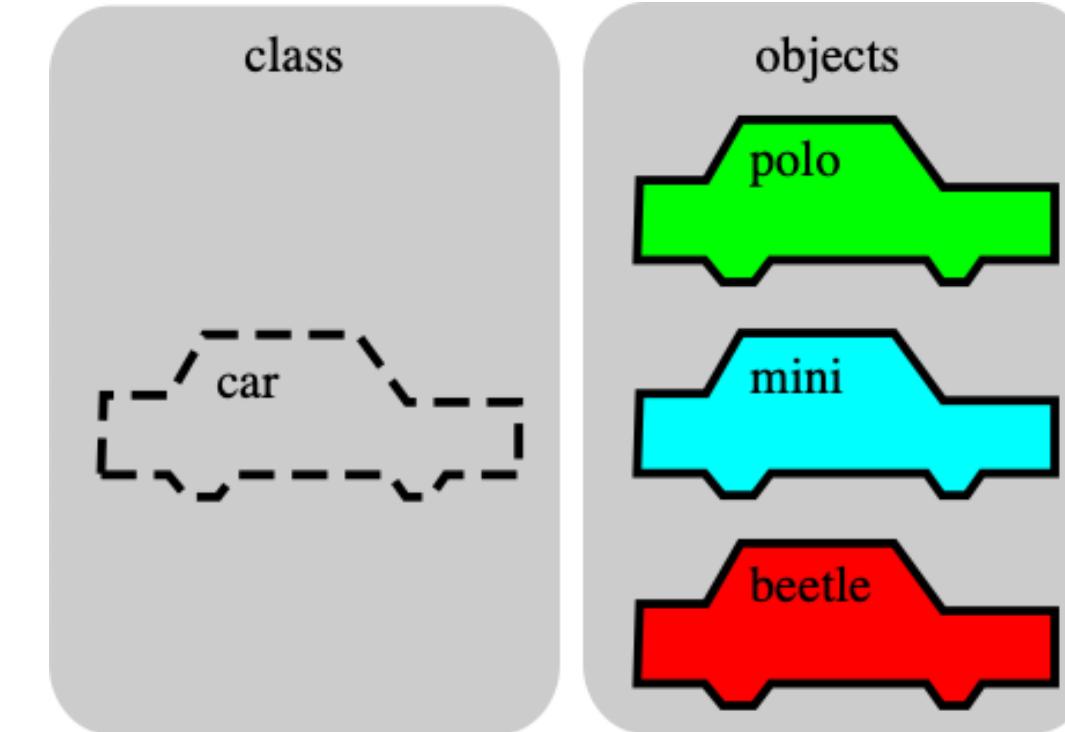
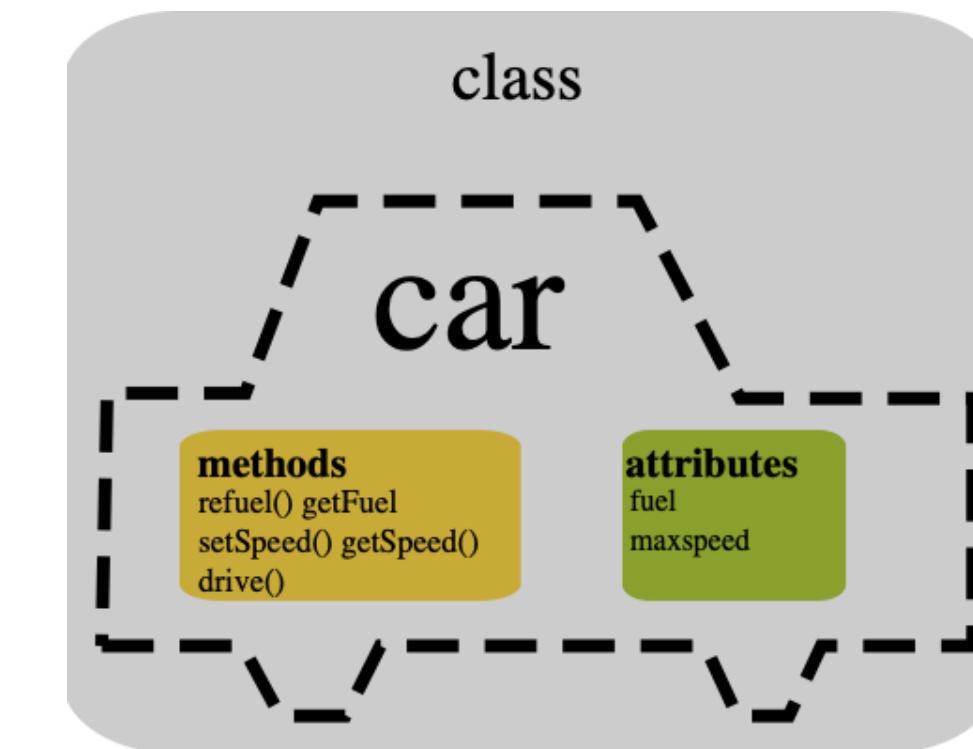
# Klassen in C++

- Vollwertige Abstrakte Datentypen
- Die sich wie “eingebaute” Datentypen verhalten
- Interne Datenstruktur:  
***Kapselung***
- Extern aufrufbare Funktionen (Methoden)



# Klassen und Objekte

- Klassen: Definitionen von Datentypen
  - Eigenschaften / Attribute
  - Methoden
- Objekte: Instanzen von Klassen
  - Objekttyp==Klasse
  - Objekte können Attribute ihrer Klassen mit unterschiedlichen Werten versehen



# Beispiel

Deklaration der Member-Funktion

Definition der Member-Funktion

Verwendung der Member-Funktion

```
#include <iostream>

class SimpleString {

    char buffer[128];
    int strSize;

public:
    int size();
};

int SimpleString::size() {
    return strSize;
}

int main() {
    SimpleString greeting;

    std::cout << greeting.size();

    return 0;
}
```

# Zugriffsrechte

```
class Oberklasse {  
    private:                                // Voreinstellung  
        int oberklassePriv;  
        void privateFunktionOberklasse();  
    protected:  
        int oberklasseProt;  
    public:  
        int oberklassePubl;  
        void publicFunktionOberklasse();  
};
```

Zugriffsrecht in der Basisklasse	Zugriffsrecht in einer abgeleiteten Klasse
private protected public	kein Zugriff protected public

// Oberklasse wird mit der Zugriffskennung public vererbt

```
class AbgeleiteteKlasse : public Oberklasse {  
    int abgeleiteteKlassePriv;  
    public:  
        int abgeleiteteKlassePubl;  
        void publicFunktionAbgeleiteteKlasse() {  
            oberklassePriv = 1;  
            // in einer abgeleiteten Klasse zugreifbar  
            oberklasseProt = 2;  
            // generell zugreifbar  
            oberklassePubl = 3;  
        }  
};
```

```
int main() {  
    AbgeleiteteKlasse objekt;  
    int m = objekt.oberklassePubl;  
    m = objekt.oberklasseProt;
```

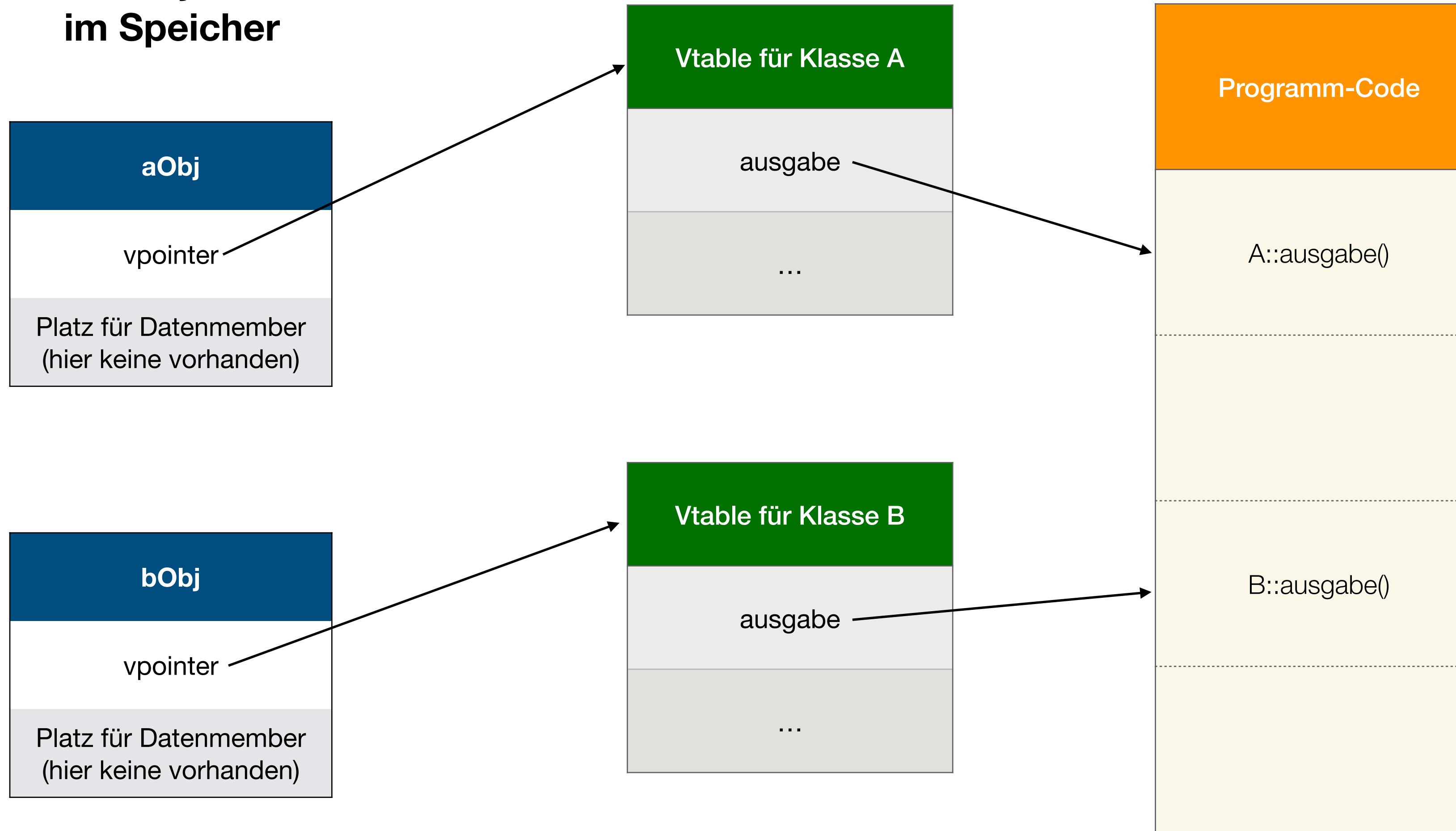
Man kann auch private oder protected erben  
— meistens nicht sinnvoll

// Fehler: nicht zugreifbar

// Fehler: nicht zugreifbar

# Polymorphie mit virtuellen Funktionen

## Repräsentation von Objekten im Speicher



Beim Aufruf von `ausgabe()` auf einem Objekt vom Typ A oder B wird zur Laufzeit ermittelt, welche Funktion wirklich ausgeführt werden muss.

# Polymorphie mit Virtuellen Methoden

```
class A {  
public:  
    virtual void ausgabe(void) {cout << "A" << endl;}  
};  
  
class B: public A {  
public:  
    void ausgabe(void) override {cout << "B" << endl;}  
};  
  
void ausgeben(A &obj) {  
    obj.ausgabe();  
}  
  
int main() {  
    A aObj;  
    B bObj;  
  
    ausgeben(aObj);  
    ausgeben(bObj);  
}
```

bewirkt Eintrag in Tabelle  
virtueller Methoden (Vtable)

Optional: Compiler prüft bei  
Übersetzung, ob virtuelle Methode  
mit dieser Signatur definiert ist.

# Virtuelle Funktionen

```
class A {  
public:  
    virtual void ausgabe(void)  
        {cout << "A" << endl;}  
};  
  
class B: public A {  
public:  
    void ausgabe(void) override  
        {cout << "B" << endl;}  
};  
  
void ausgeben(A &obj) {  
    obj.ausgabe();  
}  
  
int main() {  
    A aObj;  
    B bObj;  
  
    ausgeben(aObj);  
    ausgeben(bObj);  
}
```

- **Überschreiben von Funktionen gleicher Signatur bei Vererbung**
- **Beim Aufruf wird der dynamische Typ des Objekts ermittelt (z.B. A oder B)**
  - Richtige Funktion wird über vpointer und Vtable gefunden
- **Motivation: Basisklasse definiert Aufrufschnittstelle (z.B. ausgabe)**
  - Abgeleitete Klassen können neues Verhalten definieren
  - Generische Funktionen (wie ausgeben) können dann auf allen Objekten der Basisklasse oder abgeleiteten Klassen operieren
  - Trotzdem kann für jedes Objekt die spezialisierte Funktion aufgerufen werden

```

class Ort {
    // ...
};

class GraphObj { // Version 1
public:
    GraphObj(Ort einOrt) // allgemeiner Konstruktor
        : referenzkoordinaten{einOrt} {}

    // Bezugspunkt ermitteln
    Ort bezugspunkt(void) const {
        return referenzkoordinaten;
    }

    // alten Bezugspunkt ermitteln
    // und gleichzeitig neuen wählen
    Ort bezugspunkt(Ort nO) {
        Ort temp {referenzkoordinaten};
        referenzkoordinaten = nO;
        return temp;
    }

    // Koordinatenabfrage
    int getX(void) const {
        return referenzkoordinaten.getX();
    }
    int getY(void) const {
        return referenzkoordinaten.getY();
    }

    // Standardimplementation
    virtual double flaeche(void) = 0;

private:
    Ort referenzkoordinaten;
};

```

# Abstrakte Klassen

- **Rein virtuelle Funktionen (Pure Virtual Functions)**
- Wird mit `virtual ... = 0` deklariert, aber nicht definiert
- Abgeleitete Klassen *müssen* dann diese Funktion definieren
- Objekte vom Typ der Basisklasse (hier `GraphObj`) können nicht instanziert werden, daher **Abstrakte Klassen**

```

class Strecke : public GraphObj {
public:
    Strecke(Ort ort1, Ort ort2)
        : GraphObj{ort1},
          endpunkt{ort2} {}

    auto laenge(void) const {
        return entfernung(bezugspunkt(), endpunkt);
    }

    virtual double flaeche(void) const override {
        return 0.0;
    }
}

```

# Konstruktoren

Deklaration des Konstruktors

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

class SimpleString {

    char buffer[128];
    int strSize;

    void init(const char* initString);
public:
    SimpleString(const char* initString);
    size_t size(void) const;
};

size_t SimpleString::size(void) const {
    return strSize;
}

void SimpleString::init(const char* initString) {
    strncpy(buffer, initString, sizeof(buffer));
    strSize = strlen(buffer);
}

SimpleString::SimpleString(const char* initString) {
    init(initString);
}

int main() {
    SimpleString greeting("Moin");

    std::cout << greeting.size();

    return 0;
}
```

Definition des Konstruktors

Verwendung des Konstruktors

# Konstruktoren in C++

```
int main() {
    SimpleString greeting1("Moin");
    SimpleString greeting(greeting1);
    SimpleString name(" C++");

    greeting.add(name);

    cout << greeting.str() << endl;

    return 0;
}
```

```
class SimpleString {

public:
    SimpleString(void);
    SimpleString(const char *initString);
    SimpleString(const SimpleString &initString);

    const char *str(void) const;
    void add(const SimpleString &addedString);

private:
    char* buffer;
    int bufferSize;

    size_t increaseBuffer(size_t newSize);
};
```

- **Konstruktoren**

- Besondere Member-Funktionen
- Zum Initialisieren eines Objekts
- Heissen so wie die Klasse
- Kann man nicht direkt aufrufen (nur beim Initialisieren)
- Verschiedene Varianten

Drei grundlegende Arten von Konstruktoren

- **Default-Konstruktor:** ohne Parameter
- **Copy-Konstruktor:** Parameter vom Typ der eigenen Klasse
- **Converting-Konstruktor:** sonstige Parameter

# Destruktor

```
int main() {
    SimpleString greeting1("Moin");
    SimpleString greeting(greeting1);
    SimpleString name(" C++");

    greeting.add(name);

    cout << greeting.str() << endl;

    return 0;
}
```

```
class SimpleString {

public:
    SimpleString();
    SimpleString(const char *initString);
    SimpleString(const SimpleString &initString);
    ~SimpleString(void);

    const char *str(void);
    void add(const SimpleString &addedString);

private:
    char *buffer;
    size_t bufferSize;

    void init(const char *initString);
    size_t increaseBuffer(size_t newSize);
};
```

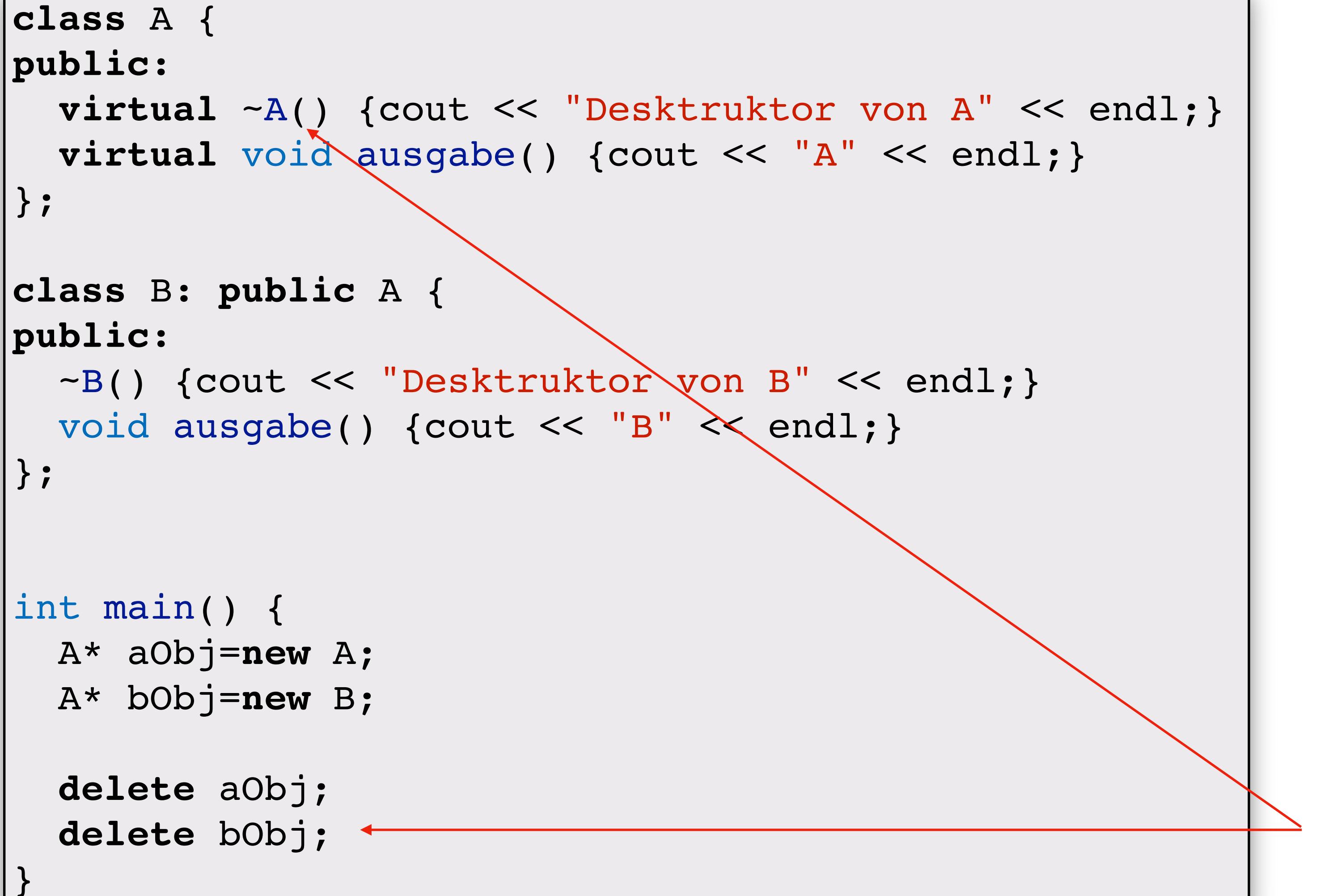
- **Lebensdauer von lokalen Variablen**

- Lokale Variablen werden in einem {}-Block instanziert
- Und beim Verlassen des Blocks automatisch gelöscht
  - Speicherplatz (für die Member-Variablen) freigeben
- Bei Objekten von Klassen möchte man noch mehr Kontrolle haben
  - Dynamisch allozierten Speicher freigeben
  - Sonstige “Aufräumaktionen”
- ➤ **Destruktoren**

```
SimpleString::~SimpleString(void) {
    delete[] buffer;
}
```

# Virtueller Destruktor

```
class A {  
public:  
    virtual ~A() {cout << "Destruktör von A" << endl;}  
    virtual void ausgabe() {cout << "A" << endl;}  
};  
  
class B: public A {  
public:  
    ~B() {cout << "Destruktör von B" << endl;}  
    void ausgabe() {cout << "B" << endl;}  
};  
  
int main() {  
    A* aObj=new A;  
    A* bObj=new B;  
  
    delete aObj;  
    delete bObj;  
}
```



# Virtuelle Destruktoren

- Immer verwenden, wenn von einer Basisklasse abgeleitet werden soll
- Vor allem, wenn Basisklassen-Zeiger oder -Referenzen auf dynamisch erzeugte Objekte verwendet werden



# Templates

- Generelle Idee: Code wiederverwenden
  - Typsicher, ohne auf Pointer-Casting zurückgreifen zu müssen (wie in C)
- Templates: Funktion/Klasse einmal definieren
  - Und dann für verschiedene Typen instanziieren

## Quicksort in C

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*comp)(const void *, const void *));
```

- base: zu sortierendes Array mit nmemb Elementen
- size: Größe eines Elements
- comp: Vergleichsfunktion
- Benutzung

```
int cmp(const void *p, const void *q) {
    return strcmp(*((char **)p), *((char **)q));
}

int main(int argc, char **argv) {
    qsort(++argv, --argc, sizeof(char *), cmp);
    while(argc-- > 0) {
        printf("%s\n", *argv++);
    }
    return 0;
}
```

# Template-Klasse

```
template <class T>
class vector {
public:
    ...
    int size() const;
private:
    int sz;
    T *p; }
```

Template-Parameter  
als Platzhalter für Datentyp

## Benutzung:

```
vector<int> vi;
vector<char> vc;
```

```
template<class T>
int vector<T>::size() const { return sz; }
```

# Template-Funktion

## Beispiel: quicksort als Template-Funktion in C++

```
template <typename T>
void qsort(const T* base, int nmemb, int (comp)(const T& e1, const T& e2)) {
    // qsort-Implementierung
    // würde irgendwann comp(element 1, element 2) aufrufen
}

int intCmp(const int& e1, const int& e2) {
    return e1<e2;
}

int main() {
    int intArray[10];
    qsort<int>(intArray, 10, intCmp);
}
```

Template-Parameter

Instanziierung der Template-Funktion

# Algorithmen

```
#include <numeric>
#include <vector>

double average(const std::vector<double> &v) {
    return std::accumulate(v.cbegin(), v.cend(), 0.0) / v.size();
}
```

# Lambda-Ausdrücke

```
#include <iostream>

int main(int argc, char **argv) {
    unsigned int (*f)(unsigned int);

    f = [](unsigned int n) { return n * n; };

    if (argc > 1)
        std::cout << f(std::stoi(argv[1])) << std::endl;
}
```

f ist Funktion  
f: unsigned int → unsigned int

[] leitet Lambda-Ausdruck („Lambda-Funktion“) ein

# Lambda-Ausdrücke

```
#include <iostream>

int main(int argc, char **argv) {
    auto f = [] (unsigned int n) { return n * n; };
    if (argc > 1)
        std::cout << f(std::stoi(argv[1])) << std::endl;
}
```

A red arrow points from the word "auto" in the first line of the code to the explanatory text in the callout box.

**Compiler leitet Deklaration automatisch ab**

# accumulate mit Lambda

```
std::vector<double> v{ -17, 3.2, 98.999, 12.7 };  
  
double avg = std::accumulate(v.cbegin(), v.cend(), 0.0,  
[ ] (double init, double val) {  
    return init + val / v.size();  
} );  
}
```

Problem: v ist weder Parameter  
der Funktion noch lokal bekannt

# accumulate mit Lambda

```
std::vector<double> v{ -17, 3.2, 98.999, 12.7 };  
  
double avg = std::accumulate(v.cbegin(), v.cend(), 0.0,  
[&v] (double init, double val) {  
    return init + val / v.size();  
} );  
}
```

**Abhilfe:** Sichtbare Variable `v` einfangen  
& erzeugt Referenzparameter

# Funktoren

```
class Data {  
    int wert = 0;  
public:  
    int operator() (void) { return wert++; }  
};  
  
std::vector<int> v;  
  
std::generate_n(insertion(v, v.end()), 100, Data());
```

**Objekt kann Zustand haben**

**Funktor muss mindestens operator() implementieren**

**Erzeugen eines anonymen Funktionsobjekts des Klasse Data  
(Wertparameter für generate\_n)**

# std::map

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

using namespace std;

int main() {
    map<string, int> preis;

    preis["Toastbrot"] = 2;
    preis["Schokolade"] = 3;
    preis["Kaffee"] = 10;

    cout << preis["Kaffee"] << endl;

    pair<string, int> ersterPreis = *preis.begin();

    cout << ersterPreis.first << ":" << ersterPreis.second << endl;

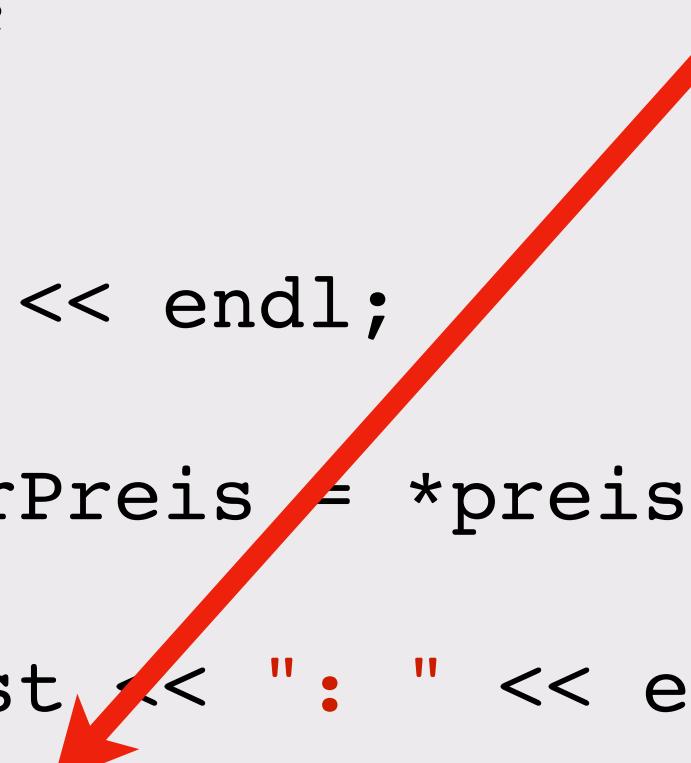
    for(auto p = preis.begin(); p != preis.end(); p++) { // gibt alle Paare aus
        cout << p->first << ":" << p->second << endl;
    }
}
```

Mal angucken...

<https://en.cppreference.com/w/cpp/container/map>  
<https://en.cppreference.com/w/cpp/utility/pair>

besser:

```
for (auto const &p: preis) {
    cout << p.first << ":" "
        << p.second << endl;
}
```



# C++ und Container

Container mit Werten

Iteratoren:

Verweise auf “Positionen”  
im Container

Algorithmen:  
Generische Funktionen,  
die über Iteratoren  
auf Container zugreifen



begin

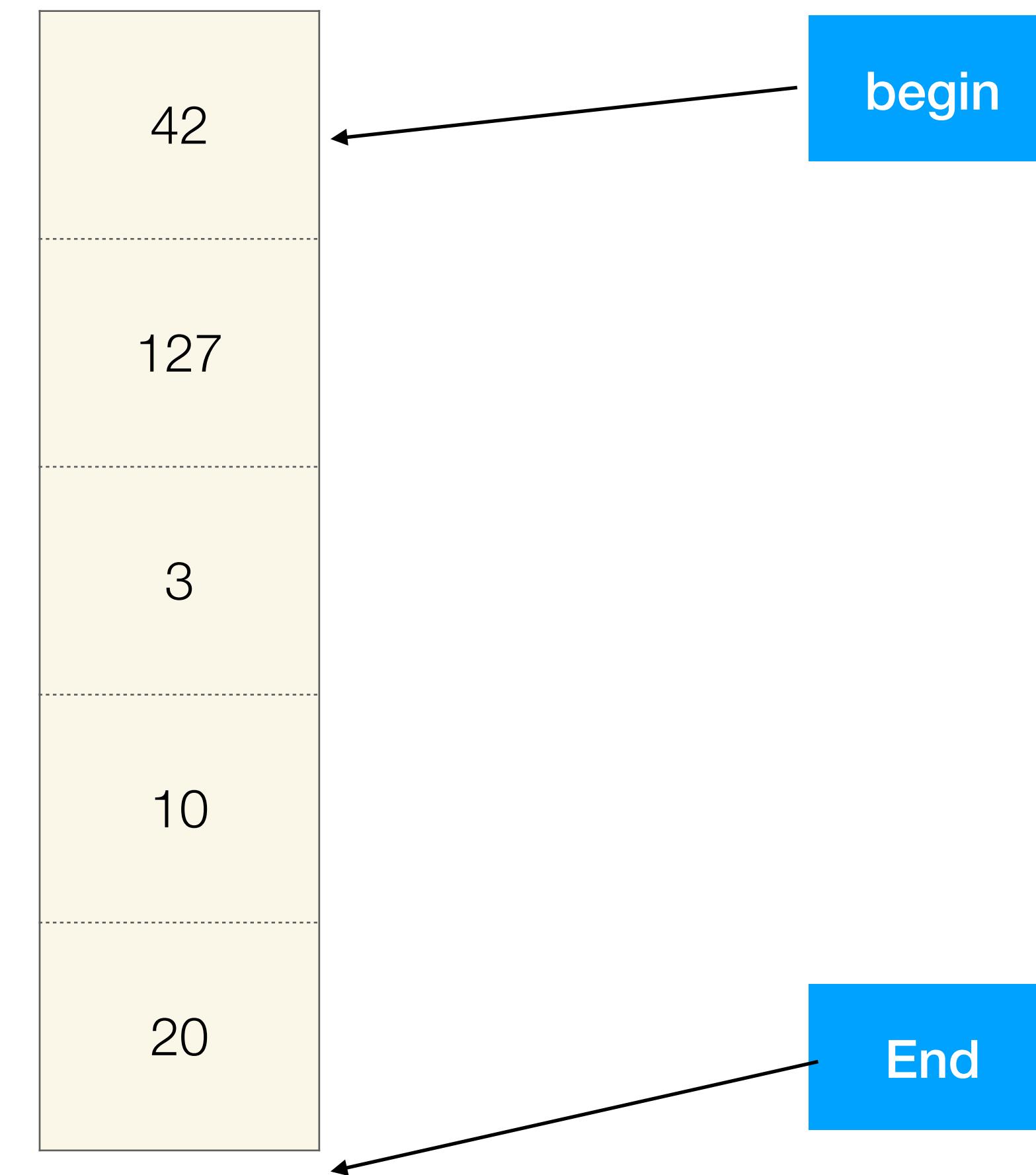
End

`find()`  
`count()`  
`sort()`  
`copy()`  
...

Alle als  
Templates definiert!

# Iteratoren

- Werden ähnlich wie Zeiger verwendet
  - `it++`: Iterator auf nächstes Element zeigen lassen
  - `it--`: Iterator auf vorheriges Element zeigen lassen
  - `*it`: Dereferenzieren (Element zurückgeben)



# Iteratoren

```
template<typename T>
class Iteratortyp {
public:
    // Konuktoren, Destruktor weggelassen
    bool operator==(const Iteratortyp<T>&) const;
    bool operator!=(const Iteratortyp<T>&) const;
    Iteratortyp<T>& operator++();                                // Präfix
    Iteratortyp<T> operator++(int);                            // Postfix
    T& operator*() const;
    T* operator->() const;
private:
    // Verbindung zum Container ...
};
```

**Beispiel (in stdlib etwas anders definiert)**

# sort implementieren

```
template<class Iterator>
void mySort(Iterator first, Iterator last) {

    if(first+1==last) return; // nur ein Element

    Iterator it=first;

    while(it != last-1) {
        Iterator next=it+1;
        if(*it > *next) swap(*it, *next);
        it++;
    }
    mySort(first, last-1);
}
```

Probleme?

```
template<class T>
void print(const T& val) {
    cout << val << " ";
}

int main() {
    vector<int> zahlen{42, 43, 44, 42, 1, 2, 5, 42};
    vector<char> zeichen{'a', 'b', 'a', 'c', 'd'};

    mySort(zahlen.begin(), zahlen.end());
    for_each(zahlen.begin(), zahlen.end(), print<int>);
    cout << endl;

    mySort(zeichen.begin(), zeichen.end());
    for_each(zeichen.begin(), zeichen.end(), print<char>);
    cout << endl;
}
```

# Verbessertes sort

```
template<class Iterator, class Compare>
void mySort(Iterator first, Iterator last, Compare comp) {

    Iterator prev = first, next = first;

    // prev != last sicherstellen, bevor next inkrementiert wird
    if (prev == last || ++next == last)
        return;

    /* Ein Containerdurchlauf. Am Ende ist next == last und
     * prev == last-1 */
    while(next != last) {
        if (comp(*next, *prev))
            swap(*prev, *next);

        prev = next++;
    }
    mySort(first, prev, comp); // Rekursion mit [first, last-1)
}
```

Erlaubt nun auch ForwardIterator  
und leere Container

# Vordefinierte Sortierreihenfolge

```
#include <functional>

template<class Iterator>
void mySort(Iterator first, Iterator last) {
    /* Aufruf mit Vergleichsfunktion < für den Elementtyp des Containers
     * auf den Iterator verweist */
    mySort(first, last, std::less<typename Iterator::value_type>());
}
```

```
int main() {
    vector<int> zahlen{42, 127, 3, 10, 20};
    forward_list<char> zeichen{'a', 'b', 'a', 'd',
        // aufsteigende Sortierung
        mySort(zahlen.begin(), zahlen.end());
        for_each(zahlen.begin(), zahlen.end(), print<int>);
        cout << endl;

        // absteigende Sortierung einer einfach verketteten Liste
        mySort(zeichen.begin(), zeichen.end(), std::greater());
        for_each(zeichen.begin(), zeichen.end(), print<char>);
        cout << endl;
}
```

Instanziieren von less mit dem Element-Datentyp des Containers (typename, weil Iterator::value\_type ein Template-Datentyp ist: mySort ist abhängig von Iterator.)

Alternative Sortierreihenfolge (absteigend)

# sort mit selbstdefinierten Datentypen

```
/operations.h:384:21: error: invalid operands to binary expression ('const Quadrat'  
and 'const Quadrat')  
    {return __x < __y;}  
    ~~~~ ^ ~~~~  
quadrat.cpp:39:9: note: in instantiation of member function 'std::less<Quadrat>::op-  
erator()' requested here  
    if (comp(*next, *prev))  
    ^
```

```
struct Quadrat {  
    int px; int py; int len;  
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};  
};  
  
int main() {  
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};  
  
    mySort(q.begin(), q.end());  
}
```

# sort mit selbstdefinierten Datentypen

```
bool operator<(const Quadrat& q1,  
                  const Quadrat& q2) {  
    return q1.len < q2.len;  
}
```

```
struct Quadrat {  
    int px; int py; int len;  
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};  
};  
  
int main() {  
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};  
  
    mySort(q.begin(), q.end());  
}
```

# sort mit selbstdefinierten Datentypen

```
#include <functional>

template<class Iterator>
void mySort(Iterator first, Iterator last) {
    /* Aufruf mit Vergleichsfunktion < für den Elementtyp des Containers
     * auf den Iterator verweist */
    mySort(first, last, std::less<typename Iterator::value_type>());
}
```

```
bool operator<(const Quadrat& q1,
                  const Quadrat& q2) {
    return q1.len < q2.len;
}
```

```
struct Quadrat {
    int px; int py; int len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};
};

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};
    mySort(q.begin(), q.end());
}
```

# std::sort

```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
           RandomAccessIterator last,
           Compare comp);
```

- Funktioniert so ähnlich wie unsere eigene Version

# std::sort

```
struct Quadrat {
    int px; int py; int len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};
};

bool kleiner(const Quadrat& q1, const Quadrat& q2) {
    return q1.len < q2.len;
}

template<class T>
void print(const T& val) {
    cout << val << " ";
}

template<>
void print<Quadrat>(const Quadrat& q) {
    cout << "Quadrat mit Kantenlaenge " << q.len << ", ";
}

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};

    std::sort(q.begin(), q.end(), kleiner);
    for_each(q.begin(), q.end(), print<Quadrat>);
    cout << endl;
}
```

# print für spezielle Datentypen

```
template <class T>
void print(const T &val) {
    cout << val << endl;
}
```

**generische Templatefunktion  
für alle Typen T**

```
template <>
void print<Quadrat>(const Quadrat &q) {
    cout << "Quadrat mit Kantenlaenge " << q.len << endl;
}
```

**Spezialisierung für Quadrat**

```
struct Quadrat {
    int px; int py; int len;
    Quadrat(int x, int y, int l):px(x), py(y), len(l){};
};

int main() {
    vector<Quadrat> q{Quadrat(0,0,5), Quadrat(5,5,10), Quadrat(10,10,2)};
    mySort(q.begin(), q.end());
    print("Quadrate"); print(-123);
    for_each(q.begin(), q.end(), print<Quadrat>());
    print(Quadrat{9,12,3});
}
```

# std::for\_each

```
#include <vector>
#include <iostream>

using namespace std;

void print(int val) {
    cout << val << " ";
}

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    for_each(zahlen.begin(), zahlen.end(), print);
}
```

# std::for\_each

```
template< class InputIt, class UnaryFunction >
constexpr UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

(since  
C++20)

- Wendet die übergebene Funktion auf alle Elemente in dem angegeben Bereich an
- Iteratoren werden derefenziert und die entsprechende Objekte dann der Funktion übergeben
- Funktion kann daher nur ein Argument bekommen
- Funktion gibt den Funktor ( $f$ ) zurück → z. B. für Zustand verwenden
- **Vorteile**
  - **Übersichtlich (eine Zeile für Schleife)**
  - **Ggf. Einfache Wiederverwendung von Funktionen**

# std::copy

```
template<class Input, class Output>
Output my_copy(Input first1, Input last1, Output output) {
    while(first1 != last1) {
        *output++ = *first1++;
    }
    return output;
}
```

stellt sicher, dass Speicher in result angelegt wird (result.reserve(...))

```
int main() {
    vector<int> zahlen{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    vector<int> result;

    my_copy(zahlen.begin(), zahlen.end(), back_inserter(result));
    for_each(result.begin(), result.end(), print);
}
```

# std::copy, std::copy\_if

```
bool isEven(int i) {
    return (i & 1) == 0;
}

int main() {
    vector<int> zahlen{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    vector<int> result;
    copy_if(zahlen.cbegin(), zahlen.cend(),
            back_inserter(result),
            isEven);
    for_each(result.begin(), result.end(), print<int>);
}
```

stellt sicher, dass Speicher in result angelegt wird (result.reserve(...))

# std::transform

```
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

void print(int val) {
    cout << val << " ";
}

int mal2(int val) {return val*2;}

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);
    for_each(zahlen.begin(), zahlen.end(), print);
}
```

# std::transform

```
template<class Input, class Output, class Func>
Output my_transform(Input first1, Input last1, Output output, Func f) {
    while(first1 != last1) {
        *output++ = f(*first1++);
    }
    return output;
}
```

```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    my_transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);
    for_each(zahlen.begin(), zahlen.end(), print);
}
```

# std::transform

## Funktionsweise:

```
template<class Input, class Output, class Func>
Output transform(Input first1, Input last1, Output output, Func f) {
    while(first1 != last1) {
        *output++ = f(*first1++);
    }
    return output;
}
```

```
int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), mal2);
    for_each(zahlen.begin(), zahlen.end(), print<int>);
}
```

Ersetzt hier bestehende Elemente,  
daher ausnahmsweise kein  
insert\_iterator notwendig.



# std::bind: Argumente binden

- Erzeugt Funktor
  - Argumente können festgelegt werden
  - Platzhalter `_1, _2, _3 ...` für nicht gebundene Argumente

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <vector>

using namespace std;
using namespace std::placeholders;

int mult(int a, int b) {
    return a * b;
}

int main() {
    vector<int> zahlen{9,8,7,6,5,4,3,2,1,0};

    transform(zahlen.begin(), zahlen.end(), zahlen.begin(), bind(mult, 5, _1));
    for_each(zahlen.begin(), zahlen.end(), print);
}
```

**Binäre Funktion,**  
bind erzeugt unären Funktor,  
Erstes Argument festgelegt (= 5)

Platzhalter für Werte von zahlen

# std::accumulate

accumulate()  
reduce()

```
#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

int mult(int a, int b) {
    return a * b;
}

int main() {
    vector<int> zahlen{9, 8, 7, 6, 5, 4, 3, 2, 1};

    int result = accumulate(zahlen.begin(), zahlen.end(), 1, mult);
    cout << result << endl;
}
```

**Binäre Funktion**

**Initialwert für Parameter init**

```
graph TD
    A[Binäre Funktion] --> B[mult]
    C[Initialwert für Parameter init] --> D[1]
```

# Stream-Iteratoren (1/2)

- Auf I/O-Streams über Iterator-Schnittstelle zugreifen
  - Template-Argument ist Element-Typ
    - d.h., Typ der Werte, die gelesen werden sollen
    - und die der Iterator liefern soll
  - Vorteile?

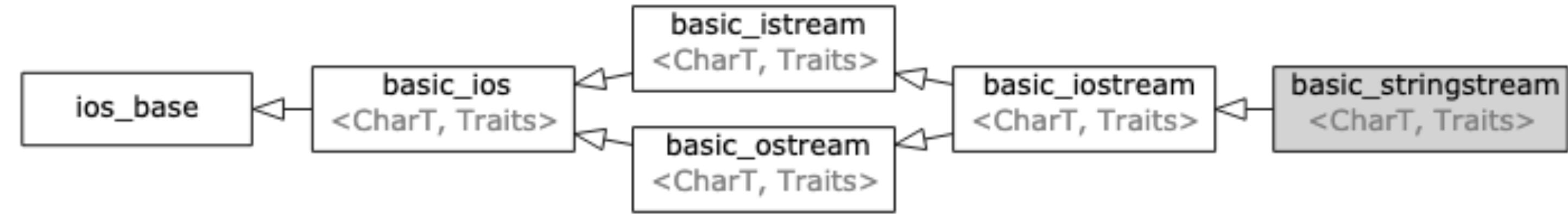
```
1 #include <fstream>
2 #include <string>
3 #include <iostream>
4
5 using namespace std;
6
7 int main() {
8     ifstream datei ("stream-it.cpp");
9     istream_iterator<string> wort(datei);
10    istream_iterator<string> ende;
11
12    while (wort != ende) {
13        cout << *wort++ << endl;
14    }
15 }
```

# Stream-Iteratoren (2/2)

- Funktioniert auch mit Ausgabe-Iteratoren
- ostream\_iterator
- Man kann dabei noch Trennzeichen als Parameter angeben

```
1 #include <fstream>
2 #include <string>
3 #include <iostream>
4
5 using namespace std;
6
7 int main() {
8     ifstream datei("stream-it.cpp");
9     istream_iterator<string> wort(datei);
10    istream_iterator<string> ende;
11    ofstream ziel ("Ergebnis.txt");
12    ostream_iterator<string> ausgabe(ziel, "*\n");
13
14    while (wort != ende) {
15        *ausgabe++ = *wort++;
16    }
17 }
```

# std::stringstream



```
1 #include <sstream>
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     stringstream s;
8
9     s << "Hallo, hallo." << endl;
10
11    cout << s.str();
12
13    return 0;
14 }
```

```
1 #include <sstream>
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     stringstream s("Von einem string "
8                  "wie aus einem istream lesen");
9     string res;
10
11    s >> res;
12    cout << res << endl;
13
14    return 0;
15 }
```

# Casting in C++

- Programmierer soll weiterhin Freiheit haben
- Aber: harmloses von gefährlichem Casting unterscheidbar machen
- Einige Cast-Operationen vom Compiler überprüfbar machen
- Daher unterschiedliche Cast-Operationen definiert:
  - `static_cast`: Typumwandlung zur Übersetzungszeit
  - `dynamic_cast`: Typumwandlung zur Laufzeit
  - `const_cast`: `const`-Eigenschaften entfernen
  - `reinterpret_cast`: Datentyp beliebig neu interpretieren

# static\_cast

Scoped Enum

```
enum class Wochentag {sonntag, montag, dienstag, mittwoch,  
                      donnerstag, freitag, samstag  
} heute = Wochentag::dienstag;  
  
int i = static_cast<int>(Wochentag::dienstag);  
heute = i;                                // Fehler, Datentyp inkompatibel  
heute = static_cast<Wochentag>(i);          // erlaubt!
```

- Typumwandlungen durchführen oder rückgängig machen
- Zur Übersetzungszeit wird geprüft, ob Typen kompatibel sind
- Auch bei Vererbung anwendbar:

```
GraphObj g(Ort(3, 17));  
Strecke s(Ort(3, 17), (Ort(0, 0));           // Strecke ist von GraphObj abgeleitet  
GraphObj *pg;  
Strecke *ps {&s};  
pg = ps;                                         // bekannte implizite Konversion  
ps = pg;                                         // verboten!  
ps = (Strecke*) pg;                            // gefährlicher C-Stil!  
ps = static_cast<Strecke*> (pg);              // nur richtig, falls pg auf ein Strecke-Objekt zeigt
```

Besser mit dynamic\_cast...

Ab C++23 auch:

```
i = to_underlying (heute);
```

# dynamic\_cast

```
class A {  
public:  
    virtual void foo();  
    void f();  
};  
  
class B: public A {  
public:  
    void g();  
};  
  
void test(A* obj) {  
    B* bobj = dynamic_cast<B*>(obj);  
    if(bobj != 0) {  
        bobj->g();  
    } else {  
        // Fehler  
    }  
}
```

Auch mit Referenzen:  
B &bobj = dynamic\_cast<B&>(ref);  
→ kann Exception `bad_cast` auslösen

# dynamic\_cast

- `dynamic_cast<T>(Ausdruck)`
- Typüberprüfung findet zur Laufzeit statt
  - Auf der Basis der dynamischen Typ-Information
  - Basisklasse muss eine virtuelle Funktion haben
- Typ  $T$  muss ein Zeiger oder eine Referenz auf eine Klasse sein
- Falls das Argument *Ausdruck* ein Zeiger ist, der nicht auf ein Objekt vom Typ  $T$  (oder abgeleitet von  $T$ ) zeigt, wird als Ergebnis 0 (Null-Pointer) zurückgegeben (Vorsicht!)
- Falls das Argument *Ausdruck* eine Referenz ist, die nicht auf ein Objekt vom Typ  $T$  (oder abgeleitet von  $T$ ) verweist, wird eine Ausnahme (Exception) vom Typ `bad_cast` erzeugt.

# const\_cast

- const-Eigenschaft: signalisiert dem Compiler: Darf man nicht ändern
  - Hilfreich, wenn man unbeabsichtigtes Ändern automatisch verhindern lassen möchte
  - Bei Parameter-Übergabe übergibt man oft const-Referenzen (z. B. `const string &`): Effizient und trotzdem sicher
- Manchmal möchte man aber doch ein Objekt verändern können, das eigentlich const ist

```
const int i {100};  
const int *ip {&i};  
*ip = 0;                      // geht nicht  
int *iq {const_cast<int*>(&i)};  // explizite Typumwandlung  
*iq = 0;                        // Wert von i wird geändert!
```

Nur in begründeten Ausnahmefällen!

# reinterpret\_cast

```
#include <iostream>

int main() {
    double d;
    std::cout.write(reinterpret_cast<char*>(&d), sizeof(d));
}
```

- Typ-Umwandlung erzwingen
- Zum Beispiel, wenn man mit Byte-Arrays umgeht
- Maximal „brutaler“ Cast
- Sollte nur verwendet werden, wenn die anderen Cast-Operationen nicht anwendbar sind