Objektorientierte Modellierung

Prof. Dr. Roland Dietrich

7. Implementierung

Implementierung von Entwurfsmodellen in C++

Vergleich mit Java

Übersicht

1. Objektorientierte Softwareentwicklung ✓
2. Anforderungsanalyse mit UML ✓
   - Anwendungsfalldiagramme
3. Statische Modellierung mit UML ✓
   - Klassendiagramme
     Objekte und Klassen, Assoziationen, Vererbung
   - Paketdiagramme
4. Der Analyseprozess und Analysemuster ✓
5. Dynamische Modellierung mit UML ✓
   - Interaktionsdiagramme (Sequenz- und Kollaborationsdiagramme)
   - Aktivitätsdiagramme
   - Zustandsautomaten
6. Entwurf mit UML ✓
7. Implementierung in C++

Klassen und Objekte

- Klasse in C++
  - Trennung von Spezifikation und Implementierung

```
//zaehler.h   Definitionsatei(→ Klassendefinition)
class Counter
{private: unsigned value;
 public:  void init();
          void inkrement();
          unsigned getValue(); };
```

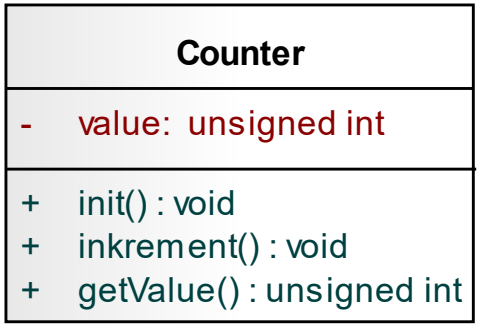
Counter
- value: unsigned int
+ init() : void + inkrement() : void + getValue() : unsigned int

```
//zaehler.cpp  Implementierungsdatei
//             (→ Definition der Methoden)
#include „zaehler.h" // Einbinden der Header-Datei

void Counter::inkrement() { value++;}
void Counter::init() { value = 0;}
unsigned Counter::getValue() { return value ;}
```

Klassen und Objekte

- Klasse in Java
  - Keine Trennung von Spezifikation und Implementierung

```
class Counter
{ private int value;
  public void inkrement()  { ... }
  public void init()  { ... }
  public int getValue()  { ... }
}
```



  - Klasse ist nur innerhalb ihres Pakets bekannt
  - Sonst `public` deklarieren

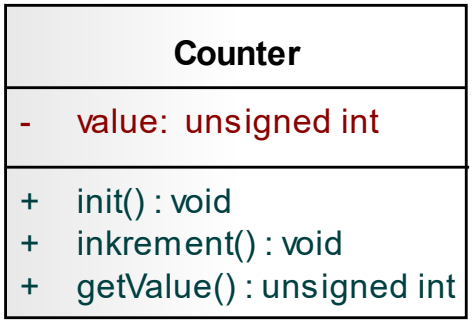
```
public class Counter {}
```

Klassen und Objekte

- Objekt in C++
  - Objekte werden wie normale Variablen behandelt
    - Statisch deklariert, automatisch verwaltet (***stack***)
    - Dynamisch erzeugt, vom Programmierer verwaltet (***heap***)
  - Dynamisch erzeugter Speicherplatz von Objekten muss explizit freigegeben werden
  - Jedes Objekt wird über seine Adresse eindeutig identifiziert
- Objekt in Java
  - Objekte können nur dynamisch (als *heap*-Variable) erzeugt werden
  - Speicherplatz wird durch *garbage collector* automatisch freigegeben

Klassen und Objekte

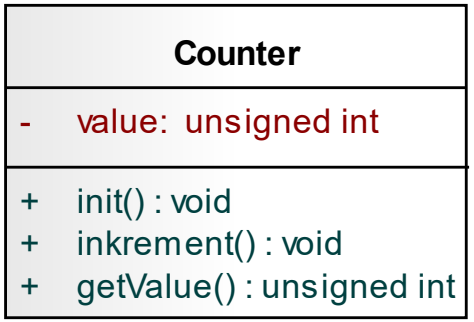
- Objekt in C++
  - Beispiel




```
Counter einZaehler;              // stack-Variable
einZaehler.init(); einZaehler.inkrement();

Counter* pZaehler;               // pointer-Variable
pZaehler = new Counter;          // heap-Variable
pZaehler -> init(); pZaehler -> inkrement();
…
delete pZaehler ;
```

Klassen und Objekte

- Objekt in Java
  - Beispiel




```
Counter einZaehler;
  // Objektvariable: Nur  Objektreferenz! -
  // - Objekt existiert (noch) nicht

einZaehler = new Counter();
          // Objekt der Klasse Counter erzeugen,
          // auf das einZaehler verweist
einZaehler.init(); einZaehler.inkrement(); ...
```

Klassen und Objekte

- Attribute in C++
  - Häufig benutzte Bezeichnung: *Member-Variable*
  - Sichtbarkeitsbereiche:
    - *public:* sichtbar für alle
    - *protected:* sichtbar innerhalb der Klasse und ihrer Unterklassen
    - *private:* sichtbar innerhalb der Klasse
    - Voreinstellung: *private*
  - Initialisierung der Attribute mittels Konstruktoren (s.u.)
    - Keine automatischen Default-Werte!
    - Parameterübergabe an den Konstruktor
      - Stack-Objekt: `Klasse obj(parameter)`
      - Heap-Objekt: `Klasse *pObj = new Klasse(parameter)`
  - Klassen-Attribute: ***static***-Attribute

```
static VariablenTyp VariablenName;
```

    - Initialisierung in der Implementierungsdatei (.cpp)

```
VariablenTyp Klasse::VariablenName = initialWert;
```

Klassen und Objekte

- Konstruktoren in C++
  - Initialisierung von Attribut-Werten
    - Insbesondere: Reservieren von Speicherplatz für andere Objekte
  - Wird aktiviert zu Beginn der Lebensdauer des Objekts
    - automatisch (*Stack-Objekte*) oder durch **`new`** (*Heap-Objekte*)
  - Mehrere Konstruktoren pro Klasse möglich
    - Unterschiedliche Parameterliste *(overloading)*
  - Syntax: ***`Klassenname(Parameterliste)`***
  - Default: Parameterloser Konstruktor
- Destruktor in C++
  - Wird aktiviert am Ende der Lebensdauer des Objekts
    - automatisch (*Stack*) oder durch **`delete`** (*Heap*)
  - Nur ein Destruktor für jede Klasse
  - Freigabe von anderen Objekten
    - spätester Zeitpunkt für Objekte, die im Konstruktor erzeugt wurden
  - Syntax: ***`~Klassenname()`***

Klassen und Objekte

- ***readOnly***-Operation in C++
  - Deklaration mit **`const`**
    - Keine Veränderungen der Attributwerte erlaubt (z.B. get-Methode)

```
bool getAttribute() const;
```

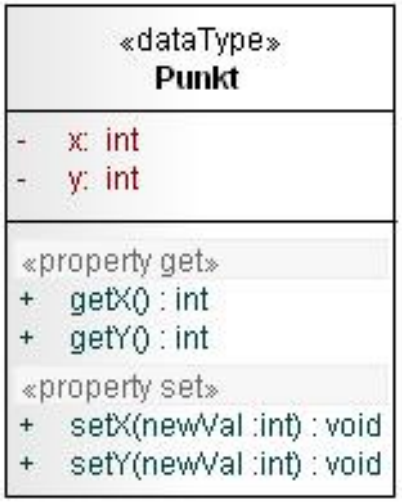
- Klassenoperation in C++
  - Deklaration mit **`static`**
  - Innerhalb einer **`static`**-Methode nur Zugriff auf Elemente, die ebenfalls **`static`** sind
  - Beim Aufruf einer Klassenoperation Angabe des Namens der Klasse mit "Gültigkeitsoperator" **`::`**

```
class Klasse {...
   static unsigned s_methode();
   ...
}

//Aufruf der Klassenoperation
i = Klasse::s_methode ();
```

Klassen und Objekte

- Beispiel (C++)

```
class Kreis {
  unsigned int radius;
  Punkt mittelpunkt;
  static unsigned anzahl;
public:
  Kreis();
  Kreis(unsigned r,int x ,int y);
  static unsigned getAnzahl();
...
};
```

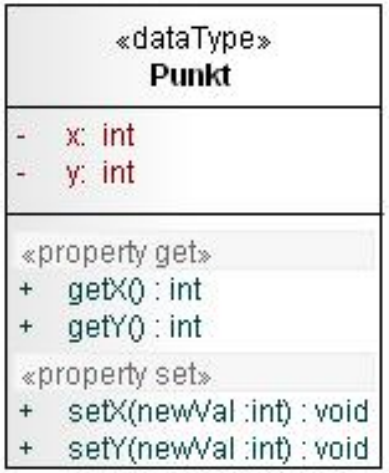



```
Kreis::Kreis() //Konstruktor 1
{  anzahl++; }

//Konstruktor 2
Kreis::Kreis (unsigned r,
              int x,int y)
{  radius = r; anzahl++;
   mittelpunkt.setX(x);
   mittelpunkt.setY(y);
}

// Implementierung Klassenoperation
unsigned Kreis::getAnzahl()
{ return anzahl; }

// Initialisierung Klassenattribut
unsigned int Kreis::anzahl = 0;

// Verwendung der Klasse Kreis
Kreis einKreis;
Kreis andrerKreis(7,-1,1);
Kreis * pKreis = new Kreis(6,2,2);
unsigned i = Kreis::getAnzahl();
      // i = 3
```

Klassen und Objekte

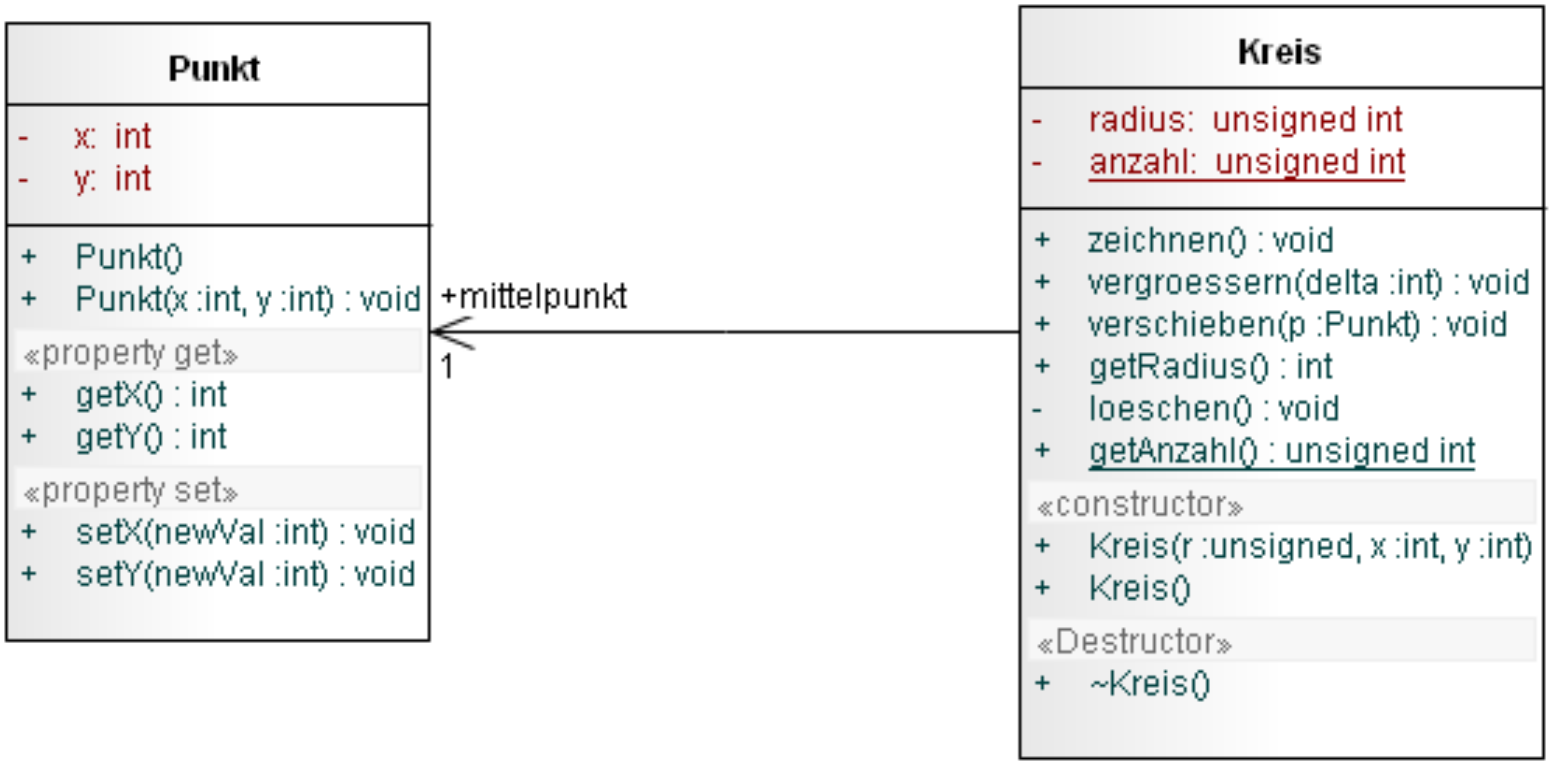
- Beispiel (Java)

```
class Kreis
{  private int radius;
   Punkt mittelpunkt;
   static int anzahl = 0; // Klassenattribut

   public Kreis()  // Konstruktor 1
   { mittelpunkt = new Punkt; anzahl ++}

   // Konstruktor 2
   public Kreis (int r, int x, int y)
   { mittelpunkt = new Punkt;
     anzahl++; radius = r;
     mittelpunkt.setX(x); mittelpunkt.setY(y);}

   static int getAnzahl() // Klassenoperation
   { return anzahl;}      ...  }

// Verwendung der Klasse Kreis
Kreis einKreis = new Kreis();
Kreis andrerKreis = new Kreis(12,3,5);
int i = Kreis.getAnzahl(); // i = 2
```



Klassen und Objekte

- Beispiel mit Destruktor (C++)
  - Der **Konstruktor** der Klasse Kreis muss einen Mittelpunkt (auf dem Heap) erzeugen
    - So ist es in Java immer!
  - Der **Destruktor** muss den Mittelpunkt wieder freigeben



Klassen und Objekte

- Beispiel mit Destruktor (C++)

```
class Kreis {
  unsigned int radius;
  Punkt * mittelpunkt;
  static unsigned anzahl;
public:
  Kreis();
  Kreis(unsigned r,int x ,int y);
  ~Kreis()
...
};
```

```
class Punkt {
  Punkt() { }; // Koordinaten
        // bleiben undefiniert
  Punkt(int px, int py)
  { x=px; y=py; }
  ...
};
```

```
Kreis::Kreis() //Konstruktor 1
{  anzahl++;
   mittelpunkt = new Punkt();
}

//Konstruktor 2
Kreis::Kreis (unsigned r,
              int x,int y)
{  mittelpunkt = new Punkt(x,y);
   radius = r; anzahl++; }

// Destruktor
Kreis::~Kreis()
{ delete mittelpunkt; anzahl--; }
```

```
// Verwendung der Klasse Kreis
Kreis einKreis;
Kreis * pKreis = new Kreis(6,2,2);
unsigned i = Kreis::getAnzahl();
                              // i = 2
delete pKreis;
i = Kreis.getAnzahl();        // i = 1
```

C++: Dateistruktur und Übersetzung

- *Wie soll ein Quellprogramm mit einer oder mehreren Klassen und einem Hauptprogramm, auf Dateien verteilt werden?*
  - Jede Klassen-Definition `KlasseX` wird in eine eigene Definitionsdatei `KlasseX.h` geschrieben.
  - Die zugehörigen Methoden-Definitionen in eine Datei `KlasseX.cpp`
    - Die Klassendefinition wird mit **`#include`** `"KlasseX.h"` eingebunden.
  - Ein (Haupt-)Programm, welches KlasseX nutzt, steht in einer eigenen Datei `XY.cpp`
    - Die Klassendefinition wird mit **`#include`** `"KlasseX.h"` eingebunden.
  - Übersetzt werden nur die .cpp Dateien
    - Dabei ersetzt zunächst der ***C++-Präprozessor*** alle **`#include`**-Anweisungen durch den Inhalt der includierten Dateien
    - Der **Übersetzter (Compiler)** erzeugt Objektcode-Dateien (`KlasseX.o`, `XY.o`)
  - Diese werden durch den **Binder (Linker)** in ein ausführbares Programm übersetzt: `XY.exe`
    - Wenn **`XY.cpp`** eine main-Funktion enthält

C++: Dateistruktur und Übersetzung

- Integrierte Entwicklungsumgebungen (IDEs):
  - Die Schritte *Präprozessor* – *Übersetzen* – Binden laufen in der Regel automatisch ab, wenn das Kommando "Ausführen" gewählt wird
  - Insbesondere werden, wenn man den Quellcode geändert hat, automatisch alle erforderlichen Nachübersetzungen durchgeführt
- Problem: Mehrfaches einbinden derselben Deklarationsdatei
  - kann verhindert werden durch entsprechende Präprozessordirektiven
    - „**bedingte Übersetzung**“:

#ifndef** Konstante **#define** Konstante **… #endif

```
// Datei KlasseX.h
#ifndef KLASSEX_H
#define KLASSEX_H
class KlasseX {
...
} ;
#endif
```

```
// DATEI KlasseX.cpp
#include "KlasseX.h"
KlasseX::KlasseX(...)
{... } ; // Konstruktor
void KlasseX::f( ... )
{ ... } ; // eine Methode
...
```

```
// DATEI XY.cpp (Hauptptogramm)
#include "KlasseX.h"
void main()
{
  KlasseX X(...); // Objetk-Def.
  X.f( ... );  // Methodenaufruf
  ...
}
```

C++ - Standardbibliothek

- Enthält zahlreiche Klassen, die Standard-Aufgaben des Programmierens lösen, u.a.
  - Zeichenketten: `string`
    → `#include <string>`
  - Standard-Ein-/Ausgabe: Strom-Objekte `cin, cout`
    → `#include <iostream>`
  - Generische Containerklassen: `set<T>, list<T>, vector<T>`
    → `#include <set>`
    → `#include <list>`
    → `#include <vector>`
  - Alle Standard-C-Deklarationsdateien *name*`.h` in der Form `c`*name*
    → `#include <cstring>  // statt <string.h>`
    → `#include <cstdio>  // statt <stdio.h>`
- Alle Elemente der Standard-Bibliothek gehören zum Namensraum **`std`**
  - Programme, die Elemente aus der Standard-Bibliothek nutzen, müssen eine entsprechende **`using`**-Anweisung enthalten:

    ```
    using namespace std;
    ```

  - Alternativ: qualifizierte Namen verwenden:

    ```
    std::string, std::cin, std::list<T>...
    ```

C++: Standard-Ein-/Ausgabeströme

- Die **`iostream`**-Bibliothek
  - definiert *Klassen* und *Objekte* mit *Operatoren* und *Methoden* zur typisierten Ein- und Ausgabe von Tastatur bzw. auf Bildschirm.
  - Ausgabe in Ströme mit dem Operator „<<“

    Strom << Ausdruck
  - Eingabe aus Strömen mit dem Operator „>>“

    Strom >> Variable
  - ***Standard-Stromobjekte***:
    - **`cout:`** Ausgabe auf dem Bildschirm (Objekt der Klasse **`ostream`**)
    - **`cin:`** Eingabe über Tastatur (Objekt der Klasse **`istream`**)
  - ***Bildschirmausgabe:***

    ```
    cout << Ausdruck1 << Ausdruck2 << ... ;
    ```

    → der Wert der Ausdrücke wird nacheinander am Bildschirm ausgegeben.
  - ***Tastatureingabe:***

    ```
    cin >> Variable1 >> Variable2 >> ... ;
    ```

    → über Tastatur können nacheinander Werte eingegeben werden, die mit den Typen der Variablen kompatibel sind;
    diese Werte werden den Variablen zugewiesen.

C++: Standard-Ein-/Ausgabeströme

- Beispiel:

```
int x,y;
cout << "Bitte zwei Zahlen eingeben: " ;
cin >> x >> y;
cout << "Die Summe der Zahlen ist: " << x+y << endl;
```

- Anmerkungen
  - Ausgabe der Konstanten `endl` bewirkt einen Zeilenwechsel
    - Alternative: Ausgabe des Zeichens `'\n'`
  - Bei der Ausgabe werden nicht-Text-Ausgaben (z.B. `int`-, `float`-Ausdrücke) in entsprechende Zeichenketten umgewandelt
  - Bei der Eingabe
    - werden stets die Zeichen bis zum nächsten Trennzeichen (Leerzeichen, Tabulator, Zeilenwechsel) verarbeitet;
    - müssen die eingegeben Texte in den Typ der entsprechenden Variablen wandelbar sein.
  - Jede Eingabe muss mit dem "Eingabe-Zeichen" ("Return", ↵) abschließen
    - Dieses Zeichen verbleibt im Eingabestrom, bis es auch verarbeitet wurde.
  - Vor der Eingabe von Zahlen werden Trennzeichen überlesen.

C++: Standard-Ein-/Ausgabeströme

- Eingabe ganzer Zeilen: `istream::getline(buf,n)`
  - Es werden Solange Zeichen aus dem Strom gelesen und in das `char`-Feld `buf` übertragen bis
    - entweder das Zeilenende-Zeichen (`'\n'`) gelesen wurde
    - oder `n` Zeichen gelesen wurden.
    - Es wird automatisch am Ende ein Nullzeichen (`'\0'`) angefügt. (D.h. `buf` ist eine nullterminierte Zeichenkette nach C-Standard)
  - Beispiel:

```
char buffer[128];
cout << "Bitte Vornamen und Nachnamen eingeben: ";
cin.getline(buffer,128);
```

  - Vorteil:
    - Das Zeilenende-Zeichen verbleibt nicht in der Eingabe
    - Leerzeichen wirken nicht als Trennzeichen
- Überlesen von Zeichen: `istream::ignore(n,delim)`
  - Es werden n Zeichen aus dem Eingabstrom entfernt ("überlesen"), allerdings höchstens bis das Zeichen (`char`) `delim` gefunden wird.
  - Beispiel:

```
int x; cin >> x; cin.ignore(129,'\n');
```

C++: Zeiger

- Siehe Präsentation Zeiger.ppt

- **Beispiel (C++):** Klasse ***MyString*** zur effektiven Verwendung von Zeichenketten
  - Klassendefiniton

```
// DATEI: MyString.h
class MyString {
   char * str ; // Zeiger auf eine Zeichenkette (char-Feld)
   int len ;    // Länge der Zeichenkette (Zeichen ohne Nullzeichen)
public:
   // Konstruktoren und Destruktor
   MyString(); // Konstruktor ohne Parameter (erzeugt "Leerstring")
   MyString(char * s) // String mit bestimmter Zeichenkette
   ~MyString();       // Destruktor
   // Methoden
   void write();   // Ausgabe auf Standardausgabe
   void read();    // Eingabe von Standardeingabe
   void copy(MyString * s) ; // Kopieren eines anderen String-Objekts
};
```

Klassen und Objekte

- **Beispiel (C++):** Klasse ***MyString*** – Methodendefinitionen

```
// Datei: MyString.cpp
#include <cstring>
#include <iostream>
#include "MyString.h"

MyString::MyString()
{ len=0;
  str=NULL;
}

MyString::MyString(char * s)
{ len=strlen(s);
  str = new char[len+1];
  strcpy(str,s);
}


MyString::~MyString()
{ if (str != NULL)
  delete[] str;
}
```

```
void MyString::write()
{ cout << str; }

void MyString::read()
{ char s[128]; // lokales char-Feld
  cin.getline(s,128);
  if (len>0) // alten String
  { delete[] str; } // löschen
  len=strlen(s);
  str = new char[len+1];
  strcpy(str,s);
}


void MyString::copy(MyString * s)
{ if (len>0) // alten String
  { delete[] str; } // löschen
  len = s->len;
  str = new char[len+1]; }
  strcpy(str,s->str);
}
```

- **Beispiel (C++):** Klasse ***MyString***
  - Objekte erzeugen und Methoden aufrufen

```
MyString s1; // Parameterloser Konstruktor: Leerstring
s1.read();   // Eingabe über Tastatur
MyString s2("Hallo!");
s2.write(); // Ausgabe "Hallo!" auf dem Bildschirm

s1 = s2;   // Möglich, aber gefährlich: "bitweises"
           // übertragen des Inhalts von s2 nach s1!
           // - Es gibt nur ein char-Feld im Speicher!
           // Wenn s2 gelöscht wird, ist der Inhalt von s1
           // Mit zerstört und umgekehrt!

// Besser:
s1.copy(&s2); // Übergabe der Adresse von s2!

// Oder so:                  // Pointer auf MyString-Objekt (Heap)
MyString * pS4 = new MyString("Hallo!") ;
s1.copy(pS4);
```

Klassen und Objekte

- Parameterkonzept in C++
  - Eingabeparameter
    - Wertübergabe ***(call by value)*** `void f(…,Typ p,…);`
    - Referenzübergabe ***(call by reference)*** `void f(…,Typ &p,…);`
  - Ausgabeparameter
    - Referenzparameter in der Parameterliste
    - Zeiger in der Parameterliste
    - Rückgabewert (***return***)
- Parameterkonzept in Java
  - **Einfach Typen**: Wertübergabe (*call by value)*, Ausgabe nur über den Rückgabewert möglich
    - Numerische Typen
    - boolean
  - **Referenztypen**: *call by reference*
    - Klassen
    - Felder
    - Schnittstellen

C++: Parameterübergabe

Wertübergabe (C/C++)	*Wertübergabe eines Zeigers (C/C++)*	*Referenzübergabe (nur C++)*

`void inkrement(int i)`
`{`
`  i=i+1 ;`
`}`
 	
`int main ()`
`{`
`  int x = 1 ;`
`  inkrement(x) ;`
`  cout << x ;`
`  return 0 ;`
`}`
`// Ausgabe: 1`
	`void inkrement(int *i)`
	`{`
	`  *i=*i+1 ;`
	`  //i dereferenziert!`
	`}`
	
	`int main ()`
	`{`
	`  int x = 1 ;`
	`  inkrement(&x) ;`
	`  cout << x ;`
	`  return 0 ;`
	`}`
	`// Ausgabe: 2`
		`void inkrement(int  &i)`
		`{`
		`  i=i+1 ;`
		`  // Dereferenzieren`
		`} // nicht erforderlich!`
		
		`int main ()`
		`{`
		`  int x = 1 ;`
		`  inkrement(x) ;`
		`  cout << x ;`
		`  return 0 ;`
		`}`
		`// Ausgabe: 2`

C++: Parameterübergabe

- Referenzübergabe:
  - Bei Aufruf einer Funktion mit Referenzparametern müssen die entsprechenden ***aktuellen Parameter*** **(**Argumente) stets ***Variablen*** sein.
  - Beim Funktionsaufruf werden formale Referenzparameter und aktueller Parameter miteinander ***identifiziert*** - sie bezeichnen dasselbe Datenobjekt.
  - Jede ***Veränderung*** eines formalen Referenzparameters im Rumpf einer Funktionsdefinition ***wirkt in gleichem Maße*** auf den aktuellen Parameter.
  - Bei der Übergabe von komplexen Datenobjekten (Felder, Strukturen, Objekte) ist Referenzübergabe ***effizienter***, da nur eine Adresse und nicht das komplette Datenobjekt übergeben wird.
  - Bei übergaben von Objekten, die Pointer auf Heap-Objekte enthalten (z.B. Klasse `MyString`), ist stets Referenzübergabe zu verwenden
    - **Sonst zerstört der Destruktor evtl. das übergebene Argument!**
      - Beispiel: Übungsblatt 3, Aufgabe 3-2

C++: Kopierkonstruktor

- Kopierkonstruktor (***copy constructor***)
  - Konstruktor, der ein Objekt derselben Klasse als Referenzparameter hat
    - Beim Aufruf wird ein existierendes Objekt übergeben
    - Das neue Objekt wird mit den Werten des übergebenen Arguments gefüllt
  - Beispiel: Kopierkonstruktor für die Klasse MyString

```
class MyString {
 char * str ;
 int len ;
 MyString( MyString& );
 ...
}

MyString::MyString(MyString &s)
{
  len = s.len ;
  str = new char[len+1];
  strcpy(str,s.str);
}
```

```
// Anwendung des
// Kopierkonstruktors

MyString s1("Hallo") ;

MyString s2(s1) ;
```