

Klassen und Objekte

Zunächst werden einige Begriffe wiederholend erklärt, dann wird auf spezielle Eigenschaften von Klassen und Objekten eingegangen.

Inhalt

1. Eine erstes Beispiel (zur Wiederholung)	2
2. Klassenspezifische Daten und Funktionen	17
2.1. Klassenspezifische Konstanten	26
3. Verschachtelte Klassen	28
4. Klassentemplates	33
4.1. Member Templates	42
4.2. Rekursive Templates	45

1. Eine erstes Beispiel (zur Wiederholung)

Zur Verarbeitung von Strings haben wir bis jetzt entweder char-Arrays oder den Datentyp `string` verwendet.

`string` ist eigentlich eine Klasse von C++. Wir werden nun eine **einfache Sting-Klasse** `mstring` selbst entwickeln, um Eigenschaften von Klassen kennen zu lernen.

Die Methodennamen sind an die Methoden der Klasse `string` angelehnt; aus Gründen der Kompatibilität zu C-String zu haben, wird die Realisation jeden String mit einem Nullbyte abschließen.

Die erste Version von `mstring` hat folgendes Aussehen:

```
$ cat mstring.h
// einfache String-Klasse. Erste, nicht vollständige Version
#ifdef mstring_h
#define mstring_h mstring_h
#include<cstdint> // size_t
#include<iostream> // ostream
```

```

class mstring {
    public:
        mstring(); // Default-Konstruktor
        mstring(const char *); // allg. Konstruktor
        mstring(const mstring&); // Kopierkonstruktor
        ~mstring(); // Destruktor
        void assign(const mstring&); // Zuweisung eines mstring
        void assign(const char *); // Zuweisung eines char*
        const char& at(std::size_t position) const; // Zeichen holen
        char& at(std::size_t position); // Zeichen holen,
        // die Referenz erlaubt Ändern des Zeichens
        std::size_t length() const { return len;} // Stringlänge zurückgeben
        const char* c_str() const { return start;} // C-String zurückgeben
        void anzeigen(std::ostream &os); // anzeigen
    private:
        std::size_t len; // Länge
        char *start; // Zeiger auf den Anfang
};

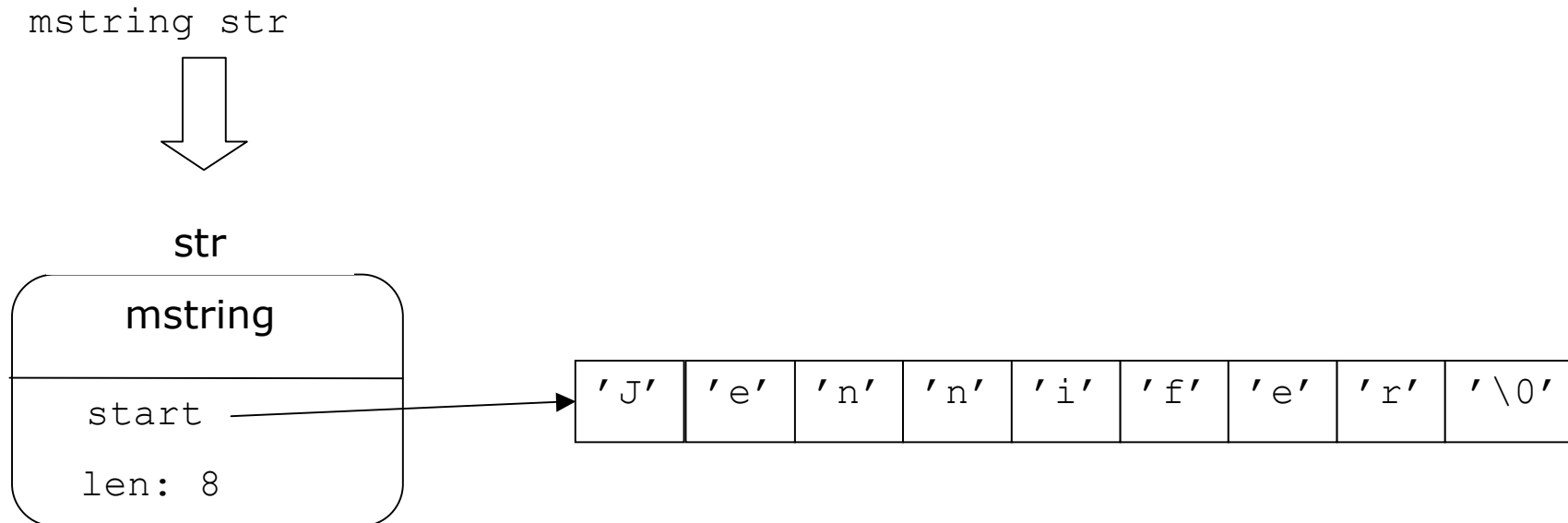
#endif // mstring_h
$

```

Die Klasse `mstring` hat nur private Daten:

- `start` ist der Zeiger auf den Anfang der Zeichenkette. Der benötigte Platz wird im Konstruktor dynamisch erzeugt.
- `len` speichert die aktuelle Länge des Strings. Damit ist die Methode `length` effizienter, als wenn man stets die Länge ermitteln müsste.

Veranschaulichen kann man sich damit ein Objekt der Klasse `mstring` als:



Eine Anwendung, die die Klasse verwendet, könnte dann etwa so aussehen:

```

$ cat strmain.cpp
#include"mstring.h"
#include<iostream>
using namespace std;

int main() {
    mstring einString("Jennifer");           // allg. Konstruktor
    einString.anzeigen(cout);

    cout << "\nzeichenweise Ausgabe:\n";
        for(size_t i = 0; i < einString.length(); ++i)
            cout << '*' << einString.at(i);
    cout << '*' << endl;

    mstring Z;
    Z.assign(einString);                     // Zuweisen

    cout << "zugewiesener String: ";
    Z.anzeigen(cout);
    cout << endl;
}
$

```

```

$ t
Jennifer
zeichenweise Ausgabe:
*J*e*n*n*i*f*e*r*
zugewiesener String: Jennifer
$

```

Die Implementierung kann wie folgt realisiert werden:

```
$ cat mstring.cpp
#include "mstring.h"
#include <cassert>
using namespace std;

// Hilfsfunktion: String kopieren (um #include <cstring> zu vermeiden)
static void copy(char *ziel, const char *quelle) {
    while ((*ziel++ = *quelle++));
}

// Hilfsfunktion: Länge eines C-Strings ermitteln
static size_t laenge(const char *s) {
    size_t sl = 0;
    while (*s++) ++sl;
    return sl;
}
```

```

/* Der Default-Konstruktor erzeugt einen leeren String der Länge 0,
   der nur aus dem Nullbyte besteht. */
mstring::mstring()                // Default-Konstruktor
: len(0), start(new char[1]) {    // Platz für '\0'
    *start = '\0';               // leerer String
}

/* Der allgemeine Konstruktor erzeugt aus einem klassischen C-String ein
   mstring-Objekt. */
mstring::mstring(const char *s)   // allg. Konstruktor
: len(strlen(s)), start(new char[strlen(s)+1]) { // Platz für s
    strcpy(start, s);
}

```

Das funktioniert nur, weil die Initialisierungsliste in der Reihenfolge der Attributdefinition im `private` Teil der Klasse abgearbeitet wird.

D.h. wäre in der Klasse definiert:

```

char *start;
size_t len;

```

dann würde der Konstruktor fehlerhaft arbeiten!

```
/* Der Kopierkonstruktor arbeitet ähnlich, nur dass er die Länge des  
Objekts, mit dem initialisiert wird, direkt übernehmen kann. */
```

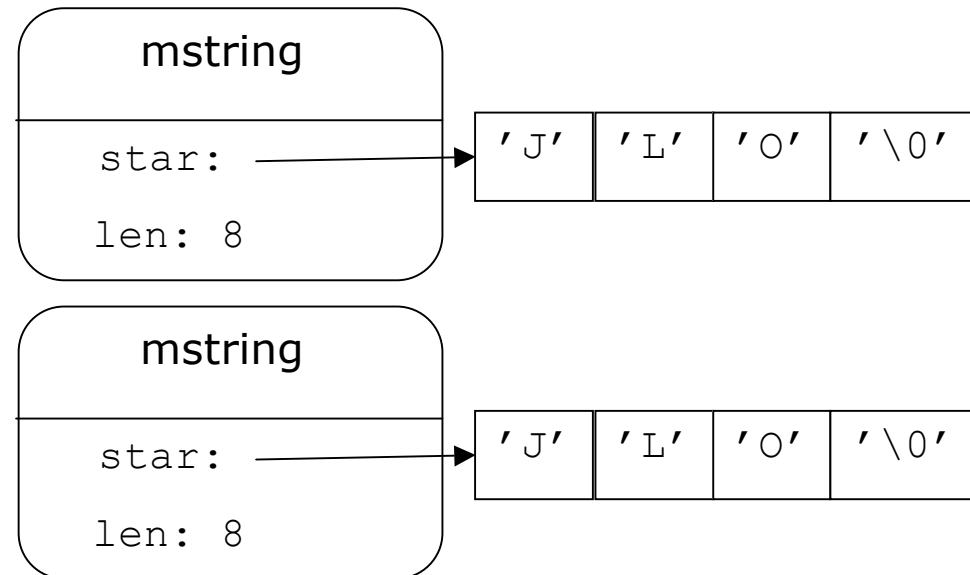
```
mstring::mstring(const mstring &m) // Kopierkonstruktor  
: len(m.len), start(new char[len+1]) {  
    copy(start, m.start);  
}
```

Der Kopierkonstruktor stellt sicher, dass die **Inhalte kopiert** werden (tiefe Kopie).

```
mstring str („JLO“);
```

```
mstring k(str);
```

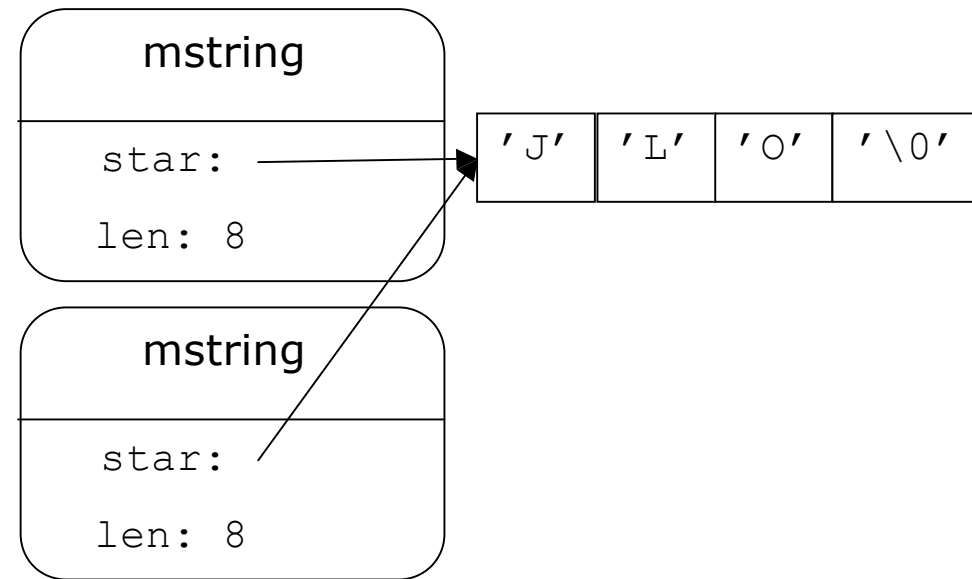
tiefe Kopie



Wäre kein Kopierkonstruktor explizite programmiert, würde der Standard Kopierkonstruktor nur die Werte der Attribute `len` und `start` kopieren (flache Kopie):

```
mstring str („JLO“);  
mstring k(str);
```

flache Kopie



```
mstring::~~mstring() { // Destruktor  
    delete [] start;  
}
```

Welche Ausgabe erzeugt das folgende Programm, wenn der Kopierkonstruktor flach implementiert ist?

```
cat mstring04/stringmain.cpp
#include"mstring.h"
#include<iostream>
using namespace std;

int main() {
    mstring einString("Jennifer");    // allg. Konstruktor
    anzeigen(cout,einString);
    cout << endl;
    {
        cout << "Kopierter String: ";
        mstring K(einString);        // Kopierkonstruktor
        anzeigen(cout,K);
        cout << endl;
    }
    anzeigen(cout,einString);
    cout << endl;
}
$
```

Wie sieht eine flache Implementierung des Kopierkostruktors aus?

-> mstring04/mstring.cpp

Die Zuweisung (assign) muss neuen Platz beschaffen, kopieren und den alten Platz freigeben.

```
void mstring::assign(const mstring &m) { // Zuweisung eines mstring
    char *p = new char[m.len+1]; // neuen Platz beschaffen
    copy(p, m.start);
    delete [] start; // alten Platz freigeben
    len = m.len; // Verwaltungsinformation aktualisieren
    start = p;
    return ;
}

void mstring::assign(const char *s) { // Zuweisung eines char*
    size_t L = laenge(s);
    char *p = new char[L+1];
    copy(p, s);
    delete [] start; // alten Platz freigeben
    len = L; // Verwaltungsinformation aktualisieren
    start = p;
    return ;
}
```

assign ist überladen, deshalb können C-Strings und `mstring`'s zugewiesen werden:

```
mstring str;           // Standardkonstruktor
mstring str1("JLO");  // allg. Konstruktor
str.assign(str1);     // Zuweisung
str.assign("AS");     // Zuweisung
```

```
char& mstring::at(size_t position)  {           // Zeichen per Referenz holen
    assert(position < len);         // Nullbyte lesen ist nicht erlaubt
    return start[position];
}

const char& mstring::at(size_t position) const { // Zeichen holen
    assert(position < len);         // Nullbyte lesen ist nicht erlaubt
    return start[position];
}

void mstring::anzeigen(ostream &os)  {
    os << start;
}
$
```

In der nächsten Version werden wir nun ermöglichen, dass die Methode `anzeigen` als **globale** Funktion realisiert ist. Dann muss der Aufruf das Objekt, das angezeigt werden soll, als zweiten Parameter mit übergeben:

```
$ cat mstring.h
// einfache String-Klasse. Zweite, nicht vollständige Version
...
```

```
class mstring {
    ...
};

void anzeigen(std::ostream&, const mstring&); // anzeigen global

#endif // mstring_h
$
```

```
$ cat mstring.cpp
...
void anzeigen(ostream &os, const mstring &m) {
    os << m.c_str();
}
```



Nun erfolgt der Aufruf von `anzeigen` in der Form:

```
mstring einString("Jennifer");
anzeigen(cout, einString);
```

Diese Form ist aus Sicht der OOP **nicht** sinnvoll. In Anwendungen braucht man aber manchmal die Möglichkeit, dass man von **nicht Elementfunktionen** auf **private Klassenattribute** zugreifen können muss.

Hierzu stellt C++ den `friend`-Mechanismus zur Verfügung:

Eine als `friend` gekennzeichnete Funktion ist **keine** Methode der Klasse, hat aber das Recht auf klassenprivate Attribute zuzugreifen.

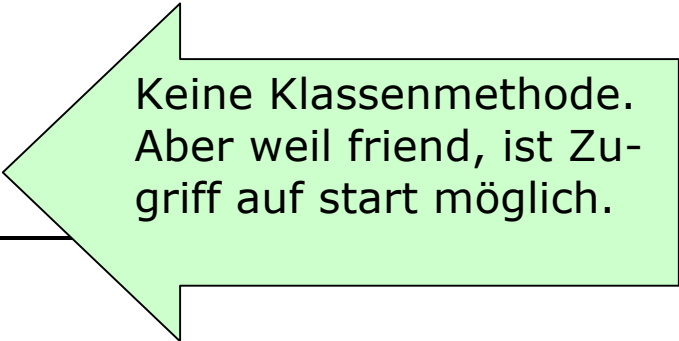
Dazu ist in der Klassendeklaration die „befreundete“ Funktion als `friend` zu deklarieren; dann kann sie global bekannt gemacht werden:

Jetzt wird anstelle der globalen Funktion eine `friend`-Methode verwendet.

```
$ cat mstring.h
class mstring {
    public:
        ...
        friend void anzeigen(std::ostream&, const mstring&); // anzeigen als friend
};

void anzeigen(std::ostream&, const mstring&); // anzeigen global

#endif // mstring_h
$
$ cat mstring.cpp
...
void anzeigen(ostream &os, const mstring &m) {
    os << m.start;
}
```



Keine Klassenmethode.
Aber weil `friend`, ist Zugriff auf `start` möglich.

Hörsaalübung:

Realisieren Sie eine Klasse CInteger zur Darstellung beliebig großer Integer als Array von Ziffern.

Methoden:

- Konstruktor, allgemeiner Konstruktor, Kopierkonstruktor
- assign
- anzeigen
- plus, mal, minus, geteilt

Verwenden Sie den friend-Mechanismus für anzeigen.

2. Klassenspezifische Daten und Funktionen

Objekte beinhalten Daten und Methoden. In vielen Anwendungsfällen braucht man eine Möglichkeit, **Daten** und **Funktionen** für **alle Objekte**, die zu einer Klasse gehören verfügbar zu haben.

Eine (**schlechte**) Möglichkeit, dies zu ermöglichen wäre die **Verwendung** von **globalen Objekten**, die von allen Instanzen einer Klasse (aber auch von Instanzen anderer Klassen) verwendet werden können. Dies ist kein guter Weg, da er alle Nachteile der Verwendung globaler Variablen mit sich bringt.

Klassenspezifische Daten sind Daten, die **nur einmal** für **alle Objekte** der Klasse **existieren**. Diese Daten sind dann gekapselt und können nur von den Objekten der Klasse (von keinen anderen Klassen) gemeinsam verwendet werden. In C++ werden diese Daten mit `static` gekennzeichnet.

Klassenspezifische Funktionen führen Aktionen aus, die an einer Klasse, nicht an einem Objekt der Klasse gebunden sind. Klassenspezifische Funktionen werden ebenfalls mit `static` gekennzeichnet. Jeder Konstruktor ist ein Beispiel für eine solche Funktion, hier entfällt `static`.

Beispiel (Bankkonto)

Ein typisches Beispiel ist eine Klasse, die nummerierte Objekte realisiert. Z.B. kann ein Bankkonto modelliert werden, dann braucht man einen Automatismus, der bei jeder **Kontoeröffnung** eine **neue Kontonummer erzeugt**.

Im folgenden Beispiel ist der Mechanismus verdeutlicht:

Wir betrachten eine Klasse, die **jedem Objekt eine eindeutige Seriennummer** mitgibt und **objektunabhängig über die insgesamt erzeugten Objekte Buch führt**.

```
$ cat numobj.h
#ifdef numobj_h
#define numobj_h mstring_h
class nummeriertesObjekt {
public:
    nummeriertesObjekt();
    nummeriertesObjekt(const nummeriertesObjekt&);
    ~nummeriertesObjekt();
    unsigned long Seriennummer() const { return SerienNr;}
    static int Anzahl() { return anzahl;}
    static bool Testmodus;

private:
    static int anzahl;
    static unsigned long maxNummer;
    const unsigned long SerienNr;
};
#endif // Ende von numobj.h
$
```



klasenspezifisch



klasenspezifisch

Die Methode **Seriennummer** ist **objektspezifisch**, sie gibt die dem Objekt zugeordnete Seriennummer (`SerienNr`) zurück. Die Methode **Anzahl** ist **klassenspezifisch**, sie gibt die insgesamt aktiven Objektinstanzen zurück.

Die Variable `Testmodus` ist dazu da, die Erzeugung und Löschung von Objekten zu visualisieren.

```
$ cat numobj.cpp
#include<iostream>
#include"numobj.h"
#include<cassert>
using namespace std;

// Initialisierung der klassenspezifischen Variablen:
int nummeriertesObjekt::anzahl = 0;
unsigned long nummeriertesObjekt::maxNummer = 0L;
bool nummeriertesObjekt::Testmodus = false;

// Default-Konstruktor
nummeriertesObjekt::nummeriertesObjekt()
: SerienNr(++maxNummer) {
    ++anzahl;
    if (Testmodus) {
        if (SerienNr == 1)
            cout << "Start der Objekterzeugung!\n";
        cout << " Objekt Nr. "
            << SerienNr << " erzeugt" << endl;;
    }
}
```

Die Initialisierung der static Variablen **darf nicht innerhalb** eines Konstruktors erfolgen, da ansonsten bei jeder Objekterzeugung die Initialisierung erfolgen würde.

Der Standardkonstruktor **initialisiert** die **objekt-spezifische** Konstante `SerienNr` und **inkrementiert** die **klassenspezifische** Variable `maxNummer`.

Der Standard-Kopierkonstruktor kann **nicht** verwendet werden, es darf hier **keine** Kopie erstellt werden, ansonsten hätten zwei Objekte dieselbe Seriennummer, also muss ein eigener Kopierkonstruktor implementiert werden.

```
// Kopierkonstruktor
nummeriertesObjekt::nummeriertesObjekt(const nummeriertesObjekt &X)
:   SerienNr(++maxNummer) {
    ++anzahl;

    if (Testmodus)
        cout << "   Objekt Nr. " << SerienNr
            << " mit Nr. " << X.Seriennummer()
            << " initialisiert" << endl;;
}
```

Der Destruktor muss die Anzahl der aktiven Objekte dekrementieren. Er wird aufgerufen, implizit bei Verlassen eines Blocks, in dem ein Objekt erzeugt wurde und explizit durch Verwendung von `delete`.

```

// Destruktor
nummeriertesObjekt::~nummeriertesObjekt() {
    anzahl--;
    if (Testmodus) {
        cout << " Objekt Nr. "
             << SerienNr << " gelöscht" << endl;

        if (anzahl == 0)
            cout << "letztes Objekt gelöscht!" << endl;

        if (anzahl < 0) // deshalb int und nicht unsigned int
            cout << " FEHLER! zu oft delete aufgerufen!"
                 << endl;;
    }
    else assert(anzahl >= 0);
} // Ende von numobj.cpp
$

```

Die Verwendung der Klasse wird in folgendem Hauptprogramm demonstriert.

```
$ cat nummain.cpp
```

```
int main() {
```


```
    // Testmodus für alle Objekte der Klasse einschalten
```

```
    nummeriertesObjekt::Testmodus=true;
```

```
    nummeriertesObjekt dasNumObjekt_X;           // ... wird erzeugt
```

```
    cout << "Die Seriennummer von dasNumObjekt_X ist: "
```

```
         << dasNumObjekt_X.Seriennummer() << endl;
```



Zugriff auf klassenspez. Variable



Objekterzeugung

```
// Anfang eines neuen Blocks
```

```
{ nummeriertesObjekt dasNumObjekt_Y; // ... wird erzeugt
```

```
// objektgebundener Aufruf:
```

```
cout << dasNumObjekt_Y.Anzahl()  
     << " Objekte aktiv" << endl;
```

schlechter Stil (objektbezogen)

```
// *p wird dynamisch erzeugt:
```

```
nummeriertesObjekt *p = new nummeriertesObjekt;
```

```
// objektgebundener Aufruf über Zeiger:
```

```
cout << p->Anzahl()  
     << " Objekte aktiv" << endl;
```

```
delete p; // *p wird gelöscht
```

schlechter Stil (objektbezogen)

```
// klassenbezogener Aufruf:
```

```
cout << nummeriertesObjekt::Anzahl()  
     << " Objekte aktiv" << endl;
```

guter Stil (klassenbezogen)

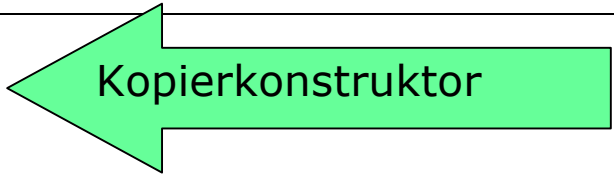
```
} // Blockende: dasNumObjekt_Y wird gelöscht
```

```
cout << " Kopierkonstruktor: " << endl;
    nummeriertesObjekt dasNumObjekt_X1 = dasNumObjekt_X;

    cout << "Die Seriennummer von dasNumObjekt_X ist: "
        << dasNumObjekt_X.Seriennummer() << endl;

    cout << "Die Seriennummer von dasNumObjekt_X1 ist: "
        << dasNumObjekt_X1.Seriennummer() << endl;

    // dasNumObjekt_X wird gelöscht
}
$
```



Kopierkonstruktor

Will man im Hauptprogramm eine Zuweisung von Objekten durchführen, die zur Klasse nummeriertesObjekt gehören, so wird der Compiler einen Fehler melden:

```

$ cat numobj02/nummain.cpp
...
cout << " Kopierkonstruktor: " << endl;
    nummeriertesObjekt dasNumObjekt_X1 = dasNumObjekt_X;

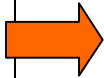
    cout << "Die Seriennummer von dasNumObjekt_X ist: "
        << dasNumObjekt_X.Seriennummer() << endl;

    cout << "Die Seriennummer von dasNumObjekt_X1 ist: "
        << dasNumObjekt_X1.Seriennummer() << endl;

// Zuweisung wird wegen const SerienNr vom Compiler verboten
dasNumObjekt_X1 = dasNumObjekt_X; // Fehler

// dasNumObjekt_X wird gelöscht
}
$

```



```

g++ -c nummain.cpp
nummain.cpp: In member function `nummeriertesObjekt&
    nummeriertesObjekt::operator=(const nummeriertesObjekt&)`:
nummain.cpp:47: non-static const member `const long unsigned int
    nummeriertesObjekt::SerienNr', can't use default assignment operator

```

Die Zuweisung würde bewirken, dass dem privaten Attribut des Objektes (`const SerienNr`), eine Konstante, ein Wert zugewiesen würde. Einer Konstanten kann aber nichts zugewiesen werden, also kann der Zuweisungsoperator nicht erzeugt werden.

2.1. Klassenspezifische Konstanten

Im letzten Beispiel ist gezeigt, dass **klassenspezifische Variable außerhalb der Klassendefinition initialisiert** werden müssen. Klassenspezifische Konstanten für Aufzählungen und einfache Typen können innerhalb der Klasse definiert und initialisiert werden.

```
class bsp {  
    enum RGB { rot = 1, gelb = 2, blau = 4};  
    static const int max = 1000;  
  
    static int messwerte[max];  
    ...  
}
```

static kann entfallen, es sind immer klassenspezifische Konstanten

Verwendung von klassensp. Konstante zur Definition einer klassensp. Variable.

Hörsaalübung:

Realisieren Sie eine einfache Klasse CKonto mit Methoden

- einzahlen
- auszahlen
- kontostand

so, dass ein Attribut Kontonummer automatisch hochgezählt wird, wenn ein Konto angelegt wird.

3. Verschachtelte Klassen

Klassen können **geschachtelt** werden. Eine Klasse wird also innerhalb einer Klasse vereinbart.

Die innere Klasse kann im öffentlichen oder im privaten Teil vereinbart werden. Der Unterschied bezieht sich auf die **Benutzbarkeit**.

Lokale Klasse B im öffentlichen Teil der Klasse A

```
class A {  
    public:  
        class B {  
            public:  
                int xB;  
        };  
        int xA;  
};  
  
A a; a.xa = 1;  
A::B b; b.xB = 2;
```

A und B verhalten sich wie 2 separate Klassen. Aber der Zugriff auf die innere Klasse erfolgt **vollqualifiziert** über den Bereichsoperator.

Klassenbezogene Definitionen am Beispiel von lokalem typedef

```
class A {
    static void fa() { cout << "A" << endl; }
public:
    class B {
        public:
            typedef unsigned int uint
            uint xB;
            void fB() { fA(); };
    };
    B::uint xA;
};
```

In B kann die Methode **fA()** verwendet werden, weil sie mit **static** spezifiziert wurde.

In A kann die **Typdefinition uint** verwendet werden (mittels **Bereichsoperator**).

xB kann in der Klasse **A nicht** verwendet werden, da **A kein B-Objekt enthält**. Dies wird im nächsten Beispiel demonstriert.

Zugriff von A auf Attribute der inneren Klasse B

```
class A {  
    public:  
        class B {  
            public:  
                int xB;  
        };  
        B b;  
        int Ax;  
        void set(int x, int y) { xA = x; b.xB = y; }  
};
```

Da A ein B-Attribut besitzt, kann auf das interne B-Attribut xB zugegriffen werden.

Lokale Klasse B im privaten Teil der Klasse A:

```
class A {
    private:
        class B {
            private:
                int xB;
            public:
                void put(int x) { xB = x; }
                int get() { return xB; }
        };
        B b;
        int xA;
    public:
        void put(int x, int y) { xA = x; b.put(y); }
        void get(int &x, int &y) { x = xA; y = b.get(); }
};
```

B ist im private-Bereich definiert; somit ist B selbst und seine Instanz b außen sichtbar. Der Zugriff muss also über die Schnittstellenmethode von B erfolgen.

Vorwärtsdeklarationen der inneren Klasse B

```
class A {
    private:
        class B; // Vorwaertsdeklaration
        B b;
        int xA;
    public:
        void put(int x, int y);
        void get(int &x, int &y);
};
class A::B { // Deklaration der Klasse B
    private:
        int xB;
    public:
        void put(int x) { xB = x; }
        int get() { return xB; }
};
void A::put(int x, int y) { xA = x; b.put(y); }
void A::get(int &x, int &y) { x = xA; y = b.get(); }

// Implementierung der Methoden der Klasse B
void A::B::put(int x) { xB = x; }
int A::B::get() { return xB; }
```

Innerhalb von A wird die Implementierung der Methode verborgen.

4. Klassentemplates

Durch Funktionstemplates sind „typunabhängige“ Funktionen möglich. Klassentemplates ermöglichen „**parametrisierte Datentypen**“ zu realisieren.

Klassentemplates sind **Typ-parameterisierte** Klassen. Der **Compiler** erzeugt daraus **Klassen für konkrete Datentypen**.

So wie ein Funktionstemplate durch Voranstellen von `template <typename Typen>` entsteht, wird aus einer Klasse durch Voranstellen von `template <class Typen>` ein Klassentemplate (man kann auch `template <typename Typen>` vor die Klasse schreiben).

Dies wird am Beispiel eines Stack gezeigt.

```
$ cat simstack.h
// ein einfaches Stack-Template
#ifndef simstack1_t
#define simstack1_t simstack1_t
#include<cassert>

const unsigned int maxSize=10;
```

```

template <class T>
class simpleStack {
public:
    simpleStack() : anzahl(0){}
    bool empty() const { return anzahl == 0;}
    bool full() const { return anzahl == maxSize;}
    unsigned int size() const { return anzahl;}
    void clear() { anzahl=0;}           // Stack leeren

    const T& top() const;              // letztes Element sehen
    void pop();                        // Element entfernen
    // Vorbedingung für top und pop: Stack ist nicht leer

    void push(const T &x);            // x auf den Stack legen
    // Vorbedingung für push: Stack ist nicht voll

private:
    unsigned int anzahl;
    T array[maxSize];                 // Behälter für Elemente
};

```

```

// noch fehlende Methoden-Implementierungen
template <class T>
const T& simpleStack<T>::top() const {
    assert(!empty());
    return array[anzahl-1];
}

template <typename T>
void simpleStack<T>::pop() {
    assert(!empty());
    --anzahl;
}

template <class T>
void simpleStack<T>::push(const T &x) {
    assert(!full());
    array[anzahl++] = x;
}
#endif // simstack1_t
$

```

Die Verwendung zeigt ein einfaches Hauptprogramm:

```
$ cat simmain.cpp
#include<iostream>
#include"simstack.h"
using namespace std;

int main() {
    simpleStack<int> einIntStack;

    // Stack füllen
    int i=10;
    while(!einIntStack.full())
        einIntStack.push(i++);
    cout << "Anzahl : " << einIntStack.size() << endl;

    // oberstes Element anzeigen
    cout << "oberstes Element: "
        << einIntStack.top() << endl;

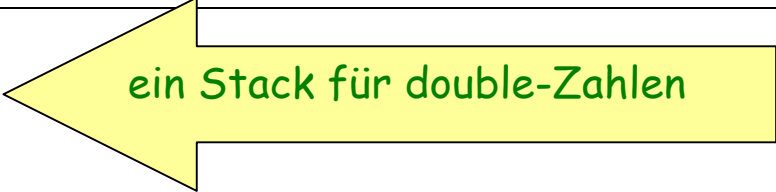
    cout << "alle Elemente entnehmen und anzeigen: " << endl;
    while(!einIntStack.empty()) {
        i = einIntStack.top();
        einIntStack.pop();
        cout << i << '\t';
    }
    cout << endl;
}
```



ein Stack für int-Zahlen

```
// ein Stack für double-Zahlen
```

```
simpleStack<double> einDoubleStack;
```



ein Stack für double-Zahlen

```
// Stack mit (beliebigen) Werten füllen
```

```
double d=1.00234;
```

```
while(!einDoubleStack.full()) {  
    d = 1.1*d;  
    einDoubleStack.push(d);  
    cout << einDoubleStack.top() << '\t';  
}
```

```
cout << "\n4 Elemente des Double-Stacks entnehmen:" << endl;
```

```
for (i=0; i<4; ++i) {  
    cout << einDoubleStack.top() << '\t';  
    einDoubleStack.pop();  
}
```

```
cout << endl;
```

```
cout << "Restliche Anzahl : "  
    << einDoubleStack.size() << endl;
```

```
cout << "clear Stack" << endl;  
einDoubleStack.clear();
```

```
cout << "Anzahl : " << einDoubleStack.size() << endl;
```

```
}  
$
```

Der Compiler sieht in main

```
simpleStack<int> einIntStack;
```

und erzeugt nun die Klasse:

```
class simpleStack {
public:
    simpleStack() : anzahl(0){}
    bool empty() const { return anzahl == 0;}
    bool full() const { return anzahl == maxSize;}
    unsigned int size() const { return anzahl;}
    void clear() { anzahl=0;}           // Stack leeren

    const int& top() const;           // letztes Element sehen
    void pop();                       // Element entfernen
    // Vorbedingung für top und pop: Stack ist nicht leer

    void push(const int &x);         // x auf den Stack legen
    // Vorbedingung für push: Stack ist nicht voll

private:
    unsigned int anzahl;
    int array[maxSize];              // Behälter für Elemente
};
...
```

und analog für double.

Die Erzeugung eines Objektes für einen konkreten Datentyp anstelle des Platzhalters T wird ***Instantiierung des Templates*** genannt.

Im o.a. Beispiel ist die Stackgröße eine **globale** Größe. **Besserer** Stil ist es, durch einen **zweiten Parameter** die maximale Größe für das Template anzugeben.

Auch wäre denkbar, einen zweiten Parameter zu verwenden, der die Größe angibt und im Konstruktor der Klasse durch `new`, entsprechend viel Platz zu erzeugen.

Hier die erste Möglichkeit, die zweite Möglichkeit werden wir im Zusammenhang mit Containern erörtern:

```

// ein einfaches Stack-Template
template <class T, int maxSize>
class simpleStack {
public:
    simpleStack() : anzahl(0){}

    ...

private:
    unsigned int anzahl;
    T array[maxSize];           // Behälter für Elemente
};
// noch fehlende Methoden-Implementierungen
template <class T, int m>
const T& simpleStack<T, m>::top() const {
    assert(!empty());
    return array[anzahl-1];
}
...

```

Die **Größe** wird nicht dem Objekt, sondern dem Template übergeben. Dadurch ist sie zur **Übersetzungszeit** bekannt. Hier handelt es sich also um **statisch** zugewiesenen Speicher, nicht um dynamische Allokation.

Frage:

Was passiert beim folgenden Programm:

```
$ cat stack3/simmain.cpp
#include<iostream>
#include"simstack.h"
using namespace std;

int main() {
    int n;
    cout << "Groesse angeben: ";
    cin >> n;
    simpleStack<int,n> einIntStack;
    ...
}
$
```

4.1. Member Templates

Member Templates sind Templates, die innerhalb einer Klasse deklariert werden.

Achtung:

Nicht alle C++ Umgebungen unterstützen Member Templates!

Betrachten wir **zunächst** eine **Klasse Pair ohne Member Template**:

```

$ cat mt.cpp
#include <iostream>
using namespace std;

template <class A, class B>
class Pair {
    public:
        A a;
        B b;

        Pair(const A& ax, const B& bx) // constructor
        : a(ax), b(bx) {
        }
};

void main() {
    Pair<int, long> z(12, 123456);
    cout << z.a << " " << z.b << endl;
}
$

```

Hier wird ein Objekt z als (int,long) Paar erzeugt und jede Komponente ausgegeben.

Will man ein Paar erzeugen, das aus einem **beliebigen anderen Paar** konstruiert werden kann, kann man ein Member Template als Konstruktor verwenden (dieser Mechanismus wird in der STL oft verwendet):

```

$ cat mt02.cpp
#include <iostream>
using namespace std;

template <class A, class B>
class Pair {
    public:

        A a;
        B b;

        Pair(const A& ax, const B& bx) // constructor
        : a(ax), b(bx) {
        }

        template <class T, class U> // template constructor
        Pair(const Pair<T,U>& p)
        : a(p.a), b(p.b) {
        }

};

int main() {
    Pair<short, float> x(37, 12.34);
    Pair<short, float> y(x);
    cout << y.a << " " << y.b << endl;
}
$

```

Member Template

y ist Objekt, das aus **einem** short, float Paar erzeugt wird.

4.2. Rekursive Templates

Eine Definition eines Template kann rekursiv sein. Dadurch wird der **Compiler gezwungen**, die **Rekursion aufzulösen** und Code zu erzeugen, bei dem mehrere Datentypen erzeugt werden, die dann zur Laufzeit des Programms verwendet werden.

Beispiel ($2^{10}=1024$)

```
$ cat zweihoch.cpp
#include<iostream>
using namespace std;
template<int n>
class Zweihoch {
public:
    enum { Wert = 2*Zweihoch<n-1>::Wert} ;
};

template<>
class Zweihoch<0> {
public:
    enum { Wert=1 };
};

int main() {
    cout << Zweihoch<10>::Wert << endl;
}
$
```

Der Compiler versucht `Zweihoch<10>::Wert` zu ermitteln. Der Aufzählungstyp `Wert` hat für jeden Typ der Klasse `Zweihoch` einen anderen Wert, der von `n` abhängt.

Der Compiler erzeugt also Code für:

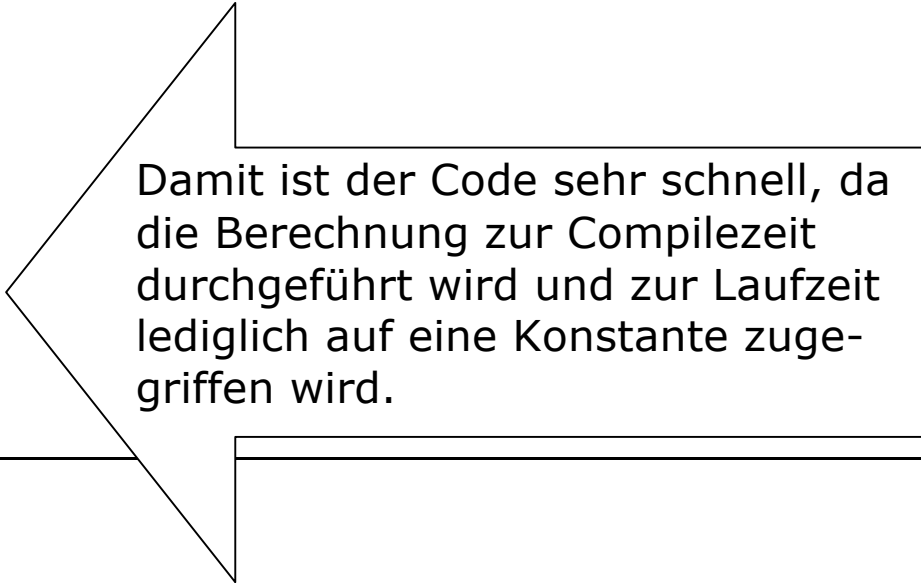
```
#include<iostream>
using namespace std;

template<int n>
class Zweihoch {
    public:
        enum { Wert = 2*Zweihoch<n-1>::Wert} ;
};

class Zweihoch<0> {
    public: enum { Wert=1 };
};

class Zweihoch<1> {
    public: enum { Wert=2 };
};
...
class Zweihoch<10> {
    public: enum { Wert=1024 };
};

int main() {
    cout << Zweihoch<10>::Wert << endl;
}
```



Damit ist der Code sehr schnell, da die Berechnung zur Compilezeit durchgeführt wird und zur Laufzeit lediglich auf eine Konstante zugegriffen wird.

Die Schreibweise im o.a. Beispiel

class ... mit nur public-Bestandteilen ist auch kürzer wie folgt möglich:

```
$ cat zweihoch02.cpp
#include<iostream>
using namespace std;

template<int n>
struct Zweihoch {
    enum { Wert = 2*Zweihoch<n-1>::Wert};
};

template<>
struct Zweihoch<0> {
    enum { Wert=1};
};

int main() {
    cout << Zweihoch<10>::Wert << endl;
}
$
```

Diese Technik ist im folgenden Programm verwendet, um Primzahlen zu berechnen:

```

$ cat primzahl02.cpp
#include<iostream>
using namespace std;

template<int p, int i>
struct istPrimzahl {
    enum {prim=(p%i) && istPrimzahl<(i>2? p:0), i-1>::prim};
};

template<int i>
struct druckePrimzahlenBis {
    druckePrimzahlenBis<i-1> a;
    enum { prime = istPrimzahl<i, i-1>::prim};
    druckePrimzahlenBis() {
        if(prime)
            cout << i << endl;
    }
};

// Spezialisierungen
struct istPrimzahl<0,0> {
    enum {prim=1};
};

```

```
struct istPrimzahl<0,1> {
    enum {prim=1};
};

struct druckePrimzahlenBis<2> {
    enum {prim=1};
};

int main() {
    druckePrimzahlenBis<17> a;
}
$
```

Die o.a. Primzahlberechnung, bei der der Compiler ja die Arbeit erledigt, **produziert Fehlermeldungen des Compilers**, wenn es mit zu großen Werten arbeitet.

Hörsaalübung:

Probieren Sie es an Ihrem Rechner aus: was ist die Größte Primzahl die Sie berechnen können?