University of Applied Sciences
HOCHSCHULE EMDEN·LEER Networked Systems

Programmieren 3
C++

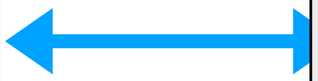
Vorlesung 06: Algorithmen, Funktoren

Prof. Dr. Dirk Kutscher
Dr. Olaf Bergmann

Wiederholung

Konstruktoren in C++

```
int main() {
  SimpleString greeting1("Moin");
}
```



```
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;
  size_t bufferSize;

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

- **Converting Konstruktor**
  - Parameter eines anderes Typs
  - Neues Objekt mit dem übergebenen Objekt initialisieren
  - Oder: Bestehendes Objekt in ein neues Objekt der Klasse konvertieren

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

```
SimpleString::SimpleString(const char *initString)
  : SimpleString() {
  size_t stringLength = strlen(initString);

  size_t sz = increaseBuffer(stringLength);
  strncpy(buffer, initString, sz);
}
```

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 instanziiert
  - 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**

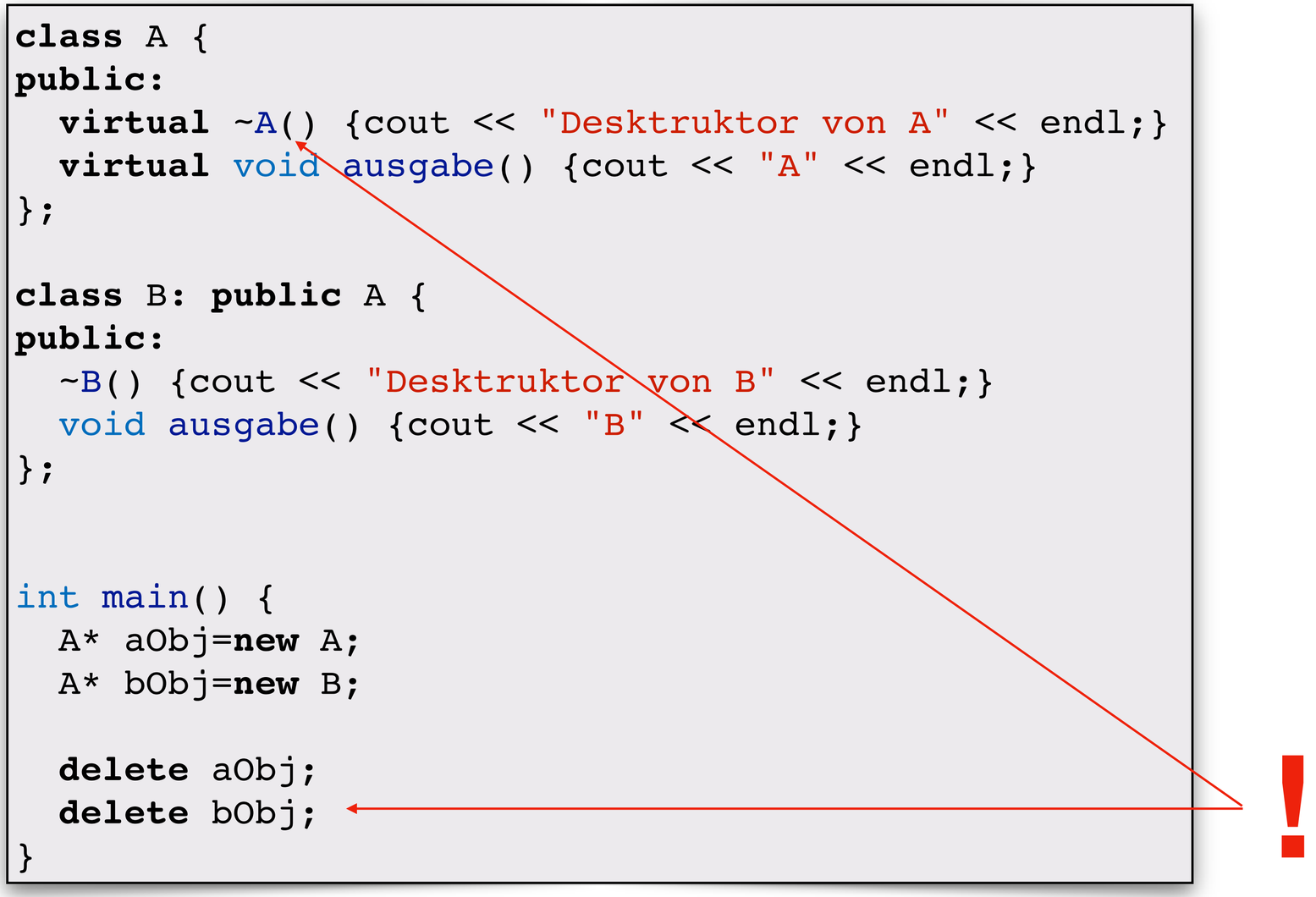
```
SimpleString::~SimpleString(void) {
  delete[] buffer;
}
```

Virtueller Destruktor

```
class A {
public:
  ~A() {cout << "Desktruktor von A" << endl;}
  virtual void ausgabe() {cout << "A" << endl;}
};

class B: public A {
public:
  ~B() {cout << "Desktruktor von B" << endl;}
  void ausgabe() {cout << "B" << endl;}
};


int main() {
  A* aObj=new A;
  A* bObj=new B;

  delete aObj;
  delete bObj;
}
```

Virtueller Destruktor

```
class A {
public:
  virtual ~A() {cout << "Desktruktor von A" << endl;}
  virtual void ausgabe() {cout << "A" << endl;}
};

class B: public A {
public:
  ~B() {cout << "Desktruktor 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

SimpleString

```
#include <cstring>
#include <cstdlib>
#include <iostream>

// SimpleString class with dynamic memory management

using namespace std;

class SimpleString {

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

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

private:
  char *buffer;
  size_t bufferSize;

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

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

  greeting.add(name);

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

  return 0;
}
```

```
/ Standard constructor
SimpleString::SimpleString(void)
  : buffer(nullptr), bufferSize(0) {
}

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

// Copy constructor
SimpleString::SimpleString(const SimpleString &initString)
  : SimpleString() {
  init(initString.buffer);
}

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

void SimpleString::init(const char *initString) {
  if (initString) {
    size_t length = increaseBuffer(strlen(initString));
    strncpy(buffer, initString, length);
  }
}

size_t SimpleString::increaseBuffer(size_t newSize) {
  if (newSize > 0) { // change only when necessary
    char *newBuffer = new char[newSize + 1]; // may throw std::bad_alloc
    if (bufferSize > 0) {
      strncpy(newBuffer, buffer, bufferSize);
      delete[] buffer;
    }

    buffer = newBuffer;
    bufferSize = newSize + 1;
  }
  return bufferSize;
}

void SimpleString::add(const SimpleString &addedString) {
  size_t newSize = size() + addedString.size();
  size_t sz = newSize + 1;
  if (bufferSize < newSize + 1) {
    sz = increaseBuffer(newSize);
  }
  strncat(buffer, addedString.buffer, sz);
}
```

Templates

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

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

Template-Parameter
als Platzhalter für Datentyp

Benutzung:

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

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

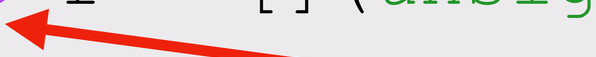
```
#include <iostream>

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;
}
```



Compiler leitet Deklaration automatisch ab

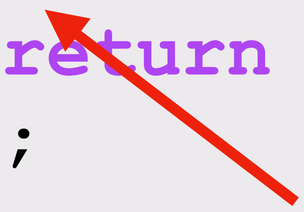
`accumulate` mit Lambda

```
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

Objekt kann Zustand haben

Funktor muss mindestens `operator()` implementieren

```
class Data {
   int wert = 0;
public:
   int operator()(void) { return wert++; }
};


std::vector<int> v;

std::generate_n(inserter(v, v.end()), 100, Data());
```

Erzeugen eines anonymen Funktionsobjekts des Klasse `Data` (Wertparameter für `generate_n`)

Containerklassen in C++

std::vector und andere Container-Klassen

- Sind alle als Template-Klassen definiert
  - Daher schreiben wir immer `vector<int>`

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

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

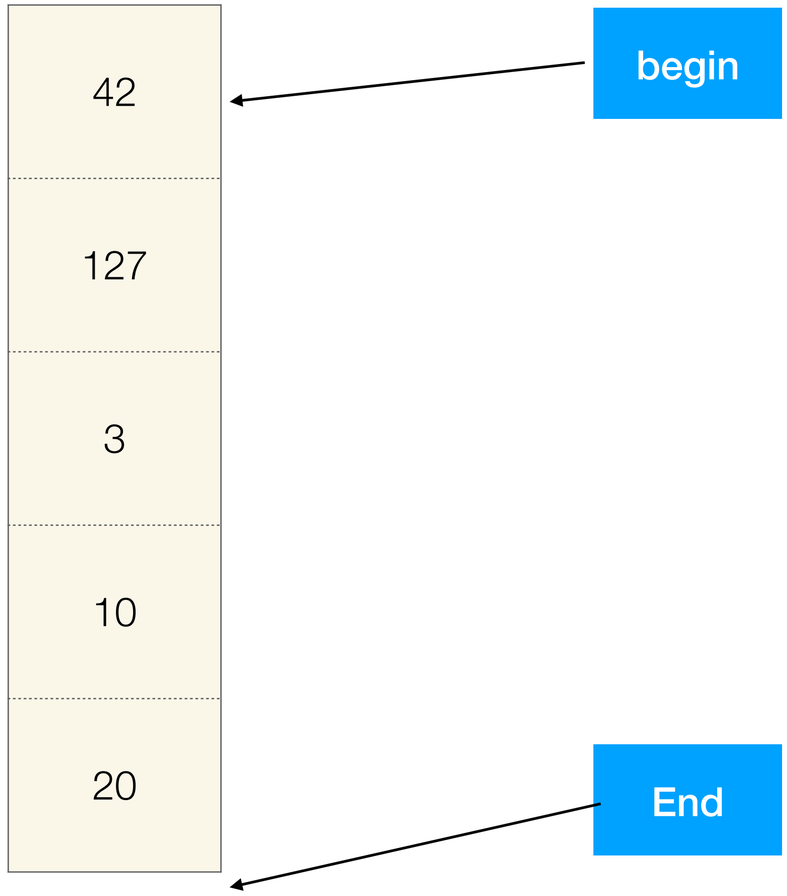
https://en.cppreference.com/w/cpp/container

C++ und Container

Container mit Werten

Iteratoren:
Verweise auf “Positionen”
im Container

Algorithmen:
Generische Funktionen,
die über Iteratoren
auf Container zugreifen




```
find()
count()
sort()
copy()
…
```

Alle als
Templates definiert!

Beispiel

find
(Funktion aus <algorithm>

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

int main() {
  std::vector<int> zahlen{42, 3, 10};
  int suchwert;

  std::cout << "Zahl: ";
  std::cin >> suchwert;

  auto result = std::find(std::begin(zahlen), std::end(zahlen), suchwert);

  if(result !=  std::end(zahlen))
    std::cout << "element " << *result << " gefunden." << std::endl;
  else
    std::cout << "element nicht gefunden." << std::endl;
}
```

find liefert Iterator

Vergleich mit Iterator für “Ende”

Übungsaufgabe

- String-Klasse auf `vector<char>` umbauen
- Neue Funktion: `zuweisung`

```
class SimpleString {

public:
   void zuweisung(const SimpleString &s);
};
```

SimpleString mit vector

```
class SimpleString {

public:
  SimpleString(void) : len(0) {}
  SimpleString(const char *initString);

  string to_string(void) const;

private:
  vector<char> buffer;
  size_t len;
};
```

```
int main() {
  SimpleString test("test");

  cout << test.to_string() << endl;

}
```

SimpleString mit vector

```
class SimpleString {

public:
  SimpleString(void) : len(0) {}
  SimpleString(const char *initString);

  string to_string(void) const;

private:
  vector<char> buffer;
  size_t len;
};

SimpleString::SimpleString(const char *initString)
  : len(strlen(initString)) {

  copy(initString, initString+len, inserter(buffer,buffer.begin()));
}


string SimpleString::to_string(void) const {
  string res;

  copy_n(buffer.begin(), len, inserter(res, res.begin()));

  return res;
}
```

```
int main() {
  SimpleString test("test");

  cout << test.to_string() << endl;

}
```

SimpleString mit vector

```
class SimpleString {

public:
  SimpleString(void) : len(0) {}
  SimpleString(const char *initString);

  string to_string(void) const;

  void add(const SimpleString &addedString);


private:
  vector<char> buffer;
  size_t len;
};
```

```
int main() {
  SimpleString test("test");
  cout << test.to_string() << endl;

  SimpleString hello("Hello"), world(", world!");

  hello.add(world);
  cout << hello.to_string() << endl;
}
```

```
void SimpleString::add(const SimpleString &addedString) {
  buffer.reserve(buffer.size() + addedString.buffer.size());
  buffer.insert(buffer.end(), addedString.buffer.begin(), addedString.buffer.end());
  len+=addedString.len;
}
```

SimpleString mit vector

```
class SimpleString {

public:
  SimpleString(void) : len(0) {}
  SimpleString(const char *initString);

  string to_string(void) const;

  void add(const SimpleString &addedString);

  bool isEqual(const SimpleString &s) const
  {return (len==s.len) && (buffer==s.buffer);};

  int find(const SimpleString &s) const;

private:
  vector<char> buffer;
  size_t len;
};
```

```
int main() {
  SimpleString test("test");
  cout << test.to_string() << endl;

  SimpleString hello("Hello"), world(", world!");
  hello.add(world);
  cout << hello.to_string() << endl;

  testIsEqual("Moin", "Moin", true);
  testIsEqual("Moin", "Hallo", false);

  testFind("Moin", "in", 2);
  testFind("Moin C++", "C++", 5);
  testFind("Moin C++", "Java", 0);
}
```

```
int SimpleString::find(const SimpleString &s) const {
  int pos = -1;
  auto it = search(buffer.begin(), buffer.end(), s.buffer.begin(), s.buffer.end());

  if (it != buffer.end())
    pos = (int)(it-buffer.begin());

  return pos;
}
```

SimpleString: Testfunktionen

```
void okMsg(bool ok) {
  cout << "Ergebnis: " << (ok?"OK":"ERROR") << endl << endl;
}

bool testIsEqual(const SimpleString &s1, const SimpleString &s2, bool expRes) {

  bool res = false;

  cout << s1.to_string() << ((res=s1.isEqual(s2))?"==":"!=") << s2.to_string() << endl;

  okMsg(expRes == res);
  return res;

}

int testFind(const SimpleString &s1, const SimpleString &s2, int expRes) {
  int res(0);

  cout << "\"" << s1.to_string() << "\".find(\"" << s2.to_string() << "\")==" << (res=s1.find(s2)) << endl;

  okMsg(expRes == res);
  return res;
}
```

```
int main() {
  SimpleString test("test");
  cout << test.to_string() << endl;

  SimpleString hello("Hello"), world(", world!");
  hello.add(world);
  cout << hello.to_string() << endl;

  testIsEqual("Moin", "Moin", true);
  testIsEqual("Moin", "Hallo", false);

  testFind("Moin", "in", 2);
  testFind("Moin C++", "C++", 5);
  testFind("Moin C++", "Java", 0);
}
```

SimpleString mit vector

```
class SimpleString {

public:
  SimpleString(void) : len(0) {}
  SimpleString(const char *initString);

  string to_string(void) const;

  void add(const SimpleString &addedString);

  bool isEqual(const SimpleString &s) const {return (len==s.len) && (buffer==s.buffer);}
  int find(const SimpleString &s) const;
  void zuweisung(const SimpleString &s)
  {len = s.len; buffer = s.buffer;}
private:
  vector<char> buffer;
  size_t len;
};
```

```
int testZuweisung(SimpleString s1, const SimpleString& s2) {
  int res = 0;

  cout << "\"" << s1.to_string()
       << "\".zuweisung(\""
       << s2.to_string() << "\")" << endl;

  s1.zuweisung(s2);
  okMsg(s1.isEqual(s2));
  return res;
}
```

```
int main() {
  SimpleString test("test");
  cout << test.to_string() << endl;

  SimpleString hello("Hello"), world(", world!");
  hello.add(world);
  cout << hello.to_string() << endl;

  testIsEqual("Moin", "Moin", true);
  testIsEqual("Moin", "Hallo", false);

  testFind("Moin", "in", 2);
  testFind("Moin C++", "C++", 5);
  testFind("Moin C++", "Java", 0);

  testZuweisung("Moin C++", "Java");
  testZuweisung("Foo", "Bar");

  return 0;
}
```

Mit Operatoren

```
class SimpleString {

public:
  SimpleString(void) : len(0) {}
  SimpleString(const char *initString);

  string to_string(void) const;

  const SimpleString &operator+=(const SimpleString &addedString);

  bool operator==(const SimpleString &s) const {return (len == s.len) && (buffer == s.buffer);}
  int find(const SimpleString &s) const;
  const SimpleString &operator=(const SimpleString& s) {len=s.len; buffer=s.buffer; return *this;}

private:
  vector<char> buffer;
  size_t len;
};
```

```
const SimpleString& SimpleString::operator+=(const SimpleString &addedString) {
  buffer.reserve(buffer.size() + addedString.buffer.size());
  buffer.insert(buffer.end(), addedString.buffer.begin(), addedString.buffer.end());
  len += addedString.len;
  return *this;
}
```

Container in C++ (Stdlib)

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

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

Associative containers

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

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

https://en.cppreference.com/w/cpp/container

std::map

```cpp
#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