Einführung in C++

Martin Holzer

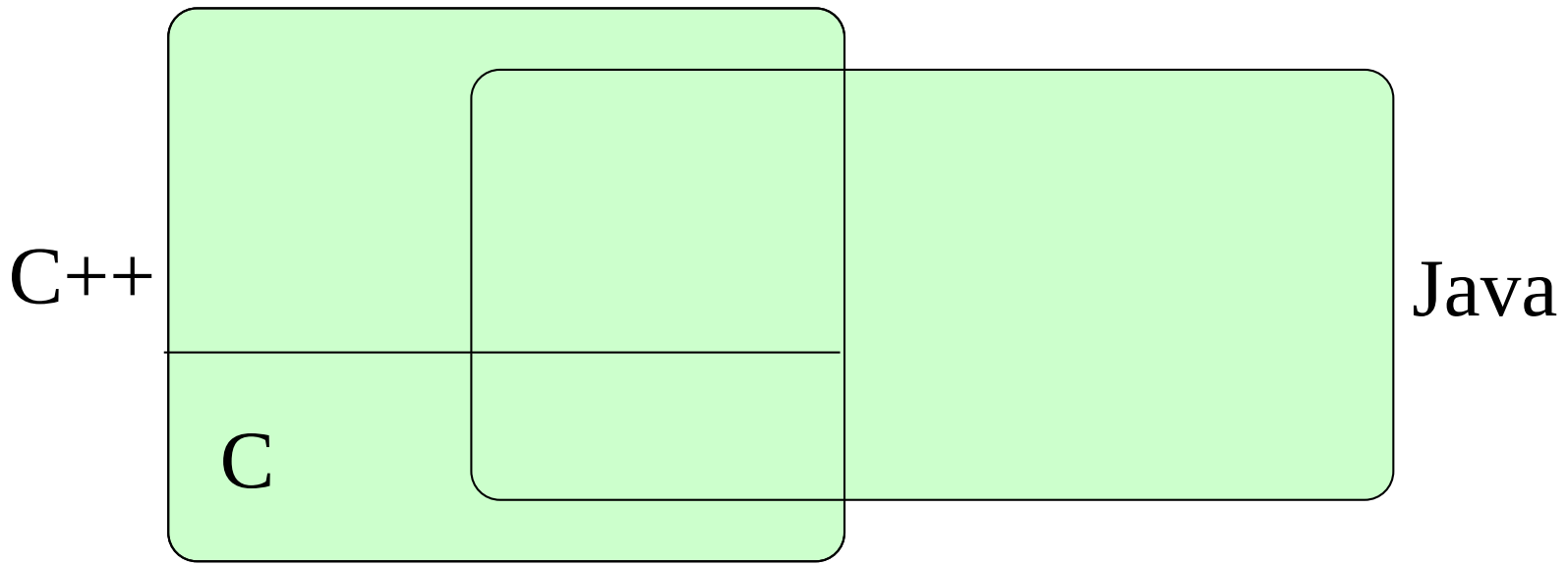
Lehrstuhl für Algorithmik I

April/Mai 2005

Zielstellung

- Vermittlung grundlegender Kenntnisse in C++
- Gegenüberstellung von Java und C++
- Voraussetzung:
  Kenntnis allgemeiner Programmierkonzepte:
  - Syntax
  - Variablen
  - Schleifen
  - Verzweigungen
  - Unterprogramme
  - Objektorientierung
  - etc.

Verwandtschaft C++ und Java



Frage

Welches sind die größten Unterschiede zwischen Java und C++?

Antwort (diskussionsfähig)

	Java	C++
Variablen/ Parameter	Referenzen	Kopien, Referenzen, Zeiger
Speicherverwaltung	automatisch (Garbage-Collection)	Selbstverwaltung
Objektorientierung	strikter	C-Erbe
Portabilität	Java-Bytecode, JVM	direkt ausführbares Programm (maschinenabhängig)
⋮	⋮	⋮

Inhalt

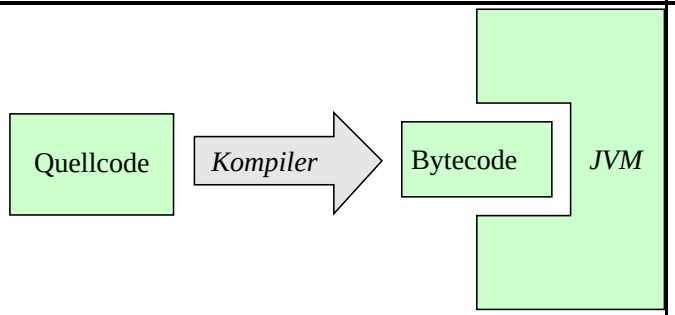
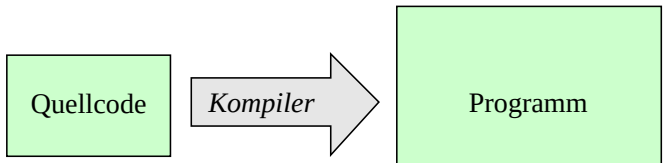
Referenzen

- Thomas Willhalm:
  Von Java nach C++
  `http://i11www.ira.uka.de/teaching/scripts/sources/java2c++.pdf`
- Bjarne Stroustrup:
  The C++ Programming Language
  Addison-Wesley
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
  Design Patterns — Elements of Reusable Object-Oriented Software
  Addison-Wesley

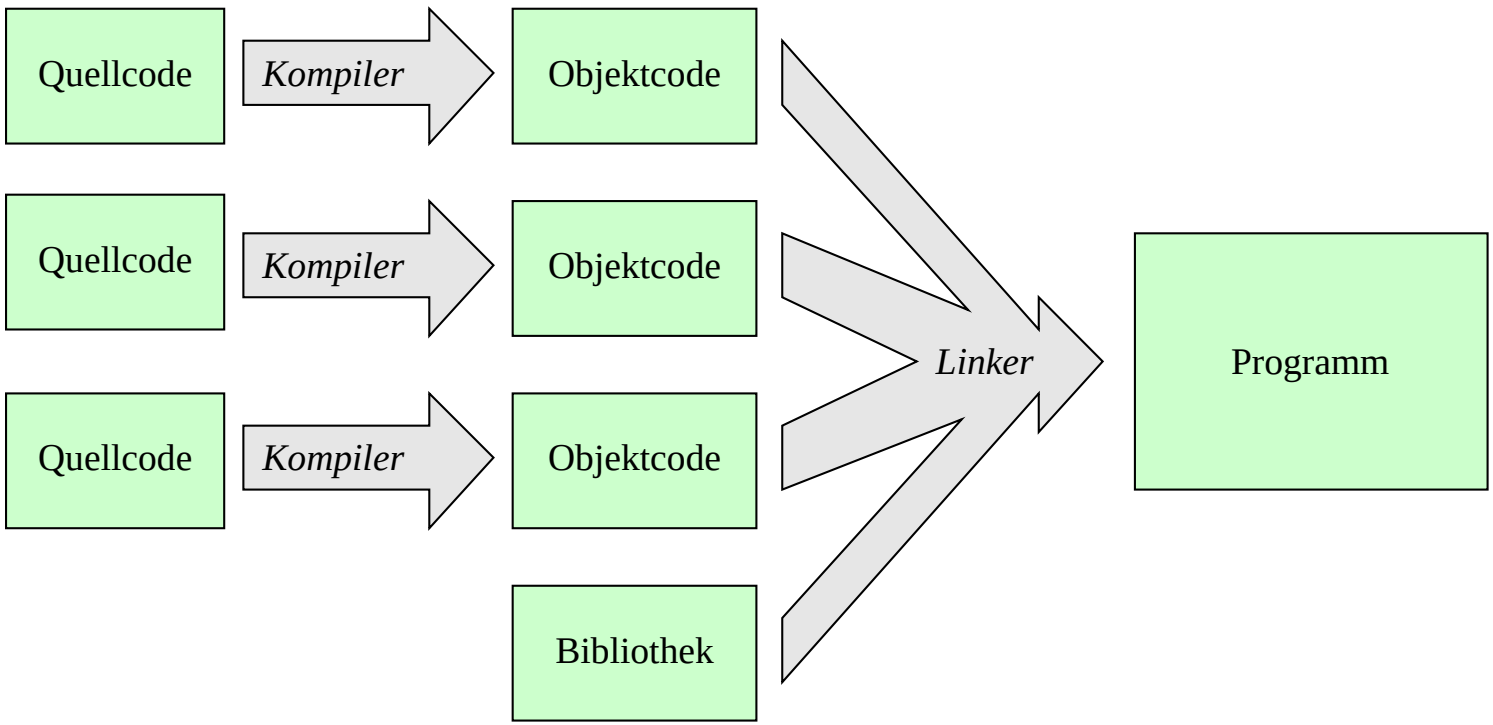
Online-Referenzen

- The `cplusplus.com` Tutorial
  `http://www.cplusplus.com/doc/tutorial/`
- Bruce Eckel:
  Thinking in C++
  `http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`
- STL Programmer's Guide
  `http://www.sgi.com/tech/stl/`

Kompilieren und Linken — Vergleich

	Java	C++
Kompilieren	 Resultat: Java-Bytecode	 (vereinfacht) Resultat: ausführbares Programm

Kompilieren und Linken in C++



Der Compiler — Syntax

```
g++ -c -Wall -g Quelldatei -o Objektdatei
```

Argumente:

- `g++ -c`: Kompilieren
- `-Wall`: alle Warnungen einschalten
- `-g`: Debugging-Informationen
- *Quelldatei*
- `-o` *Objektdatei*: Ausgabedatei

Der Linker

- Entwicklungstool, unabhängig vom Compiler
- Linker stellt fest, ob alle Programmteile vorhanden sind
- Entwickler muss vor Linken Aktualität der Objektdateien sicherstellen
- Automatisierung?
- `make` (s. u.)

Syntax

`g++ -Wall -g` ⟨*Objektdateien*⟩ `-o` *Zieldatei*

Argumente:

- `g++`: Linken
- `-Wall`: alle Warnungen einschalten
- `-g`: Debugging-Informationen
- ⟨*Objektdateien*⟩: Liste (leerzeichenseparierter) von Objektdateien
- `-o` *Zieldatei*: Ausgabedatei

Das Tool make

- Zentrale Datei mit Regeln, d. h. Information, welche Objektdateien jeweils benötigt werden
- *make* überprüft an Hand des Zeitpunkts der letzten Dateiänderung, welche Quelltexte kompiliert und welche Objektdateien gelinkt werden müssen
- Abhängigkeit kann beliebig kompliziert werden

Syntax

Ziel: ⟨*Abhängigkeiten*⟩
⟶ *Befehl*

- *Ziel*: Regelname
- ⟨*Abhängigkeiten*⟩: Liste (leerzeichenseparierter) von vorausgesetzten Dateien
- ⟶: Tabulator
- *Befehl*: Kommando zum Erstellen der Zieldatei

Beispiel

Makefile

```
myprog.o: myprog.cc
  g++ -c -Wall -g myprog.cc -o myprog.o
datastruc.o: datastruc.cc
  g++ -c -Wall -g datastruc.cc -o datastruc.o
myprog: myprog.o datastruc.o
  g++ -Wall -g myprog.o datastruc.o -o myprog
```

Arbeitsweise

Aufruf

`make` *Regel*

`make myprog`

- Überprüfung der Aktualität von `myprog` (Ziel jünger als Objektdateien)
- Falls nicht aktuell (oder nicht vorhanden):
  - Evtl. Erzeugen von `myprog.o` bzw. `datastruc.o`
  - Ausführen des Link-Befehls der Regel `myprog`

Beispiel mit Variablen

```
CC = g++
FLAGS = -Wall -g
myprog.o: myprog.cc
  $(CC) -c $(FLAGS) myprog.cc -o myprog.o
datastruc.o: datastruc.cc
  $(CC) -c $(FLAGS) datastruc.cc -o datastruc.o
myprog: myprog.o datastruc.o
  $(CC) $(FLAGS) myprog.o datastruc.o -o myprog
```

Oram/Talbott: *Managing Projects with make*.
O'Reilly & Associates, Inc.

Weitere Tools

- Entwicklungsumgebung, z. B. *Eclipse*
- Dokumentationstool, z. B. *Doxygen*
- Debugger, z. B. `ddd`
- Profiler, z. B. `valgrind`

Fragen

- Welche grundsätzlichen Merkmale hat eine Klasse?
- Was ist ein Konstruktor?
- Was ist der Unterschied zwischen Deklaration und Definition?

Antworten

- Klasse:
  Kapselung von Attributen (Variablen) und Methoden (Funktionen), privat $\leftrightarrow$ öffentlich, statisch
- Konstruktor:
  Methode, die bei der Instanziierung aufgerufen wird
- Deklaration: ‚Skelett' festlegen
  Definition: Funktionalität/‚Inhalt' festlegen

Klassen — Syntax

	Java	C++

Klassen	`class` *Klassenname* {
	`private` *Var.-/Fkt.-Dekl.*
	
	`public` *Konstruktor*
	
	`public` *Var.-/Fkt.-Dekl.*
	}
		`class` *Klassenname* {
		`private:`
		priv. Var.-/Fkt.-Dekl.
		
		`public:`
		Konstruktor
		
		öff. Var.-/Fkt.-Dekl.
		};

Beispiel

```
class Vektor {
private:
  double myX, myY;

public:
  Vektor(double x, double y) : myX(x), myY(y) {};

  Vektor addiere(Vektor v)
  {
    return Vektor(myX+v.myX, myY+v.myY);
  }
};
```

Hinweise

`private:` kann weggelassen werden (Variablen und Funktionen vor `public:` automatisch als privat deklariert).

; nach Klassendefinition notwendig.

	Java	C++
Variablen-initialisierung	`this.x = x`	`: myX(x)`

Hinweise

Variablen sollten in Reihenfolge ihrer Deklaration initialisiert werden (sonst Warnung).

Falls kein Konstruktor definiert: Standardkonstruktor (hat keine Parameter und ruft Standardkonstruktoren der Datenelemente auf).

Falls eigener Konstruktor definiert ist, wird kein Standardkonstruktor definiert.

Deklaration und Definition

Deklaration

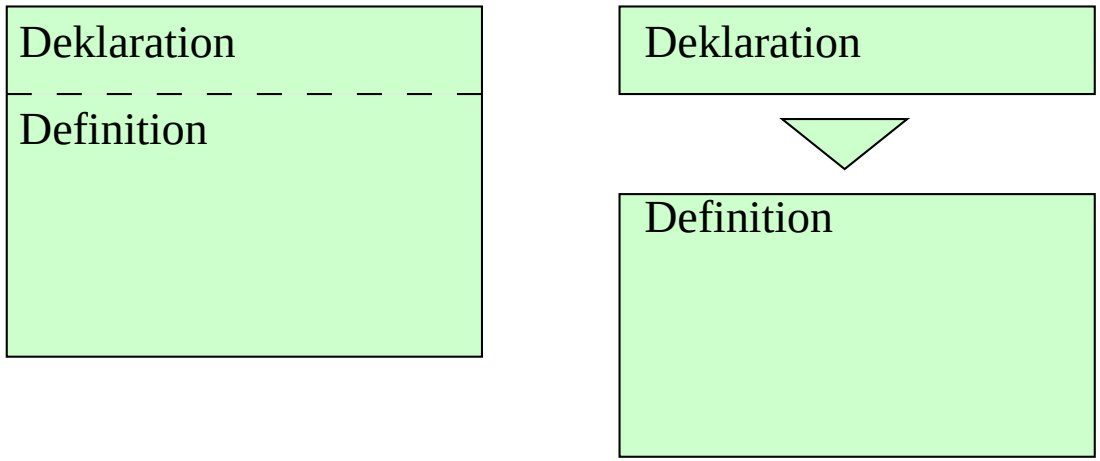
```
class Vektor {
  double myX, myY;

public:
  Vektor(double x, double y) : myX(x), myY(y) {};
  Vektor addiere(Vektor v);
};
```

Definition

```
Vektor Vektor::addiere(Vektor v)
{
  return Vektor(myX+v.myX, myY+v.myY);
}
```

Die Header-Datei

- Definition zu verarbeiten dauert relativ lange
- Von anderen Programmteilen wird nur Deklaration benötigt
- $\Longrightarrow$ Deklaration und Definition getrennt verarbeiten
- Deklaration in sog. Header-Datei



Die Deklaration

vektor.h

```
class Vektor {
  double myX, myY;

public:
  Vektor(double x, double y) : myX(x), myY(y) {};

  Vektor addiere(Vektor v);
};
```

	Java	C++
Schnitt-stelle, ‚Signatur‘	Interface	Deklaration

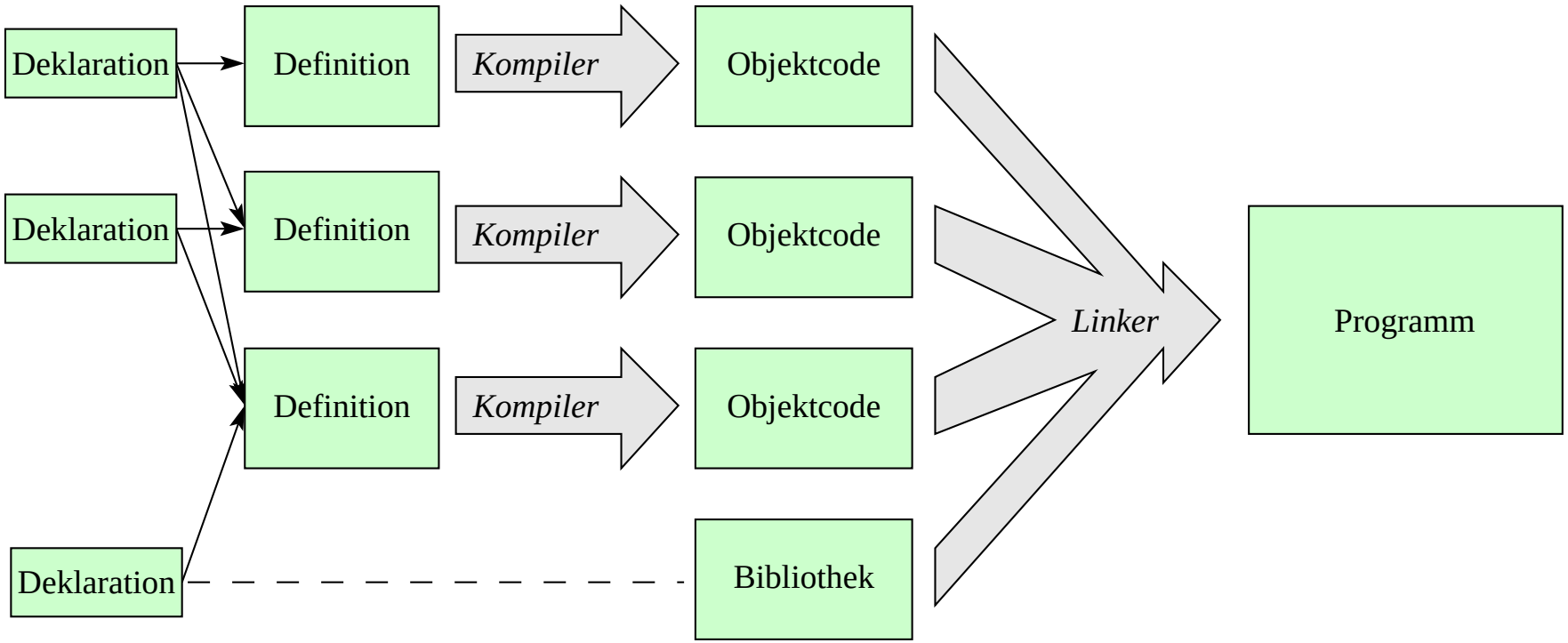
Die Definition

vektor.cc

```
#include "vektor.h"

Vektor Vektor::addiere(Vektor v)
{
  return Vektor(myX+v.myX, myY+v.myY);
}
```

	Java	C++
Klassenreferenz	*Klasse.Element*	*Klasse::Element*

Kompilieren und Linken (mit Header-Dateien)



make
(mit Header-Dateien)

```
CC = g++
FLAGS = -Wall -g
myprog.o: myprog.cc datastruc.h
  $(CC) -c $(FLAGS) myprog.cc -o myprog.o
datastruc.o: datastruc.cc
  $(CC) -c $(FLAGS) datastruc.cc -o datastruc.o
myprog: myprog.o datastruc.o
  $(CC) $(FLAGS) myprog.o datastruc.o -o myprog
```

Compiler-Guards

- Compiler überprüft, ob Klassen- (Funktions-, Variablen-)Name schon einmal deklariert wurde
  $\Longrightarrow$ sonst Fehlermeldung
- $\Longrightarrow$ Fehlermeldung, falls Header-Datei mehrfach eingebunden wird
- Automatisierung?
- Compiler-Guards:
  Definition einer Markierung, wenn Datei bearbeitet (eingebunden) wird

Beispiel

vektor.h

```
#ifndef VEKTOR_INC
#define VEKTOR_INC

class Vektor {
  double myX, myY;

public:
  Vektor(double x, double y) : myX(x), myY(y) {};

  Vektor addiere(Vektor v);
};

#endif
```

Übersicht über Compiler-Direktiven

`#include "`*Dateiname*`"`
: Platziere Inhalt von *Dateiname* an diese Stelle.

`#include <`*Dateiname*`>`
: Variante für Systemdateien.

`#ifndef` *MACRO*
: Bearbeite nachfolgenden Code nur, wenn *MACRO* nicht definiert.

`#endif`
: Ende der Bedingung.

`#define` *MACRO*
: Definiere *MACRO*.

Ausgabe — Syntax

	Java	C++
Ausgabe	`System.out.println(`*String*`);`	`std::cout <<` *String* `<< std::endl;`
Main	`public static void main(String[] args) {}`	`int main() {}` *oder* `int main(int argn, char* argv[]) {}`

Beispiel

```
#include <iostream>

int main()
{
  int a(5);
  int b(a+3);
  std::cout << "a ist " << a
            << " und b ist " << b << std::endl;
  return 0;
}
```

Fragen

- Welche Funktion haben **Bezeichner (Qualifier)**?
- Welche Bezeichner haben Sie bereits kennengelernt?
- Welche Bedeutung hat eine `static` deklarierte Variable in Java?

Antworten

- Funktion von Bezeichnern:
  Zugriffskontrolle etc.
- Bisher kennengelernt:
  `private`, `public`
- `static` deklarierte Variable:
  Nur eine Instanz pro Klasse

`static` vs. `extern`

`static`	`extern`
nur Programmteilen bekannt, die Definition sehen	allen Programmteilen bekannt, die dazugelinkt werden
Standard für Variablen	Standard für Funktionen
`static int abs(int i);` besser: private Funktion einer Klasse	`extern node_array xpos;` besser: als Parameter übergeben

`extern` deklarierte Variablen sind schlechter Stil!

`static`

`static` hat weitere Bedeutung (ähnlich Java):

`static` definierte Variable/Funktion:

- statische Variable existiert nur einmal pro Klasse
- statische Funktion darf nur statische Variablen und Funktionen verwenden

Beispiel `static`

```
class Vektor {
    ⋮
  static int maximalLength;
  static Vektor generateRandom();
    ⋮
}
```

`inline`

- Bezeichner für Funktionen
- Hinweis an Compiler: kein Funktionsaufruf, sondern direktes Einfügen des Codes
- $\oplus$ Zeitersparnis
- $\ominus$ Programmvergrößerung
- `inline`-Methode ist automatisch statisch
- Automatismus: Methode ist bei gleichzeitiger Deklaration und Definition `inline` deklariert

Beispiele `inline`

```
inline int abs(int i) { return i<0 ? -i : i; }
```

```
class Vektor {
  double myX, myY;

public:
  Vektor(double x, double y) : myX(x), myY(y) {};

  Vektor addiere(Vektor v)
  {
    return Vektor(myX+v.myX, myY+v.myY);
  }
};
```

Frage

Welche Möglichkeiten der Parameterübergabe gibt es?

Antwort

Möglichkeiten der Parameterübergabe:
Werte-Parameter, Referenz-Parameter

Übersicht

Werte-Parameter	Referenz-Parameter

`funktion(double wert)`
`{`
`  wert = 0.0;`
`}`
	`funktion(double& wert)`
	`{`
	`  wert = 0.0;`
	`}`

Änderung des Wertes lokal begrenzt	Änderung wirkt sich außerhalb aus
⊖ Kopieren notwendig	⊕ (fast) keine Laufzeiteinbußen

Hinweise

In C++ gibt es hier keine Unterscheidung zwischen eingebauten Typen und selbstdefinierten Klassen.

	Java	C++
Parameterübergabe	festgelegt	benutzerdefiniert

Referenzparameter an unerwarteter Stelle können schwer aufzuspürende Fehler nach sich ziehen!

Wie ‚versehentliches Ändern' von Referenzen verhindern?

`const`

`const`-Deklarationen

Bezeichnung	Syntax	Semantik
Konstante Referenz	*Fkt*(*Typ* `const&` *Name*)	Referenz unveränderlich
Konstante Funktion	*Fkt*() `const`	Funktion ändert Klasse nicht

Hinweise

Konstante Funktionen für Aufrufe von konstanten Klasseninstanzen.

	Java	C++

Konstanten	`final double PI = 3.14159;`
	`final String STR = new String("Hello World!");`
		`const double PI(3.14159);` *oder*
		`double const PI(3.14159);`
		`std::string const STR("Hello World!");`

Beispiele

```
funktion(double const& wert)
{
  wert = 0.0; // nicht mehr möglich
}
```

Beispiele (Forts.)

```
#include <cmath>
#include "vektor.h"

double Vektor::norm() const
{
  return sqrt(myX*myX + myY*myY);
}
⋮
Vektor const meinVektor(1, 2);
std::cout << meinVektor.norm() << std::endl;
⋮
```

Kopierkonstruktor — Verwendung

- Zweck: Instanziierung einer Klasse durch Referenz auf andere Instanz dieser Klasse
- Konstruktor mit konstanter Referenz als Parameter
- Standard-Kopierkonstruktor kopiert alle Datenelemente

Beispiel

```
class Vektor {
  double myX, myY;

public:
  Vektor(double x, double y) : myX(x), myY(y) {};

  Vektor(Vektor const& v) : myX(v.myX), myY(v.myY) {};

  Vektor addiere(Vektor v);
};
```

Beispiel

Werte-Parameter	Referenz-Parameter

`double kontostand;`
`kontostand = 1000.0;`
`double einkommen(kontostand);`
`einkommen = 0.0;`
	`double kontostand;`
	`kontostand = 1000.0;`
	`double& einkommen(kontostand);`
	`einkommen = 0.0;`

`kontostand` ist 1000	`kontostand` ist 0
nur Änderung der Kopie	Änderung von Original und Referenz

Fragen

- Was ist unter dem **Gültigkeitsbereich (Scope)** einer Variablen zu verstehen?
- Wie wird in Java Speicherplatz für ein Objekt angefordert?

Antworten

- Gültigkeitsbereich:
  Codeabschnitt, in dem Variable definiert (‚ansprechbar‘) ist (kleinster umschließender ‚Block‘)
- Speicherplatz anfordern:
  `new`-Operator

Speicherfreigabe

	Java	C++
Speicher-freigabe	Garbage-Collection	sofort bei Verlassen des Gültigkeitsbereichs; davor evtl. Aufruf des Destruktors

Destruktor

```
Klasse::~Klasse() {...};
```

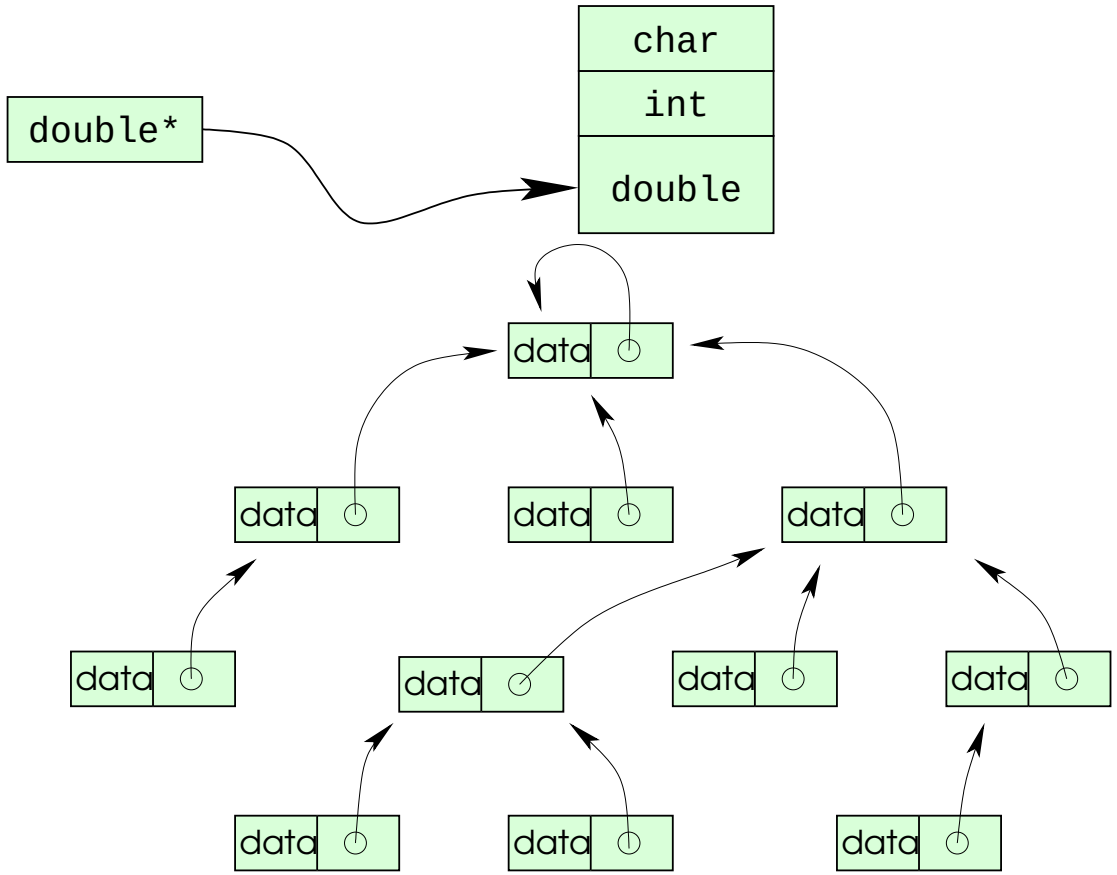
Globale Gültigkeit

Was tun, wenn Speicherbereich länger benötigt wird?
Zeiger/`new`/`delete` (s. u.)

	Java	C++
Instanziierung	`new`	‚lokal' (Scope), `new`

Zeiger — Verwendung

- verkettete Listen
- dynamische Arrays
- Bäume



Syntax

Bezeichnung	Syntax	Semantik
Zeigerdeklaration	*Typ* Zeiger*;	Zeigervariable *Zeiger* vom Typ *Typ*
Speicheradresse	&*Variable*	Adresse von *Variable*
Dereferenzierung	**Zeiger*	Inhalt von *Zeiger*

Achtung: Mehrfachbedeutungen von * und &!

Beispiel

```
int a = 5;
int b = 7;
int* p = &a;
int* q = &5;    // Fehler
int* r;
r = &b;
*p = 9;         // a ist nun 9
p = r;          // p zeigt nun auf b
*p = 8;         // b ist nun 8
```

Speicher anfordern

	Java	C++
Speicher anfordern	*Referenz* `= new` *Typ*`();`	*Zeiger* `= new` *Typ*`;`

Zeiger müssen im Gegensatz zu Referenzen nicht initialisiert werden.

Beispiel

```
#include <string>

int main()
{
  int* zeiger1 = new int;
  string* zeiger2 = new string;
  short int* zeiger3 = new short int(42);
  double const* zeiger4 = new double const(3.14159);
  return 0;
}
```

Hinweise

Bezeichnung	Syntax	Semantik
Zeiger auf Konstante	*Typ* `const*` *Zeiger*	*Zeiger* kann *Typ*-Variable nicht verändern
Konstanter Zeiger	*Typ*`*` `const` *Zeiger*	Zeiger nicht ‚umbiegbar‘

Deklarationen von rechts nach links lesen!

Konstanten (z. B. konstanter Zeiger) müssen initialisiert werden.

Speicher zurückgeben

	Java	C++
Speicher zurückgeben	Garbage-Collection	`delete` *Zeiger*;

Beispiel

```
int main()
{
  int* zeiger1 = new int;
  double* zeiger2 = new double;
      :
  delete zeiger1;
  double* zeiger3 = zeiger2;
  *zeiger3 = 1.41;
  delete zeiger3;
      :
  *zeiger2 = 1.73;   // nachfolgend undef. Verhalten!!
  return 0;
}
```

Felder

	Java	C++
Feld anlegen	*Typ*`[]` *Referenz* `=` `new` *Typ*`[`*Größe*`];`	*Typ*`*` *Zeiger* `=` `new` *Typ*`[`*Größe*`];`
Feld zurückgeben	Garbage-Collection	`delete[]` *Zeiger*`;`

Alternative: `std::vector`

Beispiel

```
int main()
{
  int* feld = new int[1000];
      :
  delete[] feld;
  return 0;
}
```

C-Strings

Syntax

```
char* Variable;
char const* Variable = "Zeichenkette";
```

Alternative: `std::string`

Dynamischer Vektor

```
class Vektor {
private:
  double* data;
  unsigned int groesse;

public:
  Vektor(unsigned int g);
  Vektor(Vektor const& v);
  ~Vektor();
    ⋮
};
```

Konstruktoren

```
Vektor::Vektor(unsigned int g)
{
  groesse = g;
  data = new double[g];
}
```

```
Vektor::Vektor(Vektor const& v)
{
  groesse = v.groesse;
  data = new double[groesse];
  for (unsigned int position=0;
       position<groesse; ++position)
    data[position] = v.data[position];
}
```

Destruktor

```
Vektor::~Vektor()
{
  delete[] data;
}
```

Fragen

- Welche Operatoren aus Java oder C++ kennen Sie bereits?
- In welche beiden Kategorien lassen sich diese einteilen?

Antworten

- Operatoren:
  –, ++, =, <<, [] etc.
- Kategorien:
  Unäre und binäre Operatoren

Syntax

Binärer Operator

```
Typ' Klasse::operatorOperatorname(Typ const& Argument2);
Objekt1 Operatorname Objekt2                          // Infix
Objekt1 Operatorname_1 Objekt2 Operatorname_2 // 'Zirkumfix'
```

Unärer Operator

```
Typ' Klasse::operatorOperatorname();
Operatorname Objekt    // Präfix
Objekt Operatorname    // Postfix
```

Zuweisungsoperator

```
Vektor& Vektor::operator=(Vektor const& v)
{
  delete[] data;
  groesse = v.groesse;
  data = new double[groesse];
  for (unsigned int position=0;
       position<groesse; ++position)
    data[position] = v.data[position];
  return *this;
}
```

Additionsoperator

```
Vektor Vektor::operator+(Vektor const& v) const
{
  if (groesse!=v.groesse)
    throw "falsche Laenge";
  Vektor ergebnis(groesse);
  for (unsigned int position=0;
       position<groesse; ++position)
    ergebnis.data[position] =
    data[position]+v.data[position];
  return ergebnis;
}
```

Zugriffsoperator

```
double& Vektor::operator[](unsigned int const index)
{
  return data[index];
}

double const&
Vektor::operator[](unsigned int const index) const
{
  return data[index];
}
```

Regel der Goldenen Drei

»**Brauchst du eine, brauchst du alle!**«
(Kopierkonstruktor, Destruktor, Zuweisungsoperator)

Beispiel

1. Kopierkonstruktor:
   `Vektor::Vektor(Vektor const& v)`
2. Destruktor:
   `Vektor::~Vektor()`
3. Zuweisungsoperator:
   `Vektor& Vektor::operator=(Vektor const& v)`

Beispiel — Definition

```
std::ostream&
operator<<(std::ostream& ausgabe, Vektor const& v)
{
  ausgabe << '(';
  for (unsigned int position=0;
       position<v.Groesse(); ++position)
    ausgabe << v[position] << ' ';
  ausgabe << ')';
  return ausgabe;
}
```

Anwendung

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

int main()
{
  Vektor v(7);
  for (int w=0; w<7; ++w)
    v[w] = w;
  std::cout << v << " ist der Inhalt von v" << std::endl;
  return 0;
}
```

Dateiausgabe

```
#include <fstream>
#include "vektor.h"

int main()
{
  Vektor v(7);
  for (int w=0; w<7; ++w)
    v[w] = w;
  std::ofstream ausgabe("ausgabe.txt");
  ausgabe << " v ist " << v << std::endl;
  return 0;
}
```

`std::ofstream` ist von `std::ostream` abgeleitet.

Fragen

- Wie lautet die Syntax in Java für das Ableiten einer Klasse?
- Was bedeutet **dynamisches Binden**?
- Was ist unter einem **Package** in Java zu verstehen?

Antworten

- Ableiten: `class` *Klasse* `extends` *Basisklasse* { ... }
- dynamisches Binden: Aufruf der entsprechenden Funktion abhängig vom **aktuellen Typ** eines Objekts
- Package: Sammlung von Klassen, Funktionen etc., die über Package-Namen angesprochen werden

Vererbung — Syntax

	Java	C++
Vererbung	`class` *Klasse* `extends` *Basisklasse* { ... }	`class` *Klasse* `: public` *Basisklasse* { ... };

Zugriffsregelung

Deklaration in Basisklasse	Zugriff
`private:`	nur Basisklasse
`protected:`	Basisklasse und abgeleitete Klassen
`public:`	‚global'

Beispiel

```
class Figur {
  void zeichne();
};

class Kreis : public Figur {
  void zeichne();
};
```

Dynamisches Binden

- Aufruf der entsprechenden Funktion abhängig vom aktuellen Typ einer Instanz
- Bezeichner `virtual` vor Funktion in Basisklasse (sonst statisches Binden der Basisklassenfunktion!)
- Funktion mit selber Signatur in abgeleiteter Klasse ebenfalls virtuell
- [⊖ Speicherplatz (Tabelle), Rechenzeit]

	Java	C++
dynamisches Binden	Standard	`virtual`

Beispiel

```
class Figur {
  virtual void zeichne();
};

class Kreis : public Figur {
  virtual void zeichne();
};
```

`virtual` bei der `zeichne`-Funktion von `Kreis` kann weggelassen werden.

Beispiel (Forts.)

```
Kreis meinKreis;
Figur* zeiger;
zeiger = &meinKreis;
(*zeiger).zeichne();
```

Zu `(*zeiger).zeichne()` alternative Syntax: `zeiger->zeichne()`.

Abstrakte Klassen

- Klasse ist abstrakt, sobald sie abstrakte Funktion hat
- Abstrakte Klasse kann nicht instanziiert werden
- Im Unterschied zu Java-Interfaces können Methoden definiert werden
- Von abstrakter Klasse kann abstrakte Klasse abgeleitet werden
- Abgeleitete Klasse ist nicht mehr abstrakt, wenn alle abstrakten Funktionen definiert sind

Syntax

```
virtual Funktionssignatur = 0;
```

	Java	C++
Schnittstelle, Abstraktion	`abstract`, Interface	abstrakte Klasse

Beispiel

```
class Figur {
  virtual void zeichne() = 0;
}

class Kreis : public Figur {
  void zeichne() { ... }
};
```

Namensräume — Syntax

	Java	C++
Namensraum	`package` *Paket*;	`namespace` *Namensraum* { ... }
Zugriff	*Paket*.*Element*	*Namensraum*::*Element*
Importieren	`import` *Paket*;	`using` *Namensraum*::*Element*; `using namespace` *Namensraum*;

Beispiel

```
namespace Bibliothek {
  void funktion();
}
⋮
int main()
{
  Bibliothek::funktion();
}
⋮
using Bibliothek::funktion;
int main()
{
  funktion();
}
```

Frage

Welche Funktion erfüllen **Casts** in Java?

Antwort

Sicherstellen des gewünschten aktuellen Typs eines Objekts.

Templates — Übersicht

- Parametrisierte Funktion/Klasse
- Abstraktion von konkreten Datentypen
- Funktion/Klasse kann für alle Typen instanziiert werden, die darin verwendete Funktionen bereitstellen

Syntax

Template-Funktion

```
template<typename Parameter> Funktion(...) {
  // Funktionsdefinition mit Parameter
}
Funktion(...);
```

Beispiel Template-Funktion

```
int    abs(int    i) { return i<0 ? -i : i; }

float  abs(float  f) { return f<0 ? -f : f; }

double abs(double d) { return d<0 ? -d : d; }
```

```
template<typename T>
T abs(T x) { return x<0 ? -x : x; }
⋮
double d = -42;
std::cout << abs(d) << std::endl;
```

Beispiel Template-Funktion (Forts.)

Gemeinsamkeiten

- Operator <
- Operator - (unär)
- Konstruktor, der 0 als Parameter akzeptiert
- Kopierkonstruktor

Hinweise

Mehrere Template-Parameter werden jeweils durch Komma getrennt.

Durch Angabe von = *Typ* kann Standardparameter festgelegt werden.

	Java	C++
Generizität	Cast, Java 1.5	Template

Syntax

Template-Klasse

```
template<typename Parameter> class Klasse {
  // Klasse mit Parameter
};
Klasse<Typ> Instanz;
```

Beispiel Template-Klasse

```
template<typename T> class Vektor {
  T* data;
  unsigned int groesse;

public:
  Vektor(unsigned int g);
  Vektor(Vektor<T> const& v);
  ~Vektor();
  Vektor<T>& operator=(Vektor<T> const& v);
  Vektor<T> operator+(Vektor<T> const& v) const;
  T const& operator[](unsigned int const index) const;
  T& operator[](unsigned int const index);
  unsigned int getGroesse() const { return groesse; }
};
```

Beispiel Template-Klasse (Forts.)

```
template<typename T>
Vektor<T>::Vektor(unsigned int g)
{
  groesse = g;
  data = new T[g];
}
⋮
template<typename T>
T& Vektor<T>::operator[](unsigned int const index)
{
  return data[index];
}
```

Methodendefinitionen verlangen ebenfalls `template<typename T>`.

Beispiel Template-Klasse (Forts.)

```
⋮
Vektor<double> meinVektor(7);
⋮
meinVektor[4] = 3.14;
⋮
```

Vor- und Nachteile

- $\oplus$ Definition einer Funktion/Klasse für mehrere Typen
- $\oplus$ Typsicherheit: Sicherstellen zur Kompilierzeit, dass verwendete Typen die notwendigen Funktionen bereitstellen und Typ einer Variablen der richtige ist (keine Casts notwendig)
- $\ominus$ Erhöhte Kompilierzeiten
- $\ominus$ Unhandliche Funktionsnamen
- $\ominus$ ‚Kryptische' Fehlermeldungen
- $\ominus$ Vergrößerung des Programms

Fragen

- Welche Datenstrukturen zählen Sie zu den grundlegendsten?
- Was ist unter einem **Iterator** zu verstehen?

Antworten

- Datenstrukturen:
  Array, Liste, Warteschlange, Stapel, Suchbaum etc.
- Iterator:
  Objekt zum Navigieren auf Datenstrukturen wie z. B. Listen

Die Standard Template Library

- Standardcontainer
- Iteratoren
- Grundlegende Funktionen und Algorithmen (z. B. Sortieren)
- ...

Vektor und Liste

Vektor

- Container: `std::vector<`*T*`>`
- Zugriffsoperator: `operator[]`
- ...

Liste

- Container: `std::list<`*T*`>`
- Anhängen (hinten): `push_back(...)`
- Entfernen (vorne): `pop_first()`
- ...

Beispiel Graphdatenstruktur

```
#include <vector>
#include <list>
using std::vector;
using std::list;

class Graph {
  struct Node {
    typedef Node* NeighbourType;
    typedef list<NeighbourType> NeighbourContainer;
    NeighbourContainer neighbours;
  };

  vector<Node> nodes;
};
```

Übersicht

- Iterator: *ContainerTyp*`::iterator`
- Konstanter Iterator: *ContainerTyp*`::const_iterator`
- Zugriffsoperator: `operator*`
- Weiterschalten: `operator++`
- Iterator, der auf erstes Element zeigt: *Container*`::begin()`
- Iterator, der hinter Ende zeigt: *Container*`::end()`

Beispiel Iterator

```
int Graph::Node::degree()
{
  int number = 0;
  for (NeighbourContainer::iterator it =
         neighbours.begin();
       it != neighbours.end(); ++it)
    ++number;
  return number;
}
```

Beispiel konstanter Iterator

```
int Graph::Node::degree() const
{
  int number = 0;
  for (NeighbourContainer::const_iterator it =
         neighbours.begin();
       it != neighbours.end(); ++it)
    ++number;
  return number;
}
```