Man beachte:

- Für den Aufruf von Elementfunktionen ist die Syntax

  ```
  st.init();
  ```

  analog zum Zugriff auf andere **`struct`**-Elemente, z.B.

  ```
  st.top;
  ```

- Innerhalb der *Definition* einer Elementfunktion entfällt aber der Präfix **`st.`**, es genügt **`top`**.

Im Aufruf `st.push(c);` ist die Variable `st` vor dem Punkt ähnlich zu verstehen wie ein zusätzliches (implizites) Argument.

Mit einer *gewöhnlichen* Funktion würde man schreiben `push(st, c);`.

Dagegen mit einer *Elementfunktion:*

- Die Funktion `push` "gehört" dem Objekt `st`.
- Ein Vorteil ist: Generische Namen wie `push`, `print` oder `open` können gebraucht werden, ohne dass die Gefahr von Namenskonflikten besteht.

`Stack` bedarf noch mehrerer Verbesserungen!

Eines der Probleme ist:

- Die obligatorische Initialisierung (d.h. der Aufruf von **`init`**) ist unschön, und auch unsicher, da diese vergessen werden kann.
- Man kann dies nicht korrigieren, indem man schreibt

  ```
  char* top = data;
  ```

  weil Verbundkomponenten nicht initialisiert werden dürfen.

Abhilfe: Klassen bieten sog. *Konstruktoren* an.

Ein Konstruktor wird *automatisch aufgerufen*, wenn ein Objekt kreiert wird. Dies geschieht

- beim Eintritt in einen Block, worin eine Variable der Klasse deklariert ist, sowie:
- beim dynamischen Erzeugen mittels **`new`** oder **`new …[…]`**.

Der Konstruktor gleicht einer Elementfunktion bis auf folgende Abweichungen:

- der Konstruktor heisst gleich wie die Klasse.
- der Konstruktor hat keinen Rückgabewert, er wird aber auch nicht als **`void`** deklariert.
- er enthält keine **`return`**-Anweisung.

```
// Stack-Klasse, jetzt mit Konstruktor
struct Stack {
    static const int SIZE = 100;
    char data[SIZE];
    char* top;
    Stack() { top = data; }       // Konstruktor
    bool empty() { return top == data; }
    void push(char item) { *top++ = item; }
    char pop() { return *--top; }
};
```

stack2.cpp

Nächste Verbesserung: Es sollen Stacks mit unterschiedlicher Maximalgrösse möglich sein.

Dazu kann ein *Konstruktor mit Argumenten* verwendet werden.

Wie bei gewöhnlichen Funktionen darf der Funktionsname mehrfach verwendet werden, solange die Signatur jedesmal eine andere ist.

Weil wir jetzt Speicher dynamisch allozieren, müssen wir ihn auch wieder freigeben. Dies macht man im *Destruktor*.

Der Destruktor wird automatisch aufgerufen, wenn ein Objekt gelöscht wird, also:

- beim Verlassen des Blocks, wo die Variable deklariert ist, oder
- beim Freigeben mit **`delete`** oder **`delete[…]`**.

Der Destruktor heisst gleich wie die Klasse mit vorangestellter Tilde (**`~`**).

stack3.cpp

```
struct Stack {
  int size;
  char* data;
  char* top;
  Stack() {                  // Default-Konstruktor
    size = 100;
    data = new char[size];
    top = data;
  }
  Stack(int size) {          // Zweiter Konstruktor
    Stack::size = size; // ::  loest Konflikt
    data = new char[size];
    top = data;
  }
  ~Stack() { delete[] data; } // Destruktor
```

Es gibt immer noch Verbesserungsmöglichkeiten:

- Wahrung der *Datenintegrität:*

  Der Benützer der Klasse **`Stack`** hat Zugriff auf die Elemente **`data`** und **`top`**. Er könnte diese fehlerhaft verwenden und damit den Stack verfälschen.

- *Kapselung* der Implementation *(Information hiding):*

  Wenn man dem Benützer Zugriffe auf **`data`** und **`top`** erlaubt, kann man die Klasse **`Stack`** nicht mehr zu Optimierungszwecken re-implentieren.

Die Lösung besteht in einer *Zugriffskontrolle:*

Der Zugriff auf jedes Klassenelement kann in drei Stufen gewährt werden:

- **`private:`** für Klassen-Elemente die nur intern (d.h. in Definitionen von Elementfunktionen) verwendet werden dürfen
- **`public:`** für erlaubte Zugriffe von aussen (solche Elemente bilden die Schnittstelle der Klasse)
- **`protected:`** wie **`private`**, aber zugreifbar auch für abgeleitete Klassen (siehe später).

In unserem Beispiel **`Stack`** sieht dies so aus:

```
struct Stack {
   private:
      int size;
      char* data;
      char* top;
   public:
      Stack() {
      ...
```

Statt **struct** kann das Schlüsselwort **class** verwendet werden.

Unterschied:

- Bei **class** ist **private** vordefiniert,
- bei **struct** dagegen **public**.

Es gibt Situationen wo ein neues Objekt vom Typ **`Stack`** erzeugt wird, ohne dass einer der zwei bis jetzt definierten Konstruktoren aufgerufen wird.

Dies geschieht namentlich bei

- Deklaration mit Initialisierung
- Zuweisung

Hier wird ein Copy-Konstruktor resp. ein Zuweisungs-Operator aufgerufen.

Default- und Copy-Konstruktor, Zuweisungs-Operator und Destruktor müssen nicht explizit programmiert werden.

C++ stellt diese wenn nötig zur Verfügung.

Sobald aber im Konstruktor dynamisch Speicher alloziert wird, sollten alle vier Funktionen explizit programmiert werden.

Der Copy-Konstruktor kann so definiert werden:

```
Stack(const Stack& s) {
      size = s.size;
      data = new char[size];
      top = data;
      char* top0 = s.data;
      while (top0 < s.top) *top++ = *top0++;
}
```

Er wird automatisch aufgerufen bei einer Initialisierung:

```
Stack a;
Stack c = a;
```

Und der Zuweisungs-Operator kann so definiert werden:

```
Stack& operator=(const Stack& s) {
      size = s.size;
      delete[] data;          // !!!
      data = new char[size];
      top = data;
      char* top0 = s.data;
      while (top0 < s.top) *top++ = *top0++;
      return *this;
}
```

Er wird automatisch aufgerufen bei einer Zuweisung:

```
Stack a;
Stack c;
c = a;    // c.operator=(a);
```

stack4.cpp

Andere Anwendung des Stacks:
"Türme von Hanoi" mit Iteration statt Rekursion.

Statt die Funktion rekursiv aufzurufen, werden die drei "Teilaufgaben" (in umgekehrter Reihenfolge!) auf einen "Aufgabenstack" geladen.

```
int main()
{
  Stack s(200);

  s.push(5);  // Anzahl Scheiben
  s.push(0);  // Turm "von"
  s.push(1);  // Turm "ueber"
  s.push(2);  // Turm "nach"
```

```
while (!s.empty()) {
   // Hole 4 oberste Zahlen
   int nach   = s.pop();
   int ueber  = s.pop();
   int von    = s.pop();
   int anzahl = s.pop();
   if (anzahl == 1) {
     cout << von << " -> " << nach << "\n";
   }
   else {
     // Teilaufgabe 3:  Turm ohne unterste
     // Scheibe von Zwischenplatz
     s.push(anzahl-1);
     s.push(ueber);
     s.push(von);
     s.push(nach);
```

```
            // Teilaufgabe 2:  Unterste Scheibe
            // allein
            s.push(1);
            s.push(von);
            s.push(ueber);
            s.push(nach);

            // Teilaufgabe 1:  Turm ohne unterste
            // Scheibe auf Zwischenplatz
            s.push(anzahl-1);
            s.push(von);
            s.push(nach);
            s.push(ueber);
        }
    }
}
```

stackHanoi.cpp

Verbleibende offensichtliche Probleme von **`Stack`** sind:

- fehlende Überlaufbehandlung
- unbenutzter Speicherplatz

Sie wären lösbar mit verketteter Liste statt Array, aber:

- Speicherplatzverschwendung für die Zeiger.

Eine gute Implementation ist etwas komplizierter.

Die Standardbibliothek bietet eine solche, allerdings:
nicht als Klasse, sondern als Klassen-*Template*.

Vorteil des (später erklärten) Templates: Es definiert nicht nur einen Stack von **`char`** (oder **`int`**), sondern Stacks basierend auf beliebigen (auch eigenen) Datentypen.

Beispiel "Datum"

Wir definieren eine Klasse **`Datum`** mit

- Elementen: **`tag, monat`** und **`jahr`**
- Operationen:

  Vortag, Folgetag, Wochentag, formatierte Ausgabe.

Auf die Elemente **`tag, monat`** und **`jahr`** soll nicht zugegriffen werden dürfen.

Dies verunmöglicht fehlerhafte Datumsberechnungen wie

```
Datum gestern = heute; gestern.tag--;
```

oder

```
stich.tag   = 29;
stich.monat = 2;
stich.jahr  = 2002;
```

```
class Datum
{
  int tag, monat, jahr;
 public:
  // Konstruktoren
  Datum(int, int, int); // Tag, Monat, Jahr
  Datum(int, int);      // T., M., (aktuelles J.)
  Datum(int);           // Tag, (akt. M. und
  Datum();              // heute (default ctor)
  Datum(const char *);  // Datum aus C-String

  // Methoden
  Datum vortag();
  Datum folgetag();
  int wochentag();   // 0: Sonntag, 1: Montag,...
  void ausgabe(int); // Ausg. in versch. Formaten
};
```

Hier wurde eine alternative Syntax verwendet:

- Innerhalb des *Klassenblocks* werden die Element-funktionen nur *deklariert*,
- *definiert* werden sie dann ausserhalb, wobei dann der Bezugsoperator `::` benötigt wird.

Oft ist es bequem, *mehrere Konstruktoren* zu haben. Die ersten vier Konstruktoren kann man aber ersetzen durch einen einzigen, unter Verwendung von *Default-Argumenten*.

Default-Argumente können beim Aufruf ganz oder teilweise (von hinten nach vorn) weggelassen werden.

Die Klasse **Datum** vereinfacht sich damit zu:

```
class Datum
{
  int tag, monat, jahr;
 public:
  // Konstruktoren
  Datum(int t=0, int m=0, int j=0);
  Datum(const char*);  // Datum aus C-String

  // Methoden
  Datum vortag();
  Datum folgetag();
  int wochentag(); // 0: Sonntag, 1: Montag, ..
  void ausgabe();  // formatierte Ausg. auf cout
};
```

Es fehlen noch die *Definitionen*.

Der erste Konstruktor kann etwa so definiert werden:

```
Datum::Datum(int t, int m, int j)
{
  tag   = t!=0 ? t : aktueller_tag();
  monat = m!=0 ? m : aktueller_monat();
  jahr  = j!=0 ? j : aktuelles_jahr();
  // Pruefe Gueltigkeit des Datums
  // ...
}
```

Die übrigen Definitionen lassen wir der Kürze halber weg.

Überladene Operatoren

Neues Beispiel: Eine Klasse **Bruch** für das Rechnen mit Brüchen kann so deklariert werden:

```
class Bruch {
    int zaehler, nenner;
  public:
    Bruch(int z=0, int n=1);
    double wert();
    Bruch operator+(const Bruch& b);
    Bruch operator-(const Bruch& b);
    Bruch operator*(const Bruch& b);
    Bruch operator/(const Bruch& b);
    Bruch operator+=(const Bruch& b);
    // etc.
};
```

Die Elementfunktion **operator+** wird dann so definiert:

```
Bruch Bruch::operator+(const Bruch& b) {
  return Bruch(
     zaehler * b.nenner + nenner * b.zaehler,
     nenner * b.nenner);
}
```

Und der entsprechende Aufruf kann z.B. lauten

```
Bruch c = a.operator+(b);
```

Oder aber, in der schöneren (und symmetrischen!) Schreibweise:

```
Bruch c = a + b;
```

Fast alle Operatoren von C++ können durch Klassenoperationen *überladen* werden, d.h. **a OP b** wird als Kurznotation für **a.operatorOP(b)** verstanden.

In der Definition von **operator+** wird ein Konstruktor aufgerufen und damit ein neues **Bruch**-Objekt erzeugt.

Bei einem Operator **operator*=** ist dies nicht nötig, da der Wert des aktuellen Objektes überschrieben werden darf:

```
Bruch Bruch::operator*=(const Bruch& b) {
  zaehler *= b.zaehler;
  nenner  *= b.nenner;
  return ??;
}
```

Was ist aber der Rückgabewert?

Bei einem Aufruf **a *= b** erwartet man das (veränderte) **a** als Rückgabewert.

Wie spricht man nun aber **a** von innerhalb der Funktionsdefinition an?

a ist ja das *implizite Argument*, das beim Aufruf **a.operator*=(b)** (wofür **a *= b** Kurzschreibweise ist) an die Funktion mitgegeben wird.

Mit **zaehler** und **nenner** kann man zwar **a.zaehler** und **a.nenner** ansprechen, man möchte aber das Objekt als Ganzes.

Zu diesem Zweck stellt C++ einen Zeiger **this** zur Verfügung, der beim Methoden-Aufruf automatisch auf das aktuelle Objekt gesetzt wird.

Die gesuchte Anweisung heisst somit: **return *this;**

Für die Ein- und Ausgabe (z.B. von **cin** resp. auf **cout**) kann man die Operatoren **>>** und **<<** überladen.

Weil das erste Argument von **<<** aber vom Typ **ostream** ist, kann man *keine Elementfunktion* verwenden (dazu müsste die Klasse **ostream** erweitert werden), sondern man braucht eine *gewöhnliche Funktion:*

```
ostream& operator<<(ostream&, Bruch&);
```

Die Funktion braucht Zugriff auf die Klassen-Elemente **zaehler** und **nenner**, die aber **private** sind. Dieses Zugriffsrecht kann beibehalten werden, wenn man die Funktion **operator<<** zum **friend** der Klasse **Bruch** macht.

bruch.cpp

Friends

Eine Friend-Funktion wird im Klassenblock deklariert mit vorangestelltem Schlüsselwort **friend**. Im Beispiel:

```
class Bruch {
    int zaehler, nenner;
  public:
    // wie bisher
    friend ostream& operator<<(ostream&, Bruch&);
};
```

Die Friend-Funktion hat dann Zugriff auf alle, auch **private**, Elemente der Klasse.

Die Friend-Funktion wird dann ausserhalb der Klasse **Bruch** definiert:

```
ostream& operator<<(ostream& s, Bruch& b)
{
  s << b.zaehler << "/" << b.nenner;
  return s;
}
```

Die neue Funktion erlaubt die bequeme Ausgabe von Brüchen wie z.B. **cout << b << endl;**

Die Verkettung funktioniert korrekt, weil **operator<<** das erste Argument **s** wieder zurückgibt (der *Wert* des Ausdrucks **cout << b** ist **cout**), und weil der Operator linksassoziativ ist.

Strings

Ein schönes Beispiel für überladene Operatoren ist auch die Klasse **`string`** aus der Standardbibliothek.

Die Klasse enthält mehrere *Konstruktoren*. Die wichtigsten sind hier demonstriert:

```
#include <string> // statt cstring (C-Strings)
string s1("abc"); // String aus C-String
string s2(5,'z'); // erzeugt "zzzzz"
string s3(s2);    // Copy-Konstruktor
string s4;        // Default-K., leerer String
```

Die *Zuweisungsoperatoren* **=** und **+=** (String anhängen) sind ebenfalls überladen. Die rechte Seite kann ein **string**, ein C-String oder ein **char** sein. Mit

```
s += c;
```

wird z.B. ein Zeichen **c** and den String **s** angehängt.

Der Benützer von **string** muss sich nicht um Speicher-Allokation und -Freigabe kümmern. Dies wird von der Klasse selber gehandhabt.

Beim Anhängen an einen String reicht der allozierte Speicherplatz manchmal nicht aus, so dass kopiert werden muss. In diesem Fall wird ein Speicherbereich alloziert, der eine gewisse "Reserve" einschliesst.

Die *effektive* Länge eines Strings kann mit **`s.length()`** abgefragt werden, **`s.capacity()`** liefert dagegen die *allozierte* Länge (die aber selten von Interesse ist).

Für lexikographische Vergleiche wurden die Operatoren **`<`**, **`<=`**, **`==`**, **`!=`**, **`>=`** und **`>`** überladen.

Für den Zugriff auf das i-te Zeichen mittels **`s[i]`** wurde der Operator **`[]`** überladen.

Es existieren weitere nützliche Funktionen. Beispiele sind:

```
string s = "ABCDEFGH";
s.c_str();              // wandelt um in C-String
s.find("CD");           // liefert 2 (Position)
s.find("CC");           // liefert -1
s.substr(4,3);          // liefert EFG
```

Die folgenden Funktionen verändern `s`:

```
s.replace(4,3,"xy");// liefert ABCDxyH
s.erase(2,1);       // liefert ABDxyH
```

string.cpp

Für die Ein- und Ausgabe von Strings wurden die Operatoren **>>** und **<<** überladen, ähnlich wie im Beispiel **Bruch** gezeigt:

```
operator>>(istream&, string&);
operator<<(ostream&, string&);
```

Die für C-Strings existierende Elementfunktion **getline** von **istream** wurde in **string** nachgebildet durch eine gewöhnliche (Friend-)Funktion.

Statt

```
cin.getline(str, 10);
```

heisst es deshalb für **string**s:

```
getline(cin, str);
```

Vererbung

Ein mächtiges Instrument der objektorientierten Programmierung ist die *Vererbung*.

Von einer *Basisklasse* ausgehend, können neue Klassen *abgeleitet* werden, wobei Datenelemente und Elementfunktionen vererbt werden.

Dadurch wird die *Wiederverwendbarkeit* von Programmcode unterstützt.

Als Basisklasse wollen wir eine Klasse **Punkt** verwenden:

```
class Punkt {
  protected:
    double x, y;
  public:
    Punkt(double X=0, double Y=0) { x=X; y=Y; }
    void moveto(double X, double Y) { x=X; y=Y; }
    double abstand(Punkt B) {
        return sqrt(SQR(x-B.x) + SQR(y-B.y)); }
    friend ostream& operator<<(ostream&, Punkt&);
};

ostream& operator<<(ostream& s, Punkt& P)
{
    s << "(" << P.x << "," << P.y << ")";
    return s;
}
```

Wenn nun eine Klasse **Kreis** benötigt wird, kann man diese von **Punkt** ableiten, indem man ein Element **radius** sowie neue Elementfunktionen hinzufügt.

Der Vorteil ist: Die Methoden **moveto** und **abstand** können von der Basisklasse übernommen werden.

Die Syntax der Klassenableitung ist

```
Klassenart neue Klasse : Zugriffsrecht
    Basisklasse { neue Elemente };
```

Die abgeleitete Klasse **Kreis** könnte deshalb etwa so aussehen:

```
class Kreis : public Punkt {
    double radius;
  public:
    Kreis(double X=0, double Y=0, double R=1)
        : Punkt(X,Y) { radius = R; }
    bool istInnen(Punkt P)
        { return abstand(P) < radius; }
    friend ostream& operator<<(ostream&, Kreis&);
};

ostream& operator<<(ostream& s, Kreis& K)
{
    s << "Mittelpunkt (" << K.x << "," << K.y <<
        "), Radius = " << K.radius;
    return s;
}
```

punkt.cpp

Bemerkungen:

- Das Zugriffsrecht nach dem Doppelpunkt, darf fehlen. Es wird dann **`private`** angenommen. Es dient dazu, die Zugriffsrechte auf Elemente der Basisklasse *weiter einzuschränken*. Im Normalfall verwendet man **`public`**.
- Der *Konstruktor* von **`Kreis`** hat eine einzige Anweisung **`radius = R;`**.
- Es gilt aber: der Konstruktor ruft *zuerst implizit* den Konstruktor der Basisklasse auf (und dieser zuerst den der Basis-Basis-Klasse, etc.).
- Und wenn die Basisklasse mehrere Konstruktoren hat? Dann kann man den gewünschten Konstruktor-Aufruf nach der Parameterliste (und einem Doppelpunkt) angeben, wie im Beispiel gemacht.

- Analog dazu ruft der Destruktor *zuletzt* implizit den Destruktor der Basis-Klasse auf.
- Im Beispiel sind keine Destruktoren programmiert, daher werden in beiden Klassen die automatisch erzeugten Destruktoren verwendet.
- In **`istInnen`** wird die Methode **`abstand`** benützt, die von der Basisklasse zur Verfügung gestellt wird.
- In der überladenen Funktion **`operator<<`** werden die von **`Punkt`** geerbten Elemente **`x`** und **`y`** benützt. Das ist möglich, weil diese in **`Punkt`** als **`protected`** deklariert sind.

Objekte in einer Klassenhierarchie sind *polymorph:*
Ein **Kreis** kann immer auch als ein **Punkt** angeschaut werden.

Beispiel:

```
Punkt* arr[4];
arr[0] = new Punkt;  arr[1] = new Kreis;
arr[2] = new Kreis;  arr[3] = new Punkt;
for (int i = 0; i < 4; i++) arr[i]->print();
```

Hier sei **print()**eine zusätzliche Methode von **Punkt**, z.B.

```
void print() {
   cout << "Punkt: " << *this << endl;
}
```

basierend auf dem vorher definierten Operator **<<**.

Die Methode **print** wird an **Kreis** vererbt.

Möchte man stattdessen in **Kreis** die analoge Methode

```
void print() {
    cout << "Kreis: " << *this << endl;
}
```

definieren, dann muss die Basisklasse das *Überschreiben* der Methode erlauben.

Dies geschieht mit dem Schlüsselwort **virtual**:

```
virtual void print() {
    cout << "Punkt: " << *this << endl;
}
```

Eine *virtuelle Methode* wird nur dann vererbt, wenn die abgeleitete Klasse keine Methode mit der *gleichen Signatur* hat.

Im Gegensatz dazu wird eine *rein virtuelle Methode* gar nie vererbt.

- Sie wird nur deklariert, aber *nicht definiert*.
- Sie muss in *jeder* abgeleiteten Klasse definiert werden.
- Sie wird gekennzeichnet durch ein `= 0` in der Deklaration, z.B.:

```
virtual void print() = 0;
```

Eine Klasse, die eine rein virtuelle Methode enthält heisst *abstrakte Basisklasse*.

Beispiel: Man möchte neben Kreisen noch Klassen für weitere geometrische Figuren wie **`Rechteck`** etc. definieren.

- Dann ist **`Punkt`** keine geeignete Basisklasse.
- Besser ist eine abstrakte Basisklasse **`Figur`**.
- Methoden wie **`flaecheninhalt`** oder **`zeichneMich`** sind nun typische *rein virtuelle* Methoden, da sie für die Basisklasse nicht sinnvoll definiert werden können.
- Sie werden stattdessen für *alle* abgeleiteten Klassen definiert.

Von einer abstrakten Basisklasse darf keine *Instanz* gebildet werden. Das bedeutet: Es darf kein Objekt dieser Klasse deklariert werden!

Der Grund liegt darin, dass Objekte keine undefinierten Methoden haben dürfen.

Objekte dürfen also nur von abgeleiteten Klassen gebildet werden.

Dagegen sind *Zeiger* auf eine abstrakte Basisklasse erlaubt. Sie ermöglichen die einheitliche Behandlung von Objekten verschiedener abgeleiteter Klassen.

Beispiel:

```
Figur* figur[N];
// …
for (int i = 0; i < N; i++) {
     figur[i]->zeichneMich();
}
```

Welches ist nun die bei `figur[i]->zeichneMich();` aufgerufene Methode?

Dies hängt vom Typ von `figur[i]` ab, der *erst zur Laufzeit* feststeht. Der Compiler muss also den Aufruf einer noch nicht feststehenden Funktion vorsehen. Man spricht hier von *dynamischer Bindung*.