

Praxis der Programmierung

**Strings, Aufteilung der Quelltexte,
Vererbung, Polymorphie**

**Institut für Informatik und Computational Science
Universität Potsdam**

Henning Bordihn

Strings

Arten von Zeichenketten

- **C-Zeichenketten**
 - char-Array (*also* Pointer auf char)
 - null-terminiert (letztes Element ist '`\0`')
 - String-Konstanten der Form "Hallo Text!" werden als **C-Strings** angelegt
- **Klasse `string`** der Standardbibliothek
 - einfacherer Umgang mit Strings
(keine Überläufe der Arraygröße, keine Nullterminierung)
 - Objekt fordert nötigen Speicherplatz selbst an
 - viele komfortable Methoden vordefiniert

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

string str;    // Aufruf des Standardkonstruktors
```

Die Klasse string

- Konstruktoren

- Standardkonstruktor (ermöglicht spätere Initialisierung)
- Initialisierungskonstruktor mit einer String-Konstante als Parameter
- `string(int n, char c)` erzeugt String aus n Zeichen c

```
string name;  
name = "Programmer";  
string ort("Bithausen");  
string trennlinie(60, '-');
```

Die Klasse string

- Konstruktoren

- Standardkonstruktor (ermöglicht spätere Initialisierung)
- Initialisierungskonstruktor mit einer String-Konstante als Parameter
- `string(int n, char c)` erzeugt String aus n Zeichen c

```
string name;  
name = "Programmer";  
string ort("Bithausen");  
string trennlinie(60, '-');
```

- Kompatibilität zu C-Strings

- C-String → string: per Zuweisung oder mit Initialisierungskonstruktor
- string → C-String: Methode `c_str()` der Klasse string
 - ↪ konstanter char-Pointer auf die Zeichenkette (*unveränderlich!*)
 - ↪ mit `strncpy()` (<cstrin>) in ein char-Array überführen

Beispiel Konvertieren in einen C-String

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

int main() {
    string ort("Potsdam");
    const char *cStringPointer = ort.c_str();

    char veraenderlich[60];
    strncpy(veraenderlich, cStringPointer, 60);
    veraenderlich[0] = 'p';    // ohne Probleme
    ort[1] = '0';              // betrifft auch cStringPointer
    cStringPointer[1] = 'u'    // Compilerfehler
}
```

Einige Methoden der Klasse string (1)

- `append(str)` fügt `str` hinten an
Alternative: `s1 = s1 + s2;` // auch `s1 += s2;` ist `s1.append(s2);`
- `at(i)` liefert Zeichen an Position `i`
Alternative: `str[i];` // ist `str.at(i);`
- `clear()` löscht alle Zeichen des Strings
- `insert(p, str)` fügt `str` an Position `p` ein
- `erase(p, n)` entfernt ab Position `p` `n` Zeichen
- `replace(p, n, str)` ersetzt ab Position `p` `n` Zeichen durch den String `str`

Einige Methoden der Klasse `string` (2)

- `length()` und `size()` liefern die Länge des Strings
- `substr(p,n)` liefert die Teilzeichenkette ab Position `p` der Länge `n`
- `compare(str)` vergleicht den String mit `str`
- `empty()` gibt `true` zurück, wenn der String leer ist

weitere Methoden auf

<http://www.cplusplus.com/reference/>

<http://en.cppreference.com/w/cpp>

↪ umfassende Dokumentationen der Klassen der Standardbibliothek

String-Iteratoren

- string-Methoden `begin()` und `end()` liefern je ein Objekt vom Typ `string::iterator`
- Iteratoren sind Pointer, die `string`-Objekte durchlaufen (Pointerarithmetik!)
- `begin()`-Pointer zeigt auf erstes Zeichen des Strings
- `end()`-Pointer zeigt hinter das letzte gültige Zeichen des Strings
- Vergleich mit dem `end()`-Pointer zeigt an, dass das Stringende erreicht wurde
- *“symmetrisch”*: `string::reverse_iterator`, `rbegin()`, `rend()`
- Gültigkeit nach Veränderung des Strings nicht mehr garantiert
 \rightsquigarrow immer `begin()` unmittelbar vor Gebrauch des Iterators!

String-Iteratoren – Beispiel

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

int main() {
    string s = "Testlauf";
    string::iterator i;
    for (i = s.begin(); i != s.end(); i++)
        cout << *i;
    cout << endl;
}
```

String-Vergleiche

ohne Bibliotheksfunktionen möglich:

==	gleich
!=	ungleich
<	in lexikalischer Reihenfolge vorher
>	in lexikalischer Reihenfolge hinterher
<=	in lexikalischer Reihenfolge vorher oder gleich
>=	in lexikalischer Reihenfolge hinterher oder gleich

Umwandlung von oder in Zahlen

- Bibliothek `sstream` stellt Klassen `istringstream` und `ostringstream` bereit
- diese Klassen erlauben, Daten mit `>>` und `<<` in bzw. aus Strings umzuleiten
- `ostringstream` stellt Methode `str()` zur Umwandlung des Streams in ein Exemplar der Klasse `string` zur Verfügung

```
int zahl = 0;  
string s = "123";  
istringstream in(s);  
in >> zahl;    // zahl = 123
```

```
zahl = 3778;  
ostringstream os;  
os << zahl;  
s = os.str();   // s = "3778"
```

Aufteilung der Quelltexte

Leitgedanken bei der Quelltext-Aufteilung

- je Klassendefinition eine Datei
- Compiler-Aufruf zum Erzeugen der Objektdaten (.o) für jede beteiligte Datei einzeln möglich: `g++ -Wall -c datei.cpp`
- Linker wird aktiv bei Aufruf von `g++` ohne Option oder mit Option `-o`
 - alle beteiligten Dateien als Argumente übergeben (.cpp oder .o, .h)
 - genau eine beteiligte Datei hat eine `main`-Funktion
 - es entsteht eine ausführbare Datei
- weitere Aufteilung einer Klasse in
 - eine Header-Datei .h mit der Definition der Klasse, in der Datenelemente und Methoden nur deklariert werden
 - eine .cpp-Datei mit den Implementierungen der Methoden

Beispiel zur Quelltextaufteilung

```
// date.h

class Date {
    int day, month, year;
    ...
};
```

```
// date.cpp
#include "date.h"
...
```

```
// highscore.h
#include "date.h"
class Highscore {
    int score;
    Date d;
    ...
};
```

```
// highscore.cpp
#include "date.h"
#include "highscore.h"
...
```

Beispiel zur Quelltextaufteilung

```
// date.h

class Date {
    int day, month, year;
    ...
};
```

```
// date.cpp
#include "date.h"
...
```

```
// highscore.h
#include "date.h"
class Highscore {
    int score;
    Date d;
    ...
};
```

```
// highscore.cpp
#include "date.h"
#include "highscore.h"
...
```

```
> g++ date.cpp -c          // liefert date.o
> g++ highscore.cpp -c     // Compilerfehler! (WARUM???)
```


Vermeiden von Doppel-Definitionen mit dem Präprozessor

```
#ifndef _DATE_h      // nur, falls die Variable _DATE_h nicht definiert ist
#define _DATE_h      // wird diese Konstante
class Date {         // und die Klasse Date definiert
    ...
};
#endif              // sonst wird alles bis hier uebersprungen
```

Analog:

```
#ifndef _HIGHSCORE_h
#define _HIGHSCORE_h
class Highscore { ... }
#endif
```

Beispiel zur Quelltextaufteilung

```
// date.h, implementiert in date.cpp
#ifndef _DATE_h
#define _DATE_h

class Date {
    int day, month, year;
    ...
};
#endif
```

```
// highscore.h, implementiert in highscore.cpp
#include "date.h"
#ifndef _HIGHSCORE_h
#define _HIGHSCORE_h
class Highscore {
    int score;
    Date d;
    ...
};
#endif
```

Applikation useHighscore.cpp (mit main) nutzt Highscore und Date:

```
// useHighscore.cpp
#include "date.h"
#include "highscore.h"
...
```

```
> g++ date.cpp highscore.cpp useHighscore.cpp -o useHighscore
```

Klassenvariablen und -methoden

Klassenvariablen und -methoden

- werden mit dem Schlüsselwort `static` vereinbart
- sind der Klasse, nicht den Objekten zugeordnet; d.h.
 - **Klassenmethoden** können auch aufgerufen werden, ohne dass ein Objekt der Klasse erzeugt wurde
 - **Klassenvariablen** sind (mit ihren Werten) allen Exemplaren der Klasse gemeinsam, sind also *klassenglobal*
 - Klassenvariablen werden vor dem ersten Exemplar der Klasse angelegt, müssen also wie globale Variablen definiert werden
- werden mit dem Klassennamen (statt mit dem Objektnamen) qualifiziert
- Klassenmethoden dürfen direkt nur auf Datenelemente und Methoden zugreifen, die ebenfalls `static` sind!

Beispiel Klassenvariablen

```
class Static_Beispiel {
    int n;
    static int exCounter;    // wird im Speicher angelegt,
public:                    // bevor das erste Objekt der Klasse erzeugt wird
    Static_Beispiel(int n) {
        this->n = n;
        exCounter++;
    }
    ~Static_Beispiel() {
        exCounter--;
    }
    void increase() { n++; }
};

int Static_Beispiel::exCounter = 0;
```

Vererbung

Achsenparallele Quadrate in der Ebene (naiv)

```
class Square {  
    // Datenelemente  
    int a;      // Radius  
    int x, y;   // Koordinaten  
public:  
    Square(int x, int y, int a);  
    int getX();  
    int getY();  
    int getSize();  
    void moveTo(int x, int y);  
    void moveRel(int dx, int dy);  
    void reSize(int a);  
    double area(); // Flaeche berechnen  
};
```

Kreise in der Ebene (naiv)

```
class Circle {  
    // Datenelemente  
    int a;      // Radius  
    int x, y;   // Koordinaten  
public:  
    Circle(int x, int y, int a);  
    int getX();  
    int getY();  
    int getSize();  
    void moveTo(int x, int y);  
    void moveRel(int dx, int dy);  
    void reSize(int a);  
    double area(); // Flaeche berechnen  
};
```


WRITE ONCE!

Prinzipien bei der Vererbung

- Definition von Klassen (*Unterklassen*), die von einer anderen Klasse (*Oberklasse*) abgeleitet sind
- Vererbung aller Datenelemente und Methoden von der Oberklasse an jede ihrer Unterklassen
- Erweiterung der Oberklasse durch neue Datenelemente und Methoden möglich
verdecken gleichnamige Elemente der Oberklasse
- Re-Definition (**Überschreiben**) geerbter Methoden (mit derselben Signatur!) können die Funktion der Oberklasse an geeigneter Stelle aufrufen:
`Oberklasse::verdeckteFunktion()`
- hierarchische Vererbung (Kinder, Enkel, Urenkel, ...)
- Mehrfachvererbung (Erben von mehreren Oberklassen gleichzeitig)

Definition einer Unterklasse

```
class Unterklasse : public Oberklasse {  
  
    // neue Datenelemente  
  
    // ggf. Konstruktoren  
    Unterklasse(...) { // ruft als erstes Oberklasse() auf  
        ...  
    }  
  
    // ggf. Destruktoren  
    ~Unterklasse(...) { // ruft als letztes ~Oberklasse() auf  
        ...  
    }  
  
    // neue Methoden  
};
```

Vererbung am Beispiel

```
class Auto {
public:
    void hupen() {
        cout << "huup" << endl;
    }
};

class Pkw : public Auto {
public:
    void airCondition() {
        cout << "fauch" << endl;
    }
};
```

```
int main() {
    Auto karre;
    Pkw pkw;
    karre.hupen() // "huup"
    pkw.hupen();  // "huup"
    karre.airCondition(); // Fehler
}
```

Überschreiben einer Funktion

```
class Auto {  
public:  
    void hupen() {  
        cout << "huup" << endl;  
    }  
};
```

```
class Pkw : public Auto {  
public:  
    void hupen() {  
        cout << "tuut" << endl;  
    }  
};
```

```
int main() {  
    Auto karre;  
    Pkw pkw;  
    karre.hupen() // "huup"  
    pkw.hupen();  // "tuut"  
}
```

Konstruktoren bei der Vererbung

- Konstruktor der Unterklasse ruft zuerst einen Konstruktor der Oberklasse auf
 - den Standardkonstruktor oder
 - den explizit angegebenen Konstruktor

```
Unterklasse(...) : Oberklasse(***)
```
- sorgt für Kompatibilität zur Oberklasse
 - ↪ *Exemplare der Unterklasse sind spezialisierte Exemplare der Oberklasse.*

```
Oberklasse allgemein;
```

```
Unterklasse spezialisiert;
```

```
allgemein = spezialisiert;    // ok
```

```
spezialisiert = allgemein;    // geht nicht
```

Zugriffsattribute bei der Vererbung

- `class Unterklasse : public Oberklasse`
~> alle Elemente (Datenelemente und Methoden) werden vererbt
(*auf private Elemente kann aber nicht direkt zugegriffen werden*)
- `class Unterklasse : Oberklasse` oder
`class Unterklasse : private Oberklasse`
~> alle geerbten Elemente werden `private`
- Definition von Elementen mit `protected`:
~> Zugriff nur für die Klasse und ihre Unterklassen

Zugriffsattribute – Beispiel

```
class Oberklasse {  
    private:  
        int privat;  
    protected:  
        int protect;  
    public:  
        int publik;  
};  
  
int main() {  
    Oberklasse o;  
    Unterklasse u;  
  
    int i = u.publik;    // Fehler  
    i = o.publik;       // ok  
    i = o.protect;      // Fehler  
}  
  
class Unterklasse : Oberklasse { // privat!  
    public:  
        int f1() { return privat; }    // geht nicht  
        int f2() { return protect; }   // ok  
        int f3() { return publik; }    // ok
```


Mehrfachvererbung

- Aufzählen der Oberklassen, durch Komma getrennt:

```
class Unterklasse : public Elternklasse, public Superklasse  
{ ... }
```

- Vorsicht vor Namenskonflikten
 - z.B. bei Datenelement `var` in beiden Oberklassen:

```
a = Elternklasse::var;  
b = Superklasse::var;
```
 - besser Mehrfachvererbung vermeiden!

Polymorphie durch Überschreiben virtueller Methoden

- Redefinieren (“Verdecken”) der Methode der Oberklasse
- bei Aufrufen wird die Implementierung der Klasse abgearbeitet, die dem Typ entspricht, mit dem das Objekt definiert wurde
 - ~> Entscheidung durch den Compiler ([statische/frühe Bindung](#))
- werden *virtuelle Methoden* (Schlüsselwort `virtual`) überschrieben, entscheidet der Typ des *Objekts*, an das der Aufruf gerichtet ist ([dynamische/späte Bindung](#))
- Aber: bei Zuweisungen von Unterklassenobjekten an Oberklassenvariablen erfolgt automatisch eine Typkonversion
 - ~> Besonderheiten der Unterklasse gehen verloren!
 - ~> Pointer verwenden, um dynamische Bindung zu erreichen!

Polymorphie, nicht virtuell

```
class Auto {  
public:  
    void hupen() {  
        cout << "huup" << endl;  
    }  
};
```

```
class Pkw : public Auto {  
public:  
    void hupen() {  
        cout << "tuut" << endl;  
    }  
};
```

```
void signal(Auto * a) {  
    a->hupen();  
}
```

```
int main() {  
    Auto karre;  
    Pkw pkw;  
    signal(&karre) // "huup"  
    signal(&pkw);  // "huup"  
}
```

Polymorphie, virtuell

```
class Auto {  
public:  
    virtual void hupen() {  
        cout << "huup" << endl;  
    }  
};
```

```
class Pkw : public Auto {  
public:  
    void hupen() {  
        cout << "tuut" << endl;  
    }  
};
```

```
void signal(Auto * a) {  
    a->hupen();  
}
```

```
int main() {  
    Auto karre;  
    Pkw pkw;  
    signal(&karre) // "huup"  
    signal(&pkw);  // "tuut"  
}
```

Polymorphie, virtuell

```
class Auto {  
public:  
    virtual void hupen() {  
        cout << "huup" << endl;  
    }  
};
```

```
class Pkw : public Auto {  
public:  
    void hupen() {  
        cout << "tuut" << endl;  
    }  
};
```

```
void signal(Auto * a) {  
    a->hupen();  
}
```

```
int main() {  
    Auto karre;  
    Pkw pkw;  
    signal(&karre) // "huup"  
    signal(&pkw);  // "tuut"  
    karre = pkw;   // Typkonvertierung  
    signal(&karre); // "huup"  
}
```