Skript Programmieren C++

Klaus Kusche

0. Kapitel: C++

C++ ist (fast) eine *objektorientierte Erweiterung* von C
(fast alle C-Programme sind auch gültiges C++).

C++ ist:

- Groß (bezüglich Möglichkeiten) und sehr flexibel (mächtiger als Java, C#, ...)
- Performance-orientiert und -optimiert (alle "langsamen"
  bzw. Overhead verursachenden Konstrukte kann man weglassen)
- Nur für erfahrene Programmierer gedacht (ohne Rücksicht auf Anfänger),
  bei fortgeschrittener Anwendung komplex und oft schwierig zu verstehen.
- Zur Laufzeit ebenso unsicher wie C (Arrays, Pointer, expl. Speicherverwaltung, ...).

1. Kapitel: Unterschiede C / C++

Die wichtigen Erweiterungen von C++ gegenüber C lernen wir im Laufe des Semesters.

Die unwichtigen Erweiterungen sowie die inkompatiblen Änderungen sind Folgende
(Liste nicht vollständig, es gibt › 50 Unterschiede im Detail):

- **File-I/O:**

  C++ verwendet anderes File-I/O: **`cin >>`** ... und **`cout <<`** ... (bzw. **`cerr <<`** ...),
  `endl` für **`\n`** plus Buffer-Flush, Header **`iostream`** (+ **`iomanip`** für Formatierung).
  C **`stdio.h`** (**`printf`**, ...) funktioniert auch in C++, aber bitte nicht mischen!

  C++ File-I/O werden wir nicht im Detail lernen...

- **Dynamische Speicherverwaltung:**

  C++ verwendet statt **`malloc`** und **`free`** eigene Sprachkonstrukte
  zur dynamischen Speicherverwaltung: **`new`** und **`delete`**

  Lernen wir später...

- **`bool`**:

  `bool` ist in C++ *ohne Header fix definiert*, man braucht keinen **`stdbool.h`** .

- **Headerfiles:**

  C++-Headerfiles haben kein **`.h`** im Namen, z.B. **`#include <iostream>`**

  Zu fast jedem Standard-C-Header gibt es einen äquivalenten Standard-C++-Header
  (ohne **`.h`**, dafür mit **`c`** vorn dran):
  `#include <string.h>` ==› **`#include <cstring>`**
  `#include <stdlib.h>` ==› **`#include <cstdlib>`** usw.

  Diese definieren meist dieselben Funktionen, Konstanten, ... wie die
  entsprechenden C-Header, aber im Namespace **`std`** (nächster Punkt).

- **Namespaces:**

  Namen können in C++ explizit einem *benannten Namespace* zugeordnet werden, um die Namen eines Projektes, einer Library usw. zusammenzufassen und Namenskollisionen mit Code aus anderen Quellen zu verhindern.

  Derselbe Name kann unabhängig voneinander in *mehreren* Namespaces vorkommen.

  Welchen Namen man jetzt meint, gibt man mit dem **Scope-Operator `::`** an.

  Beispiel:
  `std::cin` ist das vordefinierte **`cin`** aus dem Header **`iostream`**,
  `myProj::cin` ist das **`cin`** aus einem eigenen Namespace **`myProj`**.

  Alle *vordefinierten Namen* (aus den Standard-C++-Headern) gehören zum *Namespace* **`std`**. Da es lästig wäre, vor jeden solchen Namen jedesmal **`std::`** zu schreiben, kann man mit **`using`** einmal oben im Programm festlegen, dass alle Namen ohne explizites **`::`** automatisch auch im angegebenen Namespace gesucht werden sollen:
  `using namespace std;`
  (viele Firmen bestehen aber auf **`std::`** statt **`using`**)

  Alternativ kann man auch ein **`using`** auf einzelne Namen machen, die man dann ohne **`::`** davor verwenden kann:
  Nach **`using std::string;`** kann man **`string`** statt **`std::string`** schreiben.

  Den Scope-Operator **`::`** *ohne* Namespace davor kann man auch verwenden, um *explizit auf globale Variablen* zuzugreifen, wenn es lokale Variablen gleichen Namens gibt: **`::size`** ist beispielsweise die globale Variable **`size`**.

  Es gibt auch *namenlose (anonyme) Namespaces*, die die Sichtbarkeit von Namen begrenzen (auf die Namen solcher Namespaces kann von außerhalb nicht zugegriffen werden, das soll das Konzept von File-**`static`**-Variablen ablösen).

  Wie man Namespaces selbst anlegt und eigene Namen einem Namespace zuordnet, lernen wir nicht.

- **Prototypen:**

  Prototypen für alle Funktionen sind in C++ *Pflicht*!

- **Deklarationen im Code / im `for`:**

  In C++ kann man lokale Deklarationen und Befehle in beliebiger Reihenfolge schreiben (in C seit C99 auch, nur verwendet das kaum jemand).

  Weiters ist es möglich und üblich, *Schleifenvariablen* direkt im **`for`** zu deklarieren:
  `for (int i = 0; i < n; ++i) {` …

  Achtung: Hier hat sich der C++-Standard geändert!!!

  - *Früher:* Die Deklaration galt bis zum Ende der **`{ }`**, in denen das **`for`** stand.
    ==> Man konnte die im **`for`** deklarierte Variable *nach* dem **`for`** noch prüfen!
  - *Heute:* Die Deklaration gilt nur bis zum Ende der **`for`**-**`{ }`** !
    ==> Die Schleifenvariable ist unmittelbar nach dem **`for`** *unbekannt*!

- **Keine variabel großen Arrays:**

  In C++ muss die Größe von Arrays in Deklarationen *konstant*
  (d.h. zur Compile-Zeit berechenbar) sein, so wie früher in ganz altem C:
  Variabel große Arrays (z.B. `int data[argc];`) sind verboten!

  Auch der Trick, als letztes Member einer **`struct`** ein Array unbekannter Größe
  (nur **`[]`**) anzugeben (und die **`struct`** dann mit **`malloc`** in variabler Größe
  zu allokieren), ist in C++ *nicht* möglich.

- **`struct`-Tags sind automatisch Typnamen:**

  In C++ wird bei einer **`struct`**-, **`union`**- oder **`enum`**-Deklaration *automatisch*
  ein **`typedef`** mit dem angegebenen **`struct`**-Tag gemacht.

  Man kann also (*ohne* selbst ein **`typedef`** zu machen) das **`struct`** (bzw.
  das **`union`** oder **`enum`**) bei Variablen- und Parameter-Deklarationen usw. *weglassen*
  und nur das **`struct`**-Tag als Typ angeben.

  Achtung: Das gilt erst *nach* der **`struct`**-Deklaration. *Innerhalb* der Deklaration
  ist das **`struct`** nach wie vor notwendig (z.B. für Pointer auf sich selbst).

- **Typ `auto`:**

  Ab *C++ 11* kann das Wort **`auto`** als *Typ* verwendet werden (wie **`int`** oder **`bool`**),
  wenn es sich um eine *Deklaration mit Initialisierung* handelt:

  ```
  auto x = ... ;
  ```

  In diesem Fall bekommt die neue Variable **`x`** automatisch den Typ,
  den der *Initialisierungswert* … rechts vom **`=`** hat.

  `auto` hatte in den allerersten Versionen von C eine ganz andere
  (seit langem funktionslose) Bedeutung, diese wurde mit C++ 11 entfernt.
  Alte **`auto`** einfach aus dem Code entfernen!

- **`void*` muss gecastet werden:**

  In C darf man einen **`void*`**-Pointer ohne Cast auf einen beliebigen anderen
  Pointer zuweisen und umgekehrt oder als Argument für Parameter beliebigen
  Pointer-Typs übergeben.

  In C++ ist dafür *ein Cast vorgeschrieben*.

- **`nullptr`** statt **`NULL`:**

  Ein **`void*`**-Pointer kann in C++ nicht ohne expliziten Typcast auf einen anderen
  Pointer zugewiesen werden. In vielen Headern (im Besonderen alten C-Headern)
  ist **`NULL`** aber noch als **`((void *) 0)`** definiert. Daher gibt es ein Typ-Problem,
  wenn man **`NULL`** auf Pointer eines "echten" Pointer-Typs zuweisen,
  als Parameter übergeben oder mit anderen Pointern vergleichen will.

  Im neuen C++-Standard C++ 11 wurde ein eigenes Konstrukt statt **`NULL`** eingeführt:
  `nullptr`. Es sollte konsequent statt **`NULL`** verwendet werden!

  Wenn **`nullptr`** oder das vordefinierte **`NULL`** Fehler liefern (und nur dann!),
  ist es derzeit noch am besten, entweder **`NULL`** einfach als **`0`** neu zu definieren
  oder *im Code direkt* **`0`** statt **`NULL`** zu verwenden.

- **Typ-Casts:**

  Es gibt in C++ nicht nur einen Typ-Cast, sondern gleich 6 verschiedene:

  - Den **bisherigen C-Cast** **`(`**(*neuer_typ*)**`)`** *wert* , der in C++ zwar erlaubt ist, aber nicht verwendet werden sollte: Er wird intern in einen **`static_cast`**, **`const_cast`** oder **`reinterpret_cast`** verwandelt, und man sollte zur Sicherheit und besseren Lesbarkeit gleich den **`..._cast`** hinschreiben, den man haben will.
  - Casts, die aussehen und funktionieren wie **Funktionen**: *neuer_typ*(*wert*) (also z.B. **`double(n)`**).

    In C++ können dabei neben den *vordefinierten Typumwandlungen* auch *selbstdefinierte Konvertierungsfunktionen* aufgerufen werden.
  - 4 spezielle Sprachkonstrukte für Casts: **`static_cast`**, **`dynamic_cast`**, **`reinterpret_cast`** und **`const_cast`**
    - **`static_cast`**`<`*neuer_typ*`>` **`(`***wert*`)` entspricht in etwa dem alten C-Cast: Zur Laufzeit wird nichts geprüft und der Wert auch nicht verändert, sondern nur das bisherige Bitmuster als Wert eines anderen Typs betrachtet (außer bei Vorhandensein eines expliziten Typumwandlungs-Operators oder bei einer Umwandlung zwischen verschiedenen numerischen Typen).

      `static_cast` ist *nur in "harmlosen" Fällen erlaubt*: Zwischen "verwandten" Pointer-Typen (siehe später), zwischen numerischen Typen, und zwischen Typen, für die ein Typumwandlungs-Operator existiert. In allen anderen Fällen sollte der Compiler einen Fehler liefern, ebenso, wenn **`static_cast`** ein **`const`** entfernen würde.

      `static_cast` auf Pointern ist potentiell gefährlich: Man soll es nur verwenden, wenn man genau weiß, dass der Pointer immer auf ein Objekt des "richtigen" Typs zeigt und die Typänderung sinnvoll ist.
    - **`reinterpret_cast`**`<`*neuer_typ*`>` **`(`***wert*`)` ist die "Brutal-Variante" von **`static_cast`** mit weniger Compiler-Prüfungen für die "verbotenen" Fälle: Es ist auch zwischen Pointer-Typen erlaubt, die nichts miteinander zu tun haben, und zwischen Pointer und **`int`** bzw. **`long`** und umgekehrt (aber wieder nur ohne **`const`**-Änderung!).

      In beiden Fällen bleibt der Wert unverändert und wird nur anders interpretiert. Das Ergebnis von **`reinterpret_cast`** ist in einigen Fällen plattform-abhängig und daher nicht portabel.
    - **`dynamic_cast`**`<`*neuer_typ*`>` **`(`***wert*`)` ist die sichere, aber langsamere Variante von **`static_cast`** für die Konvertierung zwischen "verwandten" Pointern: Es wird *zur Laufzeit* geprüft, ob der aktuelle *wert* wirklich als *neuer_typ* verwendet werden kann. Auch hier dürfen sich die **`const`**'s nicht ändern.

      Details zu **`dynamic_cast`** später.

- **`const_cast`**`<`*neuer_typ*`>` **`(`***wert***`)`** ändert nicht den Typ selbst, sondern darf ausschließlich entweder zu einem Wert eines nicht-**`const`**-Typs ein **`const`** dazutun oder -- viel wichtiger! -- von einem **`const`**-Typ-Wert das **`const`** (oder analog das **`volatile`**) entfernen.

- **`enum`-Werte sind nicht `int`:**

  Werte von **`enum`**-Typen sind in C++ eigene, von **`int`** *verschiedene* Typen (und auch bezüglich interner Darstellung nicht unbedingt **`int`**'s).

  - Man kann zwischen verschiedenen **`enum`**'s und zwischen **`int`** und **`enum`**-Werten *nicht ohne Cast* zuweisen.
  - **`sizeof`** eines **`enum`** ist *nicht unbedingt gleichgroß* wie **`sizeof(int)`** (auch **`sizeof(bool)`** ist nicht unbedingt **`sizeof(int)`** !).

    Verschiedene **`enum`**-Typen können verschiedene auch **`sizeof`** haben.

- **Effizienz von `const`:**

  Mit **`const`** deklarierte Variablen mit "einfachem" Typ (**`int`**, **`double`**, **`enum`**, ...) sind in C++ wirklich *Konstanten* (im Unterschied zu C, wo es *read-only-Variablen* sind), d.h. sie werden direkt im Code eingesetzt und belegen keinen Speicherplatz.

  Damit sind sie so effizient wie **`#define`**-Konstanten, aber eleganter und typsicher. **`#define`** sollte daher in C++ nicht mehr für Wert-Konstanten verwendet werden.

- **Sichtbarkeit von globalen `const`-Werten:**

  Globale **`const`**-Deklarationen sind in C wie alle anderen globalen Deklarationen (ohne **`static`**) auch in anderen Files sichtbar.

  In C++ sind *globale* **`const`***-Deklarationen implizit* **`static`***, d.h. auf den eigenen File beschränkt*. Sollen sie in anderen Files sichtbar sein, muss man explizit **`extern`** davorschreiben.

  Grund: In C wird für **`const`**-Variablen zumindest theoretisch genauso wie für normale Variablen Speicher angelegt. Der Linker kann daher Code aus anderen Object-Files mit diesem Speicher verbinden. In C++ belegen **`const`**-Variablen von Basistypen per Definition *keinen* Speicher und sind daher für den Linker *unsichtbar*.

- **`extern "C"`**:

  C-Funktionen und C++-Funktionen werden *intern unterschiedlich übersetzt*.
  Will man in einem C++-Programm Funktionen aufrufen, die mit dem C-Compiler übersetzt wurden, so muss man das dem Compiler sagen:

```
extern "C" void myCfunc(int n);
```

  Soll ein C-Header mit Prototypen für C und C++ verwendbar sein, sieht das wie folgt aus:

```
#ifdef __cplusplus
extern "C" {
#endif
  int Cfunc1(int n);
  void Cfunc2(char *str);
  ...
  ...
```

```
#ifdef __cplusplus
}   // schließende Klammer vom extern "C"
#endif
```

Auch *Function Pointer* muss man **extern "C"** deklarieren, wenn sie auf C-Funktionen zeigen.

2. Kapitel: Erweiterungen für Funktionen

Default-Werte

Man kann in C++ für Funktionsparameter im Funktionskopf Default-Werte angeben.
Die Funktion kann dann aufgerufen werden, ohne einen Argumentwert
für diese Parameter anzugeben (d.h. *mit weniger Argumenten als Parametern*).
Die Parameter werden in diesem Fall *mit dem Default* statt einem Argument *initialisiert*.

Beispiel:
```
void drawLine(int width = 80, char c = '=') { … }
```

Gültige Aufrufe:
```
drawLine();
drawLine(40);
drawLine(40, '*');
```

Achtung:
Die Default-Parameter müssen immer die *rechten (letzten) Parameter* sein:
Nach einem Default-Parameter darf kein nicht-Default-Parameter mehr stehen!

Default-Parameter dürfen nicht in Prototyp und Definition stehen,
sondern (normalerweise) *nur* im Prototyp.

Default-Parameter müssen keine Konstanten sein:
Es sind auch Ausdrücke (z.B. mit globalen Variablen oder statischen Membern) erlaubt.

Überladen von Funktionen

In C++ darf es *beliebig viele Funktionen desselben Namens* geben,
solange man sie *anhand ihrer Parameter-Typen eindeutig unterscheiden* kann.

Bei jedem Funktionsaufruf sucht der Compiler in *zwei Durchläufen* die passende Funktion:

- Zuerst einmal sucht er nach einer *genau* zu den Argument-Typen passenden Funktion.
- Dann sucht er nach einer *mit impliziten Typkonvertierungen* der Parameter passsenden Funktion.

Man kann also beispielsweise im selben Programm sowohl ein
`double mittel(double a, double b)` als auch ein `int mittel(int a, int b)` deklarieren.

Achtung:

- Der Returntyp ist *kein* Unterscheidungskriterium, nur die Argument-Typen!
- Man muss auch *alle Kombinationen von Default-Parametern* berücksichtigen:
  `int func1(int n, char *txt = "")` und
  `int func1(int n, double x = -1)` kollidieren!
  (beide passen auf einen Aufruf mit einem einzigen `int`-Argument!)

Referenzen

In C++ gibt es neben Pointern auch Referenzen, d.h. *Verweise* auf oder neue Namen für bereits bestehende Variablen. Ihr Hauptzweck ist *Call by Reference* und *Return By Reference*, d.h. sie werden vor allem als *Parametertypen* und *Returntypen* verwendet.

Intern sind Referenzen Pointer, aber sie sind viel einfacher zu verwenden:

- Deklariert werden Referenzen im Funktionskopf mit **&** statt *, also z.B.
  `bool delete(int key, double &val);`
  (**`val`** wird "by Reference" übergeben und kann daher auch als *Ausgabe-Parameter* verwendet werden).
- Zugreifen kann man im Code der Funktion auf eine Referenz wie auf eine *ganz normale Variable* desselben Typs, d.h. *ohne Pointer-Dereferenzierung* (* oder ->).
- Beim Aufruf wird einem Referenz-Parameter auch eine *ganz normale Variable* (oder ein Array-Element oder **`struct`**-Member oder was immer) übergeben, *keine Adresse*. Man braucht also *kein* **&** im Aufruf!

Referenzen unterscheiden sich in wesentlichen Dingen von Pointern:

- Sie können *nie* **NULL** sein, sondern zeigen immer auf einen echten Speicherplatz.

  Eine Gefahr bleibt allerdings: Gibt man aus einer Funktion eine Referenz auf eine lokale Variable zurück, "überlebt" die Referenz die Variable und zeigt danach auf undefinierten oder fremden Speicherplatz.
- Sie sind *konstante* Pointer: Sie bekommen beim Anlegen die Adresse einer Variable und können danach *nicht mehr verändert* werden, solange sie bestehen (weil ja jede Zuweisung auf eine Referenz auf die ursprüngliche Variable wirkt und nicht auf die Referenz selbst).

  Wird eine normale Referenz-Variable (kein Parameter) deklariert, muss daher immer eine Initialisierung angegeben werden.

  Eine Referenz kann also z.B. nicht dazu benutzt werden, ein Array zu durchwandern.
- Eine **`const`**-Referenz kann auch auf den internen Speicherplatz einer Konstanten oder eines Zwischenergebnisses zeigen, man kann also einem **`const`**-Referenz-Parameter auch konstante Werte oder zusammengesetzte Ausdrücke übergeben.
- Es gibt keine Pointer auf Referenzen, keine Arrays von Referenzen und keine Referenzen von Referenzen.

Verwendet werden sie wie Pointer-Parameter:

- Für Parameter, bei denen der Wert im Aufrufer *geändert* werden soll.
- Für Parameter, die *"groß"* sind (*Strukturen, Objekte*) und daher aus *Effizienzgründen nicht kopiert* werden sollen (Objekte werden fast immer als Referenz und nicht "by Value" übergeben, auch wenn sie keine Ausgabe-Parameter sind: In diesem Fall eben als Referenz mit **`const`**).

Arrays werden nicht als Referenz, sondern wie bisher per Pointer übergeben!

Referenzen als Returnwert sind lvalues (links von = erlaubt)

- Wenn eine Funktion eine (nicht-**`const`**) Referenz auf ein Objekt als Returnwert zurückliefert, darf ein Aufruf dieser Funktion *auf der linken Seite einer Zuweisung stehen.*
- Der Effekt ist derselbe wie bei einer Funktion mit Pointer-Returnwert und **`*(func(…)) = …;`** : Der Wert auf der rechten Seite der Zuweisung wird in jenes Objekt gespeichert, auf das die Referenz zeigt, die als Returnwert von der Funktion zurückkommt.
- Beispiel: Eine Funktion **`find`** liefert eine Referenz auf ein zu suchendes Element einer Datenstruktur als Ergebnis, und mit **`find(…) = …;`** kann man direkt in das gefundene Element zuweisen.

3. Kapitel: Objektorientierte Programmierung, Grundlagen

Warum objektorientierte Programmierung?

Weiterentwicklung der "normalen" (prozeduralen) Programmierung, um

- *größere* Software-Systeme
  (= Hunderttausende / Millionen Zeilen, zehntausende Funktionen)
- schneller, fehlerfreier, klarer … entwerfen und programmieren zu können.

Im Detail:

- **Strukturierung, Modularisierung, Aufteilung auf mehrere Programmierer:**

  Probleme:

  - Zehntausende Funktionen sind *unüberschaubar*
    (Namenskollisionen, Dokumentation, …).
  - Die *Datenstrukturen* werden im Vergleich zum Code immer wichtiger, mehr und komplexer, sind aber im Code *nicht klar zu erkennen*.

  Zentrale Beobachtung:

  Es arbeiten nicht alle Funktionen mit allen Daten eines großen Softwaresystems!

  - Fast jedes Stück Code / jede Funktion arbeitet nur
    mit den Daten *einer* ganz bestimmten Datenstruktur.
  - Umgekehrt gehören zu jeder Datenstruktur meist nur
    einige wenige Funktionen, die diese Datenstruktur manipulieren.

  Lösung:

  - *Die Funktionen werden den Daten zugeordnet, auf denen sie arbeiten:*

    Eine Datenstruktur besteht aus den Daten *und den dazugehörigen Funktionen,*
    d.h. zu jedem Datentyp gehört auch der *Code* für die Operationen auf diesen Daten.

    Umgekehrt: (Fast) jede Funktion gehört zu einem Datentyp
    ==› es sollte möglichst wenig "globalen" Code außerhalb
    von Datenstrukturen geben!
  - Die Datenstruktur ist neues *Strukturierungskriterium*:
    - Die *Dokumentation* ist nach Datenstrukturen gegliedert.
    - Jede Datenstruktur bildet einen eigenen *Namensraum* für ihre Funktionsnamen (==› keine Namenskollisionen
      zwischen gleichnamigen Funktionen verschiedener Datenstrukturen!).
    - Der *Sourcecode* wird *nach Datenstrukturen in Files unterteilt*:
      Die Datenstruktur ist die Einheit der *Programm-Entwicklung*,
      große Programme werden Datenstruktur-weise
      auf mehrere Programmierer verteilt.

- **Kapselung:**

  Problem:

  - Es ist nicht klar erkennbar, *welcher Code wie auf welche Daten zugreift*.

  Lösung:

  - Möglichst *keine globalen Daten*.
  - Die Daten sind "*von außen*" (d.h. von Code, der nicht unmittelbar zu dieser Datenstruktur gehört) nicht sichtbar, *nicht direkt zugreifbar*:

    Nur "eigene" (= zur Datenstruktur gehörige) Funktionen dürfen direkt auf die Daten zugreifen!

    ==› Zugriffe von außen sind nur *durch Aufruf der "eigenen" Funktionen* der jeweiligen Datenstruktur möglich!
    (und damit nur über eine offizielle, dokumentierte Schnittstelle)

    ==› Eine Datenstruktur ist über *ihre Funktionalität / ihr Verhalten definiert* (über die Funktionen, die sie anbietet), und nicht über ihre internen Daten!

    Beispiel: Ein Stack hat die Zugriffsfunktionen **`push`** und **`pop`**
    ==› Versehentliches Einfügen / Löschen "mittendrin" ist dadurch verhindert, auch wenn es die interne Datendarstellung erlauben würde.
  - Die interne Datenstruktur und die internen Funktionen sind *versteckt*.

    ==› Das Innenleben einer Datenstruktur kann ohne Änderung des aufrufenden Codes modifiziert werden, solange die offizielle Schnittstelle gleich bleibt.

    Beispiel: Ein Stack kann mit gleicher Schnittstelle intern wahlweise als verkettete Liste oder als Array implementiert werden, nach außen sieht man nur **`push(`***x***`)`**, **`pop()`** und **`isEmpty()`**.

    ==› Solange neue Funktionalität die alte Schnittstelle nur erweitert, aber nicht ändert, brauchen bestehende Verwender der Datenstruktur ebenfalls nicht geändert werden.

- **Code-Wiederverwendung:**

  Problem:

  - Programme enthalten oft viele *Kopien* von ähnlichem Code.
    Das hat meist folgende Ursachen:
    - Mehrfache Implementierung *derselben Datenstruktur für verschiedene Element-Typen* (Liste von **`int`**, Liste von **`char*`**, Liste von **`struct`** …).

      Analog: Mehrfache Implementierung *desselben Algorithmus für verschiedene Datenstrukturen*:
      Sortieren von Adressen, von Telefonnummern, von Terminen, …
    - *Erweiterung* von bestehenden Datentypen:
      Beispiel: Button mit Icon zusätzlich zum Text (90 % des Codes ist ident zu normalem Button…), Random File (= sequentieller File + Position abfragen / setzen), Balancierter Binärbaum (Einfügen / Löschen ist anders, sonst ist der Code ident zum primitiven Baum).

Lösung: 2 verschiedene Konzepte:

- *Vererbung: Ein neuer Datentyp kann als Erweiterung / Modifikation eines bestehenden Datentyps definiert werden!*

  Nur was dazukommt / anders ist, wird frisch programmiert:

  - Neue Datenelemente kommen dazu.
  - Neue Funktionen kommen dazu.
  - Bestehende Funktionen werden anders implementiert.

- *Parametrierter Code / parametrierte Datentypen* (= Templates):

  Code kann *Datentypen als Parameter* haben!

  Beispiel:

  - Datentyp (und Code!) der Liste haben den *Elementtyp als Parameter*.
  - Das Array hat den Elementtyp als Parameter
    ==› Sortieren funktioniert mit jedem Elementtyp,
    der eine "kleiner"-Funktion hat.

Objektorientierte Software-Entwicklung:

Bisher (C):

- *Ablauforientiert* = Schritt für Schritt = am Code orientiert
- *Top-Down* =
  Zerlegung *ausgehend vom Gesamtproblem* in immer kleinere Teilaufgaben
- Daten werden erst beim Programmieren "nach Bedarf" definiert.

Problem dabei:

- Entwurf in abstrakten Abläufen (Computerprogramm ist Abfolge
  immer detaillierterer Einzelschritte) fällt den Leuten meist schwer.

Jetzt (C++):

- *Datenorientiert*: Daten statt Algorithmen stehen im Mittelpunkt!
  1. *Welche Daten* brauche ich für mein Problem?
  2. *Welche Operationen* brauche ich auf diesen Daten?
  3. ==› Zuerst Implementierung der einzelnen Daten samt Operationen darauf
     (oder Suche nach passenden "fertigen" Datentypen).
  4. ==› Dann Kombination der Datentypen zu einer Gesamtlösung.
- *Bottom-Up* = Aus kleinen Bausteinen schrittweise Größeres zusammensetzen!

  ==› Entspricht "Baustein-orientierter" Konstruktionsweise der Technik:

  - Computer wird aus Komponenten zusammengebaut, Komponenten werden aus Chips entworfen, …
  - Maschine besteht aus Rahmen, Antrieb, Steuerung, diese wieder aus (schon standardisiert vorhandenen!) Motoren, Getrieben, Kabeln, Schaltern, …

- Hoffnung: Diese "Software-Bausteine" sind wie im Maschinenbau usw. *standardisierbar* und in vielen Projekten *wiederverwendbar*!

Das "denkt sich leichter", weil

- die Realität (oder EDV-Problemstellungen) bestehen meist aus Objekten, die miteinander interagieren.

  ==› Ein Computerprogramm ist ein Modell / eine Nachbildung dieser Realität, d.h. eine *Abstraktion von realen Objekten und deren Zusammenspiel*.

  Beispiele für reale Objekte: Flug, Ticket,...
  Beispiele für Objekte in EDV-Aufgaben: Fenster im GUI, Liste von Personen, ...

Jedes Objekt hat

- *Eigenschaften / Zustandsgrößen:*

  Beispiele:
  Flug: Sitzplätze, Abflug und Ankunft, ...
  Ticket: Preis, Inhaber, Flug, Sitz, Bezahlt?, Eingecheckt?, ...
  Fenster: Position und Größe, GUI-Elemente, Titel, ...
  Personenliste: Personen, Sortierkriterium, ...

- *Funktionalität / Verhalten / Fähigkeiten:*

  Beispiele:
  Flug: "Berechne deine aktuelle Verspätung", "Belege Sitz *xxx* mit Ticket *yyy*", "Drucke deine Passagierliste", ...
  Ticket: "Werde 15 % billiger", "Setze deinen Inhaber auf *xxx*", "Gib mir dein Verkaufsdatum", ...
  Fenster: "Verschiebe dich", "Schließe dich", "Sag mir deinen Fenstertitel", ...
  Personenliste: "Gib mir die Anzahl der Personen", "Suche Person *xxx* und gib mir die Position", "Füge Person *yyy* an der richtigen Stelle ein", ...

Neue Denkweise:

Daten sind aktiv, d.h. können *Operationen auf sich selbst* ausführen.
Funktionen werden *für ein bestimmtes Objekt* aufgerufen,

d.h. in jeder Funktion gibt es einen "ich selbst"-Wert

(also den Flug, das Ticket, das Fenster oder die Personenliste, *für den die Funktion aufgerufen wurde*).

==› Ein "objektorientiert gedachtes" Programm ist ein Netzwerk zusammenspielender (einander etwas anschaffender) Datenobjekte, die Objekte "handeln auf Zuruf".

Beispiele:

- Statt *"Ich zeiche einen Button."*: *"Ich sage dem Button, er soll sich zeichnen."* (und der sagt wieder: "Mein Hintergrund, färbe dich!", "Mein Text, schreib dich!" und "Mein Rahmen, zeichne dich!")

  ==› Notation:
  In C: **`drawButton(okButton);`**
  Jetzt: **`okButton.draw();`**
  Oder bei Pointern: **`okButtonP->draw();`**

- Statt *"Ich berechne die Wurzel von x."*: *"Ich sage x 'Gib mir deine Wurzel!'."*
  `y = x.sqrt();`
  (*nicht* in C++, nur in *rein objektorientierten Sprachen*, wo auch Zahlen Objekte sind)

Konzepte und Terminologien

- **Klasse:**

  Hochtrabend: Beschreibung / Definition der Eigenschaften gleichartiger Objekte (Schnittstelle + Innenleben, Daten + Code) (z.B. aller Flüge, aller Fenster, ...)

  Technisch: Ein **`struct`**-***Typ*** *erweitert um Funktionen* und einiges mehr.

  Beispiele: Typ Button, Typ Liste, Typ Person, ...

  Eine Klasse ist ein Typ: Es gibt sie so wie einen **`struct`**-Typ nur einmal (ursprünglich nur zur Compile-Zeit ohne reale Existenz zur Laufzeit, inzwischen wird die Klassen-Information auch zur Laufzeit mitgeführt), sie enthält noch keine konkreten Werte oder Daten, sondern ist sozusagen der Prototyp bzw. die Schablone für derartige Daten.

- **Objekt:**

  Hochtrabend: Ein Objekt ist eine *Instanz* einer Klasse, also ein ganz konkreter Wert dieser Klasse.

  Technisch: Ein Objekt ist *eine* solche "erweiterte **`struct`**-***Variable***" im Speicher (und zusätzlich ev. interne Pointer auf Verwaltungsinformation, z.B. die dazugehörigen Funktionen).

  Jedes Objekt *gehört zu einer Klasse* (bzw. im Fall von Vererbung auch zu mehreren), so wie jede Variable einen Typ hat.

  Objekte einer Klasse kann es zur Laufzeit beliebig viele (oder auch keine) geben, sie werden wie normale **`struct`**-Variablen deklariert (das Objekt entsteht dann beim Anlegen der Variable verschwindet wieder, wenn die Variable aufhört zu existieren). Man kann sie auch dynamisch anlegen (mit **`new`** statt **`malloc`**) und erhält dann einen Pointer auf das neue Objekt.

  Beispiel: Klasse Button ==> Objekte "Speichern", "Abbrechen", "Ok", ...

- **Methode** (oder "**Member-Funktion**"):
  Zu einer Klasse gehörige ***Funktion***.

- **Member-Variable** (ganz selten auch: "**Attribut**"):
  Zu einer Klasse gehöriges *Datenelement* (wie bisher ***Struktur-Member***).

  Member können Basis-Typen, Arrays, andere Objekte, aber in der Praxis vor allem Pointer auf andere Objekte sein (z.B.: Fenster enthält Menüleiste, Ticket zeigt auf Person, ...).

Objektorientierte Sprachen und Anwendungen

Ursprünge der objektorientierten Programmierung:

- Programmierung *grafischer Benutzer-Oberflächen*:

  Ist ohne Objektorientierung praktisch unmöglich: *Alle* modernen *GUI-Toolkits* (Qt, MFC, GTK, Java AWT/SWT, Microsoft .Net, wxWidgets, ...) sind *objektorientiert*.

  (GTK baut die Objektorientierung in reinem C nach!)

  Moderne GUI's wurden ab 1970 im Xerox Palo Alto Research Center von Alan Kay erfunden, als "Abfallprodukt" entstand dabei die erste allgemein bekannte objektorientierte Sprache **Smalltalk** (das Musterbeispiel der objektorientierten Sprachen, standardisiert und veröffentlicht 1980). Alan Kay hat für seinen Beitrag zur objektorientierten Programmierung den Turing Award bekommen, die höchste Auszeichnung der Informatik.

- *"Discrete Event Simulation"*:

  Modellierung realer Welten mit konkreten Ereignissen und Objekten, z.B. Auto / Zapfsäule / Tankwart oder Lift / Stockwerk / Passagier.

  ==› Sprache "**Simula**", erste (und recht mächtige!) objektorientierte Sprache **(1962!)**

- *Symbolische Mathematik:*

  Komplett andere Bezeichnungen (nicht "objektorientiert"), aber fast idente Ansätze: Lange Integers, Brüche, Polynome, ... sind alles "Zahlen". Das Computeralgebra-System "Axiom" ist beispielsweise ein Musterfall objektorientierter Programmierung mit Klassen usw.

C++:

... kam relativ spät: Es wurde erst ab 1979 von *Bjarne Stroustrup bei AT&T* (amerikanische Telefongesellschaft) in den "Bell Lab's" als Erweiterung von C entwickelt (ursprünglich als Präprozessor, der C++ auf normales C übersetzt hat).
Stroustrup übernahm dabei Konzepte von Simula, das er gut kannte.
1985 wurde C++ veröffentlicht, 1998 formal als ISO-Standard definiert.
2011 wurde eine deutlich erweiterte C++-Version (C++-11) standardisiert.
Seitdem gibt es eine kontinuierliche Erweiterung der Sprache und alle 3 Jahre eine neue Sprachversion (C++-14, C++-17).

Heute:

- Objektorientierung für "fast alles" (praktisch alle wichtigen Sprachen außer C).

Varianten:

- *Rein objektorientierte Sprachen: "Alles ist ein Objekt":*
  Auch **`char`**'s, **`int`**'s, ... (und in Smalltalk sogar der Programmcode!) sind Objekte!

  ==› Smalltalk
  ==› Ruby, Eiffel

  Langsamer, aber konzeptionell schöner / einfacher.

- *Gemischt objektorientierte Sprache: Nicht alle Daten sind Objekte.*
    - *By Design:* C++, Java, C#, Objective C, Python, Delphi, …
    - *Spezialfall,* etwas anderes Konzept der Objektorientierung: JavaScript, Lua
    - *Nachträglich objektorientiert* erweitert: PHP, Perl, Visual Basic, …

  Schneller, aber *konzeptionell problematisch*:

  Beispiel für die Problematik:
  Allgemeine Implementierung "Liste von Objekten" für Objekte beliebiger Art.
  Aber: **`int`** ist kein Objekt ==› Liste ist *nicht* für "Liste von **`int`**" verwendbar!
  (weil Objekte und **`int`**'s speichermäßig anders gehandhabt werden)

  Lösungen:

    - Z.B. in C++: *Templates* (Code mit Typen als Parametern,
      die sowohl Objekte als auch Basis-Typen sein dürfen,
      vom Compiler bei Bedarf mehrfach aufgelöst ==› komplex!)
    - Z.B. in Java: *Autoboxing* (automatisches Anlegen und Entfernen
      von "Wrapper"-Objekten rund um Werte primitiver Datentypen bei Bedarf
      ==› langsamer!)

Typische Größenordnungen:

- Java enthält rund 20000 *vordefinierte Klassen,* einige 100000 *vordefinierte Methoden*.
- wxWidgets ist eine "kleine" GUI-Bibliothek und umfasst knapp 1000 Klassen,
  einige 1000 Methoden.

Auch in C++ gibt es mehrere große *Klassenbibliotheken:*

- Die **STL (Standard Template Library, Bestandteil des C++-Sprachstandards)**:
  Die wichtigsten Datenstrukturen (Liste, Baum, Hashtable, …),
  wichtige Algorithmen darauf (Suchen & Sortieren)
- **Boost** (nicht im C++-Standard, aber heute oft verwendet, Teile werden vermutlich in der nächsten C++-Version als C++-Bestandteil standardisiert): Sehr umfangreich, Sammlung und Vereinheitlichung vieler bestehender Libraries für verschiedenste Zwecke (mehr Algorithmen und Datenstrukturen, "smart Pointer", Mathematik, Threads, Parallele Programmierung, I/O, Netzwerk-Zugriffe, Regular Expressions, Compilerbau, …)
- Diverse GUI-Bibliotheken: **Qt** (bei nichtkommerzieller Verwendung Open Source, fast alle Plattformen), **MFC** (Microsoft- bzw. Windows-Gui-Programmierung vor C# / .net), wxWidgets (ähnlich Qt, kleiner und weniger verbreitet, auch für kommerzielle Verwendung Open Source), FLTK, gtkmm (C++-Port von GTK), …

==› C++-Programmierung mit Klassenbibliotheken ohne Online-Doku ist *sinnlos*
(vordefinierte Methoden nicht auswendiglernen!).

==› Oft auch "Learning by example".

4. Kapitel: Klassen und Objekte in C++

Deklaration einer Klasse

- Schlüsselwort **`class`** statt **`struct`**
  (**`struct`** bietet etwas weniger Möglichkeiten, aber funktioniert auch weiterhin).
- Enthält neben den *Deklarationen der Member-Variablen* (normale Deklarationen wie in **`struct`**) auch die *Deklarationen der Methoden* der Klasse (wie normale *Prototypen*: mit Argumenten und Returntyp) und optional auch gleich deren *Code* (gleich anschließend).
- Zugriffsregelung mit **`public:`** und **`private:`** (und später **`protected:`**):
  - **`public:`** Im gesamten Programm sichtbar / zugreifbar / aufrufbar.
  - **`private:`** Nur für Methoden der Klasse sichtbar,
    für Code außerhalb der Klasse unsichtbar.

  ==› Alles, was **`public`** ist, ist die *öffentliche Schnittstelle* der Klasse,
  alles, was **`private`** ist, ist das *(geheime) Innenleben*.
  - **`public`** und **`private`** gelten für *alle nachfolgenden Member und Methoden*
    bis zum Ende (**`}`**) der **`class`** oder bis zum nächsten **`public`** / **`private`**.
  - Unterschied **`struct`** / **`class`**: In einer **`struct`** ist per Default alles **`public`**,
    in einer **`class`** ist per Default alles **`private`**.
  - ***Stil:*** Bitte immer *zuerst* immer **`public`**, dann **`private`**
    (denn meistens interessieren den Leser nur die öffentlichen Teile,
    und die sollten daher gleich oben stehen!).

Objekte deklarieren / anlegen

- Ganz normale Variablen-Deklaration wie für **`struct`**-Variablen,
  Klassenname ist zugleich Typ (ohne **`class`**): **`myClass objA;`**
- *Initialisierung* mit **`=`** ist erlaubt, ruft den *Copy-Konstruktor* oder
  einen *Typumwandlungs-Konstruktor* auf (siehe unten): **`myClass objB = objA;`**
- *Neu:* Eventuell mit **`(…)`** nach dem Variablennamen in der Deklaration:
  `myClass objC(x, 42);`
  Siehe unten:
  Statt dem Standard-Konstruktor wird ein anderer *Konstruktor* mit
  den angegebenen Argumenten zur Initialisierung der Objekt-Variable aufgerufen.
- *Neu:* Man kann innerhalb einer Rechnung auch *anonyme, temporäre Objekte*
  erzeugen (z.B. als Parameter eines Funktions- oder Methoden-Aufrufs),
  zu denen es keine eigene Variable gibt, und die nur innerhalb der Rechnung leben:

  Man macht dazu einen *Konstruktor-Aufruf mittels Klassennamen und Parametern*
  (statt mit Variablennamen und Parametern): **`myFunc(myClass(x, 42), …);`**
- *Später:* Dynamisches Anlegen mit **`new`** (statt **`malloc`**).

Definition des Codes von Methoden

- Entweder *gleich bei der Deklaration* im **`class`** (empfohlen nur für kurze Methoden, bis rund 3-5 Zeilen ohne Schleifen) ==› automatisch **`inline`** !
- Oder separat *außerhalb* des **`class`** (*danach,* nicht davor!).

  Dann muss der *Klassenname* mit **`::`** vor den Methodennamen geschrieben werden:
  `double myClass::myMeth(double *data, int size) {` …
  (damit der Compiler weiss, dass das eine Methode und keine globale Funktion ist, und zu welcher Klasse die Methode gehört!)

  Das funktioniert auch für die Definition eines *Konstruktors* / *Destruktors* außerhalb von **`class`**, nur eben *ohne Returntyp* in der Kopfzeile.

  Code außerhalb von **`class`** ist nur inline, wenn man explizit **`inline`** davorschreibt, sonst wird die Methode ganz normal aufgerufen.

  Achtung:

  - **`inline`** wirkt *nicht* rückwirkend,
    sondern nur für Aufrufe *nach* dem **`inline`**!
  - Bei Aufspaltung in **`.h`**-File und **`.cpp`**-File:
    (Öffentliche) **`inline`**-Definitionen gehören in den **`.h`**-File,
    sonst sind sie sinnlos!
  - **`inline`** hat Vor- und Nachteile:
    Weniger Overhead, bessere Optimierung, aber mehr Code.
    (Compiler entscheiden heute meist selbst ganz gut,
    sie dürfen **`inline`** ignorieren oder selbst dazutun!)
  - **`inline`** ist in C++ im Unterschied zu C wichtig und häufig,
    da es sehr viele sehr kurze Methoden gibt,
    bei denen der Overhead zu hoch wäre!

Konstruktoren

- Im objektorientierten Jargon: ***"Ctor"***
- Methode mit *Namen der Klasse* selbst, *kein Returntyp*
  (wirklich keiner, auch nicht **`void`**).
- Es kann *mehrere* Konstruktoren pro Klasse geben
  (mit unterschiedlichen Argument-Typen).
- Wird automatisch beim *Anlegen eines neuen Objektes* aufgerufen:
  - Beim Betreten des Blockes für *lokale* Objekt-Variablen.
  - Beim Aufruf der Funktion für "by Value" übergebene *Objekt-Parameter,* und beim Return aus einer Funktion für "by Value"-Objekt-Returnwerte.
  - Beim Start des Programmes (noch vor **`main`** !!!)
    für *globale und statische* Objekt-Variablen
    (Achtung bei Abhängigkeiten zwischen globalen Objekten:
    Reihenfolge undefiniert!!!)
  - Beim **`new`** bei *dynamisch* angelegten Objekten.

- Beim Konstruktor des *übergeordneten Objektes* bei Member-Objekten (Konstruktoren der Subobjekte werden *vor* dem übergeordneten Konstruktor ausgeführt).
- Kann auch *explizit* aufgerufen werden ("temporäres / anonymes Objekt").
- *Nicht* (!!!) beim Anlegen eines Pointers,
  nur beim Anlegen des Speichers für das Objekt selbst!

  ==› Pointer auf Objekte (bzw. Member-Variablen, die Pointer auf Objekte sind) zeigen *nicht* automatisch auf ein Objekt und bewirken *nicht*, dass automatisch ein Objekt angelegt wird, sondern bleiben uninitialisiert, d.h. zeigen (wie in C) auf zufällige Speicherbereiche, bis ihnen ein Wert zugewiesen wird!

- Konstruktoren dienen zur *Initialisierung*, ein Objekt sollte immer gleich mit gültigem Inhalt bzw. voll funktionsfähig "geboren" werden! (z.B. Anlegen von Speicher für Pointer-Member)

Arten von Konstruktoren

- **Standard-Konstruktor** (Default-Konstruktor): *Ohne* Parameter.

  Wird *automatisch* angelegt, wenn die Klasse *gar keinen expliziten Konstruktor* hat (weder einen expliziten Standard-Konstruktor noch irgendeinen anderen), kann aber auch explizit definiert werden.

  Wenn automatisch: Tut "nichts" ==› Member-Variablen haben *undefinierten Inhalt*. (Ausnahme: Bei statisch oder global angelegten Objekten sind die Member auch ohne expliziten Konstruktor auf 0 initialisiert, analog zu allen anderen nicht explizit initialisierten statischen / globalen Variablen).

  Wird *automatisch* aufgerufen, wenn ein Objekt neu und ohne Parameter oder Initialisierungs-Zuweisung angelegt wird (das Objekt muss *ohne* **()** in der Deklaration stehen, nicht mit leeren **()** !).

- **Copy-Konstruktor**: Hat eine *Referenz* (normalerweise eine **`const`**-Referenz, da das Original ja nicht geändert wird) auf *ein Objekt der eigenen Klasse* als *einzigen* Parameter.

  Wird *automatisch* angelegt, kann aber auch explizit definiert werden.

  Wenn automatisch erzeugt: Alle Member sind bitweise Kopie
  ==› Original und neues Objekt haben *identen Inhalt*,
  ihre Pointer zeigen auf dieselben Daten, nicht auf eine Kopie (siehe eigenes Kapitel!).
  Bei Membern, die selbst wieder Objekte sind, ruft der automatisch erzeugte Copy-Konstruktor deren impliziten oder expliziten Copy-Konstruktor auf.

  Ein *expliziter Copy-Konstruktor* ist also vor allem dann erforderlich, wenn die *Objekte Pointer als Member-Variablen enthalten*. Weiters ist er notwendig, wenn Objekte Member enthalten, die einen *eindeutigen Wert* enthalten müssen und nicht einfach kopiert werden dürfen, z.B. eine fortlaufende *Seriennummer*, oder wenn beispielsweise die Anzahl der Objekte global mitgezählt werden soll.

Der Copy-Konstruktor wird aufgerufen:

- Bei *Initialisierungs-Zuweisung* und *explizitem Konstruktor-Aufruf*.
- Implizit bei *Call by Value* | *Return by Value!*
- *Nicht* bei normaler Zuweisung! (hier wird ein =-Operator aufgerufen!!!)

- **Typumwandlungs-Konstruktor** (Konvertierungs-Konstruktor):
  Hat *einen Parameter anderen Typs* (meist **`const`**, bei Objekten: Als **`const`**-Referenz).

  Wird aufgerufen:

  - Implizit bei Initialisierungs-Zuweisung mit fremdem Typ und Parameter-Übergabe mit fremdem Typ (ev. Zuweisung mit fremdem Typ).
  - *Explizit* als *Typ-Cast* oder *Typumwandlungs-Funktion*, sowie als *Konstruktor mit einem Parameter* in einer Deklaration.

  Wenn man vor den Konstruktor **`explicit`** hinschreibt, sind *nur explizite* Aufrufe des Konstruktors möglich: Der Konstruktor wird dann *nicht automatisch* für implizite Typ-Umwandlungen verwendet.

- **Anderer expliziter Konstruktor** ("Zuweisungskonstruktor" ???):

  Mit Parametern, Angabe der Parameter-Werte in **`()`** bei der Deklaration bzw. bei explizitem Konstruktor-Aufruf.

"Verhinderte" Konstruktoren

- Wenn man *irgendeinen expliziten Konstruktor* definiert (aber keinen Standard-Konstruktor), verhindert man, dass der Compiler den automatischen Standard-Konstruktor erzeugt. Das verhindert, dass Objekte *"leer"* (uninitialisiert) erzeugt werden können.
- Wenn man einen bzw. alle Konstruktoren **`private`** oder **`protected`** macht, verhindert man, dass Objekte dieser Klasse außerhalb des Codes der eigenen Klasse angelegt werden können: In diesem Fall kann nur die eigene Klasse ein Objekt erzeugen (oder abgeleitete Klassen, siehe später).

  Das hat Anwendungen z.B. bei sogenannten *Singleton-Klassen*, d.h. Klassen, von denen es nur ein einziges Objekt geben darf.

- Wenn man einen (bzw. alle) Konstruktoren überhaupt *nur deklariert*, aber nicht definiert (d.h. *nie Code* dafür angibt), unterbindet man ihren Aufruf vollständig (wenn der Konstruktor doch verwendet wird, compiliert das zwar, aber es kommt ein Linker-Fehler). Das betrifft nicht nur explizite Aufrufe, s ondern auch alle impliziten Aufrufe des Konstruktors.

  Ab C++-11 gibt es ein *eigenes Sprachkonstrukt*, das ausdrückt, dass dieser Konstruktor *nicht existiert*: ... **`= delete`** hinten an den Prototyp anhängen.

  Am häufigsten macht man das *für den Copy-Konstruktor*, um zu verhindern, dass ein Objekt versehentlich kopiert wird (bzw. versehentlich "by Value" als Argument oder Returnwert übergeben wird).

  Weiters geschieht das bei *rein statischen Klassen*, die nur eine Sammlung von Konstanten, globalen Variablen und Funktionen darauf sind.

Destruktoren

- Im objektorientierten Jargon: ***"Dtor"***
- Methode mit dem *Namen der Klasse selbst mit* **~** *davor*,
  kein Returntyp, *keine* Parameter.
- Daher nur *ein* Destruktor pro Klasse (nicht wie bei Konstruktoren mehrere!).
- Wird automatisch beim *Abbau eines Objektes* aufgerufen:
  - Verlassen des Blockes mit der Deklaration der Objekt-Variablen.
  - Ende des Programms bei globalen / statischen Objekten.
  - **`delete`** (Gegenstück zu **`new`**) bei dynamischen Objekten.
  - Destruktor des übergeordneten Objektes bei Member-Objekten.
  - Vernichtung eines temporären Objektes.
- ==> *Aufräumarbeiten* (z.B. bei Pointer-Membern: Dynamische Daten freigeben!)

Initialisierungsliste

- *Nur in Konstruktoren* im Kopf nach der **`)`**, vor der **`{`**.
- Aussehen: … **`:`** *member1***`(`***wert1***`),`** *member2***`(`***wert2***`),`** … **`{`** …

  (entspricht *member1* **`=`** *wert1***`,`** …)

  Beispiel (für eine Klasse **`Point`** mit Member-Variablen **`mRGB`**, **`mX`** und **`mY`**):

  ```
  Point(const Color &color, int x, int y) : mRGB(color), mX(x), mY(y)
  { …
  ```
- Die *Member* müssen *in der Reihenfolge der Deklaration* im **`class`** angegeben sein
  (bzw. werden in Reihenfolge der Deklaration initialisiert,
  auch wenn sie in der Initialisierungs-Liste in anderer Reihenfolge angegeben sind!).
- Die *Werte* können beliebige Ausdrücke mit Konstanten, globalen Variablen,
  Parametern des Konstruktors und schon initialisierten Membern sein.
- Für Member, die selbst wieder Objekte sind, kann statt einer simplen Zuweisungs-Initialisierung ein *beliebiger Konstruktor-Aufruf* in der Initialisierungsliste stehen!
- *Notwendig* zur Initialisierung von **`const`**-Membern und Membern, die *Referenzen* sind (beide müssen gleich beim Anlegen initialisiert werden
  und dürfen nicht nachträglich zugewiesen werden!).
- *Notwendig*, wenn für Member, die selbst wieder Objekte sind, nicht deren Standardkonstruktor, sondern ein *explizit angegebener Konstruktor* aufgerufen werden soll (auch das kann nur in der Initialisierung geschehen,
  wenn das Member neu angelegt wird: Später kann für ein Member
  kein Konstruktor mehr aufgerufen werden!).
- Später: *Notwendig* in abgeleiteten Klassen, wenn für die Basis-Klasse
  nicht deren Standard-Konstruktor, sondern ein explizit angegebener Konstruktor aufgerufen werden soll: Der *Konstruktor-Aufruf der Basis-Klasse* steht als Erstes,
  vor den Member-Initialisierungen.

- *Effizienter* als Zuweisungen im Konstruktor-Rumpf:
  Statt einer Initialisierung in zwei Schritten (Objekt und Subobjekte zuerst "leer" anlegen und dann einzeln per Zuweisung mit Werten füllen) werden gleich die richtigen Werte angelegt.

`this`

- Nur *innerhalb von Methoden* einer Klasse definiert, nicht in normalen Funktionen.
- Sinngemäß wie eine vordefinierte und vorinitialisierte Variable:
  Zeiger auf das eigene Objekt
  (das Objekt, für das die Methode aufgerufen wurde, d.h. der *"0-te Parameter"*)
  ==› Typ von **`this`** : *"Pointer auf eigene Klasse"*

Zugriff auf Member

- *Member des eigenen Objektes* werden *wie normale Variablen* angesprochen.
- Member liegen in der Namens-Such-Reihenfolge
  zwischen den lokalen Variablen bzw. Parametern und den globalen Variablen.
- Wenn ein *gleichnamiger Parameter* / eine gleichnamige lokale Variable existiert:
  Zugriff auf das Member mit **`this->`**...
  (der Name allein spricht den Parameter bzw. die lokale Variable an)
- *Member fremder Objekte*: Zugriff mit *obj*`.`*member* oder *obPtr*`->`*member*
- Man kann auf Member von *fremden Objekten der eigenen Klasse* zugreifen,
  auch wenn diese Member **`private`** sind
  (**`public`** und **`private`** gelten nicht pro Objekt, sondern *pro Klasse*).

`const`-Methoden

- Bei der Deklaration einer Methode im **`class`** (und dann auch bei der Definition derselben Methode außerhalb des **`class`** falls diese getrennt erfolgt)
  kann *nach* den **`( )`** mit den Parametern **`const`** stehen.
- Das bedeutet:
  Diese Methode *ändert* **`this`**, also das Objekt, für das sie aufgerufen wurde, *nicht*
  (d.h. macht keine Zuweisungen auf Member-Variablen
  und ruft intern nur **`const`**-Methoden für das eigene Objekt auf).
- Auf Objekte, die **`const`** deklariert sind,
  dürfen nur **`const`**-Methoden angewendet werden.

Übliche Methoden: `get`... und `set`...

In fast allen objektorientierten Sprachen hat sich eingebürgert:

- Member-Variablen sind *nicht* **`public`**
  (auch nicht solche, die direkt von außerhalb der Klasse verwendet werden sollen).
- Für jedes Member, das von außen lesbar sein soll,
  gibt es eine öffentliche *get-Methode*.

Diese heißt üblicherweise auch **`get`**..., hat meist *keine Parameter*, und besteht im einfachsten Fall nur aus einem **`return`** *der jeweiligen Member-Variable*.

- Für Member, die von außen *änderbar* sein sollen,
  gibt es eine öffentliche *set-Methode*.

  Diese heißt üblicherweise **`set`**..., hat meist *einen Parameter und keinen Returnwert*, und besteht im einfachsten Fall nur aus *einer einzigen Zuweisung* des Parameters auf die Member-Variable.

  (es gibt aber auch komplexere Fälle von **`get`**... und **`set`**..., im **`get`** könnte der Wert beispielsweise auch erst berechnet oder gesucht werden, und ein **`set`** könnte auch mehrere oder komplexere Wertänderungen durchführen).
- Beide sollten so gut wie immer **`inline`** sein, **`get`** sollte auch **`const`** sein.

Aufruf von Methoden

- Methoden des *eigenen Objektes* im Code einer Methode: *Wie normale Funktionen* (oder mit **`this->`***meth***`(...)`**).
- Methoden *fremder Objekte*: Zugriff mit *obj***`.`***meth***`(...)`** oder *obPtr***`->`***meth***`(...)`**
- Aufruf von normalen Funktionen: Wie bisher.

Daher: Ein "normaler" Aufruf (ohne Objekt davor) ist

- entweder ein Aufruf einer normalen Funktion
- oder ein Aufruf einer *Methode der eigenen Klasse für das eigene Objekt* (**`this`**) bzw. im Konstruktor für das gerade konstruierte Objekt.

Bei jedem *Aufruf einer Methode* wird ein *Pointer auf das Objekt*, für das die Methode aufgerufen wird, intern *implizit als zusätzlicher Parameter* übergeben.
In der Methode repräsentiert **`this`** diesen versteckten Parameter.

Statische Member und Methoden

Man kann vor Member-Variablen und vor Methoden-Prototypen **`static`** schreiben.

Bei einer *Member-Variable* bewirkt das, dass die Variable *einmal pro Klasse und nicht einmal pro Objekt* existiert: Alle Objekte greifen auf *denselben* Wert zu!

Das nutzt man z.B. für *Zähler*, die die Anzahl der erzeugten Objekte der Klasse zählen (==> Objekt-Nummern), oder für Datenstrukturen, die alle Objekte der Klasse enthalten.

- **Achtung:** Statische Member müssen nicht nur in der Klasse deklariert werden, sie müssen auch *definiert* werden (d.h. es muss *explizit Speicher für sie angelegt werden*, weil sie ja nicht pro Objekt beim Anlegen des Objektes entstehen),
  und zwar *außerhalb* des **`class`**, z.B.

  ```
  int myClass::myCounter = 1;
  ```

  Das ist auch die einzige Stelle, an der man statische Member initialisieren kann!

  Bei Aufteilung in Header- und Code-Files:
  Diese Zeile gehört in den **`.cpp`**-File der Klasse, *nicht* in den **`.h`**-File!

- Der Speicher für statische Member wird *beim Programmstart* angelegt und existiert bis zum Programmende. Er ist *per Default auf 0 initialisiert* (wie für globale Variablen oder statische Variablen in Funktionen).

  Falls das Member selbst wieder ein Objekt ist, wird (noch vor **`main`** !) sein Konstruktor ausgeführt. Wie bei globalen Objekten gilt:
  Die Reihenfolge, in der Konstruktoren beim Programmstart ausgeführt werden, ist zufällig, das kann bei voneinander abhängigen statischen / globalen Objekten zu ganz kniffeligen Fehlern führen!
  (d.h. man darf sich im Konstruktor nicht darauf verlassen, dass andere statische / globale Variablen schon in einem wohldefinierten Zustand sind!).

- Im Code der Klasse erfolgt der Zugriff ganz normal mit dem Namen.
  Falls statische Member **`public`** sind,
  kann von außen auch *ohne Objekt* mit **`::`** auf sie zugegriffen werden, z.B.:

  ```
  myClass::myCounter = -1;
  ```

Analog zu statischen Member-Variablen gibt es *statische Methoden*,
die *nicht für ein bestimmtes Objekt*, sondern allgemein *für die Klasse aufgerufen* werden,
selbst dann, wenn es gar keine Objekte der Klasse gibt
(man darf sie auch ganz normal für ein Objekt aufrufen, aber das ist selten).

- Der Aufruf in "fremdem" Code erfolgt mit **`::`**, z.B. **`myClass::statMeth(10);`**
  (innerhalb des Codes der Klasse kann der Aufruf "ganz normal" erfolgen).
- Da es ja kein aktuelles Objekt gibt, ist **`this`** in statischen Methoden *nicht* definiert,
  und es darf nur auf statische Member-Variablen, aber nicht auf normale Member zugegriffen werden!
- Einer der *Hauptanwendungen* statischer Methoden sind *set- und get-Methoden für statische Member*.

Statische Member und Methoden heißen auch *"Klassenvariablen"* und *"Klassenmethoden"*.

`const`-Member

... sind kompliziert. Es gibt 3 Fälle:

- *Normale* **`const`***-Member-Variablen* (existieren *ein Mal pro Objekt*),
  die nur mittels Initialisierungsliste *im Konstruktor initialisiert*
  und dann nicht mehr geändert werden können.

  Das betrifft **`const`**-Member-Variablen *ohne* **`static`** und *ohne* Initialwert in der Deklaration.

- *Echte Konstanten*, die zur Laufzeit gar keinen Speicher belegen,
  sondern gleich vom Compiler als konstante Werte in den Code eingesetzt werden.

  Das betrifft **`const`**-Member-Variablen mit einem *"einfachen" Typ* (**`int`**, **`double`**, ...)
  und *mit* **`static`**, denen gleich direkt *in der Deklaration ein Initial-Wert* zugewiesen wird (vergleichbar mit **`enum`**-Konstanten).

  Obwohl solche Member **`static`** deklariert sind, darf dafür
  keine getrennte Definition / Initialisierung außerhalb der Klasse codiert werden,
  die Initialisierung muss bei **`static`**-**`const`**-Membern von einfachen Typen immer gleich in der Deklaration erfolgen.

- *Echte konstante Klassenvariablen*: Sie werden *einmal im Speicher angelegt* und *beim Programmstart initialisiert* und können dann nicht mehr geändert werden.

  Das betrifft Member, die **`static const`** deklariert sind und einen *komplexen Typ* haben (Strukturen, Arrays, Objekte, …).
  Hier darf *keine Initialisierung bei der Deklaration* stehen:
  Die Initialisierung muss bei der separaten Definition erfolgen!

- *Ohne* **`static`** durften **`const`**-Member früher *nicht* in der Deklaration initialisiert werden, egal ob sie einen einfachen oder komplexen Typ haben.

  Seit C++ 11 sind *Initialisierungen von Membern* direkt in deren Deklaration im **`class`** *erlaubt*. Der Effekt ist, dass das Member mit dem Init-Wert in seiner Deklaration initialisiert wird, wenn es nicht in der Init-Liste des Konstruktors vorkommt:
  Der Init-Wert in der Deklaration ist sozusagen der *Default-Init-Wert*, der verwendet wird, wenn im Konstruktor *keine* Initialisierung für dieses Member angegeben ist.

  Das gilt sowohl für **`const`**- als auch für nicht-**`const`**-Member (solange sie nicht **`static`** sind), nur dass **`const`**-Member nach ihrer Initialisierung eben nicht mehr geändert werden dürfen.

Pointer auf Methoden

C++ erlaubt Pointer auf Methoden *analog zu Funktions-Pointern*. Das sind aber implementierungstechnisch verschiedene Dinge, sie können nicht gemischt werden (Methoden haben einen impliziten "nullten" Parameter: Das **`this`**-Objekt).

Achtung: Methoden-Pointer dürfen *nicht* auf **`void *`** konvertiert werden und umgekehrt, denn sie sind intern nicht notwendigerweise Pointer!

Achtung:
Pointer auf **`static`**-Methoden sind *normale* Funktions-Pointer, keine Methoden-Pointer!

- Wegen der komplexen Syntax definiert man sich am besten einen Typ für den Methoden-Pointer:

  ```
  typedef double (myClass::*myMethPtrType)(double x, double y);
  ```

  definiert einen Methoden-Pointer-Typ **`myMethPtrType`** für Objekte der Klasse **`myClass`** mit zwei **`double`**-Parametern und einem **`double`**-Returnwert, der dann verwendet werden kann, um Pointer oder Argumente dieses Typs zu deklarieren. Falls der Pointer nur auf **`const`**-Methoden zeigen soll, gehört vor den **`;`** noch ein **`const`** .

- Um einen solchen Pointer-Wert zu erhalten, muss man die Methode im Normalfall mit dem Klassennamen versehen:

  ```
  myMethPtrType myMethPtr = &myClass::meth1;
  ```

- Für der Aufruf gibt es eigene Operatoren (Achtung auf die Klammern!):

  ```
  (myObj.*myMethPtr)(xVal, yVal);
  (myObjPtr->*myMethPtr)(xVal, yVal);
  ```

  `.*` und **`->*`** sind eigene Operatoren, nicht **`.`** oder **`->`** gefolgt von **`*`** .

Objekte als Parameter und Returnwert

Für Objekte als Parameter und Returnwert gilt dasselbe *wie für Strukturen*:

- Rein technisch können sie *"by Value"* als Argument und Returntyp übergeben werden, d.h. es wird eine eigenständige Kopie des Objektes gemacht (beim **`return`** werden Objekte in ein temporäres Objekt im Aufrufer kopiert).
- Da das Kopieren aber sowohl Zeit als auch Speicher kostet, ist es sinnvoll und allgemein üblich, Objekte *möglichst immer "by Reference"* zu übergeben (bevorzugt als Referenz, alternativ als Pointer), auch wenn sie nur gelesen werden bzw. der Wert im Aufrufer nicht geändert werden soll (in diesem Fall bitte **`const`** !).

  Es lauern aber dieselben Gefahren:

  - Es ist nicht sinnvoll, eine Referenz oder einen Pointer auf ein lokales oder temporäres Objekt als Returnwert zurückzugeben (er zeigt nach dem **`return`** "ins Leere"!).
  - Es ist nicht sinnvoll, einen Pointer oder eine Referenz, die als Parameter übergeben wurden, über den Funktionsaufruf hinaus zu speichern (z.B. in einer Member-Variablen oder einer dynamischen Datenstruktur), wenn das als Argument übergebene Objekt ein temporäres Objekt sein könnte (aus demselben Grund).

Vorab-Klassendeklaration

So wie indirekt rekursive Funktionen einander gegenseitig aufrufen, können *Klassendeklarationen gegenseitig voneinander abhängen* (als Argument- oder Returntyp, als Member, später als Ableitung, ...).

So wie man für Funktionen in diesem Fall einen Prototypen vorab braucht, braucht man bei Klassen ein **`class`** *xxx* **`;`** als *Vorab-Deklaration*, damit der Compiler weiß, das *xxx* der Name einer Klasse ist (ähnlich wie bei **`struct`**'s ein **`struct`** ...**`;`** ohne **`{ }`**).

Ebenso wie bei einem Vorab-**`struct`** können vorab deklarierte **`class`**-Namen im Wesentlichen *nur für die Deklaration von Pointern und Referenzen* (in Member-Deklarationen, Prototypen usw.) verwendet werden.

`friend`

In einem **`class`** können auch folgende Zeilen stehen:

- **`friend class`** *xxx***`;`**
- **`friend`** *prototyp***`;`**

  (also z.B. **`friend void print(const myClass &a);`**)

In beiden Fällen bekommt der *Code der angegebenen Klasse bzw. Funktion vollen Zugriff auf alle Member und Methoden der eigenen Klasse* (als ob es Code der eigenen Klasse wäre), kann also auch **`private`**-Member lesen und ändern und **`private`**-Methoden aufrufen.

Wichtig und häufig ist das z.B. für selbstgeschriebene Ausgabe-Operatoren **`<<`** .

Ein häufiger "**Schmutztrick**" ist die Nutzung von **`friend class`** *xxx* in einer Klasse *yyy*, wenn diese Klasse *yyy* einzig und allein von der Klasse *xxx* verwendet werden soll und für alle anderen Klassen unsichtbar bleiben soll: Wenn man alle Member und Methoden in *yyy* **`private`** macht, dann darf nur die **`friend`**-Klasse *xxx* darauf zugreifen.

Angewendet wird das beispielsweise für *Datenstrukturen wie Listen oder Bäume*:
Die Klasse *yyy* ist in diesem Fall die Klasse für *einzelne Listen- oder Baumknoten*,
und die Klasse *xxx* ist die Klasse, die *die Liste oder den Baum als Ganzes* repräsentiert:
Nur diese Klasse darf auf die einzelnen Knoten zugreifen, "Außenstehende" dürfen nicht.

Das saubere Design für solche Fälle wäre aber eine *"nested class"*, nicht **`friend`**.

Aufteilung in Dateien

In den meisten Firmen / Programmier-Richtlinien ist gefordert:

- Ein **`.h`**-File und ein **`.cpp`**-File *pro Klasse*
  (**`.cpp`** nur, falls es Methoden gibt, die nicht inline sind).

  Plus weitere **`.cpp`**- / **`.h`**-Files für nicht-Klassen-Code
  (z.B. **`main`**, allg. Typen, ...).
- Der **`.h`**-File enthält neben dem **`class`** (und ev. anderen Deklarationen wie bisher) alle **`inline`**-*Methoden*.
- Weiters gehören *Templates* (siehe später) in den **`.h`**-File, nicht in den **`.cpp`**-File.
- Die *Definitionen* (Speicher anlegen und initialisieren) statischer Member gehören hingegen in den **`.cpp`**-File!

Die Idee dahinter ist Folgende:

- Alles, was *alle* von der Klasse wissen bzw. verwenden sollen
  (die *öffentliche Schnittstelle*), gehört in den **`.h`**-File:

  Der **`.h`**-File muss alles von der Klasse enthalten, was ein **`.c`**-File,
  der die Klasse verwendet, braucht, damit er compiliert werden kann.
- Die *Implementierung* (der Code, d.h. die Methodendefinitionen
  und die Deklarationen, die Speicher belegen) gehört in den **`.cpp`**-File.

Technisch gilt Folgendes:

- Der **`.h`**-File darf nur Deklarationen enthalten, d.h. Dinge, die weder Code erzeugen noch globalen / statischen Speicher anlegen: Da ein **`.h`**-File mehrmals (in der Praxis: In vielen **`.cpp`**-Files) inkludiert wird, würde alles, was aus dem **`.h`**-File an Funktionscode und Daten entsteht, in *mehreren* **`.o`**-Files enthalten sein, und der Linker würde sich über doppelte Funktionen / Methoden / globale Daten mit gleichem Namen beschweren.
- Inline-Code resultiert nicht in eigenständigem Funktionscode mit eigenem Namen, sondern wird nur im Kontext der aufrufenden Funktionen erzeugt.
  Er resultiert daher nicht in mehrfachem Code gleichen Namens,
  auch wenn er mehrfach compiliert wird.

  Inline-Code muss *im Header* stehen, damit der zu verwendende Code
  allen Aufrufern bekannt ist.

Includes:

Hier gilt das im früher Semester für C Gesagte: So viele wie nötig, so wenig wie möglich:

- Jeder File (egal, ob **.cpp** oder **.h**) inkludiert die Header all jener Klassen, die er *direkt verwendet*, d.h. deren Klassenname oder sonstige Elemente in seinem Source vorkommen.
- Jeder **.cpp**-File inkludiert jedenfalls den **.h**-File der *eigenen* Klasse.
- Kommt eine Klasse X in der *Schnittstelle* der Klasse Y vor (z.B. im Prototyp einer Methode oder als Member), so muss der Header von Y den Header von X inkludieren. Kommt X hingegen nur in der *Implementierung* (aber nicht in der Schnittstelle) von Y vor, so inkludiert nur der **.cpp**-File (aber nicht der **.h**-File) von Y den Header von X.

Wie im vorigen Semester besprochen, kann es passieren, dass derselbe **.h**-File beim Compilieren eines **.cpp**-Files *mehrmals* inkludiert wird (direkt und / oder auf verschiedenen Wegen indirekt). Sein gesamter Inhalt muss daher in ein **#ifndef** / **#define** / **#endif** verpackt sein, um doppelte Deklarationen zu verhindern: Nur beim ersten **#include** wird der File wirklich eingefügt, alle weiteren **#include** sehen einen leeren File, weil das **#ifndef** fasch und der gesamte Code damit auskommentiert ist.

Enumerationen innerhalb `class`

Innerhalb von **class** können *Enumerations-Typen* definiert werden.

- Wenn so ein Typ *nicht* **public** ist, ist er nur innerhalb der Klasse verwendbar und kann dort direkt angesprochen werden.
- Wenn er **public** ist, ist er von außen sichtbar, aber sowohl der Typname als auch die Werte müssen *mit dem Klassennamen versehen* werden: **myClass::myEnumType** und **myClass::myEnumVal1**

  Öffentliche Enumerationstypen sind der übliche Weg, mit dem eine Klasse *Konstanten oder Flags* nach außen zur Verfügung stellt. Die Flags der File-I/O-Klassen zur Angabe der Optionen beim Open oder der Formatierungs-Details sind beispielsweise Konstanten öffentlicher **enum**-Typen dieser Klassen.

Dynamische Speicherverwaltung: `new` und `delete`

- **new** *legt dynamisch Speicher* an (wie **malloc**), *ruft den Konstruktor auf* (falls es sich um Objekte handelt) und liefert *einen Pointer* auf das Objekt oder das Array als Ergebnis.

  Es gibt *3 Varianten*:

  - ```
    int *p = new int;            // legt einen int dynamisch an
    myClass *p = new myClass;    // legt ein myClass-Objekt dynamisch an
                                 // und ruft den Standard-Konstruktor auf
    ```
  - ```
    int *p = new int[10];
        // legt ein Array mit 10 int's dynamisch an
    myClass *p = new myClass[size];
        // analog für ein Array von size vielen myClass-Objekten,
        // für jedes wird der Standard-Konstruktor einzeln aufgerufen.
    ```

- **`myClass *p = new myClass(10, i + 1);`**
  // legt *ein* **`myClass`**-Objekt an
  // und ruft einen *Konstruktor mit Argumenten* auf
- Es gibt *kein* **`new`** für ein Array mit explizitem Konstruktor!
  ==› Hat eine Klasse keinen Standard-Konstruktor,
  kann kein Array von Objekten dieser Klasse angelegt werden!

- **`delete`** *ruft den Destruktor auf* (bei Arrays von Objekten für *jedes einzelne Element*) und *gibt den Speicher frei* (wie **`free`**).

  Es gibt *2 Varianten,* die zum **`new`** passend verwendet werden müssen:

  - **`delete p;`** // wenn **`p`** auf einen *einzelnen* Wert / ein Objekt zeigt
  - **`delete [] p;`** // wenn **`p`** auf ein *Array* zeigt

Hinweise:

- Der Speicher ist bei einfachen Typen bzw. beim Anlegen von Objekten mittels Default-Standard-Konstruktor so wie bei **`malloc`** *uninitialisiert*.
- **`delete`** darf auch mit dem **`NULL`**-Pointer aufgerufen werden (tut nichts).
- Die Fehlerbehandlung von **`new`** ist dubios:
  - Früher kam im fehlerfall ein **`NULL`**-Pointer zurück.

    Dieses Verhalten kann man in aktuellen Systemen auch erreichen, z.B.
    `new (nothrow) myClass[10];`
  - Nach dem aktuellen Standard wird eine Exception geworfen (siehe später) oder eine spezielle Fehlerfunktion aufgerufen.
- Man kann die Speicherverwaltung von **`new`** und **`delete`** optional selbst programmieren (kommt später).
- **`new`**-Aufrufe mit zusammengesetzten Typen (vor allem **`*`** für Pointer) sind erlaubt, werden aber bei komplexen Typen schnell kryptisch und unverständlich.
  Für komplexere Fälle ist ein **`typedef`** sinnvoll!

  `double **data = new double * [size];`
  // **`data`** ist ein Array von **`double`**-Pointern
- Es gibt kein **`new`**, mit dem man *mehrdimensionale Arrays* variabler Größe anlegen kann: Alle Dimensionen *außer der ersten* (vordersten) müssen konstant sein!

  (mehr dazu später)

5. Kapitel: Vererbung

Grundlagen

Die Vererbung ist ein zentrales Konstrukt der Objektorientierung:

Eine Klasse lässt sich als Erweiterung
einer schon bestehenden Klasse deklarieren.
Sie "erbt" dann deren Schnittstelle und
alle Member und Methoden dieser Klasse.

Man nennt das auch *"ableiten"*: Die neue Klasse ist die *abgeleitete Klasse* (engl. *"derived class"*), die ursprüngliche die *Basisklasse* (engl. *"base class"*), manchmal auch Vaterklasse (nur im Deutschen) genannt.

Beispiele:

- Alle Dialogelemente eines GUI's (Buttons, Textfelder, Listboxen, …) haben viel gemeinsam (z.B. Position, Größe, Textfarbe, …). Sie erweitern alle eine gemeinsame Klasse "DialogElemente" um ihre spezifischen Daten (z.B. Beschriftungstext, eingegebener Text im Textfeld, Listenelemente, …). Sie haben auch viele gemeinsame Methoden ("zeichne dich", "aktiviere / deaktiviere dich", …), die spezifisch angepasst werden (ein Button zeichnet sich anders als ein Textfeld).
- Ähnliche Objekte in einer Simulation verhalten sich bis auf spezifische Unterschiede gleich: Ein PKW, ein LKW und ein Motorrad haben vieles gemeinsam, unterscheiden sich aber in Geschwindigkeit, Beschleunigung, Lärm und Abgasen.

Die *geerbten Member und Methoden* heißen im Englischen *"inherited members / methods"*:

- *Bestehende Member* werden automatisch *unverändert übernommen*,
  sie können **nicht umdeklariert** werden[1].
- *Bestehende Methoden* werden automatisch *unverändert übernommen*. Hierbei wird sowohl die Deklaration (Prototyp im `class`) als auch die Implemenierung (Code) automatisch übernommen, beides muss in der abgeleiteten Klasse nicht mehr wiederholt werden.

  Geerbte Methoden dürfen in der abgeleiteten Klasse aber **auch neu implementiert** werden (d.h. die Methode hat in der abgeleiteten Klasse zwar *denselben Prototyp*, aber möglicherweise *anderen Code*, siehe unten), Objekte der abgeleiteten Klasse können sich also bei derselben Methode *anders verhalten* als die der Basisklasse.

  Üblicherweise listet man im `class` der abgeleiteten Klasse nur jene Methoden auf, für die man in dieser Klasse auch explizit Code angibt: Die überschriebenen und die neu dazugekommenen, nicht aber die unverändert geerbten (und bei den überschriebenen wäre nicht einmal das notwendig).
- Für *Konstruktoren und Destruktoren* gibt es **spezielle Mechanismen**:
  Sie werden nicht geerbt, sondern existieren in jeder Klasse separat und werden alle nacheinander ausgeführt (siehe unten).

1 In der abgeleiteten Klasse könnte ein neues Member gleichen Namens *zusätzlich* deklariert werden, aber da das nur Verwirrung stiftet, sollte man es tunlichst vermeiden!

- *Neue Member und neue Methoden* können **nach Belieben dazudeklariert** werden.

Man erbt immer alles:
Es ist nicht möglich, geerbte Member oder Methoden zu verstecken oder zu löschen. Eine abgeleitete Klasse "kann" daher immer mindestens so viel wie die Vaterklasse, jede Methode der Vaterklasse kann auch für Objekte der abgeleiteten Klasse aufgerufen werden, und jedes Objekt der abgeleiteten Klasse enthält alle Member der Vaterklasse.

Die Vaterklasse wird unmittelbar nach dem Klassennamen im **`class`** mit **`public`** angegeben, beispielsweise:

`class Button: public DialogElement {` …

Damit sind (ohne dass man sie nochmals angeben müsste) alle Member und Methoden von **`DialogElement`** auch für **`Button`** deklariert.

Die Zugriffsrechte von fremden Klassen auf Member und Methoden einer abgeleiteten Klasse, die geerbt sind, sind der "kleinste gemeinsame Nenner" der in der Basisklasse dafür festgelegten Zugriffsrechte und des Zugriffsrechts für die Basisklasse im **`class`** der abgeleiteten Klasse. Da fast immer **`public`** abgeleitet wird, werden die Rechte der Basisklasse meist unverändert übernommen (es stiftet nur Verwirrung, wenn Dinge, die in der Basisklasse **`public`** sind, in der abgeleiteten Klasse plötzlich nicht mehr zugänglich sind, man sollte das daher nicht tun!).

`protected`

`protected` ist wie **`public`** und **`private`** ein Schlüsselwort zur Festlegung der Zugriffsrechte auf Member und Methoden. Es liegt zwischen **`public`** und **`private`** :

- *Nach außen* (für fremde Klassen) sind **`protected`**-Member und -Methoden *unsichtbar*, wie bei **`private`**.
- Der *Code abgeleiteter Klassen* hat hingegen *vollen Zugriff* auf **`protected`**-Member und -Methoden, als ob sie **`public`** wären.

`private`-Member und -Methoden der Basisklasse sind *auch für abgeleitete Klassen unsichtbar*.

Design-Hilfe…

Wenn beim Design eines Programmes eine Klasse X irgendetwas von Klasse Y übernimmt, stellt man sich oft die Frage, ob X jetzt von Y abgeleitet sein soll, oder ob X jetzt ein Objekt der Klasse Y als Member enthält.

Diese Frage lässt sich oft klären, wenn man versucht, die Abhängigkeit in deutschen Worten zu formulieren:

- Wenn man sinnvollerweise sagen kann **"X ist (auch) ein Y"**, also z.B. *"Ein Rechteck ist ein geometrisches Objekt"*, dann ist das ein Indiz dafür, dass *X von Y abgeleitet* gehört.
- Wenn es eher sinnvoll ist, **"X hat (oder "enthält") (auch) ein Y"** zu sagen, also z.B. *"Ein Rechteck hat eine Farbe"*, dann ist es vermutlich richtig, X *nicht* von Y abzuleiten, sondern *in X eine Member-Variable vom Typ Y* zu deklarieren (oder eine Member-Variable in X, die ein Pointer auf ein dynamisch angelegtes Y ist).

Das Erben und Überschreiben von Methoden

Es ist erlaubt – und sogar ganz zentral und allgemein üblich – in einer Klasse Code für eine Methode anzugeben, für die es schon in der Vaterklasse Code gibt. Man nennt das *Überschreiben* der Methode.

Gibt es für eine Methode mehrere Definitionen, so werden diese nicht irgendwie nacheinander ausgeführt, sondern es kommt *immer der "zutreffendste" Code* der Methode für die jeweilige Klasse – und nur dieser! – zur Anwendung: Wird eine Methode **myFunc** der Klasse X aufgerufen, so wird zuerst einmal geprüft, ob in X Code für diese Methode angegeben ist. Wenn ja, wird dieser ausgeführt. Wenn nein, wird der Code für die Methode in der Vaterklasse gesucht. Ist dort auch nichts definiert, wird in deren Vaterklasse weitergesucht usw. (nur Konstruktoren und Destruktoren funktionieren grundsätzlich anders, siehe unten).

Es ist daher nicht notwendig, identen Code von der Vaterklasse in die abgeleitete Klasse zu kopieren: Wenn die abgeleitete Klasse *keinen eigenen Code* für eine Methode hat, erbt die abgeleitete Klasse *automatisch den Code der Vaterklasse*.

Virtuelle Methoden

In der Regel ist es so, dass in einer Anwendung Objekte verschiedener abgeleiteter Klassen gemischt vorkommen: Ein GUI-Dialog-Fenster hat Buttons, Textfelder, Choiceboxes usw. In diesem Fall sind Variablen, die auf beliebige solcher Objekte verweisen können, *mit der gemeinsamen Basisklasse aller dieser Klassen deklariert*, also beispielsweise "Referenz auf Dialog-Element" für ein einzelnes, konkretes Dialog-Element oder "Array von Pointern auf Dialog-Elemente" zur Auflistung aller GUI-Elemente eines Dialog-Fensters.

Daher gilt:

Ein Pointer, der gemäß seiner Deklaration
auf ein Objekt der Klasse X zeigt
(oder eine Referenz auf ein Objekt der Klasse X),
kann entweder auf ein Objekt der Klasse X selbst
oder auf ein Objekt einer davon
direkt oder indirekt abgeleiteten Klasse zeigen!

Es gilt also:

Man darf auf einen Pointer,
der als "Pointer auf Klasse X" deklariert ist,
auch einen Pointer auf ein Objekt
einer davon abgeleiteten Klasse zuweisen,
aber nicht umgekehrt!

Also z.B.:

```
myBaseClass *p;
...
p = new mySubClass(...);   // in diese Richtung erlaubt, in die andere nicht!
cout << p->myFunc();
```

Klar ist, dass man mit **p->** *nur jene Methoden* aufrufen kann, die *schon in* **myBaseClass** *deklariert* sind: Wäre **myFunc** erst in **mySubClass** deklariert,
könnte **p** ja auch auf Objekte von Klassen zeigen, die gar keine **myFunc** haben
(zum Beispiel auf **myBaseClass**-Objekte)!

Das ist auch technisch begründet: Der Compiler muss beim Übersetzen prüfen, ob die aufgerufene Methode überhaupt existiert, und er kann dazu nur die Deklaration von **p** und die Methoden der Klasse aus der Deklaration heranziehen: Von davon abgeleiteten Klassen weiß er im Moment der Übersetzung nichts, sie können in einem anderen File stehen oder überhaupt erst später geschrieben werden.

Die Frage ist jetzt:

Wird bei **p->myFunc()**
das **myFunc** *aus* **myBaseClass**
oder aus **mySubClass** *aufgerufen?*

Die Antwort ist wie so oft "das hängt davon ab":

- Steht vor der Deklaration von **myFunc()** in **myBaseClass** das Wort **virtual** ,
  so wird das **myFunc** aus **mySubClass** aufgerufen.

  Man nennt das *"dynamische" Bindung*, weil die richtige Methode *erst zur Laufzeit unmittelbar beim Aufruf ermittelt wird* und bei jedem derartigen Aufruf eine andere sein kann (je nach tatsächlicher Klasse des Objektes, auf das **p** gerade zeigt).
  Aufgerufen wird also nicht die Methode, die dem deklarierten Typ von **p** entspricht, sondern es wird zur Laufzeit nachgeschaut, zu welcher Klasse das Objekt, auf das **p** gerade zeigt, wirklich gehört, und *die Methode aus der tatsächlichen Klasse des Objektes aufgerufen*.

  Das ist in den meisten Fällen das Verhalten, das man erwartet
  (und das, was z.B. in Java oder C# immer geschieht).

  virtual vererbt sich: Ist eine Methode in einer Klasse als **virtual** deklariert, so ist sie auch in allen davon abgeleiteten Klassen automatisch **virtual**.
  In unserem Beispiel wäre der Aufruf also auch dann dynamisch, wenn **myFunc** nicht in **myBaseClass**, sondern in einer der Vaterklassen von **myBaseClass** virtuell deklariert ist.

- Ist das nicht der Fall (*kein* **virtual**),
  wird das **myFunc** aus **myBaseClass** aufgerufen.

  Man nennt das *"statische" Bindung*, weil es *zur Compilezeit geschieht* und fix der Aufruf einer ganz bestimmten Methode aus einer ganz bestimmten Klasse im compilierten Code steht: Der Compiler bestimmt die Methode nach dem deklarieren Typ des Pointers oder der Referenz: Er nimmt die *Methode jener Klasse, auf die der Pointer laut Deklaration zeigt,* denn das Objekt, auf das der Pointer zeigt, gehört auf jeden Fall auch zu dieser Klasse (möglicherweise indirekt).

  Im Normalfall ist das allerdings nicht das gewünschte Verhalten: Es wird eine "zu allgemeine" Methode aufgerufen statt der speziell für dieses Objekt geschriebenen.
  Immer, wenn man eine geerbte Methode überschreibt, die nicht **virtual** ist, sollte man sich daher überlegen, ob da ein **virtual** fehlt oder ob man wirklich ausnahmsweise statische Bindung für diese Methode haben möchte.

Achtung: Es nutzt nichts, die Methode erst in einer von **myBaseClass** abgeleiteten Klasse virtuell zu machen! Wenn der Compiler **p->myFunc()** compiliert, kann er sich nur danach richten, wie **myFunc()** in **myBaseClass** deklariert ist. In **mySubClass** schaut der Compiler in diesem Moment nicht nach, ein **virtual** nur dort *"übersieht"* er!

Erst nach der Auswahl der Klasse, deren Methode aufgerufen werden soll, beginnt wie im vorigen Punkt beschrieben die Suche in der Ableitungshierarchie von dieser Klasse abwärts nach der ersten (zutreffendsten) Code-Definition.

In fast allen anderen objektorientierten Sprachen sind Methodenaufrufe grundsätzlich und automatisch immer virtuell, weil das das "natürliche" bzw. "einleuchtende" Verhalten ist. Da der **virtual**-Mechanismus aber *Laufzeit und Speicherplatz kostet* (jedes Objekt mit virtuellen Methoden enthält *als verstecktes Member einen Zeiger auf die Methodentabelle seiner Klasse*, und dort muss bei jedem Aufruf nachgeschaut werden), hat man in C++ beschlossen, dass standardmäßig die "billigere" statische Aufrufmethode verwendet wird (statische Aufrufe sind genauso schnell wie normale C-Funktions-Aufrufe) und nur auf ausdrücklichen Wunsch die virtuelle.

Typumwandlungen zwischen abgeleiteten Klassen

Aus dem vorigen Kapitel folgt, dass ein mit **myBaseClass *p;** deklarierter Pointer auf ein Objekt der Klasse **myBaseClass** oder auf ein Objekt irgendeiner davon direkt oder indirekt abgeleiteten Klasse zeigen kann. Daraus ergeben sich zwei Fragen:

- *Wie stellt man fest, zu welcher Klasse das Objekt wirklich gehört,*
  auf das **p** gerade zeigt?
- Wenn man weiß, dass **p** in Wirklichkeit auf ein Objekt der Klasse **mySubClass** zeigt: *Wie spricht man Member und Methoden an, die es nur in dieser Klasse gibt*
  (denn mit **p->** können ja nur Methoden aus **myBaseClass** aufgerufen werden)?

Den ersten Punkt kann man

- entweder "C-like" (wie bei **union**'s) lösen, indem man in der Basisklasse in einem Member selbst explizit Informationen über die Klasse mitspeichert (*unschön!!!*),
- oder über eine virtuelle Funktion, die in jeder abgeleiteten Klasse überschrieben wird und jeweils einen unterschiedlichen Wert zur Typerkennung liefert (besser),
- oder mit den dafür in C++ eingeführten Mechanismen **RTTI (RunTime Type Information)** und dem **typeid**-Konstrukt (elegant, aber potentiell hoher Overhead, deshalb oft abgeschaltet, besprechen wir hier nicht weiter).

Für den zweiten Punkt muss man **p** in einen Pointer auf **mySubClass** verwandeln. Dazu gibt es zwei Möglichkeiten:

- **static_cast**: Ein **static_cast** ist zwischen zwei Pointer-Typen erlaubt, deren Klassen voneinander abgeleitet sind, und zwar in *beide* Richtungen:
  - Ein **static_cast** von einem Pointer der abgeleiteten Klasse *auf einen Pointer der Basisklasse* **ist sicher und funktioniert in jedem Fall richtig** (hilft uns aber in unserem konkreten Beispiel nicht weiter).

    In Richtung *zur Basisklasse* sollte man daher aus Effizienzgründen immer **static_cast** verwenden.

- Ein **`static_cast`** von einem Pointer der Basisklasse *auf einen Pointer der abgeleiteten Klasse* ist erlaubt, aber **unsicher**: Es wird nicht geprüft, ob der Pointer wirklich auf ein Objekt dieser abgeleiteten Klasse zeigt.

  Zeigt der Pointer in Wirklichkeit auf ein Objekt der Basisklasse oder einer anderen davon abgeleiteten Klasse, so führt der erste Aufruf von Methoden der abgeleiteten Klasse über diesen Pointer zu undefiniertem Verhalten (meist einem *Absturz*)!

  Man sollte einen Pointer daher nur dann mittels **`static_cast`** in einen Pointer auf eine abgeleitete Klasse umwandeln, wenn man ganz sicher ist, dass der Pointer auch wirklich gerade auf ein Objekt dieser Klasse zeigt!

- Ist man sich nicht ganz sicher, auf welches Objekt der Pointer gerade zeigt, sollte man **`dynamic_cast`** verwenden, das *eigens für diesen Fall geschaffen* wurde: Es verwandelt **einen Pointer der Basisklasse in einen Pointer auf eine davon abgeleitete Klasse**, *prüft aber dabei zur Laufzeit*, ob das Objekt, auf das der Pointer gerade zeigt, wirklich ein Objekt dieser abgeleiteten Klasse (oder einer davon weiter abgeleiteten Klasse) ist. Deshalb ist **`dynamic_cast`** auch *langsamer*.

  Zeigt der Pointer auf ein Objekt einer "*falschen*" Klasse, liefert **`dynamic_cast`** **als Ergebnis einen NULL-Pointer**. Man kann also anschließend den Erfolg der Konvertierung explizit prüfen und entsprechend reagieren. Folglich darf man ein **`dynamic_cast`** auch "auf Verdacht" machen und auf diesem Weg indirekt feststellen, zu welcher Klasse das Objekt, auf das der Pointer zeigt, wirklich gehört.

  `dynamic_cast` funktioniert nur auf Klassen *mit mindestens einer virtuellen Methode* (weil es intern zur Klassenprüfung die Informationen über die virtuellen Methoden verwendet, die jedes Objekt versteckt mitführt - Objekten ohne virtuelle Methoden fehlt diese Informationen). **`dynamic_cast`** benötigt in manchen Implementierungen **RTTI** (RunTime Type Information). Da RTTI zur Laufzeit einen gewissen Overhead verursacht, wird C++ auf Kleinst-Systemen und Echtzeit-Systemen oft ohne RTTI compiliert.

  `dynamic_cast` funktioniert analog zu Pointern auch für Referenzen, aber da es keine NULL-Referenz gibt, wird in diesem Fall beim Fehlschlagen der Typumwandlung eine Exception (siehe später) ausgelöst.

Konstruktoren und Destruktoren in abgeleiteten Klassen

1.) Reihenfolge:

Beim Anlegen eines Objektes werden automatisch die *Konstruktoren aller seiner Klassen* aufgerufen, und zwar *in Ableitungsreihenfolge*: Zuerst der Konstruktor der untersten Basisklasse, die von keiner anderen Klasse abgeleitet ist, dann der Konstruktor der davon abgeleiteten Klasse, usw. bis zum Konstruktor der Klasse, zu der das Objekt selbst gehört.

Das Objekt wird also wie eine Zwiebel Schicht für Schicht *von innen nach außen* initialisiert: Man darf sich im Konstruktor der abgeleiteten Klasse darauf verlassen, dass der Konstruktor der Basisklasse schon gelaufen ist und die Member der Basisklasse schon initialisiert sind. Deshalb müssen in der Initialisierungsliste eines Konstruktors auch immer genau die auf dieser Schicht dazugekommenen Member

initialisiert werden: Eine direkte Initialisierung der Member-Variablen der Vaterklasse ist nicht mehr möglich, denn diese müssen schon fertig initialisiert sein, wenn die Initialisierungsliste der abgeleiteten Klasse abgearbeitet wird.

Beim *Destruktor* ist es *genau umgekehrt*: Es werden der Reihe nach die Destruktoren aller beteiligten Klassen von der Klasse des Objektes selbst zurück bis zur untersten Basisklasse ausgeführt, das Objekt wird also Schicht für Schicht *von außen nach innen* aufgelöst.

2.) Welcher Konstruktor der Basisklasse wird aufgerufen?

Normalerweise der *Standard-Konstruktor*.
Es kommt daher ein Fehler, wenn die Basisklasse keinen Standard-Konstruktor hat!

Im Besonderen ruft der *automatisch erzeugte Standard-Konstruktor* einer Klasse den (automatisch erzeugten oder expliziten) *Standard-Konstruktor der Basisklasse* auf.

Will man einen *anderen Konstruktor* der Basisklasse aufrufen, muss man diesen Konstruktor-Aufruf an die **erste Stelle der Initialisierungsliste** schreiben.

Trickreich und gefährlich ist die Lage beim Copy-Konstruktor:

- Ein *automatisch erzeugter Copy-Konstruktor*
  ruft implizit den *Copy-Konstruktor der Basisklasse* auf, egal,
  ob die Basisklasse einen impliziten oder expliziten Copy-Konstruktor hat.
- Ein *explizit programmierter Copy-Konstruktor*
  ruft hingegen immer den *Standard-Konstruktor der Basisklasse* auf,
  d.h. die Member der Basisklasse werden *nicht automatisch kopiert*,
  sondern sind uninitialisiert!

  Will man die Member der Basisklasse kopiert haben, muss man
  in der Initialisierungsliste des Copy-Konstruktors der abgeleiteten Klasse als Erstes explizit den Copy-Konstruktor der Basisklasse aufrufen!
  (egal, ob der Copy-Konstruktor der Basisklasse explizit oder automatisch ist)

3.) **`virtual`** :

Konstruktoren können nicht virtuell sein: Erstens ist der **`virtual`**-Mechanismus zum Zeitpunkt des Konstruktor-Aufrufs noch nicht funktionsfähig, zweitens wäre es sinnlos, den obersten Konstruktor zu starten, bevor der "Unterbau" fertig ist, und drittens ist im Moment des Anlegens ohnehin genau bekannt, um welche Klasse es sich handelt: Man braucht nicht nachzusehen, zu welcher Klasse das Objekt wirklich gehört.

Destruktoren können sehr wohl *virtuell* deklariert werden,
und das ist auch *sinnvoll und notwendig*, wenn

- *irgendein Destruktor einer Klasse X oberhalb der Basisklasse Code enthält*,
- und es Pointer geben könnte, die mit einer Klasse Y "unterhalb" dieser Klasse X deklariert sind und auf Objekte dieser Klasse X oder einer davon abgeleiteten Klasse zeigen könnten:

Wäre der Destruktor nicht virtuell deklariert, würden beim Freigeben des Objektes, auf das der Pointer zeigt, nur die Destruktoren von seiner "deklarierten" Klasse Y zurück bis zur Basisklasse ausgeführt. Die Destruktoren von der wahren

Klasse des Objektes bis zurück zur Klasse Y (also auch der Destruktor der Klasse X) würden "vergessen" werden! Zu unserem Beispiel vor ein paar Seiten: Ist der Destruktor nicht virtuell, so wird bei **`delete p`** nur der Destruktur von **`myBaseClass`** ausgeführt, auch wenn **`p`** auf ein **`mySubClass`**-Objekt zeigt!

Ist der Destruktur hingegen (schon in der Klasse **`myBaseClass`** !) **`virtual`** deklariert, so ist sichergestellt, dass alle Destruktoren ausgeführt werden: Beim Freigeben eines Objektes wird nachgeschaut, zu welcher Klasse es wirklich gehört, und dann werden alle Destruktoren von dieser Klasse abwärts ausgeführt.

Die Faustregel *"Immer wenn eine Klasse eine virtuelle Methode enthält, muss man auch den Destruktor virtuell machen"* ist zwar aus technischer Sicht nicht richtig (es gibt Gegenbeispiele in beide Richtungen), aber deckt in der Praxis 95 % der Fälle richtig ab und ist leicht zu merken.

Selbst wenn der Destruktor nichts tut, muss man explizit Code dafür angeben (eben leer, mit **`{ }`**), sobald man eine **`virtual`**-Deklaration für den Destruktor macht: Der implizite leere Destruktor wird dadurch nämlich nicht mehr erzeugt, und der Linker beschwert sich über einen fehlenden Destruktor.

4.) Virtuelle Aufrufe:

Achtung: Im Code von Konstruktoren und Destruktoren ist der **`virtual`***-Mechanismus abgeschaltet!*

Wenn im Code eines Konstruktors *Aufrufe virtueller Methoden* vorkommen, wird (im Unterschied zu Aufrufen virtueller Methoden in anderem Code) immer die entsprechende Methode *unmittelbar aus der Klasse des Konstruktors* aufgerufen (d.h. im Konstruktor der Basisklasse die der Basisklasse, im Konstruktor der abgeleiteten Klasse die der abgeleiteten Klasse), im Basis-Konstruktor daher nie die Methode aus der abgeleiteten Klasse, zu der das Objekt wirklich gehört (denn diese kann man zu diesem Zeitpunkt noch gar nicht aufrufen!).

Auch im Basis-Destruktor wird bei virtuellen Aufrufen nicht die Methode der abgeleiteten Klasse aufgerufen (denn die Funktionalität der abgeleiteten Klasse ist im Objekt schon abgebaut, wenn der Destruktor der Basisklasse läuft), sondern die aus der Klasse des Destruktors selbst.

Aufruf einer Methode der Vaterklasse

In vielen Fällen ist es nötig, im Code einer Methode eine Methode (meist: dieselbe Methode) der Vaterklasse aufzurufen.

Das häufigste Beispiel ist der Zuweisungs-Operator **=** (wir lernen später, dass man den selbst programmieren kann): Der erste Schritt ist normalerweise, die Zuweisung für die geerbten Member zu machen, und dazu ruft man den Zuweisungs-Operator der Basis-Klasse auf, bevor man sich dann um die eigenen Member kümmert.

Schreibt man aber nur den Methoden-Namen, wird entweder die entsprechende Methode der eigenen Klasse aufgerufen oder mittels **`virtual`**-Mechanismus die "höchste" Methode dieses Namens (d.h. die eigene oder die einer abgeleiteten Klasse).

Um explizit die Methode einer Vaterklasse aufzurufen, muss man die gewünschte Klasse mit **`::`** vor den Methodennamen stellen:
`myBaseClass::myFunc(…)` bei Aufrufen für das eigene Objekt und
`someObj.myBaseClass::myFunc(…)` oder **`ptr->myBaseClass::myFunc(…)`**
für andere Objekte.

In Java gibt es dafür einfach das Schlüsselwort **`super`**, aber da es in C++ *mehrere* Vaterklassen geben kann, muss man explizit angeben, welche Vaterklasse gemeint ist.

Rein virtuelle Methoden, abstrakte Klassen

Oft ist es so, dass eine Methode zwar **in der Basisklasse deklariert** werden soll, weil sie eben von allen abgeleiteten Klassen in gleicher Form (d.h. mit denselben Parametern und demselben Returntyp) angeboten wird, **aber nicht implementiert** werden kann, weil ihre Implementierung für jede abgeleitete Klasse verschieden ist und oft auch von Daten abhängt, die es erst in den abgeleiteten Klassen gibt.

Solche Basisklassen sind typischerweise abstrakte Gebilde, die nur angelegt werden, um die Gemeinsamkeiten verschiedener Objektarten zusammenzufassen: Es ist gar nicht beabsichtigt, jemals Objekte der Basisklasse selbst anzulegen, sie ist nur "Unterbau" bzw. "kleinste gemeinsame Schnittstelle" für andere Klassen.

Beispiele:

- Bei unserem GUI-Beispiel wird schon die Klasse Dialog-Element eine **`draw`**-Methode bieten, denn wir möchten ja ganz allgemein in einer Schleife über alle Dialog-Elemente eines Dialogs für jedes Objekt **`draw`** aufrufen können, ganz unabhängig, um welches Element es sich konkret handelt.

  Andererseits wird es in der Klasse Dialog-Element keine allgemeine Implementierung von **`draw`** geben, denn jedes reale Dialogelement (Button, Textfeld, …) muss anders gezeichnet werden.

  Das ist aber kein Problem, weil ohnehin nie ein Dialog-Element selbst erzeugt wird und gezeichnet werden muss, sondern immer nur Buttons, Textfelder usw.

- Bei den so oft als Beispiel zitierten Fahrzeugen gibt es (wenn wir eine beispielhafte Implementierung für die öffentliche Verwaltung betrachten) sicherlich schon in der Basisklasse "Fahrzeug" eine Methode zur Berechnung der Steuer, denn jedes registrierte Fahrzeug zahlt in irgendeiner Form Steuer.

  Implementiert wird die Methode aber erst in den konkreten Klassen PKW, LKW, Motorrad, landwirtschaftliches Nutzfahrzeug usw., denn sie wird für jede dieser Fahrzeugtypen anders berechnet.

  Es werden auch nur Objekte der abgeleiteten Klassen angelegt, nie Objekte der allgemeinen Klasse "Fahrzeug": Jedes denkbare Fahrzeug *muss* zu einer der abgeleiteten Klassen gehören.

Solche Methoden heißen (solange kein Code für sie angegeben wird) *"rein virtuelle" Methoden (englisch: "pure virtual")*. Basisklassen, von denen gar keine Objekte direkt angelegt werden sollen, sondern die nur bestimmte gemeinsame Eigenschaften von Klassen / Objekten zusammenfassen, heißen *"abstrakte" Klassen.*

In C++ wird eine rein virtuelle Methode durch ein **`= 0`** am Ende ihrer Deklaration im **`class`** gekennzeichnet. Beispiel: **`virtual void draw() = 0;`**
Damit ist gesagt, dass *für diese Methode in dieser Klasse kein Code definiert werden soll*.

Abstrakte Klassen werden *nicht eigens gekennzeichnet*, sondern ergeben sich *automatisch* aus der Deklaration rein virtueller Methoden.

Daher gilt zusammenfassend:

- Eine Klasse ist automatisch *genau dann abstrakt*,
  wenn sie *mindestens eine rein virtuelle Methode* enthält.
- Die rein virtuelle Methode und die Eigenschaft, eine abstrakte Klasse zu sein, werden *vererbt*.
- Soll eine abgeleitete Klasse *nicht mehr abstrakt* sein,
  muss sie *für alle geerbten rein virtuellen Methoden Code angeben*.
- Von einer abstrakten Klasse können *keine Objekte angelegt* werden,
  nur von davon abgeleiteten, nicht mehr abstrakten Klassen.
- Da von einer abstrakten Klasse keine Objekte angelegt werden können,
  kann man mit dieser Klasse keine Variablen oder Parameter und auch keine Arrays deklarieren, und man kann in anderen Klassen auch keine Member mit dieser Klasse deklarieren:

  In allen diesen Fällen wird ja sofort ein Objekt der Klasse angelegt und ein Konstruktor aufgerufen, und genau das ist bei abstrakten Klassen nicht möglich.

  Sehr wohl möglich und der Normalfall ist hingegen, Variablen, Parameter, Arrays oder Member zu haben, die *Pointer* (teilweise auch Referenzen) *auf Objekte einer abstrakten Klasse* sind: Beim Anlegen eines Pointers entsteht ja kein neues Objekt, sondern man lässt den Pointer auf ein anderweitig erzeugtes *Objekt einer abgeleiteten Klasse zeigen*.

Konstruktoren können nicht virtuell sein, daher auch nicht rein virtuell.

Destruktoren können sehr wohl virtuell sein, und auch rein virtuell, und das hat den erwarteten Effekt: Man kann keine Objekte der Klasse selbst erzeugen, nur Objekte abgeleiteter Klassen. Man macht das beispielsweise, wenn man eine Klasse abstrakt machen möchte, die sonst keine **`virtual`**-Methode hat, die man rein virtuell machen könnte.

Da aber beim Abbau eines Objektes *alle* Destruktoren von allen Vaterklassen der Reihe nach abgearbeitet werden müssen, muss man *trotzdem Code* (und zwar eine leere **`{ }`**) für den Destruktor definieren, obwohl er rein virtuell ist.

Der **`clone`**-Trick

Pointer auf Objekte zuzuweisen funktioniert wie oben beschrieben, auch wenn diese Pointer mit der Basisklasse deklariert sind und in Wirklichkeit auf Objekte abgeleiteter Klassen zeigen: Die *Inhalte der Objekte* werden dabei ja *weder gelesen noch überschrieben*.

Aber **`*dest = *src;`** funktioniert ***nicht***: Dabei wird ja der *Inhalt des Objektes kopiert*, d.h. in das Objekt, auf das **`dest`** zeigt, sollten die Daten des Objektes gespeichert werden, auf das **`src`** zeigt.

Dazu müsste aber erstens sichergestellt sein, dass das **src**-Objekt und das alte **dest**-Objekt *zur selben (abgeleiteten) Klasse gehören* (denn sonst "passt" das **src**-Objekt ja nicht in den schon vorhandenen Speicher des **dest**-Objektes), und zweitens hängt es von der *tatsächlichen Klasse* der beiden Objekte ab, *welche Daten wie zugewiesen* bzw. kopiert werden müssen. Man kann zwar den Zuweisungsoperator selbst definieren, sogar virtuell (siehe nächstes Kapitel), aber dieses Problem lässt sich damit trotzdem meist nicht praktikabel lösen.

Die erste Idee wäre daher, den **dest**-Pointer auf ein *neues, dynamisch angelegtes Objekt* zeigen zu lassen, das man *per Copy-Konstruktor* als Kopie des **src**-Objektes erstellt, womit die eigentliche Zuweisung *nur eine Pointer-Zuweisung* wäre:

```
dest = new myBaseClass(*src);
```

Aber der *Denkfehler* ist auf den ersten Blick sichtbar: Egal, zu welcher abgeleiteten Klasse das **src**-Objekt gehört: Das neue **dest**-Objekt ist *immer* ein **myBaseClass**-Objekt, und es werden *nur die Daten der Basisklasse kopiert*.

Um generell ein *Objekt zu kopieren*, auf das ein *Pointer der Basisklasse* zeigt, das aber auch *von einer abgeleiteten Klasse* sein könnte, bräuchte man einen *virtuellen Copy-Konstruktor*. Da Konstruktoren (und damit auch *Copy-Konstruktoren*) aber *nicht virtuell* sein können, gibt es hier keine direkten Lösungsmöglichkeiten.

Falls der *Inhalt des Objektes* ***src** kopiert oder zugewiesen* werden soll, auch wenn **src** *auf Objekte einer abgeleiteten Klasse zeigen* kann, hilft oft der **clone-Trick**:
Die Idee ist, den *nicht-virtuellen Copy-Konstruktor in eine virtuelle Methode zu verpacken*, die *in jeder abgeleiteten Klasse überschrieben* wird (mit dem *"richtigen" Copy-Konstruktor*).

- Man verlagert die Arbeit des *Kopierens* vom **operator=** in eine *Methode* **clone** (man kann die Methode nennen, wie man will, aber **clone** ist allgemein *üblich*) und ersetzt die Objekt-Zuweisung durch eine *simple Pointer-Zuweisung*:

  ```
  dest = src->clone();
  ```

  Die Methode **clone** hat folgende Aufgabe: *"Returniere einen Pointer auf eine neue, dynamisch angelegte Kopie des Objektes, für das du aufgerufen wurdest!"*

  Es wird also nicht per Zuweisung in das bestehende Objekt kopiert, auf das **dest** zeigt, sondern es wird eine *neue Kopie des Quellobjektes* erstellt, und man lässt **dest** dann *auf diese neue Kopie zeigen*.

  Achtung: Vorher ist meist ein **delete dest;** notwendig, damit das alte Objekt, auf das **dest** bisher gezeigt hat, freigegeben wird!

- **clone** wird in der Basisklasse **myBaseClass** wie folgt deklariert:

  ```
  virtual myBaseClass* clone() const;
  ```

- Alle Klassen, deren Objekte kopiert werden sollen, erhalten folgende immer gleiche Implementierung von **clone** unter Verwendung des *Copy-Konstruktors der jeweiligen abgeleiteten Klasse*:

  ```
  virtual Abgel* clone() const {
    return new Abgel(*this);
  }
  ```

  (In der Basisklasse ist **clone** entweder ebenso definiert oder pure virtual.)

Mehrfachvererbung

C++ erlaubt Mehrfachvererbung:
Nach dem **`class`** einer Klasse können *mehrere Basisklassen* angegeben werden, z.B.

`class myClass : public Base1, public Base2`

- Die erste Problematik ergibt sich, wenn **`Base1`** und **`Base2`** Member oder Methoden mit *demselben Namen* enthalten. In diesem Fall muss bei jeder Verwendung mit **`Base1::`** oder **`Base2::`** angegeben werden, *aus welcher Basisklasse* der Namen kommen soll.
- Die zweite Problematik ergibt sich, wenn *dieselbe "Großvaterklasse"* über zwei davon abgeleitete Vaterklassen *doppelt geerbt* wird: Dadurch würden die Objekte der aktuellen Klasse *zwei* Instanzen der Objekte der Großvaterklasse enthalten.

  In diesem Fall muss die Großvaterklasse in den Vaterklassen mit dem Schlüsselwort **`virtual`** geerbt werden, dann gibt es die Großvaterklasse im aktuellen Objekt nur einmal (außer es sind wirklich 2 Instanzen gewünscht, dann ist wieder **`::`** statt **`virtual`** erforderlich).

Wir befassen uns mit diesem Thema nicht weiter. Auch wenn die genannten Probleme damit technisch gelöst sind, ist das Ergebnis schwer zu lesen. Solche Konstrukte sollten daher vermieden werden: Man sollte sich bemühen, sein Klassendesign *einfach* zu halten, z.B. nach den Regeln von Java (nur *eine* Vaterklasse darf wirklich Member enthalten, alle anderen Vaterklassen sind reine Schnittstellen-Klassen, d.h. enthalten nur Prototypen, keine Member).

In Java heißt eine abstrakte Klasse, die nur eine Schnittstelle (Methoden) definiert, aber keine Daten (Member) enthält, "Interface". In C++ wird nicht zwischen "echten" Klassen und Interfaces unterschieden.

6. Kapitel: Operator Overloading

Bisher kennen wir zwei Wege, Funktionen und Methoden *mit demselben Namen* mehrfach zu definieren:

- Das *Overloading*:

  Hier handelt es sich um *völlig getrennte, unabhängige Funktionen* (ohne Vererbungs-Beziehung), die sich *durch die Typen der Parameter unterscheiden*.

  Der Compiler entscheidet *zur Compile-Zeit auf Grund der Typen der Argumentwerte*, welche der Funktionen aufzurufen ist.

- Der *Polymorphismus mittels Ableitung und Überschreiben*:

  Hier handelt es sich um *verschiedene Implementierungen derselben Schnittstelle*, d.h. einer einzigen Funktionsdeklaration *mit denselben Parameter-Typen*, in verschiedenen Klassen, die in einer Ableitungs-Beziehung zueinander stehen.

  Hier entscheidet entweder der Compiler auf Grund des deklarierten *Typs des Objektes* (bei Methoden ohne **`virtual`**) oder das Laufzeit-System auf Grund des tatsächlichen Typs des Objektes (bei Methoden mit **`virtual`**), welche Implementierung der Methode (aus welcher Klasse) zu verwenden ist.

Alle diese Möglichkeiten wollen wir nicht nur für Funktionen, sondern *auch für Operatoren* nutzen können:

- Wir wollen *selbst Code für Operatoren* definieren können, damit Operatoren wie **+** oder Vergleiche wie **<** auch auf *Objekte* anwendbar sind.
- Wir wollen denselben Operator dabei *mehrfach für verschiedene Operanden-Typen* definieren können (Overloading).
- Wir wollen *Operatoren* nicht nur als globale Funktionen, sondern auch *als Methoden einer Klasse* definieren können,
  - … damit sie auch auf private Member zugreifen können,
  - … und damit Mechanismen wie Vererbung, Überschreiben und **`virtual`** auch für Operatoren nutzbar sind.

In C++ geht das alles ganz einfach:

Man kann Operatoren *völlig analog zu normalen Funktionen* deklarieren und definieren, indem man als Funktionsname "**`operator`**…" verwendet, also z.B. **`operator+`** oder **`operator==`** .

Man hat dabei freie Wahl bei den Operanden-Typen:
Links und rechts können verschiedene Typen stehen.

<< und **>>** für Ein- und Ausgabe sind beispielsweise auf diese Weise definiert.

Folgende Eigenschaften von Operatoren können allerdings *nicht umdefiniert* werden:

- *Anzahl* der Operanden:
  Man kann beispielsweise weder ein binäres **++** noch ein unäres **/** definieren.
- Die *Priorität* (Vorrangregeln):
  Auch ein selbstdefiniertes **+** kommt immer nach einem ***** .

- Die *Bindungsrichtung*, d.h. ob gleichrangige Operatoren von links nach rechts oder von rechts nach links berechnet werden: Mehrfache Zuweisungen laufen beispielsweise immer von rechts nach links, mehrfache Rechenzeichen immer von links nach rechts.

Methode oder Funktion?

Operatoren können sowohl *als globale Funktion* als auch *als Methode* definiert werden.

Üblicherweise richtet sich das nach dem Typ des ersten (linken) Operanden:

- Soll der Operator als **Methode** aufgerufen werden, muss er eine *Methode der Klasse des **linken** Operanden* sein: Es ist nicht möglich, einen Operator als Methode seines rechten Operanden aufzurufen.

  Ein Operator, der als Methode aufgerufen wird, hat *einen Parameter weniger* als er Operanden hat: Der *linke Operand* wird nicht als Parameter übergeben, sondern wird automatisch das eigene Objekt **`this`**.
  Unäre Operatoren haben daher gar keine Parameter.
- Ist das nicht möglich, weil der linke Operand von einem Basistyp wie **`int`** oder von einer vordefinierten bzw. fremden Klasse ist, so muss man den Operanden als **globale Funktion** definieren.

  In diesem Fall hat die Funktion *so viele Parameter*, wie der Operator *Operanden* hat.

  Der wichtigste derartige Fall sind die Operatoren **<< und >> für Ein- und Ausgabe**: Links steht das **`iostream`**-Objekt, nicht das Objekt, das man einlesen oder ausgeben möchte. Man kann **<<** und **>>** daher nicht als Methoden der Klasse definieren, deren Objekte man ein- und ausgeben möchte.

  Man könnte eine eigene Klasse von **`istream`** oder **`ostream`** ableiten und dort das **<<** oder **>>** als Methode hinzufügen, das ist aber völlig unüblich: **<<** und **>>** für I/O-Zwecke werden immer *als globale Funktionen* definiert.

Operator-Funktionen und `friend`

Operator-Funktionen, im speziellen **<<** und **>>**, werden oft als **`friend`** der Klassen der Operanden deklariert, damit sie auf deren Member zugreifen können, obwohl sie nicht zur Klasse gehören. Im **`class`** von **`myClass`** steht dann beispielsweise

```
friend ostream &operator<<(ostream &outFile, const myClass &val);
```

Überladbare Operatoren

Eine Liste aller überladbaren Operatoren samt zu verwendenden Prototypen gibt es hier:

http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Überladbar sind:

- Alle arithmetischen und bitweisen Operatoren.
- Die boolschen Operatoren und Vergleiche.
- Die Zuweisung und die kombinierten Zuweisungs-Operatoren, der Typ-Cast **(***typ***)** .

- Der Index-Operator **[ ]**, die Dereference-Operatoren ***** und **->**
  und der Address-Of-Operator **&** (wir betrachten nur **[ ]**, siehe unten).
- Der Nacheinander-Ausführungs-Operator **,** (betrachten wir nicht weiter).
- Der Funktions-Aufruf **( )** (siehe unten).

Hinweise

- Die Operanden werden normalerweise als **const**-**Referenz** übergeben.
  Operanden, die *geändert* werden sollen, *müssen als Referenz* und nicht als Kopie übergeben werden!

  Auch die Operator-Methode selbst wird **const** deklariert (nach den Parameter-**()**), wenn sie das eigene Objekt (den linken Operanden) nicht ändert.

  Eine Ausnahme sind beispielsweise **++** und **--** und vor allem der Zuweisungs-Operator **=**: Da sie ihren Operanden ändern, fällt das **const** der Methode weg!
- Alle Rechen-Operatoren liefern normalerweise ein *neues Objekt "by value"*
  als Returnwert zurück, d.h. das Ergebnis-Objekt wird in den Aufrufer kopiert.

  Eine Referenz oder ein Pointer sind nicht möglich, weil das Ergebnis ja üblicherweise nicht einer der beiden Operanden ist, sondern in einer neuen, internen, lokalen Variable berechnet wird, die mit dem Return wieder freigegeben wird, und auf die man daher keinen Pointer / keine Referenz zurückgeben darf (eine Ausnahme sind Operatoren, die einen Pointer auf ein mit **new** dynamisch angelegtes Ergebnis zurückgeben).
- Alle Operatoren, die *einen ihrer Operanden als Ergebnis* zurückliefern
  (vor allem die Zuweisung **=**), müssen eine *Referenz* darauf als Returntyp haben:
  In diesem Fall darf *keine Kopie* des Returnwertes gemacht werden,
  sondern der Operand muss im Original per Referenz zurückgegeben werden:
  `return *this;` bei Methoden und
  `return outFile;` bei einer **<<**-Funktion mit linkem Parameter **`outFile`** .
- Man ist nicht an die Argument- und Returntypen der oben zitierten Tabelle gebunden, sie listet nur die Prototypen auf, die die Operatoren normalerweise haben.

  Man darf aber auch Operatoren mit anderen Typen definieren, also beispielsweise mit **+** einen **`int`** als rechten Operanden zu einem Objekt der eigenen Klasse dazuzählen oder einen **%**-Operator schreiben, der aus zwei Objekten der eigenen Klasse einen **`int`** berechnet.
- Es ist nicht möglich (und wäre auch nicht sinnvoll),
  bei Operator-Funktionen Default-Parameter zu definieren.
- Der **`::`** funktioniert auch mit **`operator`**... , also z.B.: **`myClass::operator+(...)`**

Der Zuweisungs-Operator (Assignment Operator)

Der Zuweisungs-Operator kann *nur als Methode*, nicht als globale Funktion definiert werden. Sein Prototyp ist normalerweise

```
myClass &myClass::operator=(const myClass &value);
```

Definiert man keinen solchen Operator, erzeugt der Compiler einen *Default-Assignment-Operator*, der einfach eine *bitweise Kopie* macht (genauer: Der Default-Assignment-Operator ruft für jedes einzelne Member wieder **=** auf. Bei Basistypen und Pointern ist das die normale Zuweisung, bei Membern, die selbst wieder Objekte sind, deren impliziter oder expliziter **operator=**).

Ein expliziter **operator=** ist (so wie ein expliziter Copy-Konstruktor) vor allem dann erforderlich, wenn die Objekte *Pointer als Member-Variablen* enthalten, siehe nächstes Kapitel. Weiters ist ein **operator=** notwendig, wenn Objekte Member enthalten, die einen eindeutigen Wert enthalten müssen und nicht einfach kopiert werden dürfen, z.B. eine fortlaufende Seriennummer, oder wenn beispielsweise die Anzahl der Objekte global mitgezählt werden soll.

Der rechte Operand kann aber auch beliebigen andern Typ haben, man kann z.B. in einer Array-Klasse einen **operator=** mit einem **double**-Wert auf der rechten Seite definieren, der dann in allen Array-Elementen gespeichert wird.

Für **operator=** gelten mehrere wichtige Regeln:

- Der Returnwert ist *immer* eine Referenz des eigenen Objekts, d.h.
  return *this;

  Dies unter anderem deshalb, damit *geschachtelte Zuweisungen* und Mehrfach-Zuweisungen funktionieren.

- Der Fall, dass der rechte Operand das *eigene Objekt* ist (im Fall einer Zuweisung **a = a;**), ist *immer zu prüfen* und geeignet zu behandeln. In den meisten Fällen ist bei einer Selbstzuweisung *nichts zu tun*, also gleich am Anfang

  if (this == &value) return *this;

- Falls notwendig, muss man für die alten Werte des eigenen Objektes zuerst *Aufräum-Arbeiten* durchführen, bevor man sie mit den neuen Werten überschreibt (z.B. den dynamisch angelegten alten Speicher freigeben, auf den die eigenen Pointer-Member zeigen, bevor man sie auf neu angelegten Speicher zeigen lässt).

- Definiert man einen Zuweisungs-Operator in einer abgeleiteten Klasse, so wird man in fast allen Fällen zuerst einmal die Werte der *Basisklasse* mit deren Zuweisungsoperator kopieren:

  this->myBaseClass::operator=(val);

Will man die Zuweisung von Objekten einer Klasse vermeiden (d.h. den impliziten **operator=** unterdrücken), kann man ihn **private** definieren oder explizit deklarieren und undefiniert lassen. Seit C++11 kann / soll man ihn stattdessen mit **= delete** löschen.

Der Index-Operator

Der Index-Operator ist in der Praxis neben **=** und **<<** / **>>** der häufigste selbstgeschriebene Operator: Er wird bei vielen *zusammengesetzten Datenstrukturen* verwendet, um den *Zugriff auf einzelne Elemente* zu implementieren.

Dabei muss in den **[ ]** keine Zahl stehen:
Es kann ebensogut ein Schlüsselwort (String) oder ein komplexes Objekt als Index dienen!

Der **[]**-Operator muss immer eine *Referenz auf das ausgewählte Element* der Datenstruktur zurückliefern, wenn er auch zum Schreiben von Elementen (auf der linken Seite einer Zuweisung) verwendet werden soll: In einen "per Kopie" gelieferten Returnwert kann man nichts speichern, in eine Referenz auf das Original-Element innerhalb der Datenstruktur schon.

Von diesem Operator werden normalerweise immer *zwei* Varianten implementiert (*stets als Methode*, **[ ]** als globale Funktion ist nicht möglich):

```
elemClass &operator[] (int index);
elemClass const &operator[] (int index) const;
```

Beide enthalten in vielen Fällen denselben Code, sind aber für verschiedene Zwecke (die der Compiler automatisch richtig unterscheidet):

- Die erste Variante kann auf der *linken und auf der rechten Seite einer Zuweisung* verwendet werden (d.h. man kann das ausgewählte Element *lesen oder schreiben*), aber kann *nicht* verwendet werden, wenn die Datenstruktur als Ganzes **const** ist.
- Die zweite Variante kann nur zum *Lesen* eines Elementes verwendet werden, aber dafür auch bei *konstanten Datenstrukturen*.

Die Funktionen << und >>

Die Funktionen **<<** und **>>** für Ein- und Ausgabe lauten normalerweise

```
ostream &operator<<(ostream &outFile, const myClass &val);
istream &operator>>(istream &inFile, myClass &var);
```

Sie enden normalerweise mit **return outFile;** (oder **inFile**), damit man mehrere **<<** oder **>>** aneinanderhängen kann.

Der Typ-Cast-Operator

Dieser Operator hat als Namen einen *Typ-Namen*, also z.B. **operator double()** . Dieser Typ *muss* zugleich der Returntyp sein, ein Typ-Cast-Operator hat daher *keinen expliziten Return-Typ*.

Er kann *nur als Methode* und nicht als globale Funktion definiert werden und hat daher *kein* Argument: Ein Typ-Cast hat immer nur ein Argument, und dieses wird implizit das **this**-Objekt des Cast-Operators. Sein Zweck ist,

das eigene Objekt (explizit oder implizit) in einen beliebigen anderen Typ zu verwandeln.

Er ist damit das *Gegenteil vom Typumwandlungs-Konstruktor*,

der aus einem beliebigen anderen Typ ein Objekt der eigenen Klasse erzeugt.

Beispiel (als Methode einer Klasse, die Brüche darstellt, mit zwei Membern **z** und **n** für Zähler und Nenner, zur Umwandlung des Bruches in einen **double**):

```
operator double() const {
  return double(z) / double(n);
}
```

Increment- und Decrement-Operator

Zur Unterscheidung von Prefix-**++** (**++i**) und Postfix-**++** (**i++**) wird beim Postfix-**++** im Prototyp ein zusätzlicher **int**-Parameter angegeben, der ignoriert wird. Dasselbe gilt für **--**.

Die *Präfix*-Operatoren geben eine *Referenz auf das eigene Objekt* als Returnwert zurück, die *Postfix*-Operatoren liefern einen *neuen Wert "by value"* (da ja der Returnwert vom Wert des eigenen Objektes verschieden ist).

Der Function-Call-Operator **( )**

Dieser Operator hat ein paar Besonderheiten:

- Er hat eine *beliebige Anzahl von Argumenten* beliebigen Typs.
- Er hat einen *beliebigen Returntyp*.
- Das Objekt, für den der Operator aufgerufen wird, steht statt dem Funktionsnamen *vor* den **( )** : Es ist *nicht* das erste Argument!

Man kann also *Objekte wie Funktionen* aufrufen, wenn in deren Klasse ein Function-Call-Operator definiert ist.

Beispiel:

```
class eps_equal
{
  public:
    eps_equal(double epsilon) : eps(epsilon) {}

    bool operator()(double a, double b) const {
      return (fabs(a - b) < eps);
    }

  private:
    double eps;
};

  double current, final;
  eps_equal raw_equal(0.1), fine_equal(1e-10);
  ...
  while (!raw_equal(current, final)) ...
```

Höhere C++-Bibliotheken (STL und Boost) machen umfangreichen Gebrauch von derartigen Konstrukten.

Die Operatoren `new` und `delete`

Es gibt auch selbstdefinierbare Operatoren **`new`** und **`delete`**. Diese sind allerdings *nicht ident* mit den Befehlen **`new`** und **`delete`**, sondern nur ein *Teil* davon:

- Die Operatoren **`new`** und **`delete`** betreffen ausschließlich die *Speicherverwaltung*: Mit selbstdefinierten **`new`** und **`delete`** kann man dafür sorgen, dass die Befehle **`new`** und **`delete`** für Objekte einer bestimmten Klasse (nämlich der, für die man eigene Operatoren **`new`** und **`delete`** definiert hat) *statt der eingebauten Speicherverwaltung eine selbstgeschriebene* aufruft (um beispielsweise Speicher aus einem fixen Array zu nehmen oder freigegebene Objekte in einer eigenen Liste zur Wiederverwendung zu halten).
- Die *logische Funktionalität* der Befehle **`new`** und **`delete`** (Aufruf der Konstruktoren bzw. Destruktoren in richtiger Reihenfolge) ist *nicht* änder- oder überschreibbar.

Wir werden uns nicht mit dem Überschreiben von **`new`** und **`delete`** befassen.

Probleme von virtuellen Funktionen und Operatoren

Leider funktioniert der Mechanismus der virtuellen Methoden nicht immer so, wie man das auf den ersten Blick gerne hätte. Das Problem trifft grundsätzlich alle Methoden (egal ob Operator oder nicht), tritt aber bei Operatoren (und hier speziell beim Zuweisungsoperator) besonders häufig auf.

Folgendes muss erfüllt sein, damit eine Methode in einer abgeleiteten Klasse zum Prototypen in der Basisklasse passt:

- Der **Return-Typ** muss *exakt ident* sein, oder er muss beides Mal ein *Pointer oder eine Referenz* auf eine Klasse sein, und die betreffenden Klassen müssen voneinander *abgeleitet* sein.

  Damit können alle Operatoren wie **`operator+`**, deren Returntyp ein Objekt der eigenen Klasse *"by value"* ist (und kein Pointer / keine Referenz darauf) schon *nicht* mehr **`virtual`** sein: Der Return-Typ in der Basisklasse und in der abgeleiteten Klasse ist effektiv verschieden, und damit handelt es sich in der Basisklasse und in der abgeleiteten Klasse um *zwei verschiedene Operatoren*, nicht um zwei Implementierungen desselben virtuellen Operators.
- Die **Parameter-Typen** müssen *exakt ident* sein. Die oben für Returntypen erlaubte Ausnahme für Pointer und Referenzen abgeleiteter Klassen gilt für Parametertypen *nicht*.

  Damit ergibt sich folgendes Problem:

  - Angenommen, wir haben folgende Deklaration in der Basisklasse:

    ```
    virtual Basis &operator=(const Basis &a);
    ```
  - Erster Versuch in der abgeleiteten Klasse `Abgel` wäre folgender:

    ```
    virtual Abgel &operator=(const Abgel &a);
    ```

    Der Returntyp würde passen, aber die Parametertypen sind *verschieden.* Damit handelt es sich um *zwei verschiedene Operatoren* statt um ein Überschreiben des virtuellen Operators der Basisklasse:

Der **operator=** aus **Abgel** kommt nie zur Anwendung, wenn auf Grund der deklarierten Typen zur Laufzeit nach der passenden Implementierung des virtuellen **operator=** aus **Basis** gesucht wird.

- Zweiter Versuch in Abgel wäre folgender:

  ```
  virtual Abgel &operator=(const Basis &a);
  ```

  Das klappt theoretisch: Dieser Operator überschreibt tatsächlich den der Basisklasse und wird zur Laufzeit auch als solcher gefunden.

  Leider kommt dieser Operator aber immer dann zum Zug, wenn die linke Seite der Zuweisung ein Objekt von **Abgel** ist, egal, ob rechts ein Objekt der Klasse **Abgel** steht (der "erwünschte" Fall), oder ein Objekt der Klasse **Basis**, oder schlimmstenfalls ein Objekt einer Klasse **Abgel2**, die parallel zu Abgel von **Basis** abgeleitet ist.

  Mit den Mitteln, die wir lernen, ist es nicht möglich, alle diese Fälle sinnvoll zu unterscheiden und zu behandeln (es geht z.B. mittels Verwendung von **dynamic_cast** oder RTTI, aber es ist sehr aufwändig).

Ähnliches gilt auch für andere Operatoren: Ein virtueller **operator==** wäre z.B. technisch möglich, weil der Returntyp **bool** in allen Ableitungen ident ist, aber der Operator wird nach dem Typ des linken Objektes gewählt, und rechts steht nicht notwendigerweise ein Objekt desselben Typs, sondern es kann jeder von der Basisklasse abgeleitete Typ vorkommen.

In der Praxis kommt daher **virtual operator=** nur selten und mit sehr komplexem Code vor, und die meisten anderen Operatoren sind fast nie **virtual**. In einigen Fällen reicht es, in jeder betroffenen Klasse einen nicht-virtuellen **operator=** zu definieren.

Falls wirklich etwas im Sinne von ***dest = *src;** benötigt wird, auch wenn **src** auf *Objekte einer abgeleiteten Klasse* zeigen kann und für jede dieser Klassen verschiedene Kopiermethoden notwendig sind, so hilft oft **clone** (siehe Kapitel "Vererbung").

7. Kapitel: Pointer und dynamische Daten in Objekten

In vielen Fällen sind unter den *Member-Variablen* einer Klasse auch *Pointer*, beispielsweise auf *dynamisch angelegte* Sub-Objekte oder auf dynamisch angelegte Arrays variabler Größe. In diesen Fällen sind einige Dinge zu beachten.

`new` und `delete` in Objekten

Wenn Objekte Pointer-Member enthalten, wird üblicherweise im *Konstruktor* **`new`** aufgerufen, um Speicher anzufordern und die Pointer-Member darauf zeigen zu lassen. Das kann direkt in der Initialisierungs-Liste geschehen: **`mPtr(new ...)`**

Analog dazu sollte im *Destruktor* ein **`delete`** aufgerufen werden, um den dynamisch angelegten Speicher wieder freizugeben, auf den die Pointer-Member zeigen:
Klassen, die Pointer enthalten, haben daher *praktisch immer einen expliziten Destruktor*.

Shallow und Deep Copy / Compare

Eine *"seichte" Kopie* ist eine Kopie, bei der nur das Objekt selbst (d.h. seine Member-Variablen) kopiert wird, aber nicht die Daten, auf die das Objekt zeigt: Die Pointer-Member werden einfach bitweise kopiert, d.h. die Pointer-Member in der Kopie haben denselben Wert wie im Original und zeigen daher auf dieselben (nicht kopierten!) Daten: Original und Kopie "teilen" sich sozusagen ihre dynamischen Daten: Nur "an der Oberfläche" wird kopiert, die tiefer darunterliegenden Daten werden nicht kopiert.

Normalerweise ist das eine *ganz schlechte Idee:*

- Gibt eines der beiden Objekte die dynamisch angelegten Daten frei (zum Beispiel in seinem Destruktor), *zeigen die Pointer im anderen Objekt auf nicht mehr gültigen Speicher.*
- Ändert eines der beiden Objekte den Inhalt der Daten, auf die seine Pointer zeigen, so *ändert sich auch der Inhalt der dynamischen Daten im anderen Objekt.*

Eine *"tiefe" Kopie* kopiert nicht nur das Objekt selbst, sondern auch (ev. rekursiv!) alle Daten, auf die die Member-Pointer im Objekt zeigen, und läßt die Pointer des kopierten Objektes auf die neu angelegten, kopierten Daten zeigen.

Das ist das, was man sich *in fast allen Fällen wünscht:* Wenn jedes Objekt seine eigenen dynamischen Daten hat (bzw. die *Daten immer nur zu einem einzigen Objekt gehören* und nie zu mehreren gemeinsam), dann treten die oben genannten Probleme nicht auf.

Analog unterscheidet man *Shallow Compare und Deep Compare* zweier Objekte: Vergleicht man nur die Pointer-Member miteinander (ob sie dieselbe Adresse enthalten, d.h. auf denselben Block von Daten zeigen), oder vergleicht man den Inhalt der Daten, auf die die Pointer-Member zeigen?

Im ersten Fall ist das Ergebnis nur dann "gleich", wenn die Objekte eine Shallow Copy voneinander sind (d.h. sich ihre dynamischen Daten teilen):
Für Objekte, deren Pointer auf verschiedene Adressen zeigen, ergibt sich "ungleich", selbst dann, wenn die Daten, auf die sie zeigen, wertmäßig ident sind.

Im zweiten Fall (dem meist gewünschten) ist das Ergebnis genau dann "gleich", wenn die Inhalte der dynamischen Daten gleich sind, auch wenn es getrennte Kopien sind.

Programmierung

Leider macht es C++ *per Default genau verkehrt*: Sowohl der automatisch erzeugte Copy-Konstruktor als auch der automatisch erzeugte **`operator=`** machen eine *bitweise Kopie* des Objektes, ohne sich um Pointer oder dynamisch angelegte Daten zu kümmern, also eine *"Shallow Copy"*.

Wenn eine Klasse Pointer enthält, wird man also fast immer *selbst einen Copy-Konstruktor und einen* **`operator=`** *implementieren* müssen:

- Der Copy-Konstruktor wird neue dynamische Daten anlegen und die Werte des Originals hineinkopieren.
- Der **`operator=`** muss vor dem **`new`** die alten dynamischen Daten des eigenen Objektes mit **`delete`** freigeben, neue anlegen, und die Werte des rechten Objektes hineinkopieren (außer, die alten und die neuen dynamischen Daten sind gleich groß: Dann kann er die alten gleich mit den neuen Werten überschreiben, ohne die eigenen dynamischen Daten vorher freizugeben und frisch anzulegen).

Generell gilt die Faustregel, dass Copy-Konstruktor, **`operator=`** und Destruktor *in 90 % aller Fälle gemeinsam explizit programmiert werden müssen*:
Wenn man einen davon selbst schreibt, sollte man sich immer gut überlegen, ob man nicht auch die beiden anderen braucht!

Den *Vergleich* muss man in beiden Fällen *selbst ausprogrammieren*, indem man die Operatoren **`==`** und **`!=`** selbst definiert: Die vordefinierten Operatoren **`==`** und **`!=`** sind generell nicht auf Objekte anwendbar, es gibt nicht einmal einen automatischen bitweisen Vergleich von Objekten.

Mehrdimensionale dynamische Arrays

Der Versuch, selbst beispielsweise eine Klasse Matrix zu definieren, die intern ein zweidimensionales Array *variabler* Größe enthält, scheitert meist:

- Ein **`new`** für zwei- oder mehrdimensionale Arrays ist nur mit Einschränkungen möglich: Alle Dimensionen außer der ersten (vordersten) müssen *konstant* sein!

  Ein **`new`**, das eine quadratische Matrix variabler Größe liefert, ist also *nicht möglich.*
- Selbst wenn die hinteren Dimensionen konstant sind, ist die richtige Codierung anspruchsvoll und das resultierende Programm oft schwer zu lesen.

  Rein technisch legt **`new`** nämlich immer nur eindimensionale Arrays an:
  Die vorderste Dimension gibt die Anzahl der Elemente an, die das von **`new`** angelegte eindimensionale Array haben soll, und alle anderen Dimensionen gehören nicht zum **`new`**-Aufruf selbst, sondern zur Beschreibung des Typs der Elemente (der natürlich *nicht variabel groß* sein kann).
  `new` liefert dann als Ergebnis einen Pointer auf ein eindimensionales Array, dessen einzelne Elemente wieder Arrays sind.

  Am besten ist es daher, ein **`typedef`** für ein Array mit allen Dimensionen außer der vordersten zu machen (also im zweidimensionalen Fall ein **`typedef`** für ein eindimensionales Array), und dann mit **`new`** ein eindimensionales Array anzulegen, dessen Elemente vom definierten Array-Typ sind.

Es gibt drei Möglichkeiten, in mehreren Dimensionen variabel große Arrays nachzubauen:

- Man legt statt einem zweidimensionalen Array ein eindimensionales Arrays mit **`zeilenanzahl * spaltenanzahl`** Elementen an und macht die Index-Rechnung bei jedem Zugriff selbst:

  `array[zeile * spaltenanzahl + spalte]`

- Man verwendet (so wie wir das in C besprochen und geübt haben) statt eines zweidimensionalen Arrays ein eindimensionales Array von Pointern (so viele, wie man Zeilen hat), von denen jeder wiederum auf ein eindimensionales Array zeigt (ein Array pro Zeile mit je einem Element pro Spalte).

  In diesem Fall braucht man im Konstruktor und an jeder Stelle, an der eine neue Matrix erzeugt werden soll, zuerst ein einzelnes **`new`** für das Pointer-Array und dann in einer **`for`**-Schleife **`zeilenanzahl`** weitere **`new`** für die einzelnen Zeilen. Analog dazu muss man im Destruktor zuerst in einer Schleife die **`delete`**-Aufrufe für die einzelnen Zeilen machen und dann ein einzelnes **`delete`** für das Array von Pointern.

- Man verwendet Typen aus Libraries:
  - Ein üblicher Weg ist, das Template **`vector`** aus der STL zu schachteln, also ein **`vector`** mit **`vector`**-Elementen
    (Templates und die STL behandeln wir später).
  - Noch besser eignet sich das Template **`multi_array`** aus der Boost-Library (die wir allerdings nicht behandeln).

8. Kapitel: GUI-Programmierung

Klassenbibliotheken ("GUI Toolkits")

Wie in Kapitel 3 erwähnt, ist die GUI-Programmierung eines der Hauptanwendungsgebiete von C++. Im *C++-Standard* gibt es jedoch ***keine*** GUI-Klassen (im Unterschied zu Java und C#, wo die GUI-Klassen Bestandteil des Standards sind), aber in der Praxis haben sich einige große C++-GUI-Klassenbibliotheken etabliert.

Abgesehen von proprietären und / oder plattformspezifischen Klassenbibliotheken (Hauptvertreter **MFC = "Microsoft Foundation Classes"** für Windows und GTK für Linux, wobei GTK die Objektorientierung unter C nachbaut, aber auch eine C++-Portierung hat: **gtkmm**) sind folgende freie und portable (Windows, Linux, Mac, embedded Systems, …) GUI-Bibliotheken erwähnenswert:

- **Qt:** Mit Abstand die bedeutendste und umfangreichste freie, portable C++-GUI-Bibliothek. Weit verbreitet, u.a. Grundlage von KDE und VLC.

  Vorteile:

  - Sehr ausgereift und stabil.
  - Plattformen: Windows, Linux, Mac, iOS, Android, embedded ("bare Metal"), …
  - Sehr umfangreich, viele Features, incl. Audio/Video, HTML, SVG, XML, OpenGL & OpenVG, Datenbankanbindung, Skriptsprachen-Interpreter, …
  - Wird von eigener Firma entwickelt, die nur für Qt da ist
    ==› Zahlreiche Entwickler
    Unterstützung von vielen Firmen (Ex-Nokia, SuSE) und Projekten

  Probleme:

  - Sehr groß ==› hoher Ressourcenbedarf! (Entwicklung & Laufzeit)
  - Nicht reines C++: Für das Event-Handling (siehe unten) wurden Spracherweiterungen eingeführt, die von einem eigenen Precompiler (**moc**) nach C++ übersetzt werden.
  - Die nicht-GUI-Klassen duplizieren einen Großteil der Funktionalität von STL und Boost: Eigene Listen-, Vektor- und String-Klassen, …

    Man hat in größeren Projekten mit GUI- und nicht-GUI-Teil dann mitunter Qt und STL/Boost parallel in Verwendung, mit doppeltem Code und Konvertierungsproblemen.
  - Früher nur eingeschränkt frei, jetzt "dual licensed" ((L)GPL + kommerziell), rechtliche Lage dürfte sich mit Version 6 eher wieder verschlechtern.
  - Alle 7-10 Jahre nicht rückwärtskompatibles Major Release

  ==› Verwenden wir nicht…

- **wxWidgets:** Reicht GUI-mäßig fast an Qt heran (ist aber optisch nicht so schön), aber hat viel weniger umfangreiche nicht-GUI-Klassen und bei weitem nicht die Verbreitung von Qt. Verwendet u.a. für CodeBlocks, Codelite, Audacity, Filezilla, Veracrypt und andere Multi-Plattform-Tools.

Vorteile:

- Geringerer Ressourcen-Verbrauch als Qt
  (bei der Entwicklung und zur Laufzeit).
- Unter Windows kein Visual Studio nötig.
- Wirklich frei, auch für kommerzielle Projekte uneingeschränkt verwendbar
  (LGPL mit Zusätzen).
- Ebenso gut und umfangreich dokumentiert wie Qt.
- API (Klassenschnittstellen) änderte sich früher zwischen Versionen öfters.
  Inzwischen sehr stabiles API, sehr wenige Programm-Anpassungen nötig.

Nachteile:

- Hat nicht die Qualität / Schönheit von Qt, sehr kleine Entwickler-Gemeinde,
  weniger Bugfixes, sehr langsamer Entwicklungs-Fortschritt.
- Einige böse Makro-Hacks, vor allem in alten Versionen
  (inzwischen an vielen Stellen durch modernes C++ ersetzt).
- Der nicht-GUI-Teil dupliziert auch zum Teil die STL, ist aber primitiver.

==› Schauen wir uns an…

- **FLTK:** Eine "minimale" nur-GUI-Klassenbibliothek, besonders klein,
  auch frei für freie und kommerzielle Projekte. Im "Freak"-Bereich gern verwendet,
  aber kaum für größere Projekte eingesetzt. Uralt, nutzt objektorientierte Konzepte
  teilweise nicht sauber bzw. etwas primitiv.

Programmierkonzepte

- **GUI-Elemente** (Buttons, Textfelder, Menüzeilen, …) sind Objekte von
  entsprechenden Klassen, die in einer komplexen **Ableitungs-Hierarchie** zueinander
  stehen (GUI-Objekte ==› Fenster, Dialogelemente ==› Button, Textfeld usw.).
- Daneben gibt es auch noch eine **Anordnungs-Hierarchie** (*geometrische
  Verschachtelung* der GUI-Elemente): Ein "normales" Fenster hat normalerweise
  eine Menü- und eine Statuszeile, Toolbars, Splitter (Trennstriche für Teilfenster)
  und Text- oder Grafikbereiche, ein Dialogfenster hat diverse Dialogelemente,
  ein Fenster mit Tabs hat pro Tab ein Unterfenster, ….
- Neben den sichtbaren Elementen spielen – vor allem bei Dialogen – die **Sizer**
  eine wichtige Rolle: Ein Sizer bekommt einen bestimmten Teilbereich des Fensters
  zugeteilt und *verteilt diesen Platz* nach bestimmten Regeln (gleichmäßig, …) auf
  die in ihm liegenden GUI-Elemente: Er kümmert sich um das **Layout**, d.h. passt die
  Größe und Lage der Subelemente an, wenn die Fenstergröße neu festgelegt wird.
- **Events**:

  Ein GUI-Programm funktioniert **Event-getrieben**, d.h. der Code wird
  normalerweise nicht der Reihe nach "in einem Rutsch" von Anfang bis Ende
  sequentiell ausgeführt wie bei einem normalen Programm, sondern das Programm
  wartet auf Events (meist *Benutzer-Interaktionen wie Mausklicks und Tastendrücke*,
  aber auch *abgelaufene Timer*) und führt bei jedem Event ein kurzes Stück Code –
  den **Event Handler** – aus.

Daraus folgt:

- GUI-Programme haben meist ein *fix vorgegebenes* **`main`** bzw. basieren auf einem immer gleichen zentralen Codestück, der **Event Loop**: Sie wartet auf einen Event, ruft die zum Event gehörende Methode auf, wartet wieder usw.
- Ein GUI-Programm besteht daher *zu einem großen Teil* aus **Event Handlern** (manchmal auch *"Callbacks"* genannt, im Grunde normale Funktionen oder Methoden) sowie aus der *Zuordnung der Event Handler zu Events* (in Qt: "Signals" und "Slots").

  Diese Zuordnung kann *statisch* (mittels *fix initialisierter Tabellen*) oder *dynamisch* (durch *Aufruf einer Funktion zum Verbinden* eines bestimmten *Events* mit dem Funktions- bzw. *Methoden-Pointer* des dazugehörigen Handlers) erfolgen (in wxWidgets: **`Bind`** oder **`Connect`**)
- Vordefinierter Code in der Library sorgt dafür, dass ein Event zum richtigen Handler findet. Das geschieht normalerweise *von innen nach außen*: Jeder Klick geht zuerst einmal an das *angeklickte Element*, jeder Tastendruck an das Element, das gerade den *Tastatur-Fokus* hat. Dann wird er von dort *in der Schachtelung der GUI-Elemente (Anordnungs-Hierarchie)* nach außen durchgereicht, bis hin zum Hauptfenster ganz außen, bis sich ein Element "zuständig fühlt" und einen passenden Event-Handler hat, der die zum Klick oder Tastendruck gehörende Funktion repräsentiert.
- Problematisch ist das Konzept von Event Loop und Event Handlern bei **langlaufenden Berechnungen**, weil die *gesamte GUI-Applikation völlig blockiert* ist (nicht auf Tastendrücke, Klicks, Window-Close oder sonstwas reagiert), solange ein Event Handler rechnet: Wenn die Event Loop einen Event Handler aufgerufen hat, kann sie *keine neuen Events* entgegennehmen und verteilen, bis dieser Handler zurückgekehrt ist.

  Wenn Berechnungen notwendig sind, die über einige Millisekunden hinausgehen, ist es daher notwendig, die Anwendung **multithreaded** zu programmieren: Ein Thread führt die Event Loop aus, "Langläufer" rechnen parallel dazu in eigenen Threads.

- Die **Programmiermethodik** für GUI-Programme ist in etwa Folgende:
  - Man *leitet* von den vordefinierten "äußeren" Klassen (Applikation, Toplevel-Fenster, Dialog-Fenster) *eigene Klassen ab* (pro Fenster / Dialog eine). Die vordefinierten Klassen sind normalerweise *leere Schablonen*, d.h. enthalten keine Subelemente und keine Anwendungs-Funktionalität.
  - Man definiert die *Sub-Elemente* (Dialog-Elemente, Sizer, Splitter, Sub-Frames, ...) als *Membervariablen* der eigenen Toplevel-Klassen (nur nötig, wenn Event-Handler darauf zugreifen sollen: Sub-Elemente, die nur im Konstruktor bzw. Init angesprochen werden, müssen *nicht* Member sein).
  - Man erweitert diese abgeleiteten Klassen um die *Event-Handler-Methoden*. Da die Event-Handler Methoden z.B. der eigenen Toplevel-Fenster-Klasse sind, können sie auf deren Member, d.h. deren Sub-Elemente, zugreifen.

- Man *initialisiert* die Sub-Elemente, deren Anordnung (Zuordnung zu Sizern) und deren Event-Handler entweder *im Konstruktor* der eigenen Toplevel-Klasse oder durch Überschreiben seiner virtuellen **`Init`**-Methode.
  Auch die Menüs, Statuszeilen usw. werden auf diesem Weg initialisiert.

  Ebenso gehört das Aufräumen in den Destruktor oder ein geerbtes **`Quit`**.

Unterlagen

… online suchen: wxWidgets Klassenreferenz, Tutorial, Beispiel-Code, …

Weiters: Mein "Hello"-Beispiel (online), Buch Jürgen Wolf "C++ von A bis Z" (alte Version)

Selbst Zeichnen in wxWidgets

Grundsätzlich muss man sich um das eigentliche Zeichnen in GUI-Programmen nicht kümmern: Das GUI-Toolkit zeichnet alle Dialogelemente, Menüzeilen usw. selbständig im richtigen Moment neu. Nur wenn man ein Unterfenster selbst mit grafischen Inhalten, schön formatiertem Text usw. füllen will (zum Beispiel das *Dokumenten-Subfenster* einer Office-Applikation oder die *Grafik-Anzeige* eines Spiels), muss man *selbst zeichnen*.

In wxWidgets dient dazu die abstrakte Klasse **`wxDC`** ("Device Context"): Sie bietet ein *einheitliches Interface* zum Zeichnen von Linien, geometrischen Figuren, Bitmaps und Text auf einer *rechteckigen Pixelfläche*. Die Ableitungen dieser Klasse realisieren diese Zeichen-Methoden auf den verschiedenen *Darstellungs-Medien*: Fenster am Schirm (oder der Schirm als Ganzes), Drucker-Seiten, reine in-Memory-Bitmaps, Postscript-Files usw.. Dabei sind auch *Koordinatentransformationen* möglich: Ein DC kann skaliert werden, oder es kann in einem Fenster nur ein scrollbarer Ausschnitt eines DC's angezeigt werden.

Für das *Zeichnen am Bildschirm* in einem GUI-Programm (bzw. in einem Teilbereich von dessen Fenster) ist die von **`wxDC`** abgeleitete Klasse **`wxPaintDC`** zuständig (oder die davon *abgeleiteten* Klassen **`wxAutoBufferedPaintDC`** und **`wxBufferedPaintDC`**, die sich bezüglich Doppelpufferung des Bildschirmspeichers unterscheiden, um unschöne Effekte bei Bildschirm-Updates zu vermeiden).

Grundsätzlich gibt es **zwei Momente**, in denen etwas am Bildschirm zu zeichnen ist:

1. *Das Programm will etwas zeichnen*, z.B. weil der *Benutzer* etwas mit der Maus gezeichnet, eingetippt oder sonstige Events ausgelöst hat, die die *Darstellung im Fenster gezielt verändern*, oder weil neue darzustellende Daten eingetroffen sind.

   ==› Die Zeichenoperationen geschehen gedanklich in *"normalen" Event-Handlern*.

2. *wxWidgets selbst will das Grafikfenster neu zeichnen*, ohne dass der Benutzer irgendeine Zeichenaktion getätigt hat. Das passiert beispielsweise, wenn die *Fenstergröße geändert* wird (Resize), wenn die Anwendung *ganz oder teilweise verdeckt* war und wieder in den Vordergrund kommt, oder wenn man eine Anwendung mit mehreren Karteiblättern (Tabs) hat und zwischen diesen umschaltet.

   ==› In diesem Fall löst wxWidgets einen *Paint-Event* aus, für den das Programm so wie für andere Events auch einen *Event-Handler schreiben* und mit dem Event verbinden muss, und der dann für das Neuzeichnen aufgerufen wird.

Früher waren *beide Fälle* in wxWidgets *direkt* unterstützt: Man konnte von *jeder Stelle im Code* direkt zeichnen. Da dies aber nicht mehr auf allen Plattformen möglich ist, die von wxWidgets unterstützt werden, ist heute ***nur mehr Fall 2 erlaubt***.
Direkt Zeichnen *aus anderen Event-Handlern* kann auf manchen Plattformen noch den gewünschten Effekt haben, aber das ist Glückssache und *nicht mehr offiziell unterstützt*.
Man muss *immer Fall 2 programmieren* und *Fall 1 indirekt über Fall 2 erledigen*.

Zu Fall 2:

- Man leitet für das ***Hauptfenster*** eine *eigene Klasse* von **`wxFrame`** ab, z.B **`myFrame`**.
  Diese Klasse kümmert sich u.a. um *alle anderen GUI-Elemente* und *deren GUI-Events* sowie z.B. um den Close-Event.
- Man leitet von **`wxWindow`** eine *eigene Klasse für die **Zeichenfläche*** ab, z.B. **`myWin`**.

  Deren *Konstruktur* hat meist das *übergeordnete (Haupt)fenster als Parameter*.
- Die **`myFrame`**-Klasse bekommt einen **`myWin`**-*Pointer* als *Member*.

  Im **`myFrame`**-Konstruktor wird ein **`myWin`**-Objekt *dynamisch* angelegt
  (dabei wird in den meisten Fällen **`this`** als Vater-Window mitgegeben!)
  und in diesem Member gespeichert.

  Weiters wird dieses **`myWin`**-Objekt im **`myFrame`**-Konstruktor gemeinsam
  mit allen anderen GUI-Elementen *durch Sizer im Hauptfenster angeordnet*.
- **`myWin`** bekommt einen ***Event-Handler*** für **`wxPaintEvent`**.
  Er wird *im* **`myWin`**-*Konstruktor* an den Paint-Event *gebunden*.

  Dieser Event-Handler von **`myWin`** legt als erstes ein ***lokales* `wxPaintDC`**-*Objekt* an
  (*nicht* dynamisch oder global anlegen!), typischerweise mit **`this`** als *Parameter*
  beim Anlegen.

  Gibt es dabei ein Problem mit *Bildschirm-Flimmern* oder *halb gezeichneten Frames*,
  so kann man **`wxAutoBufferedPaintDC`** statt **`wxPaintDC`** verwenden.

  Falls der Event-Handler *zu umfangreich* und in *mehrere* Funktionen *aufgespalten*
  wird: Das **`wxPaintDC`**-Objekt *immer "by Reference"* übergeben, *nie "by Value"*!

  Die *eigentlichen Zeichen-Operationen* werden alle durch Aufruf entsprechender
  `wxPaintDC`-*Methoden* (bzw. **`wxDC`**-Methoden, siehe *Online-Doku für* **`wxDC`**!)
  für dieses **`wxPaintDC`**-Objekt *innerhalb* dieses Event-Handlers ausgeführt.
- *Welche Informationen* die Zeichenfläche wo und wie darstellen soll,
  wird meist *in Membern von* **`myWin`** *gespeichert*
  (globale Variablen wären technisch auch möglich, sind aber weniger schön).
  Diese Member repräsentieren beispielsweise das *Spielfeld* oder *Daten einer Grafik*.
  Sie werden typischerweise *von anderen Programm-Teilen gesetzt*
  und im **`wxPaintEvent`**-Handler gelesen, ihr Inhalt *steuert die Zeichen-Operationen*.

  Auch *Hilfsfunktionen* zum Berechnen und Befüllen dieser Member
  werden typischerweise als *Methoden von* **`myWin`** implementiert.

Zu Fall 1:

- Alle *Code-Stücke und Event-Handler*, die *die Anzeige inhaltlich ändern* wollen,
  zeichnen nicht direkt, sondern *ändern nur die Member* von **`myWin`**,
  die festlegen, welche Informationen im Zeichenfenster angezeigt werden sollen.

In vielen Fällen gehören diese Event-Handler zu **myFrame**,
können aber über das **myWin**-Pointer-Member von **myFrame**
auf die **myWin**-Member und die dazugehörigen **myWin**-Hilfsmethoden zugreifen.

- Danach müssen alle Code-Stücke, die *die Anzeige tatsächlich aktualisieren* wollen, an ihrem Ende die Methode **Refresh** für das **myWin**-Member *aufrufen*.

  Das bewirkt, dass sich ***"das Programm selbst einen* wxPaintEvent *schickt"***, d.h. dass *nach dem Ende* des aktuellen Event-Handlers der **wxPaintEvent**-*Handler ausgeführt* wird und die Zeichenfläche entsprechend den neuen Member-Werten *neu zeichnet*.

  Wenn *nur ein Ausschnitt* der gesamten Zeichenfläche *aktualisiert* werden soll, kann man **RefreshRect** statt **Refresh** aufrufen und dabei *ein Rechteck angeben*, das die neu zu zeichnende Teilfläche angibt.

Mausklicks in die Zeichenfläche:

- Falls die Zeichenfläche auf *Mausklicks* reagieren soll, bekommt **myWin** auch einen *Event-Handler* für **wxMouseEvent**.

  Auch für diesen Event-Handler gilt: Die **myWin**-Member entsprechend *ändern* und dann am Ende **Refresh** *aufrufen* (in diesem Fall für sich selbst) (*nicht* selbst zeichnen!).

- Zum Zeichnen braucht man oft auch exakte *Maus-Event-Koordinaten*:
  Hier gilt dasselbe wie bei vielen anderen Events auch:
  Der Maus-Event-Handler bekommt den *Maus-Klick-Event als Parameter* übergeben, und dieser *Event* ist selbst ein *Objekt*, das nähere Informationen zum Event (zum Beispiel die *Maus-Koordinaten*) als *interne Member* enthält, die man mit vordefinierten Get-Methoden abfragen kann.

Langlaufende Zeichen-Operationen, Reihenfolge des Zeichnens:

Leider hat diese wxWidgets-spezifische Zeichen-Strategie ausschließlich mittels **Refresh** und **wxPaintEvent**-Handler nicht immer den gewünschten Effekt:

1. Mit der bisher vorgestellten Methode lassen sich nur Event-Handler realisieren, die die Anzeige *pro Event* ***ein einziges Mal am Ende*** *aktualisieren*.
   Ein Programm, das *längere, durchgehende Berechnungen* ausführt und *dabei* ***laufend*** *die Anzeige aktualisiert*, ist auf diesem Wege *nicht* möglich:

   Der durch ein **Refresh** ausgelöste **wxPaintEvent**-Handler läuft ja *erst* ***nachdem*** der Event-Handler *geendet* hat, der das **Refresh** ausgelöst hat, und die Event Loop den *nächsten* Event abarbeiten kann:
   Solange der aktuelle Event-Handler noch läuft, wirkt das* Refresh *nicht, d.h. es wird nichts gezeichnet!

2. Wenn ein Event-Handler *zuerst* etwas *zeichnen will* und *danach* beispielsweise eine *Messagebox anzeigt*, so wird *zuerst die Messagebox* angezeigt, und die *Zeichenoperation* erfolgt erst, *nachdem* die Messagebox *geschlossen* wurde:

   Der **wxMessageBox**-Aufruf zeigt die Messagebox *sofort beim Aufruf* an, während das durch **Refresh** ausgelöste *Zeichnen erst nach der Rückkehr* des **wxMessageBox**-Aufrufs und des Event Handlers in der Event Loop erfolgt.

Beide Fälle erfordern eine der konkreten Aufgabestellung angepasste *Lösungs-Strategie*:

- Im einfachsten Fall hilft ein Aufruf von **wxYield()** oder **wxSafeYield()** *unmittelbar nach* dem **Refresh()**: Beide bewirken, dass innerhalb des aktuellen Event-Handlers sofort eine *zweite, geschachtelte Event Loop gestartet* wird, die alle ausstehenden Events abarbeitet und daher den **wxPaintEvent**-Handler zum **Refresh** ausführt.
  Erst *danach* wird der Code des aktuellen Event-Handlers *fortgesetzt*.

  Diese Strategie hilft einerseits, wenn ein langlaufender Event-Handler *laufend* die Anzeige updaten will, und andererseits, wenn ein Event-Handler *zuerst* die Anzeige aktualisieren und *erst danach* z.B. eine Messagebox anzeigen will.

  Das **Problem** ist, dass die geschachtelte Event Loop nicht nur den **wxPaintEvent** abarbeitet, sondern ***alle*** *ausstehenden Events und Benutzer-Interaktionen*.
  Viele Event-Handler und wxWidgets-Funktionen sind aber *nicht darauf ausgelegt*, geschachtelt bzw. innerhalb eines anderen Event-Handlers zu laufen, was zu *seltsamen Effekten bis hin zu Abstürzen* führen kann (beispielsweise dann, wenn ein **Close**-Event des Systems ansteht und aufräumt, aber der eigene Event-Handler danach noch Code nach dem **Yield** ausführt).
  wxSafeYield() ist *etwas sicherer*, weil es während der geschachtelten Event Loop weitere *Benutzer-Interaktionen blockiert*, aber es ist auch kein Allheilmittel.

  Diese Lösungs-Strategie sollte man also nur *nach sehr sorgfältigem Testen einsetzen*.
  Speziell für das Programmende ist meist eine *mehrstufige* Vorgehensweise zu implementieren:
  - Der langlaufende Event-Handler setzt vor jedem **wxYield()** ein *Flag*, an dem *andere Event-Handler erkennen* können, dass sie *geschachtelt* laufen.
  - Jeder "kritische" Event-Handler (vor allem für das Beenden) *prüft* dieses Flag: Läuft er *geschachtelt*, setzt er *nur irgendwelche Flags* (und tut sonst *nichts*), *sonst* führt er *die gewünschte Aktion* wie üblich direkt aus.
  - Der langlaufende Event-Handler setzt nach jedem **wxYield()** sein Flag zurück, *prüft* dann, ob andere Flags *ausständige Operationen* anderer Events anzeigen, und führt gegebenenfalls entsprechende Aktionen aus.

- Eine weitere häufige (aber aufwendige) Lösungs-Strategie besteht darin, die langlaufende Berechnung aus dem Event-Handler *in einen explizit programmierten parallelen Thread auszulagern*, der unabhängig von der Event Loop und den Event-Handlern läuft und daher die *Event Loop nicht mehr blockiert*. Damit werden vom Thread ausgelöste *Paint Events abgearbeitet, während der Thread rechnet*.

- Eine dritte Strategie ist, den Event-Handler so zu programmieren, dass er als Reaktion auf einen Event *nicht einmal lang läuft, sondern viele Male kurz*:
  Der Event-Handler kann bei jedem Durchlauf nach dem Refresh mit **QueueEvent** *"seinen eigenen" Event sofort wieder auslösen* und sich dann vorläufig beenden.
  Dadurch kommt die Event Loop zum Zug, arbeitet zuerst einmal den Paint-Event und andere aufgestaute Events ab, und ruft danach sofort wieder den langlaufenden Event-Handler auf.

9. Kapitel: Exceptions

Die klassische Fehlerbehandlung in C *verunstaltet* das Programm
und macht es schwer lesbar / wartbar:

- Durch die *Fehlerprüfung* ergeben sich viele zusätzliche **`if`**'s,
  "Schleifen-Notausgänge" und **`return`**'s.
- In allen Funktionen, in denen potentiell ein Fehler auftreten kann, müssen
  entweder *Werte im Returnwert für Fehlercodes reserviert* oder *zusätzliche Parameter*
  eingeführt werden. Das geht allerdings *nicht bei Konstruktoren und Destruktoren*,
  weil man dort keine Möglichkeit zu Fehler-Returnwerten hat.

  Eine Lösung mit globaler Variable (**`errno`**) widerspricht
  modernem Software-Design und ist *nicht Multithreading-fest*
  (Fehler in einem Thread überschreiben den Fehlerstatus anderer Threads).
- Die *Fehlerbehandlung* selbst ist über das gesamte Programm verstreut
  und daher meist inkonsistent und schwer zu warten / zu ändern
  (es muss an vielen Stellen modifiziert werden).

Eine erste Notlösung in C war eine Fehlerbehandlung mit **`setjmp`** / **`longjmp`**.
Diese ist aber nicht auf C++ übertragbar, weil sie Funktionen am **`return`** vorbei verlässt
und dadurch die *Destruktoren lokaler Objekte nicht ausgeführt* werden.

In C++ (und auch Java, C#, …) wurden daher **"Exceptions" ("Ausnahmen")** eingeführt,
ein neues Konzept zur Fehlerbehandlung.

Sie haben folgende Vorteile und sollten daher im Normalfall auch verwendet werden:

- Außer dem **`if`** zur Erkennung des Fehlers selbst
  bleibt die *Programmlogik und -struktur unverändert*:
  Exceptions funktionieren *unabhängig vom normalen Programmablauf*,
  der Fehlerbehandlungs-Code ist vom normalen Programmcode *getrennt*.
- Die Fehlerbehandlung kann im Programm beliebig plaziert werden:
  Entweder nahe am Auftreten des Fehlers, oder ganz zentral für alle Fehler
  im **`main`**, oder irgendwo dazwischen.
- Exceptions verwenden ein eigens erzeugtes *Fehlerobjekt*,
  um Informationen über den Fehler zu speichern und vom Auftreten des Fehlers
  bis zu seiner Behandlung zu übertragen. Sie benötigen daher
  weder zusätzliche Returnwerte oder Parameter noch globale oder statische Variablen.

  Damit sind Exceptions auch *threadsafe*.
- Exceptions kümmern sich um das *korrekte Aufräumen* von Objekten (siehe unten).

Exceptions sind in der Implementierung allerdings ziemlich *teuer* (*auch dann, wenn sie nicht verwendet werden*, weil zur Laufzeit zusätzliche interne Verwaltungsinformation
für alle Daten und Funktionen gespeichert werden muss). Viele Compiler bieten daher
die Möglichkeit, Code ohne irgendwelche Vorkehrungen für Exceptions zu compilieren.
Dieser Modus wird auf Kleinstrechnern und in hoch echtzeit-kritischen Systemen
oft verwendet, der Exception-Mechanismus steht dann nicht zur Verfügung.

Grundidee

- Tritt ein Fehler auf, wird eine Exception *"geworfen"*,
  d.h. ein *Fehlerwert* (mit Informationen zum Fehler).
- Das *unterbricht die normale Programmlogik*:
  Das Programm macht *nicht* mit dem nächsten Befehl weiter,
  sondern sucht das Programmstück, das *"die Exception fängt"*,
  d.h. sich speziell um die Behandlung von Fehlerwerten dieses Typs kümmert,
  und macht dort weiter.

  Dabei wird **{ }** für **{ }** und Funktionsaufruf für Funktionsaufruf
  "*von innen nach außen*" gesucht (und dabei die betroffenen Funktionen *verlassen*),
  bis man den Fehlerbehandlungscode gefunden hat,
  der *am nächsten* bei der Fehlerstelle liegt.
- Danach macht das Programm *nach dem Fehlerbehandlungscode* normal weiter
  (*nicht* nach der Stelle, an der der Fehler aufgetreten ist!).

Das Werfen von Exceptions

... geschieht mit **throw** *wert*`;` , wobei *wert* der zu werfende Fehlerwert ist.

Beispiele (**throw** steht praktisch immer in einem **if**):

```
throw "Too much data";
throw SyntaxError(errNr, fileName, lineNr, colNr);
```

Was wirft man normalerweise?

Grundsätzlich kann man *Beliebiges* werfen: **int**'s (z.B. eine Fehlernummer)
oder den fehlerhaften Wert, der das Problem verursacht hat,
oder einen String (z.B. die Fehlermeldung) oder Objekte beliebiger Klassen.

Typischerweise definiert man sich für Exceptions *eigene Klassen*, deren Member-Variablen *Informationen zum Fehler* speichern (wie im Beispiel die Klasse **SyntaxError**,
deren Objekte die Position des Fehlers im File speichern) oder die im einfachsten Fall ganz leer sind (d.h. es kommt *nur* darauf an, *zu welcher Klasse* das Fehlerobjekt gehört).

Man sollte solche Klassen für Exceptions von der *Standard-Exception-Klasse* **exception** *ableiten* (muß aber nicht), die eigens für diesen Zweck eingeführt wurde
und von der auch alle *vordefinierten* Exceptions abgeleitet sind.

Alle von **exception** abgeleiteten Klassen erben eine Methode
`const char *what(void) const;` , die so zu überschreiben ist,
dass sie einen String liefert, der *den aufgetretenen Fehler beschreibt*.

Achtung: Man schreibt üblicherweise *kein* **new** im **throw**, sondern nur einen Konstruktor:

- *Ohne* **new** wird ein *temporäres Objekt* erzeugt, das Objekt selbst geworfen,
  und nach dem **catch** *automatisch* wieder gelöscht ***==> Gut!***
- *Mit* **new** wird ein *dynamisch angelegtes Objekt* erzeugt und ein *Pointer* darauf geworfen (man muss daher im **catch** einen Pointer fangen!).
  Das Objekt wird *nicht* automatisch gelöscht, man muss es
  im oder nach dem **catch** *selbst freigeben* ***==> Unnötig kompliziert!***

Achtung: *Keinen Pointer* auf ein bestehendes *lokales Objekt* oder Array werfen! (der Pointer kommt zwar im **catch** an, aber das Objekt, auf das er zeigt, gibt es dann nicht mehr…)

Das Fangen von Exceptions

Code, der Exceptions werfen könnte, muss man in ein **try { … }** packen,
und unmittelbar nach dem **try { … }** folgen dann **catch (***typ param***) {** *code* **}**
für die einzelnen Arten von Fehlerwerten (nach jedem **try** muss mindestens ein **catch** stehen, es können beliebig viele sein).

(*typ param***)** ist mit einer Parameter-Deklaration vergleichbar:
typ ist der Typ, der gefangen werden soll, d.h. der entscheidet, ob dieses eine **catch** für den geworfenen Wert *"zuständig"* ist, und *param* ist ein *Parameter* des **catch**-Codestücks, der *mit dem gefangenen Fehlerwert initialisiert* wird.

Beispiel:

```
try {
     ReadFile(myFile);
}
catch (const char *msg) {
     cerr << "Internal error: " << msg << endl;
     exit(EXIT_FAILURE);
}
catch (const SyntaxError &err) {
     cout << err.what() << endl;
     ++errCnt;
}
```

Was passiert:

- Der Code im **try { … }** (inklusive aller aufgerufenen Funktionen) wird ganz normal ausgeführt und dabei *auf Exceptions geprüft*.
- Tritt dabei *keine Exception* auf (weder im Code im **try** selbst noch in irgendeiner davon aufgerufenen Funktion oder Methode), werden danach alle **catch** übersprungen, und das Programm macht *nach* dem **try** / **catch** normal weiter.
- Tritt irgendwo im **try** eine *Exception* auf, wird die Ausführung unterbrochen, der Stack aufgeräumt (siehe unten), *das erste vom Typ her passende* **catch** gesucht (der sogenannte *"Exception Handler"*) und dessen Code ausgeführt.

  Reihenfolge beim Suchen des **catch**: *"Von innen nach außen"*

  - Die das **throw** umgebenden **try** werden *von innen nach außen* geprüft (in umgekehrter Schachtelungs-Reihenfolge innerhalb einer Funktion, in umgekehrter Aufrufsreihenfolge von der gerade laufenden Funktion zurück bis zum **main**).
  - Bei jedem einzelnen **try** werden die **catch** (falls es mehrere gibt) *von oben nach unten* geprüft.

  Danach macht das Programm ebenfalls *nach diesem* **try** / **catch** normal weiter (*nicht* nach dem **throw** !). Es wird immer nur *ein* **catch** ausgeführt.

- Passt *kein* **catch** zur Exception, fliegt die Exception weiter, ohne dass ein **catch** ausgeführt wird, und es wird *im nächsten umgebenden* **try** / **catch** nach einem passenden **catch** gesucht.

Für Objekte macht man im Normalfall ein **catch** auf eine **const**-*Referenz* der entsprechenden Klasse (das ist effizienter und sicherer als ein "by Value"-Parameter).

Achtung:

Der Typ im **catch** muss *genau* passen, inklusive eventueller **const**:
Wird eine String-Konstante (**"..."**) geworfen, so wird sie nur von einem **catch(const char *msg)** gefangen, bei einem **catch(char *msg)** fliegt sie vorbei!

Einzige Ausnahme von "genau passen":
Ein **catch**-Parameter einer *Basisklasse* (bzw. ein Parameter, der als *Referenz auf die Basisklasse* deklariert ist) fängt auch Objekte davon *abgeleiteter* Klassen,
ebenso fängt ein Pointer auf eine Basisklasse Pointer auf Objekte abgeleiteter Klassen.

Das `catch(...)`

Das **catch(...)** (wörtlich "**...**") fängt *alle Exceptions*, egal welchen Typs.
Es muss daher *als Letztes* in der **catch**-Liste stehen,
da die **catch** von oben nach unten nach einem passenden durchsucht werden.

Das *Fehlerobjekt* selbst steht in diesem Fall *nicht* zur Verfügung, da sein Typ ja nicht bekannt ist und bei den **...** kein Parameter-Name stehen kann.

Weiterwerfen von Exceptions

Innerhalb des Codes eines **catch** darf ein **throw;** (*ohne* zu werfendem Wert oder Objekt) stehen. In diesem Fall wird *die aktuelle Exception weitergeworfen* und – hoffentlich! – von einem weiter außen liegenden **catch** *nochmals gefangen*.

Das ist sinnvoll, wenn man z.B. relativ nahe am **throw** ein **catch** macht, um lokale Aufräumarbeiten zu erledigen, und dann den Fehler an das globale **catch** im **main** weiterwirft, um die allgemeine Fehlermeldung und -behandlung auszulösen.

Was passiert mit ungefangenen Exceptions?

Das Programm bricht ab. Ob dabei eine sinnvolle Meldung ausgegeben wird und welche, ist implementierungsabhängig, ebenso, ob ein Core Dump erzeugt wird oder nicht.
Die Funktion, die dabei aufgerufen wird (**void terminate()**),
kann man umdefinieren.

Dasselbe passiert,

- wenn in einem globalen / statischen Konstruktor oder Destruktor (vor / nach **main**) eine Exception ausgelöst wird,
- wenn in der Exception-Behandlung eine weitere Exception auftritt (siehe unten),
- oder wenn eine Funktion eine Exception entgegen ihrer Spezifikation wirft (siehe unten).

Das Aufräumen bei Exceptions

Wird eine Exception geworfen, werden *auf dem Weg nach außen alle Blöcke und Funktionen sauber verlassen*: Für alle lokalen Objekte, deren Gültigkeitsbereich verlassen wird (d.h. alle Objekte, die innerhalb des **`try`** lokal angelegt wurden und noch leben), wird ihr *Destruktor* aufgerufen und anschließend das Objekt freigegeben.

Man nennt dieses Aufräumen des Stacks *"Stack Unwinding"*.

Das Stack Unwinding tritt allerdings naturgemäß nur bei Objekten in Aktion:

Pointer (und vor allem die Arrays bzw. Objekte, auf die sie zeigen!) werden davon nicht erfasst.

Diese Arrays und Objekte bleiben daher übrig, obwohl der auf sie zeigende Pointer freigegeben wird. Wird das Programm nach einer Exception nicht beendet, sondern werden Exceptions wiederholt im normalen Programmablauf eingesetzt, so ergibt sich daraus im Laufe der Zeit ein potentiell beträchtliches Memory Leak.

C++ bietet hier keine Unterstützung: Der Programmierer muss selbst dafür sorgen, dass bei Exceptions keine Pointer verloren gehen, ohne dass deren Daten freigegeben werden (indem z.B. alle derartigen Pointer außerhalb des **`try`** / **`catch`** deklariert und im **`catch`** händisch aufgeräumt werden).

Das ist auch der Grund, warum in allen größeren Klassenbibliotheken *Autopointer-Klassen* als "Verpackung" für normale Pointer eingeführt wurden (siehe eigenes Kapitel):

- Diese "Pseudo-Pointer" sind *Objekte* und haben einen *Destruktor*.
- Dieser Destruktor wird auch aufgerufen, wenn der "Pseudo-Pointer" durch eine Exception (oder auch durch ein normales **`return`**) freigegeben wird.
- Im Destruktor kann dann das Array oder das Objekt, auf das der Pointer zeigt, freigegeben werden (typischerweise setzen diese Autopointer-Klassen intern Reference Counting ein, um festzustellen, ob das der letzte Pointer auf einen dynamisch angelegten Speicherbereich ist).

Achtung:

Passiert *im Stack Unwinding* einer Exception beim Ausführen eines Destruktors eine *zweite Exception*, hilft kein **`catch`** mehr: Das Programm wird abgebrochen.

Die Deklaration von Exceptions im Funktionskopf

Achtung: *Dieses Feature wurde in C++11 und C++17 grundlegend geändert!*

Man kann im Prototyp einer Funktion bzw. Methode optional *angeben*, dass *sicher keine Exceptions* aus dieser Funktion heraus geflogen kommen. Das geschieht mit dem Wort **`noexcept`** hinter der Parameter-Liste.

Wird in der Funktion trotzdem etwas geworfen und nicht wieder gefangen, so wird das Programm *abgebrochen*.

Sinn dieses Features ist, dass der Compiler einen Großteil des mit Exceptions verbundenen *Overheads weglassen* kann.

Vordefinierte Exceptions

Es gibt einige im C++-Standard *vordefinierte Exceptions*. Dabei werden *Objekte* von vordefinierten Klassen geworfen, die alle von der Klasse **`exception`** abgeleitet sind.

- Ein **`new`**, das *keinen Speicher* mehr findet, wirft eine **`bad alloc`**-Exception (das ist abschaltbar).
- Die C++-Standardklassen können bei *ungültigen Aufrufen* (z.B. Index-Überschreitungen, unzulässigen Parametern, Über- oder Unterlauf usw.) Exceptions werfen.
- Die Typumwandlung **`dynamic_cast`** und das RTTI **`typeid`** (Laufzeit-Typsystem) können bei *Typfehlern* Exceptions werfen (behandeln wir beides nicht).
- Beim C++ *File-I/O* kann man wählen, ob bei Fehlern Exceptions ausgelöst oder nur Fehlerwerte gesetzt werden sollen (standardmäßig werden *keine* Exceptions ausgelöst).

10. Kapitel: Templates

Von den beiden in Kapitel 3 im Punkt "Code-Wiederverwertung" genannten Mechanismen haben wir bisher erst einen kennengelernt: Die **Vererbung**, bei der derselbe Code *für mehrere "verwandte" Klassen bzw. Datentypen* verwendet wird. Sie ist ein Kernprinzip fast aller objektorientierter Sprachen.

Die zweite Grundidee, **"parametrierter Code"**, wird in den verschiedenen objektorientierten Sprachen sehr unterschiedlich (zum Teil auch gar nicht) realisiert. In C++ geschieht dies durch Templates.

Templates erlauben es, Funktionen und Klassen zu definieren, die *Typen als Parameter* haben. Templates kommen dort zum Einsatz, wo *derselbe Code für verschiedene Datentypen verwendet* werden soll, die **nicht** voneinander abgeleitet sind:

- Ein wesentlicher Anwendungsfall sind allgemeine *Hilfsfunktionen*, die *für viele Typen verwendbar* sind, z.B. eine Funktion **`max`**, die den größeren der beiden Parameter zurückliefert (egal, ob die Parameter jetzt **`int`**-Werte, **`double`**-Werte, **`string`**-Objekte oder andere Objekte mit selbstdefiniertem **`operator<`** sind), oder eine Funktion **`swap`**, die den Inhalt der beiden als Referenz übergebenen Parameter vertauscht (für jeden Typ, für den ein **`operator=`** existiert).
- Der zweite wesentliche Nutzen von Templates sind *Datenstruktur-Klassen* (Queues, Vektoren, Listen, assoziative Arrays, ...), bei denen der *Element-Typ* frei wählbar sein soll: Dieselbe Listen-Klasse soll Listen von **`int`**-Werten, von Pointern, von **`string`**-Objekten und sogar wieder von Listen ermöglichen.

Template-Deklarationen

Template-Deklarationen beginnen mit **`template < ... >`** gefolgt von genau *einer* Klassen-Deklaration oder Funktions-Deklaration.

Die Typ-Parameter in den **`< ... >`** können entweder mit dem Schlüsselwort **`class`** oder mit dem Schlüsselwort **`typename`** deklariert werden, die Wirkung ist dieselbe (auch bei **`class`** können auch Typen eingesetzt werden, die keine Klassen sind). Es können beliebige Namen verwendet werden, üblich sind Großbuchstaben **`T`**, **`T1`**, ...

Innerhalb der darauffolgenden Klassen- oder Funktions-Deklaration können die *Typ-Parameter wie ganz normale Typen* (**`typedef`**-Namen) verwendet werden, um Parameter- oder Returntypen, lokale Variablen, Member usw. zu deklarieren.

Beispiele:

```
template <typename T>
void swap(T &x, T &y)
{
  T tmp;
  tmp = x; x = y; y = tmp;
}

template <typename T>
T max(T a, T b)
{
  return (a > b) ? a : b;
}
```

```
const int MAX = 100;   // das machen wir gleich besser!
template <class T>
class Stack {
  public:
    Stack() : m_elems (0) {}
    ~Stack() {}

    void push(const T &val) { … }
    T pop() { … }

    int size() const { … }
    bool full() const { … }
    bool empty() const { … }

  private:
    T m_data[MAX];
    int m_elems;
};
```

(Anmerkung: Die STL enthält bereits ähnliche Funktionen **swap** und **max** und auch eine **stack**-Klasse.)

Templates sind nicht auf einen Typ-Parameter beschränkt: Ein besonders schönes und trotzdem einfaches Beispiel eines Templates mit *zwei* Typ-Parametern ist die universelle **convert**-Funktion auf https://www.kompf.de/cplus/artikel/stream4.html .

Man kann für die Template-Parameter auch Default-Typen angeben, z.B.

```
template<class T = int>
class Stack { … };
```

Es ist sogar möglich, dass ein Template-Parameter wieder ein Template ist, aber das behandeln wir nicht weiter. Auch variabel viele Template-Parameter sind ab C++-11 möglich, wir behandeln sie ebenfalls nicht.

Wenn man Methoden außerhalb des **class** definiert, muss man ebenfalls ein Template verwenden, ebenso bei der Definition statischer Member. Da ein **template < … >** nur für jeweils eine Deklaration oder Definition gilt, braucht man *ein* **template < … >** *pro Methode*. Damit der Compiler weiß, dass die Methode zu unserer Template-Klasse gehört, muss man die Klasse *als Template* vor den **::** schreiben:

```
template <class T>
T Stack<T>::pop() { … }
```

Numerische Template-Parameter

Unser Stack-Template hat noch eine Schwäche: Die fixe, für alle Stacks gleiche Größe. Das lässt sich vermeiden, indem man auch die Größe als Template-Parameter übergibt: Templates können nicht nur Typen, sondern auch *Werte als Parameter* haben, diese werden so wie die Typen bei der Verwendung des Templates überall durch den angegebenen Wert ersetzt (textuell).
Wie bei Funktions-Parametern und **class**-Template-Parametern ist es auch für einen numerischen Parameter möglich, im **template < … >** einen Default-Wert anzugeben.

```
template <class T, int MAX = 100>
class Stack {
  …
    T m_data[MAX];  // MAX ist wie eine Konstante!
};
```

Numerische Template-Parameter müssen mit *konstanten* Werten instanziert werden, nicht mit zur Laufzeit berechneten Ausdrücken. Sie müssen einen *integralen* Typ haben, **`double`** ist nicht möglich, auch keine Objekte oder Strings.

Globale *Pointer- oder Referenz-Konstanten* (z.B. *Pointer auf Funktionen* oder globale Arrays) als Template-Parameter sind hingegen möglich: Einige in der STL vordefinierte Templates bekommen z.B. eine Vergleichsfunktion als Parameter übergeben.

Innerhalb des Templates ist ein numerischer Template-Parameter eine *Konstante*: Man kann ihn beispielsweise als Größe in einer Array-Deklaration verwenden, aber man darf ihn nicht ändern.

"Aufruf" von Templates

Konzept

Wird ein Template verwendet, spricht man von *Instanzierung* des Templates, so wie man beim Anlegen von Objekten einer Klasse von Instanzierung der Klasse spricht.

Der Vergleich mit den Objekten beschreibt den Sachverhalt ziemlich gut: Pro Verwendung des Templates mit *verschiedenen Typen* als Parametern macht der Compiler (zumindest konzeptionell) nämlich *eine neue Kopie der gesamten Template-Definition* (d.h. des Funktionscodes bzw. der Klassendefinition incl. aller Member und Methoden), *setzt die Argument-Typen ein*, und *compiliert den so entstandenen Code* (in den Anfängen von C++ geschah das tatsächlich textuell mittels Präprozessor, heute verwenden die Compiler dazu Kopien der internen Repräsentation des eingelesenen Templates, nicht mehr eine textuelle Kopie).

Das hat auch eine wichtige Konsequenz für Fehler: Wenn ein Template gelesen wird, wird nur die syntaktische Korrektheit (Klammernschachtelung, Strichpunkte, …) geprüft, aber noch keine inhaltliche Prüfung und keine Code-Erzeugung durchgeführt.

Erst wenn das Template *verwendet* wird, wird eine Typprüfung gemacht, nach passenden Funktionen gesucht, usw.. Diesbezügliche Fehler (auch wenn sie offensichtlich schon im Template selbst liegen) werden daher immer für die Zeile der Template-Verwendung ausgegeben: Erst dann bemängelt der Compiler beispielsweise, dass wir unsere **`max`**-Funktion für einen Typ verwenden, der gar keinen **`operator<`** bereitstellt.

Da für Template-Funktionen und -Methoden erst bei deren Verwendung Code entsteht, gehören Template-Definitionen (soweit sie in mehreren Files verwendet werden sollen) normalerweise *in den Header-File* und nicht in den **`.cpp`**-File.

Manche Compiler generieren in diesem Fall dummerweise mehrfach Code: Was man dann tun muss, um ein in einem oder mehreren **`.cpp`**-Files verwendetes Template doch in einem separaten **`.cpp`**-File (und nicht Header-File) zu definieren, findet man am Internet (besser einen guten Compiler verwenden!).

Verwendung im Code

Template-Funktionen kann man in fast allen Fällen wie ganz normale Funktionen aufrufen:

```
int a, b, m;
...
m = max(a, b);
```

Der Compiler sucht in diesem Fall ein Template, das zu den Argument-Typen passen könnte (der Return-Typ bleibt wie beim Overloading bei der Suche unberücksichtigt), und erzeugt davon automatisch eine Instanz mit dem "richtigen" Typ-Parameter. Allerdings müssen die Argument-Typen bei einem Template *genau* passen: Im Unterschied zu normalen Funktionen macht der Compiler für Templates keinen zweiten Versuch mit impliziten Typkonvertierungen der Argumente, wenn im ersten Versuch keine passende Funktion gefunden wird.

Wenn die implizite Instanzierung auf Grund der Argument-Typen nicht klappt (z.B. weil der Template-Parameter-Typ nicht ausdrücklich in den Funktions-Parametern vorkommt), kann man die Template-Funktion auch explizit instanzieren:

```
double x, top;
...
top = max<double>(x, 100);  // (double, int) ==> implizites Template passt nicht
```

In diesem Fall versucht der Compiler auch, die Argumente mit impliziten Konvertierungen an das instanzierte Template anzupassen.

Wenn man Objekte von *Template-Klassen* anlegen will (bzw. generell überall, wo man eine Template-Klasse verwendet), muss man die Template-Argumente *immer* in **<** ... **>** hinter dem Klassennamen angeben, denn implizite Template-Klassen-Instanzen gibt es nicht:

```
Stack<string, 30> callStack;
```

Es ist in vielen Fällen klug, sich pro Template-Instanz ein **typedef** zu machen: Es macht den Code leserlicher und leichter wartbar:

```
typedef Stack<string, 30> stringStack;
...
stringStack callStack;
```

Achtung: Auch *innerhalb* der **template**-**class**-Deklaration wird die eigene Klasse als *Template-Instanz mit dem Template-Parameter* angeschrieben, nur der Konstruktor- und Destruktor-Name bleibt ganz normal *ohne* **<** ... **>** ! Würde man unseren Stack um einen Copy-Konstruktor und einen Zuweisungs-Operator erweitern, sähe das im **class** wie folgt aus:

```
  Stack(const Stack<T, MAX> &orig);
  Stack<T, MAX> &operator=(const Stack<T, MAX> &val);
```

(C++ ergänzt im Template die Parameter des eigenen Templates automatisch, wenn man nur den eigenen Klassennamen angibt. Ob man sie trotzdem explizit hinschreibt oder nicht, ist eine Frage des Stils bzw. der Programmierrichtlinien.)

Templates und `friend`-Deklarationen

Will man in einer Template-Klasse eine **`friend`**-Funktion deklarieren, die ebenfalls ein Template ist (was sie fast immer sein wird), wird es kompliziert. Normale Funktionen und Template-Funktionen sind für den Compiler betreffend **`friend`** nämlich zwei verschiedene Dinge: Schreibt man in einer Template-Klasse eine "normale" **`friend`**-Deklaration, deklariert diese eine *nicht*-Template-Funktion, und die anschließende Template-Funktion wird nicht als **`friend`** erkannt.

Damit das funktioniert, sind folgende Schritte in dieser Reihenfolge nötig (die genauen Regeln sind noch um Einiges komplizierter):

1. Vorab-Deklaration der Template-Klasse (damit sie im folgenden Funktions-Prototyp als Parameter verwendet werden kann):

   ```
   template <class T>
   class myClass;
   ```

2. Prototyp für die Template-Funktion (damit sie im folgenden **`friend`** bekannt ist):

   ```
   template <class T>
   ostream &operator<<(ostream &outFile, const myClass<T> &a);
   ```

3. Deklaration der Template-Klasse:

   ```
   template <class T>
   class myClass { …
   ```

4. **`friend`**-Deklaration der Template-Funktion innerhalb der Klasse (Achtung auf das **`<T>`**, es macht den wesentlichen Unterschied!):

   ```
   friend ostream &operator<< <T>(ostream &outFile,
                                  const myClass<T> &a);
   ```

5. Definition der Template-Funktion nach der Klasse:

   ```
   template <class T>
   ostream &operator<<(ostream &outFile, const myClass<T> &a) { …
   ```

Template-Spezialisierungen

Es ist möglich, zu einem bereits deklarierten Template weitere Templates zu deklarieren, die *für bestimmte Spezialfälle von Parametern* gelten: Wird das Template verwendet, schaut der Compiler zuerst, ob eines der Spezial-Templates auf die Typen in der Template-Instanzierung passt. Wenn ja, nimmt er den Code aus dem Spezial-Template, wenn nein, verwendet er das allgemeine Template.

Beispielsweise könnten wir für unsere globale **`max`**-Funktion eine Spezial-Variante für C-Strings (die ja nicht mit **`<`** verglichen werden) definieren:

```
template <>
char *max<char *>(char *a, char *b)
{
  return (strcmp(a, b) > 0) ? a : b;
}
```

Wenn es sich nicht um eine globale Template-Funktion handelt, sondern um eine Methode einer Template-Klasse, so muss vor dem Methoden wie üblich der Klassenname mit **::** stehen, und zwar mit dem Typ, für den wir spezialisieren wollen, in **< >**,
also beispielsweise für die Spezialisierung des Standard-Konstruktors von **myClass<T>**:

```
template<>
myClass<char *>::myClass() ...
```

Mit diesem Mechanismus befassen wir uns nicht weiter.

Template-Klassen und Vererbung

Wenn eine Klasse *von einer Template-Klasse abgeleitet* wird, dann ist sie selbst wieder eine *Template-Klasse*. Im einfachsten Fall haben beide Klassen *denselben Typ-Parameter* **T**.

Ausnahme: Die abgeleitete Klasse ist von einer Template-Klasse mit *fix eingesetztem Typ* abgeleitet, beispielsweise **myParentClass<string>**.
In diesem Fall ist die abgeleitete Klasse *nicht zwingend ein Template*.

Das funktioniert *weitgehend intuitiv* und wie erwartet, wenn man Folgendes **beachtet**:

- Im Unterschied zur *eigenen Klasse*, die man *innerhalb* des eigenen **class** fast überall auch *ohne* **<T>** schreiben kann, muss man das **<T>** *bei der Vaterklasse immer dazuschreiben*, beispielsweise
  - Bei der *Angabe der Vaterklasse* oben in der **class**-*Deklaration*:

    ```
    template <typename T>
    class myClass : public myParentClass<T>
    { ...
    ```

  - Beim *Aufruf des Vaterklassen-Konstruktors* in der *Initialisierungs-Liste*:

    ```
    myClass(T &tVal) : myParentClass<T>(tVal), ... { ... }
    ```

  - Beim *expliziten Aufruf* einer *Methode der Vaterklasse*:

    ```
    int myMeth(int i) {
        int ret = myParentClass<T>::myMeth(i);
        ...
    ```

- Auf Member und Methoden, die *von nicht-Template-Klassen geerbt* wurden, kann man bekanntlich *"einfach so"* (nur mit dem Namen) *zugreifen*.

  Beim *Zugriff auf Member und Methoden*, die *von Template-Klassen geerbt* wurden, muss vereinfacht gesagt irgendetwas *direkt beim Namen* stehen,
  das *ein Wert oder Objekt* mit **T** im *Typ* oder eine *Template-Klasse* mit **T** ist,
  denn sonst *findet* der Compiler das geerbte Member / die geerbte Methode *nicht*:
  - Beispielsweise ein *Objekt der eigenen Klasse* (oder ein *Pointer* darauf) oder irgendein *Wert*, dessen Typ **T** ist oder ein **T** *enthält*, sei es mit **.** oder **->** *vor* dem Member oder der Methode oder als *Argumentwert* im Methoden-Aufruf.
  - Oder die *Vaterklasse* mit **<T>** und mit **::** .

  Am häufigsten tritt das Problem beim *Zugriff auf Member des eigenen Objektes* auf, bzw. beim *Aufruf von geerbten Methoden für das eigene Objekt*.
  In diesem Fall ist **this->...** *verpflichtend* und löst das Problem!

Template-Klassen mit statischen Membern

Das ist nicht ganz so intuitiv. Wir erinnern uns: Konzeptionell wird von Template-Code *je eine Kopie pro verschiedenem für* **T** *eingesetzten Typ* erstellt, und jede dieser Kopien wird dann *unabhängig voneinander kompiliert*.

Daraus folgt auch, dass es *pro verschiedenem für* **T** *eingesetzten Typ eine separate Instanz* der statischen Member des Templates gibt:

- Bildet man von einem Template *ein oder mehrere Instanzen*, die beispielsweise *alle* **int** für **T** einsetzen, so gibt es das statische Member so wie bisher *genau ein Mal* für *alle* Objekte *aller* dieser Instanzen *gemeinsam*.
- Bildet man vom selben Template hingegen *eine Instanz* mit **int** für **T** und eine *weitere Instanz* mit **string** für **T**, so gibt es *zwei voneinander unabhängige Kopien* des statischen Members: Eine für alle Objekte des **int**-Templates gemeinsam und eine für alle Objekte des **string**-Templates gemeinsam.

Daraus folgt:

- Auch das *Anlegen des Speichers* für das statische Member außerhalb des **class** ist ein *Template*, und die Klasse muss *mit* **<T>** angegeben werden:

  ```
  template <typename T>
  int myTemplClass<T>::count = 0;
  ```

- Beim *Zugriff* auf statische Member mittels **::** muss die Template-Klasse *mit* **< >** angegeben werden, ebenso wie beim *Aufruf* statischer Methoden:
  - Mit **<T>**, wenn dieser Zugriff *innerhalb von Template-Code* steht.
  - Beispielsweise mit **<string>**, wenn der Zugriff *in normalem Code* steht, z.B. im **main** :

    ```
    cout << myTemplClass<string>::count << endl;
    cout << myTemplClass<string>::getCount() << endl;
    ```

11. Kapitel: Die STL (Standard Template Library)

Die **STL** ist die *im C++-Standard* definierte Library. Sie ist um ein Vielfaches umfangreicher und mächtiger als die C-Standard-Library und auch komplexer zu verstehen, da sie größtenteils *Templates anstatt einfacher Funktionen und Klassen* definiert.

Zusätzlich zur STL hat sich die noch viel umfangreichere **Boost** als *quasi-Standard* etabliert; mit der Aufnahme einiger Boost-Klassen in den C++-Standard ist zu rechnen.

Weiters sind 18 der **Standard-C-Header** auch als Bestandteil der STL standardisiert.

Ein Arbeiten mit der STL ist nur mittels Online-Doku sinnvoll (siehe z.B. http://www.cplusplus.com/reference/), deshalb wird hier nur ein ganz kurzer und unvollständiger Überblick gegeben:

1.) Die STL enthält eigene, von C **`stdio`** unabhängige Klassen für die **File-Ein- und -Ausgabe** (Header **`iostream`**, **`fstream`** usw.). Sie unterscheiden sich von C neben der Verwendung (**`<<`** und **`>>`**) vor allem in der Fehlerbehandlung (kein **`errno`** !). Wir haben davon bisher nur **`cin`** und **`cout`** verwendet.

   Neben den File-Klassen zu erwähnen sind die **String-Streams** **`<sstream>`**, die *formatiertes Einlesen aus Strings und Ausgeben in Strings* erlauben (analog zu **`sscanf`** und **`sprintf`**), sowie der Header **`<iomanip>`** für die Formatierung.

   `errno`-ähnliches *Fehlercode-Handling* findet man im Header **`<system_error>`** (**`errc`** ist z.B. eine Enumeration für die vordefinierten Fehlercodes).

2.) C++ enthält eine eigene Klasse **`string`** für **Strings**.

   `string`'s sind *weniger effizient*, aber viel *komfortabler* und *sicherer* als C-Strings, weil sie das benötigte Array intern in der richtigen Länge dynamisch verwalten (d.h. bei Bedarf *wachsen*) und die üblichen *Operatoren* **`= + == <`** usw. bieten.

   Die Klasse **`string`** ist auch ein *Container* und bietet *Iteratoren*, womit ein Großteil der Features in Punkt 3. und 4. auch für **`string`**-Objekte verfügbar ist.

3.) Der größte und wichtigste Teil der STL sind die **Datenstrukturen ("Container")**:

   - Arrays *fixer* Größe (**`array`**), dynamisch große, "*wachsende*" Arrays (**`vector`**), an *beiden* Enden erweiterbare Pseudo-Arrays (**`deque`**), sowie *mathematische* Arrays (**`valarray`**)
   - Einfach (**`forward_list`**) und doppelt *verkettete Listen* (**`list`**)
   - Assoziative sortierte *Mengen von Schlüssel-Wert-Paaren* (**`map`** und **`multimap`**), einfache sortierte Mengen (**`set`** und **`multiset`**), dasselbe auch unsortiert (**`unordered_map`**, **`unordered_multimap`**, **`unordered_set`**, **`unordered_multiset`**)
   - *Queues* (**`queue`**) und *Stacks* (**`stack`**), Priority Queues (**`priority_queue`**).
   - *Bit-Arrays* (**`bitset`** und **`vector<bool>`**).

   Eigenschaften und Anwendungen dieser Container sind weiter unten beschrieben.

   Zu den Containern sind *Iteratoren* definiert, siehe unten und in der Doku zum Header **`<iterator>`**.

Weiters ist es möglich, Container durch eine Aufzählung von Werten in **{ }** zu initialisieren, z.B. **vector<int> primes = {3,5,7,11,13,17,19,23};**

Der Header **<initializer_list>** deklariert die Klasse **initializer_list**, und die Initialisierungs-**{ }** generieren implizit ein Objekt dieser Klasse.
Gibt man einer Klasse einen *Konstruktor* mit **initializer_list**-*Parameter*, lässt sich dieser Initialisierungs-Mechanismus *auch für selbstgeschriebene Klassen* verwenden.

Zwei weitere vordefinierte Datenstrukturen sind

- *Paare* von *zwei Werten* gleichen oder verschiedenen Typs (**pair**, im Header **<utility>** versteckt).
- *Tupel* von *mehreren Werten* gleichen oder verschiedenen Typs (**<tuple>**).

4.) Auf diesen Containern sind in **<algorithm>** einige **Algorithmen** definiert: Zahlreiche *Such-Funktionen* (sequentiell und binär), *Vergleich* zweier Container, *Anwenden einer Funktion* auf alle Werte, Extrahieren, Ersetzen oder Füllen von Teilbereichen, Löschen doppelter Elemente, Mengen-Operationen (Vereinigung, Schnittmenge, ...), Umdrehen, Umordnen und Permutieren, Aufteilen nach einer Bedingung und vor allem *Sortieren und sortiert Vereinigen* (siehe unten).

Der **<numeric>**-Header enthält *mathematische* Hilfsfunktionen für Container.

Ein eigener Header **<functional>** befasst sich mit **Funktions-Objekten**.
Diese sind an sich universell auch in eigenem Code verwendbar, kommen aber in der Praxis vor allem bei den Container-Funktionen in **<algorithm>** vor.

Weiters definiert **<algorithm>** die **Hilfsfunktionen min** und **max** nicht nur für Container, sondern auch *für zwei normale Werte*.

Aus **<utility>** kommt eine universelle Funktion **swap** zum *Vertauschen*.
swap kann nicht nur *zwei normale Werte vertauschen*,
sondern auch den *Inhalt zweier Container austauschen*.
(vor C++11 war **swap** in **<algorithm>**).

Neben **pair** und **swap** enthält der Header **<utility>**

- im Namespace **rel_ops** die Vergleichsoperatoren **!= <= > >=** definiert basierend auf **<** und **==**
  (damit man für eigene Klassen nur **<** und **==** selbst schreiben muss und die anderen vier Operatoren automatisch daraus erzeugt werden)
- sowie **move** und **forward** für den Umgang mit *rvalues* (siehe Kapitel "Neuerungen C++11").

5.) Für mathematische und physikalische Anwendungen gibt es
- einen **complex**-Datentyp (**komplexe Zahlen**)
- und eine **valarray**-Klasse für **mathematische Vektoren** (siehe unten).

Der Header **<random>** enthält **Zufalls-Generatoren**:
Diese können nicht nur **int**'s wie in C, sondern auch **double**'s liefern, und zwar nicht nur gleichverteilt in einem gewissen Bereich, sondern je nach Wunsch verschiedenen *mathematischen Verteilungen folgend* (normalverteilt, Chi-Quadrat-verteilt, exponentialverteilt usw.).
Außerdem ist die eigentliche *Zufallsgenerator-Engine wählbar*,

es stehen auch viel bessere Engines als in C zur Verfügung (u.a. Mersenne-Twister).

6.) Weiters definiert die STL Klassen für einige **Standard-Exceptions** (siehe u.a. **<exception>** und **<stdexcept>**).

7.) Seit C++11 gehören einige Header für **Multithreading**, Synchronisation, parallelitätsfesten Speicherzugriff usw. zur STL und damit zum C++-Standard: **<thread>**, **<mutex>**, **<condition_variable>**, **<future>** und **<atomic>**

8.) Der Header **<regex>** bietet Funktionen und Klassen für das Suchen nach **Regular Expressions** (d.h. String-Mustern) bzw. *Pattern Matching*.

9.) Außerdem bietet die STL **Smart-Pointer-Klassen** (siehe eigenes Kapitel). Daneben enthält der dafür zuständige Header **<memory>** weitere Definitionen für die Speicherverwaltung und den direkten Zugriff auf Speicherbereiche.

Iteratoren

Iteratoren sind das wichtigste allgemeine Konzept der Container und Algorithmen in der STL: Iteratoren leisten auf beliebigen Datenstrukturen das, was man in C auf Arrays üblicherweise mit *Pointern* macht: Sie dienen dazu, die **Elemente in Containern zu durchwandern** (z.B. in einer **for**-Schleife) bzw. den **Anfang und das Ende** eines Containers (oder eines Teilbereiches in einem Container) zu repräsentieren oder z.B. als Such-Ergebnis auf das gefundene Element zu "zeigen":

- Für jede Container-Klasse erhält man mit **::iterator** den *Typ des dazugehörigen Iterators*. Beispiel:

  ```
  vector<float>::iterator pos;
  ```

  Lies: "**pos** ist eine Variable vom Typ **iterator** für die Klasse **vector<float>**", d.h. der Scope-Operator wird hier nicht auf einen Variablen- oder Methoden-Namen, sondern auf einen Typ-Namen angewendet, der zu einer Klasse gehört.

  Analog gibt es einen **const_iterator** für nur *lesenden* Zugriff.

- Für jedes Container-Objekt **c** liefert **c.begin()** einen Iterator, der auf *das erste Element* zeigt, und **c.end()** einen Iterator, der ***hinter** das letzte Element* zeigt (und daher *nicht* mehr dereferenzierbar ist).
  Weiters liefern alle *Such-Funktionen* der STL einen Iterator, der auf das gefundene Objekt zeigt (bzw. liefern **end()** für den "nicht-gefunden"-Fall).

- Wenn **pos** ein Iterator ist, erhält man mit ***pos** das Element, auf das **pos** gerade zeigt. Ist das ein Objekt oder zusammengesetztes Element, kann man mit **pos->**... auf seine Komponenten und Methoden zugreifen.

- Mit **++pos** lässt man den Iterator auf das *nächste Element im Container* zeigen, mit **--pos** auf das *vorige* (falls der Container rückwärtslaufende Iteratoren kennt). **+** und **-** funktionieren auch, wenn der Container das unterstützt, ebenso **pos[n]** für ***(pos + n)** .

- Mit **==** und **!=** kann man prüfen, ob zwei Iteratoren *auf dasselbe Element zeigen*, einige Container bieten auch **<** und **>** für "weiter vorne" und "weiter hinten". Weiters gibt es eine Funktion **distance** oder schlicht **-** für den *Abstand* (in Elementen) zwischen zwei Iteratoren.

Im Header **<iterator>** sind auch allgemeine Methoden zur *"Pointerarithmetik"* mit Iteratoren definiert, die z.B. *"Abstand zweier Iteratoren"* oder *"Schiebe den Iterator um n Elemente vor"* mit Schleifen (und daher langsam!) realisieren, wenn die Iteratoren direktes Rechnen nicht unterstützen.

- Analog zum normalen **swap** *vertauscht* **iter_swap** aus **<algorithm>** die beiden Werte, auf die die beiden angegebenen Iteratoren zeigen.

Beispiel für das *Durchlaufen einer Liste* mit Aufruf der Methode **notenMittel** für jedes Element, und für das *Suchen in einer Liste*:

```
class Stud;
list<Stud> sem2;
…
for (list<Stud>::iterator pos = sem2.begin();
     pos != sem2.end();
     ++pos) {
  cout << pos->notenMittel();
}
…
Stud current;
…
current = …;
…
list<Stud>::iterator pos = find(sem2.begin(), sem2.end(), current);
if (pos != sem2.end()) { … }
```

Es gibt *verschiedene Arten* von Iteratoren (je nachdem,
welche Operationen auf der zu Grunde liegenden Datenstruktur möglich sind):
- Nur *schrittweise vorwärts* (**++**)
- *Schrittweise vorwärts und zurück* (**++** und **--**)
- *In beliebigen Schritten* vor und zurück (**+** und **-**, Vergleich mit **<** und **>**, **[]**)

Bei Containern der zweiten und dritten Art gibt es auch *"rückwärts-laufende"* Iteratoren **reverse_iterator**, die man mit **rbegin()** und **rend()** bekommt:
Bei diesen geht **++** auf das *vorige* Element.

Weiters gibt es Iterator-Klassen (die *Input- und Output-Iteratoren*),
mit denen nicht Datenstrukturen, sondern *Files durchlaufen* werden.

typename

Bei Deklarationen mit komplexen Typen (*Typen mit Scope-Operatoren*)
innerhalb eines Templates (und nur dort!) ist der Compiler nicht mehr im Stande,
von selbst zu erkennen, ob es sich jetzt um *ein Member oder um einen Typ*
des Templates vor dem **::** handeln soll. In diesem Fall ist es notwendig,
das Schlüsselwort **typename** vor das Konstrukt zu setzen.

Besonders häufig trifft das Deklarationen von Iteratoren:

- **Falsch**
  (der Compiler sucht in **vector<T>** nach einem *Member* namens **iterator**) :
  ```
  template <class T>
    …
    vector<T>::iterator i;
  ```
- **Richtig:**

```
template <class T>
  …
  typename vector<T>::iterator i;
```

auto statt Iterator-Typdeklarationen

Seit C++ 11 werden Iterator-Variablen meist direkt in der ersten Zuweisung mit Typ **auto** deklariert anstatt mit explizitem …**::iterator** .
Der Compiler gibt der Variablen bei **auto** den Typ, den der Wert rechts vom **=** hat:

```
for (auto pos = sem2.begin(); pos != sem2.end(); ++pos) { …
auto pos = find(sem2.begin(), sem2.end(), current);
```

"for :"-Schleifen für Container

Seit C++ 11 gibt es eine eigene Variante von **for**-Schleifen, um *Container*
(und auch normale Arrays sowie explizite Aufzählungen in **{ }**) *zu durchwandern*:

for (*variable* **:** *container* **) {** … **}**

Beispiele:

```
for (auto &stud : sem2) { … stud … }
for (int i : { 2, 3, 5, 7, 11, 13, 17, 19 }) { … i … }
```

Diese Schleife belegt die Variable **stud** *der Reihe nach* mit den Werten
aus dem Container **sem2** bzw. **i** der Reihe nach mit den Werten in den **{ }**
und macht für jeden Wert einen Schleifendurchlauf.

stud ist also *kein Iterator* und kein Pointer, sondern *der Wert des Elementes selbst*!

auto als Typ heißt hier *"gib der Variable automatisch den Typ, den sie für die Elemente des Containers rechts haben muss"*. Dabei sind zwei Varianten möglich:

- Bei **auto stud** ist **stud** eine *eigenständige Variable* vom Typ der Container-Elemente, und vor jedem Durchlauf wird das nächste Element des Containers *in diese Variable kopiert*. Ändert man die Variable in der Schleife, wirkt sich das *nicht* auf das betreffende Element im Container aus: Man ändert ja nur die Kopie.
- Bei **auto &stud** ist **stud** eine *Referenz*, die der Reihe nach *auf die Original-Werte im Container* verweist. Änderungen von **stud** *ändern* in diesem Fall das aktuelle Element des Containers, da **stud** direkt das jeweilige Element des Containers repräsentiert (und keine Kopie ist).
- Aus Effizienz- und Lesbarkeits-Gründen sollte man **const auto &stud** verwenden, wenn man den Wert in der Schleife *nicht ändern* will:
  const verhindert versehentliche Änderungen,
  und **&** erspart das Kopieren der Werte.

Die vordefinierten Funktionen auf Containern

Die STL enthält (vor allem in **<algorithm>**) viele *Funktions-Templates*, die ganz allgemein *auf Containern arbeiten* und das Ziel haben, viele ***immer wiederkehrende Schleifenkonstrukte durch kürzeren, eleganteren und schnelleren Code zu ersetzen.***

Damit diese Funktionen möglichst universell verwendbar sind,
bekommen sie typischerweise

- den oder die ***Container*** (oder Teilbereiche davon), die sie bearbeiten sollen, *als Iteratoren übergeben*,
- und die ***Funktion oder Bedingung***, die sie bei der Verarbeitung der Werte verwenden sollen, als *Funktions-Parameter*.

Iterator-Parameter

Die meisten dieser Funktionen haben ***zwei Iteratoren** als Parameter*: **`first`** und **`last`**.
Die Funktionen verarbeiten der Reihe nach die Elemente des Containers
zwischen **`first`** (einschließlich) und **`last`** (*ausschließlich!*).

- Will man eine solche Funktion daher auf *alle Elemente eines Containers* anwenden, übergibt man **`c.begin()`** (oder **`c.cbegin()`**, wenn die Elemente nur gelesen werden) als **`first`** und **`c.end()`** oder (**`c.cend()`**) als **`last`**.

  Je nach Container-Typ kann man einen *Rückwärts-Iterator* (**`rbegin`** und **`rend`**) übergeben, damit die Funktion die Elemente rückwärts durchläuft.
- Will man die Funktion auf ein *normales Array* anwenden, übergibt man statt Iteratoren *normale Pointer* in das Array, für eine Anwendung *auf das ganze Array* **`arr`** bzw. **`&(arr[0])`** als **`first`** und **`arr+size`** bzw. **`&(arr[size])`** als **`last`**.

Selbstverständlich lassen sich die Funktionen aber *nicht nur für ganze Container* aufrufen,
sondern *auch für beliebige Teilbereiche* eines Containers.

Funktions- und Prädikats-Parameter

Ein Großteil der im nächsten Teilkapitel aufgelisteten Funktionen aus **`<algorithm>`**
akzeptiert ***eine Funktion als Parameter***, teils *verpflichtend*, in vielen Fällen *optional*.

Teilweise kann das eine *beliebige unäre oder binäre Funktion* sein (z.B. bei **`transform`**),
teilweise ist es sinnvollerweise eine Funktion mit Returntyp **`void`** (z.B. bei **`for_each`**),
manchmal auch eine *Rechenoperation* (z.B. bei **`accumulate`**),
aber in den meisten Fällen ist es ein sogenanntes ***"Prädikat"***, also eine Funktion,
die ***Returntyp*** **`bool`** haben muss, d.h. **`true`** oder **`false`** als Ergebnis liefert.

Normalerweise ist mit *"Prädikat"* ein *"unäres Prädikat"* gemeint,
d.h. ein Prädikat, das *einen* Parameter hat, d.h. *einen einzelnen Wert prüft*.

In einigen Fällen werden in **`<algorithm>`** aber auch *"binäre Prädikate"* verwendet,
d.h. Prädikatsfunktionen mit *zwei Parametern*, typischerweise *Vergleichsfunktionen*.
Insbesondere kann man bei fast allen Funktionen aus **`<algorithm>`**,
die implizit Werte mit **`==`** oder **`<`** *vergleichen*, *optional* einen Funktionsparameter angeben,
um den normalen Vergleich durch eine *eigene Vergleichsfunktion* zu ersetzen.

Ein Blick auf die Prototypen der vordefinierten Funktionen zeigt aber,
dass hier *keine Function Pointer wie in C* z.B. bei **`qsort`** übergeben werden.
Die Frage ist daher einerseits, was da technisch abläuft,
und andererseits, was man als Programmierer für so einen Parameter übergeben kann.

Zumindest bei den vordefinierten Funktionen aus **`<algorithm>`** usw., die ja alle
Templates sind, ist der *Typ des Funktionsparameters* ein *Typ-Parameter des Templates*,

d.h. beim Aufruf eines solchen Funktionstemplates passt sich der Template-Parameter automatisch an die Funktion an, die man im Aufruf als Argument übergibt.

Eine zentrale Rolle spielt in diesem Zusammenhang der Header **<functional>**, der zahlreiche *Klassen und Funktionen im Zusammenhang mit Funktionsobjekten* enthält und in C++11 sehr stark erweitert wurde.

Der Programmierer hat bei der Übergabe von Funktionen *größtmögliche Freiheit*, solange die Argumenttypen und der Returntyp passen:

- Man kann wie in C *Funktionsnamen* direkt als Argument übergeben, sowohl selbstgeschriebener als auch vordefinierter Funktionen.
- Man kann *Variablen* von einem passenden *"Function Pointer"-Typ* übergeben.
- Man kann *Methoden-Pointer* übergeben (haben wir nicht gelernt). Dann wird diese Methode des entsprechenden Objektes aufgerufen.
- Man kann *Member-Pointer* übergeben (haben wir auch nicht gelernt). Dann wird der *Wert* dieses Members *als Returnwert* des Aufrufs verwendet.
- Man kann *Funktionsobjekte* übergeben, also Objekte einer Klasse, die einen **operator()** definiert und deren Objekte man daher *wie eine Funktion aufrufen* kann.

  Das können *selbstdefinierte* Funktionsobjekte sein oder Objekte der am Ende dieses Kapitels erwähnten *vordefinierten Funktionsobjekt-Klassen*. Typischerweise erzeugt man von diesen Klassen *direkt im Aufruf temporäre Objekte*, z.B. mit **greater_equal<string>()** ein Objekt, das zwei **string**-Objekte mit **>=** vergleicht.
- Man kann *Lambda-Ausdrücke* (siehe Kapitel "Neue C++11 Features") übergeben, d.h. den *Code*, der verwendet werden soll, direkt als Argument *in den Aufruf* schreiben.

  ```
  if (all_of(data.begin(), data.end(),
             [](int i){ return (i % 2 == 0); })) { …
  ```

  prüft beispielsweise, ob alle Elemente von **data** gerade Zahlen sind.

  Für *kurzen selbstgeschriebenen Funktionscode* ist das der in der Praxis häufigste und praktischste Fall.
- Man kann sogenannte **bind**-*Ausdrücke* angeben: Diese nehmen eine beliebige *Funktion mit zu vielen Parametern*, belegen einen oder mehrere dieser Parameter *mit fixen Werten*, und erzeugen dadurch eine *Funktion mit entsprechend weniger Parametern*.

  ```
  double my_add(double x, double y) { return x + y; }
  …
  transform(data.begin(), data.end(), result,
            bind(my_add, _1, 10.0));
  ```

  addiert zu jeden Wert in **data** die Zahl **10** und speichert die Ergebnisse in **result** (**_1** heißt "dieser Parameter bleibt als erster Parameter erhalten").

  Durch Verwendung der vordefinierten **plus**-Funktionsobjektklasse kann man sich die Hilfsfunktion **my_add** auch sparen:
  … **bind(plus<double>(), _1, 10.0)** …

- Man kann *zusammengesetzte (geschachtelte) Funktionsausdrücke* angeben.
  Dabei wird *zuerst die innere Funktion* aufgerufen,
  und dann *auf deren Returnwert die äußere Funktion*.

  Die beiden wichtigsten *vordefinierten* Funktionstemplates,
  die eine andere Funktionen als Parameter haben,
  sind **not1** (negiert ein *unäres* Prädikat) und **not2** (negiert ein *binäres* Prädikat).

  IsOdd sei eine Klasse mit einem **bool operator()(int x)**,
  der prüft, ob **x** *ungerade* ist. **IsOdd()** erzeugt ein Funktionsobjekt dieser Klasse.

  ```
  int anz = count_if(data.begin(), data.end(), not1(IsOdd()));
  ```

  berechnet daher, wie viele Elemente von **data** *gerade* (nicht ungerade) sind.

Intern sind die Funktions-Parameter in den **<algorithm>**-Funktionen
(zumindest gedanklich) vom Typ her *immer Funktionsobjekte*, also Objekte einer Klasse,
die einen **operator()** mit entsprechenden Parametertypen definiert.
Übergibt man einer **<algorithm>**-Funktion ein Funktionsobjekt, passt daher schon alles.

Übergibt man ihr *etwas anderes* (z.B. einen Function Pointer),
passiert im allgemeinen Fall folgendes:

- **<functional>** enthält eine *Template-Klasse* **function**. Von dieser Klasse
  wird *für jede Kombination von Argument- und Returntypen eine Instanz* erstellt.
- Diese Instanz von **function** definiert einen **operator()**
  mit genau diesen Argument- und Returntypen, Objekte dieser Instanz der Klasse
  function sind daher *aufrufbare Funktionsobjekte vom richtigen Typ*.
- Weiters enthält **function** *zahlreiche Konstruktoren*,
  die beispielsweise u.a. *einen Function Pointer als Parameter* haben und diesen
  Function Pointer *in ein Funktionsobjekt* dieser **function**-Instanz ***"verpacken"***,
  sodass beim Aufruf dieses Funktionsobjektes der darin enthaltene Function Pointer
  aufgerufen wird.
- Wenn daher von einer **<algorithm>**-Funktion ein Funktionsobjekt erwartet wird,
  aber im Aufruf der Funktion ein *Function Pointer als Argument* angegeben wurde,
  dann verwendet der Compiler intern *implizit diesen Konstruktor*,
  um aus dem Function Pointer ein **function**-Objekt zu *erzeugen*,
  und ruft die **<algorithm>**-Funktion mit dem so erzeugten **function**-Objekt auf.

In der Praxis werden viele Fälle auch direkt durch *Template-Spezialisierungen* abgedeckt,
also wie am Ende des Template-Kapitel beschrieben durch *Spezialcode für bestimmte Sonderfälle von Template-Parametern*. Beispielsweise könnte ein Spezialfall für ein Funktions-Template aus **<algorithm>** existieren, das für den Fall, dass ein Function Pointer übergeben wird, Code mit direktem Aufruf dieses Function Pointers enthält, anstatt den Pointer in ein Funktionsobjekt zu konvertieren und dieses aufzurufen (was langsamer wäre).

Neben **function**, **bind**, **not1**, **not2** und vielem anderen mehr stellt **<functional>**
auch *vordefinierte Funktionsobjekt-Klassen für fast alle Operatoren* von C++ bereit:
plus für die Addition, **equal_to**, **greater**, **less** usw. für Vergleiche,
logical_not für das "nicht", **negate** für das Vorzeichen-Minus usw..

`plus<double> my_plus`;

legt daher ein neues *Funktionsobjekt* **`my_plus`** an,
das die Additionsfunktion für zwei **`double`**-Parameter realisiert.

`transform(data.begin(), data.end(), result, negate<double>());`

speichert die Werte von **`data`** mit umgekehrtem Vorzeichen in **`result`**.

Die Funktionen in `<algorithm>` *(nur auszugsweise!)*

Durchlaufen, Vergleichen, Suchen

- **`for_each`**

  Ruft eine *Funktion* der Reihe nach für jedes Element im Container-Bereich aus.

- **`transform`**

  Die erste Variante durchläuft alle Elemente eines Containers,
  ruft für jedes Element eine gegebene *unäre Funktion* auf,
  und sammelt deren *Returnwerte in einem neuen Container*.

  Die zweite Variante durchläuft alle Elemente von *zwei* gegebenen Containern im Gleichschritt, ruft für die jeweils zwei korrespondierenden Elemente eine *binäre Funktion* auf, und sammelt deren *Returnwerte in einem neuen Container*.

- **`equal`**

  Prüft, ob zwei Container-Bereiche *dieselben Werte* enthalten
  (bzw. ob ein binäres Prädikat für alle korrespondierenden Elemente **`true`** liefert).

- **`lexicographical_compare`**

  Vergleicht zwei Container-Bereiche *lexikografisch* und liefert **`true`**,
  wenn der erste kleiner als der zweite ist (optional mit Vergleichs-Prädikat).

- **`mismatch`**

  Vergleicht zwei Container-Bereiche und liefert zwei Iteratoren,
  die auf die *ersten unterschiedlichen Elemente* zeigen
  (bzw. auf die Elemente, für die ein binäres Prädikat das erste Mal **`false`** liefert).

- **`is_permutation`**

  Prüft, ob zwei Container-Bereiche *dieselben Werte in beliebiger Umordnung*
  enthalten (optional wieder mit binärem Prädikat statt **==**).

- **`all_of`**, **`any_of`**, **`none_of`**

  Liefert ein Prädikat für *alle* / zumindest *ein* / *kein Element* im Container-Betreich **`true`**?

- **`find`**

  Findet einen *gegebenen Wert* im Container-Bereich.

- **`search_n`**

  Findet *n aufeinanderfolgende Vorkommen* eines gegebenen Wertes
  im Container-Bereich.

- **`find_if`**, **`find_if_not`**

Findet das erste Element im Container-Bereich, für das ein Prädikat **`true`** bzw. **`false`** liefert.

- **`find_first_of`**

  Findet das erste Vorkommen irgendeines Elementes des zweiten Container-Bereiches im ersten Container-Bereich.

- **`search`**, **`find_end`**

  Findet das erste / das letzte komplette Vorkommen eines Container-Bereiches in einem anderen Container-Bereich.

- **`adjacent_find`**

  Findet die ersten benachbarten gleichen Elemente
  (bzw. die ersten benachbarten Elemente, für die ein binäres Prädikat **`true`** liefert).

- **`count`**

  Zählt, wie oft ein *gegebener Wert* in einem Container-Bereich vorkommt.

- **`count_if`**

  Zählt, wie of ein *Prädikat* für die Elemente eines Container-Bereiches **`true`** liefert.

- **`min_element`**, **`max_element`**, **`minmax_element`**

  Liefert das *kleinste bzw. größte Element* im Container-Bereich
  (optional mit Vergleichs-Prädikat).

Container umordnen, Inhalt ändern

Generell gilt: Gibt es eine Version *ohne* ...**`_copy`** und eine *mit* ...**`_copy`**,
so *modifiziert* die erste den *bestehenden* Container,
und die zweite liefert ihr Ergebnis in einem *neuen* Container.

- **`swap_ranges`**

  Vertauscht die Inhalte von *zwei* Container-Bereichen
  (hier werden wirklich die Werte der *einzelnen Elemente der Reihe nach elementweise* vertauscht, im Unterschied zu **`swap`** auf zwei ganzen Container-Objekten,
  das im Normalfall *das gesamte Innenleben* der Container vertauscht).

- **`copy`**, **`copy_n`**, **`copy_backward`**

  Kopiert die Elemente in einem Container-Bereich.
  `copy_backward` arbeitet von hinten nach vorne, für den Fall,
  dass sich Quelle und Ziel so überlappen, dass **`copy`** Werte verlieren würde.

- **`move`**, **`move_backward`**

  Dasselbe mit C++11 **`move`**-Operationen statt **`copy`**-Operationen für jedes Element.

- **`copy_if`**

  Kopiert nur diejenigen Elemente eines Container-Bereiches,
  für die ein Prädikat **`true`** liefert.

- **`fill`**, **`fill_n`**

  Füllt einen Container-Bereich mit einem gegebenen Wert.

- **`generate`**, **`generate_n`**

Füllt einen Container-Bereich, indem es für jedes Element eine *Funktion* aufruft.

- **`replace`**, **`replace_copy`**

  Ersetzt alle Vorkommen eines gegebenen Wertes durch einen anderen Wert.

- **`replace_if`**, **`replace_if_copy`**

  Ersetzt alle Elemente, für die ein Prädikat **`true`** liefert, durch einen neuen Wert.

- **`remove`**, **`remove_copy`**

  Entfernt alle Vorkommen eines gegebenen Wertes.

- **`remove_if`**, **`remove_if_copy`**

  Entfernt alle Elemente, für die ein Prädikat **`true`** liefert.

- **`unique`**, **`unique_copy`**

  Entfernt aufeinanderfolgende gleiche Werte.

- **`reverse`**, **`reverse_copy`**

  Dreht die Reihenfolge der Werte im Container-Bereich um.

- **`rotate`**, **`rotate_copy`**

  Rotiert die Elemente in einem Container-Bereich.

- **`random_shuffle`**, **`shuffle`**

  Ordnet die Elemente in einem Container-Bereich *zufällig* um.

Sortieren, Operationen auf sortierten Containern

- **`sort`**, **`stable_sort`**

  Sortiert einen Container-Bereich.

  `partial_sort`, **`partial_sort_copy`**

  Ermittelt und sortiert die *n kleinsten Elemente* eines Container-Bereiches.

  `is_sorted`, **`is_sorted_until`**

  Prüft, ob ein Container-Bereich sortiert ist,
  bzw. liefert das erste nicht richtig sortierte Element.

- **`merge`**, **`merge_inplace`**

  Vereinigung zweier sortierter(!) Container-Bereiche zu einem sortierten Ergebnis (wie beim Merge-Sort).

- **`includes`**, **`set_union`**, **`set_intersection`**, **`set_difference`**, **`set_symmetric_difference`**

  Mengen-Operationen auf sortierten(!) Container-Bereichen:
  “Ist Teilmenge von”, Vereinigung, Durchschnitt, “In A aber nicht in B”,
  “In einem aber nicht in beiden”

- **`binary_search`**, **`equal_range`**

  Sucht einen gegebenen Wert in einem sortierten(!) Container-Bereich
  mittels binärer Suche.

`lower_bound`, **`upper_bound`**

Sucht das erste Element *größergleich* bzw. größer einem gegebenen Wert in einem sortierten(!) Container-Bereich mittels binärer Suche.

Andere speziell angeordnete Container

- **`partition`**, **`stable_partition`**, **`partition_copy`**

  Partitioniert einen Container-Bereich *nach einem Prädikat*, d.h. teilt die Elemente in einen Bereich mit allen Elementen, für die das Prädikat **`true`** liefert, und einen Bereich mit allen Elementen, für die es **`false`** liefert.

  `is_partitioned`, **`partition_point`**

  Prüft, ob ein Container-Bereich schon partitioniert ist, und ermittelt die *Grenze* zwischen den beiden Teilbereichen.

- **`nth_element`**

  Partitioniert einen Container-Bereich *nach der Größe (Sortierreihenfolge)* der Elemente, sodass der *Teilungspunkt beim n-ten Element* liegt.

- **`push_heap`**, **`pop_heap`**, **`make_heap`**, **`sort_heap`**, **`is_heap`**, …

  Funktionen, die einen *Heap*, d.h. eine *Priority Queue*, in einem Container verwalten, bzw. aus einer Priority Queue eine sortierte Reihenfolge erstellen (Heapsort).

- **`next_permutation`**, **`prev_permutation`**

  Liefert die *lexikografisch nächste / vorige Permutation* eines Container-Bereiches.

Die Funktionen in <numeric> *(nur auszugsweise!)*

- **`accumulate`**

  Summiert die Elemente innerhalb eines Container-Bereichs auf.
  Der *Initialwert*, zu dem die Elemente dazugezählt werden, ist *Parameter*.
  Als *optionaler Parameter* kann eine *binäre Funktion* angegeben werden, die statt **+** benutzt wird (z.B. Zusammenmultiplizieren, Quadrate summieren, …).

- **`inner_product`**

  Liefert das *Skalarprodukt* zweier Container **`x`** und **`y`** (d.h. die Summe der Produkte **`x[i]*y[i]`**), wieder mit der Möglichkeit, optional andere Operatoren als **+** und ***** anzugeben.

- **`iota`**

  Füllt einen Container-Bereich mit *aufsteigenden Werten*, z.B. **1**, **2**, **3**, **4**, **5**, …

12. Kapitel: Die vordefinierten Container der STL

`array`

Ein ***ganz normales Array fixer, konstanter Größe***
mit einigen *typischen Container-Methoden* (**`size`** usw.) und ***mit Iteratoren***.

Es hat *dasselbe Zugriffs-Verhalten* und ist *genauso effizient* wie normale Arrays:
Der *Zugriff per* **`[]`** *auf das i-te Element* ist *sehr schnell* und größen-unabhängig (O(1)),
Elemente nachträglich einfügen/löschen/anhängen ist *nicht* möglich, und das Suchen eines Elementes nach seinem Wert ist nur mit Hilfsfunktion oder Schleife möglich und sehr ineffizient (O(n)).

`array` wird überall dort verwendet, wo man eine Datenstruktur mit dem Verhalten, der Effizienz und dem geringen Overhead von *normalen Arrays* haben will, aber *Iteratoren braucht* (z.B. weil man vordefinierte Funktionen darauf aufrufen will, die einen beliebigen Container mit Iteratoren als Parameter erwarten).

Bei Bedarf liefert **`array`** auch einen *direkten Pointer* auf das interne Array.

`vector`

Nach außen verhält sich **`vector`** wie ein ***Array, das bei Bedarf automatisch wächst***
(d.h. man muss vorab *keine Größe angeben*, und wenn man ein Element anhängt, dann sorgt **`vector`** intern dafür, dass es auch tatsächlich Platz dafür gibt)
bzw. das man jederzeit und ohne Verlust des Inhalts *vergrößern* kann.

Auch intern enthält **`vector`** ein *einziges, dynamisch angelegtes Array*
(**`vector`** erlaubt auch einen direkten Zugriff auf dieses Array).
Wird es zu klein, wird automatisch *ein größeres Array angelegt und der Inhalt **umkopiert***.
Dies geschieht aber *nicht bei jedem* einzelnen Anhängen eines Elementes,
sondern typischerweise in Zweierpotenzen: Die interne Größe wird *jedesmal verdoppelt*,
sodass nur bei jedem 2^n-ten Anhängen eines Elementes ein Vergrößern und Umkopieren nötig wird.

Daraus folgt, dass ein **`vector`** am hinteren Ende typischerweise *angelegte, aber unbenutzte Elemente* enthält (der englische Fachbegriff für die *intern angelegte Größe* heißt ***"Capacity"***, im Unterschied zu seiner *befüllten Größe*, der ***"Size"***).
Wenn man weiß, dass der **`vector`** nicht mehr wachsen wird (oder wenn man am hinteren Ende viel gelöscht hat), dann kann man mit **`shrink_to_fit`** die Capacity auf die tatsächliche Size reduzieren, d.h. unbenutze Elemente hinten *freigeben*.

Im Unterschied zu einem normalen C-Array speichert ein **`vector`** intern nicht nur seine Capacity, sondern *auch seine Size*, d.h. die Anzahl der tatsächlich befüllten Elemente.
Man kann sie mit **`size`** *abfragen*. Wie bei vielen Containern ist **`[]`** ein *ungeprüfter Zugriff* (schneller, aber ein ungültiges **`i`** wird *nicht* erkannt und liefert *"irgendetwas"*)
und **`at`** dasselbe, aber *geprüft und langsamer* (im Fehlerfall wirft **`at`** eine *Exception*).

Es gilt:

- Der *Zugriff* per **`[]`** auf das i-te Element ist *sehr schnell und größen-unabhängig* (O(1)), ebenso das *Anhängen* neuer Elemente *am hinteren Ende* (**`push_back`**) und das *Löschen* des *letzten Elementes* (**`pop_back`**).

- Die *Suche* nach einem bestimmten Wert ist *sehr ineffizient* (O(n)).
  Das *Einfügen oder Löschen von Elementen mittendrin* oder am Anfang ist ebenfalls *sehr langsam* (O(n)).
- *"Pointerarithmetik"* auf Iteratoren (Iterator + n, Abstand zweier Iteratoren) funktioniert effizient.

`string`

Die **`string`**-Klasse bietet (zusätzlich zu den typischen String-Funktionen) ein Verhalten ähnlich einem **`vector<char>`**-*Container* (Iteratoren usw.).

deque

deque ist eine Datenstruktur ähnlich einem **`vector`** (d.h. gedanklich wie ein ***Array, das bei Bedarf automatisch wächst***), mit dem Unterschied, dass es ***an beiden Enden wachsen und schrumpfen*** kann, d.h. dass man *an beiden Enden* relativ effizient (in konstanter Zeit, O(1)) *neue Elemente anfügen oder Elemente entfernen* kann.

Das Zugriffsverhalten ist ähnlich einem Array oder einem **`vector`**: Der **`[]`**-*Zugriff auf das i-te Element ist schnell* und in konstanter Zeit (größen-unabhängig, O(1)) möglich (aber *etwas langsamer* als bei Array und **`vector`**). Das *Einfügen und Löschen* von Elementen mittendrin ist zwar *möglich*, aber *langsam*: Wie beim Array müssen dabei möglicherweise *alle Elemente davor/dahinter verschoben (umkopiert)* werden (O(n)).

Die interne Darstellung besteht typischerweise aus *ein oder mehreren dynamisch angelegten Arrays*, ist also etwas *komplexer* als bei **`vector`**.

`valarray`

`valarray` ist ein *"mathematisches" Array*, (d.h. die Elemente sind typischerweise **`int`**, **`double`**, **`complex`** o.ä.), aber nur *teilweise ein Container*:

`valarray` bietet zwar globale Funktionen **`begin`** und **`end`**, die Iteratoren liefern, aber keine Methoden **`begin`** und **`end`** und keinen eigenen Iterator-Typ (deshalb ist **`valarray`** normalerweise nicht bei den Containern dokumentiert).

Es wächst auch *nicht* automatisch: Die Größe wird beim Konstruktor angegeben, eine nachträgliche Änderung ist nur unter Verlust des Inhalts der Elemente möglich.

Dafür bietet es eine Vielzahl von Operationen:

- ***Arithmetische Operationen für "Vektorrechung":***

  Diese addieren, subtrahieren, … entweder zwei **`valarray`**'s *elementweise*, oder ein **`valarray`** und eine *einzelne Zahl* (die dann *zu allen Elementen* des **`valarray`**'s dazugerechnet wird).

  Mit **`=`** kann auch *ein Wert auf alle Elemente* des **`valarray`**'s *zugewiesen* werden.

  Neben **`+ - * / %`** sind auch die *Bitoperatoren* incl. Shift, die *logischen Operatoren* und die *Vergleichsoperatoren* verfügbar.

  Weiters gibt es ein **`apply`**, um eine *beliebige unäre Funktion* auf alle Elemente eines **`valarray`**'s anzuwenden, und Hilfsfunktionen wie **`min`** und **`max`** oder *Aufsummieren* aller Elemente.

  Der **`valarray`**-Header deklariert auch Versionen der *mathematischen Funktionen*

(**sin**, **log**, **sqrt**, …), die *elementweise* auf ganzen **valarray**'s arbeiten.

- ***Der Operator* `[]` *ist stark erweitert:***
  - Er kann nicht nur einzelne Elemente auswählen, sondern auch *Teilbereiche* eines **valarray**, sogenannte *Slices*. Diese müssen nicht einmal fortlaufend sein (*"wähle jedes 5. Element, beginnend bei 2"*, sogar geschachtelt möglich).
  - Weiters kann ein **bool-valarray** oder ein Index-**valarray** als *Selektor* im `[]` angegeben werden: Die `[]` liefern dann beispielsweise ein Array, das genau diejenigen Elemente enthält, für die der Selektor **true** war.
- Die Elemente eines **valarray** können ***effizient verschoben oder rotiert*** werden.

Eine zentrales Feature von **valarray** ist, dass die Operationen auf *großen* **valarray**'s bei der ***Implementierung auf Supercomputern*** *typischerweise* ***automatisch effizient parallelisiert***, d.h. auf mehrere Prozessoren verteilt, werden.

`list`

list implementiert eine ***doppelt verkettete Liste mit Head- und Tail-Pointer***.
Daraus folgt:

- *Hinten und vorne einfügen oder entfernen* sowie der *Zugriff auf das erste und das letzte Element* funktionieren *sehr schnell* (O(1)).

  Weiters lassen sich Listen sehr effizient *vereinigen* (zusammenhängen), ebenso effizient ist das *Aufteilen* einer Liste in zwei Listen, wenn man die Teilungsstelle schon kennt (beides O(1)).

- Der *Zugriff auf das i-te Element* sowie das *Suchen* eines Elementes mit einem bestimmten Wert sind beide *sehr langsam* (O(n), nur mit *Schleife oder Hilfsfunktion*, kein `[]` und kein **find**) und sollten daher die Ausnahme sein:
  Für einen Zugriff per *Index* sind die *arrayartigen Container* (**array**, **vector**, **deque**) viel besser geeignet, für das *Suchen* nach einem Wert sind die **map**- und **set**-Varianten viel effizienter.

  Der Zugriff auf den *Vorgänger / Nachfolger* eines Elementes ist *schnell* (O(1)), damit ist auch das *Durchlaufen der Liste in beide Richtungen* effizient (aber *etwas langsamer* als bei **vector** usw.).

- Das *Einfügen oder Löschen eines Elementes oder eines Listenstückes* ist *sehr effizient* (O(1)), wenn man *schon an der richtigen Stelle* der Liste ist (d.h. bereits einen *Iterator* auf diese Stelle hat), aber *sehr langsam* (O(n)), wenn man zuvor erst einmal *die richtige Stelle suchen* muss.

  Das *Filtern einer Liste* (Entfernen aller Elemente, die eine bestimmte Bedingung erfüllen), geht im Vergleich zu anderen Containern auch *vergleichsweise schnell*.

- Die Listen in C++ führen Buch über ihre Länge: Die *Anzahl der Elemente* lässt sich daher mit **size** in O(1) ohne Durchzählen der Liste ermitteln.

- Listen lassen sich in O(n) Zeit in der Reihenfolge *umdrehen*.

  Weiters ist für **list** bereits eine **sort**-Methode definiert. Sie läuft in O(n*log(n)), also im Prinzip sehr *effizient*, aber langsamer als gute Sortiermethoden auf Arrays. Sind zwei Listen *bereits sortiert*, lassen sie sich auch effizient (O(n)) zu einer sortierten Liste *vereinigen* (**merge**).

- Der *Speicherbedarf* ist *deutlich höher* als bei **vector**: Für jedes Element wird *ein eigener Block* im Speicher *dynamisch angelegt*, und dieser enthält neben den Nutzdaten noch *zumindest 2 Pointer* (ähnlich wie bei **map** usw.).

forward_list

forward_list ist eine ***einfach verkettete Liste ohne Tail-Pointer***.
Sie führt auch nicht Buch über die aktuelle Anzahl der Elemente.

Sie ist also speichermäßig und performancemäßig *ein bisschen effizienter* als eine normale **list**, aber sie *kann weniger*: Man kann sie *nur vorwärts und nicht rückwärts* durchlaufen, man kann nur auf das erste und *nicht auf das letzte Element* zugreifen, und man kann *nur vorne Elemente effizient anhängen und entfernen* und nicht hinten. Wenn man einen *Iterator* auf ein Element hat, so kann man *nur hinter diesem Element einfügen* (und nicht davor), und man kann nur den Nachfolger des Elementes *löschen* und nicht das Element selbst. Und einen direkten Zugriff auf das Element Nummer i oder das Element mit dem Wert x bietet **forward_list** natürlich auch nicht.

reverse, **sort** und **merge** sowie die *Filter-Methoden* sind aber verfügbar.

Die verfügbaren Operationen sind dafür alle relativ effizient, d.h. O(1), wenn sie nur ein Element betreffen, und sonst in der Regel O(n), wie bei der normalen **list**.

map

Eine **map** ist wörtlich eine *Zuordnung von Nutzdaten zu Schlüsselwerten*:
Unter *Angabe des Schlüsselwertes* wird auf die dazugehörigen Nutzdaten zugegriffen.
Eine **map** speichert daher eine ***beliebig große Anzahl von Paaren***
(d.h. jedes Element einer **map** hat den vordefinierten Typ **pair**),
und jedes Paar besteht aus ***einem Schlüsselwert und den dazugehörigen Nutzdaten***
(wobei Schlüssel und Nutzdaten meist *verschiedenen Typ* haben).

Vergleichbar ist eine **map** mit einer *relationalen Datenbank*:
Jedes Element der **map** entspricht *einer Zeile* in der Datenbank,
und der *Schlüsselwert* entspricht dem *Primary Key*, nach dem gesucht und sortiert wird bzw. mit dem auf einzelne Zeilen zugegriffen wird.
Daraus folgt auch:
- Der Schlüsselwert muss *eindeutig* sein: Mehrere Elemente können die gleichen Nutzdaten haben, aber *nicht den gleichen Schlüsselwert*.
- Der Schlüsselwert eines Elementes kann/darf/soll *nachträglich nicht geändert* werden!

Eine **map** ist daher die *bevorzugte Datenstruktur*, wenn der ***Zugriff auf die Elemente über Ihren Schlüsselwert*** erfolgen soll und *nicht über Ihre Position* (Index bzw. Einfüge-Reihenfolge). Dies wird auch durch die Schreibweise ausgedrückt: **map** bietet einen **[]**-Operator. In den **[]** steht aber ***kein Index, sondern der Schlüsselwert***
(das muss daher kein **int** sein, sondern kann ein **string**, ein **double**,
oder auch ein komplexes Objekt sein, solange dieses Objekt Vergleichsoperatoren bietet, d.h. sortierbar ist). Als Ergebnis liefern die **[]** *nicht* das ganze Paar Schlüssel+Nutzdaten, sondern *nur die zu diesem Schlüssel gehörenden Nutzdaten*, die mit den **[]** sowohl *gelesen* als auch *geändert* werden können. Ruft man **[]** mit einem *nicht existierenden Schlüssel* auf, wird ein *neues Element* mit diesem Schlüssel erzeugt und in die **map** eingefügt. Verwendet man **at** statt **[]** (das für bereits existierende Schlüssel dieselbe Funktion hat), liefert ein nicht existierender Schlüsselwert eine *Exception*. Umgekehrt liefert **insert**

mit einem schon existierenden Schlüssel das schon bestehende Element.

Damit *unterscheidet* sich eine **`map`** in ihrer *Charakteristik und Anwendung* grundsätzlich sowohl von **`array`**, **`vector`**, **`stack`** usw. als auch von **`list`**.

Intern wird eine **`map`** als ***balancierter binärer Suchbaum*** gespeichert, mit dem ***Schlüsselwert als Such- und Sortierkriterium***. Daher ist der *Zugriff mittels Schlüsselwert* und auch das *Einfügen und Löschen* von Elementen in ***O(log(n))*** möglich (nicht nur im Mittel, sondern *auch im Worst Case*!), was *um Vieles besser* ist als das Einfügen und Löschen in einem sortierten Array oder in einer sortierten Liste.

Weiters folgt daraus, dass man zu jedem Element relativ einfach das ***sortierordnungsmäßig nächstkleinere und nächstgrößere Element*** ermitteln kann und alle Elemente effizient *in **sortierter** Reihenfolge vorwärts und rückwärts durchlaufen* kann. Eine **`map`** ist daher nicht nur naheliegend, wenn der Zugriff per Schlüssel effizient sein muss, sondern auch dann, wenn Elemente *in beliebiger Reihenfolge eingefügt* werden, aber dann *in Sortierreihenfolge abgerufen bzw. durchlaufen* werden sollen (die Effizienz von "Einfügen in eine **`map`** und diese durchlaufen" ist zwar in absoluten Zeiten langsamer als das Sortieren eines Arrays, aber in derselben Größenordnung O(n*log(n))).

Ebenfalls möglich ist die *unscharfe Suche* mit einem nicht existierenden Schlüsselwert *k*: *"Gib mir das erste Element größergleich k!"*

`set`

`set` ist eine ***vereinfachte*** **`map`**: **`set`** speichert *keine Paare* aus Schlüsselwert+Nutzdaten, sondern *nur einfache Werte*, d.h. nur *Schlüsselwerte ohne Nutzdaten* (wobei diese Schlüsselwerte natürlich auch beliebig komplexe Objekte sein können).

Ansonsten ist die Funktionalität, Anwendung und Performance ident zu **`map`**.

`unordered_map` und `unordered_set`

Logisch unterscheiden sich diese Container vor allem dadurch von **`map`** und **`set`**, dass sie ***kein sortiertes Durchlaufen der Elemente*** erlauben: Sie unterstützen zwar *Iteratoren*, die man zum "nächsten" Element verschieben kann und mit denen man *alle Elemente komplett durchlaufen kann* (sogar relativ schnell, aber *nur vorwärts*, nicht rückwärts), aber welches das "nächste" Element ist, und *in welcher Reihenfolge* die Elemente durchlaufen werden, ist *undefiniert*: Der Durchlauf erfolgt *in zufälliger Reihenfolge*, nicht in Sortierreihenfolge, und auch nicht in der Reihenfolge des Einfügens.

Abgesehen davon sind die Funktionalität, Einsatzzweck, Vorteile und Nachteile im wesentlichen ident zu **`map`** und **`set`**: Hauptzweck ist das ***rasche Einfügen, Wiederfinden und Löschen von Elementen nach ihrem Schlüsselwert*** (nicht nach ihrer Position). Der Zugriff nach Schlüsselwert erfolgt wie bei **`map`** mit **`[`***key***`]`**. Dafür sind diese Container de facto ***optimal***: *Einfügen, Suchen und Löschen eines Wertes* brauchen im Mittel ***O(1), also konstante Zeit***, unabhängig von der Anzahl der Elemente. Das ist besser als **`map`** und **`set`**, die beide O(log(n)) brauchen.

Wenn es also um ***schnelles Suchen nach Werten (bzw. Schlüsseln und den Nutzdaten dazu), Einfügen und Löschen*** geht, und wenn man ***auf das sortierte Durchlaufen verzichten*** kann, dann sind die **`unordered`**-Varianten von **`map`** oder **`set`** zu verwenden!

Intern verwenden die **`unordered`**-Varianten keinen Binärbaum, sondern eine ***Hashtable***.

Daraus folgt auch, dass die ***Worst-Case-Laufzeit*** von Suchen, Einfügen und Löschen ***O(n)*** beträgt. Dieser Worst Case tritt aber nur ein, wenn die Hashfunktion die Schlüsselwerte *nicht gut verteilt* (was aber *extrem selten* ist!), oder wenn die Hashtable durch ein neues Element so *voll* geworden ist, dass sie *vergrößert und komplett umorganisiert* werden muss.

Um dieses aufwändige Umorganisieren zu *vermeiden*, kann man *vor dem Befüllen* des Containers mit **reserve** die *ungefähre Anzahl der erwarteten Elemente* setzen, damit die Hashtable gleicht in der richtigen Größe angelegt wird.

multimap, multiset, unordered_multimap und unordered_multiset

Diese Datenstrukturen sind bezüglich Funktion, Innenleben und Effizienz weitgehend ident zu den nicht-**multi**-Datenstrukturen, aber sie erlauben es, ***mehrere Elemente mit demselben Schlüsselwert*** einzufügen (was bei den *normalen Varianten* einen *Fehler* liefert). Diese werden auch *getrennt gespeichert*, und es gibt eine Methode **equal_range**, die *alle* zu einem Schlüsselwert passenden Elemente zurückliefert, sowie eine Methode **count**, die die *Anzahl* der Elemente mit einem bestimmten Schlüsselwert liefert (beide sind typischerweise auch relativ effizient, d.h. es werden nicht alle Elemente in der Datenstruktur durchsucht, sondern nur der betroffene Bereich).

Einen **[]**-Operator gibt es *nicht* (weil das Ergebnis *nicht eindeutig* wäre).
find liefert einen Iterator auf das *erste* Element mit dem gesuchten Schlüssel.
Die *weiteren Elemente* mit demselben Schlüssel folgen unmittelbar danach, wenn man diesen Iterator inkrementiert.

queue und stack

Eine **queue** (***Warteschlange***, FIFO, d.h. *"first in first out"*) ist eine lineare Datenstruktur (wie ein Array oder eine Liste), bei der Elemente ***nur am hinteren Ende angefügt*** und ***nur am vorderen Ende entnommen*** werden (wie bei einer Warteschlange an der Supermarkt-Kasse).

Ein **stack** (***Stapel***, LIFO, d.h. *"last in first out"*) ist eine lineare Datenstruktur, bei der Elemente nur ***am selben (vorderen) Ende angefügt und wieder entnommen*** werden (wie ein Bücherstapel, auf den man nur oben drauflegt und nur oben wegnimmt).

Die erlaubten Operationen (vorderstes/oberstes Element auslesen, am Ende anfügen, am Ende entnehmen), funktionieren alle in *konstanter Zeit* (O(1)), unabhängig von der Anzahl der Elemente. Wenn man daher *nur das oben genannte Verhalten* braucht, sollte man **queue** bzw. **stack** verwenden.

queue und **stack** haben *bewusst minimalistische Interfaces*: Sie bieten *keine Iteratoren* und *keinen Zugriff auf die "inneren" Elemente*. Wenn man das braucht, muss man die Queue oder den Stack entweder *selbst implementieren* (was z.B. basierend auf **deque** ganz einfach ist), oder eine eigene Klasse von **queue** oder **stack** *ableiten und entsprechend erweitern* (in der abgeleiteten Klasse hat man Zugriff auf den internen Container und seine Iteratoren, näheres findet man am Internet).

Queues und Stacks kann man unter Verwendung von Arrays oder verketteten Listen realisieren. Die STL erlaubt beides: Defaultmäßig verwenden **queue** und **stack** intern eine **deque**, wahlweise können beide auch auf eine **list** aufgesetzt werden, **stack** akzeptiert auch einen **vector** als interne Datenstruktur.
Die Default-Implementierung mittels **deque** *vereint die Vorteile* der üblichen händischen

Implementierungen mittels Array oder mittels Liste:
Einerseits ist sie *zeit- und speicherplatz-effizienter als eine listenbasierte Implementierung*,
und andererseits muss man *vorab keine maximale Größe* festlegen,
weil die Datenstruktur automatisch je *nach Bedarf beliebig vergrößert* wird.

`priority_queue`

Eine **`priority_queue`** ist eine spezielle Datenstruktur (in Algorithmenkunde-Büchern oft auch ***"Heap"*** genannt), die für einen ***ganz bestimmten Zweck*** optimiert ist:
Sie enthält eine beliebige Anzahl von Elementen, die *jederzeit und in beliebiger Reihenfolge hinzugefügt* werden können, aber erlaubt nur das *Lesen und das Entfernen des momentan sortierordnungsmäßig größten Elementes*: Alle anderen Elemente sind unsichtbar,
bis sie durch Entfernen aller größeren Elemente selbst zum größten Element werden.

Der *Zugriff* auf das aktuell größte Element ist *"sofort"* möglich (O(1), unabhängig von der Anzahl der Elemente), das *Einfügen* neuer Elemente und das *Entnehmen* des größten Elementens dauert auch jeweils nur O(log(n)). Auch *speichermäßig* ist **`priority_queue`** relativ *effizient*: Die Elemente werden intern lückenlos in einem **`vector`** gespeichert, nicht in einem dynamisch angelegten Objekt pro Element.

Derselbe Effekt wie eine **`priority_queue`** kann durch die Funktionen **`make_heap`**, **`push_heap`**, und **`pop_heap`** aus dem Header **`<algorithm>`** auf array-artigen Containern wie **`vector`** und **`deque`** selbst realisiert werden.

`bitset` und `vector<bool>`

`bitset` entspricht einem ***Array von Bits mit fixer, konstanter Größe***.
Es verhält sich ähnlich wie ein Array von **`bool`**-Werten,
aber *jedes Element belegt nur 1 Bit Speicher* (statt zumindest einem Byte pro Element).

`bitset` ist ***kein Container*** im üblichen Sinne: Es unterstützt *keine Standard-Iteratoren*.
Dafür gibt es die üblichen *Bit-Operatoren* (**`&`** **`|`** **`^`** **`~`** Shift-**`<<`** Shift-**`>>`**) für **`bitset`**-Werte, und **`bitset`**'s können *ein- und ausgegeben* sowie *in Strings und zurück verwandelt* werden (als Folge von **`'0'`** und **`'1'`**). Außerdem gibt es Tests auf *"alles 0"* und *"alles 1"* sowie Methoden, die das i-te Bit oder alle *Bits setzen, löschen, invertieren oder testen*.

`vector<bool>` verhält sich wie ein normaler **`vector`** (*wächst* also bei Bedarf usw.), aber sollte pro **`bool`**-Element *nur ein Bit und nicht ein Byte* Speicher belegen,
d.h. ist *auf Platz-Effizienz und nicht auf Geschwindigkeits-Effizienz* optimiert.
Er ist also *speichermäßig genauso effizient* wie **`bitset`**, verhält sich aber bezüglich variabler Größe, Iteratoren usw. flexibler und *wie ein normaler Container*.
Dafür *fehlen* ihm die *Bit-Operatoren* usw. von **`bitset`**.

13. Kapitel: Smart Pointer

Das RAII-Prinzip

RAII steht für ***"Ressource Acquisition Is Initialization"***, also "Ressourcenbelegung ist Initialisierung". Dabei handelt es sich um ein *allgemeines Programmier-Prinzip*, das z.B. auch in den Programmiersprachen Ada und Rust intensiv genutzt wird.

RAII bedeutet wörtlich, dass das *Belegen/Öffnen einer Ressource* immer unmittelbar beim *Anlegen eines für diese Ressource zuständigen Verwaltungs-Objektes* passiert, also typischerweise in dessen *Konstruktor*. Schlägt das Belegen/Öffnen der Ressource fehl, muss garantiert sein, dass erstens kein oder ein als "leer" markiertes Verwaltungsobjekt entsteht, und dass zweitens (außer dem leeren Verwaltunsobjekt) keine Ressourcen belegt bleiben.

Ressourcen können dabei unter anderem sein:
- Dynamisch angelegter *Speicher*
- *Files*
- *Netzwerk- oder Datenbank-Verbindungen*
- Parallele *Threads*
- *Locks* für Zugriffe auf gemeinsame Daten

Das Wesentliche an RAII ist aber gar nicht, dass die Ressourcenbelegung gemeinsam mit der Initialisierung des Verwaltungs-Objektes passiert.
Das in der Praxis sehr viel wichtigere Prinzip ist,

dass das Freigeben oder Schließen einer Ressource immer automatisch erfolgt, wenn das dazugehörige Verwaltungsobjekt verschwindet.

D.h. das Belegen/Öffnen und das Freigeben/Schließen einer Ressource wird *nicht mehr explizit durch Funktions- oder Methodenaufruf programmiert* bzw. ausgelöst, sondern geschieht *implizit*, indem man ein entsprechendes *Verwaltungsobjekt dafür erzeugt*, und indem dieses Verwaltungsobjekt irgendwann *wieder vernichtet* wird.

Aus technischer Sicht bedeutet das, dass diese Verwaltungsobjekte tatsächlich ***echte "Objekte"*** sein müssen (und nicht nur einfache Variablen oder einfache Pointer), und dass diese Objekte einen ***Destuktor*** haben, der *die zum Objekt gehörenden Ressourcen garantiert freigibt* (Speicher *freigeben*, Datei *schließen*, Datenbank-Verbindung *beenden*, ...).
Es kann also *nicht* passieren, dass eine Ressourcenbelegung das dazugehörige Verwaltungsobjekt "überlebt".

Die Ressourcen selbst sind typischerweise keine lokalen Objekte, sondern entweder dynamisch oder global angelegte Ressourcen oder z.B. Ressourcen des Betriebssystems. Daher würden sie typischerweise "ewig" (bis zum Programmende) belegt bleiben, wenn sie nicht explizit freigegeben werden.

Das dazugehörige *Verwaltungsobjekte* sind hingegen in fast allen Fällen *lokale Variablen* irgendeiner Funktion: Sie liegen am Stack und *verschwinden automatisch, sobald der Funktionsaufruf endet* (egal wie). Damit ist aber sichergestellt, dass am Ende der Funktion implizit deren Destruktor ausgeführt wird und damit *alle innerhalb der Funktion belegten Ressourcen automatisch wieder freigegeben werden*.

Und selbst wenn die Verwaltungsobjekte nicht lokal, sondern ebenfalls dynamisch

angelegt sind, passiert das Richtige: Sobald ein dynamisch angelegtes Verwaltungsobjekt freigegeben wird (sei es explizit oder sei es implizit durch das Aufräumen eines anderen Verwaltungsobjektes), läuft dessen Destruktor, und die mit dem Objekt verbundenen Ressourcen werden sauber aufgeräumt.

Was gewinnt man dadurch?

- Man *spart sich viele explizite Aufrufe* eines **`delete`**, eines **`close`** usw..
  Der Programmcode wird dadurch kürzer und übersichtlicher.
- Es besteht *keine Gefahr*, auf das Freigeben einer Ressource zu *vergessen*, also z.B. im Fall von dynamischem Speicher durch fehlende **`delete`**'s *"Memory Leaks"* zu verursachen.
- Auch andere *typische Fehler*, die bei einer expliziten Freigabe oft passieren, werden durch eine implizite, beim Verschwinden des Verwaltungsobjektes automatisch erfolgende Freigabe *vermieden*:
  - *Doppeltes Freigeben* (vor allem bei Speicherfreigaben fatal)
  - *Zugriff nach der Freigabe* (denn sobald das Verwaltungsobjekt weg ist, gibt es keinen Weg mehr, auf die Ressource zuzugreifen).

Absolut essentiell ist die implizite Freigabe durch Verwaltungsobjekte in Kombination mit ***Exceptions***:

- Vor jedem *"geplanten"* Ende einer Funktion (d.h. vor jedem **`return`**) könnte man mit entsprechender Mühe und Sorgfalt alle Ressourcen explizit freigeben bzw. schließen.
- Aber eine *Exception* (irgendwo tief in irgendeiner aufgerufenen Funktion) kann die aktuelle Funktion unvorhersehbar und *an jeder Stelle beenden*. Ein sauberes explizites Aufräumen zu programmieren wäre in diesem Fall *gar nicht möglich*.

  Das *implizite* Freigeben durch Verwaltungsobjekte hingegen *funktioniert*, weil bei Exceptions garantiert ist, dass *für alle lokalen Objekte* aller durch die Exception verlassenen Funktionen *ordnungsgemäß der Destruktor ausgeführt* wird (das ist der Hauptgrund, warum der Exception-Mechanismus relativ "teuer" ist).

RAII und die Speicherverwaltung in C++

Die *dynamische Speicherverwaltung* ist einer der *wichtigsten Anwendungsfälle* von RAII in C++: In einem "modernen" C++-Programm sollten gar *keine* expliziten **`new`** und **`delete`** (oder gar **`malloc`** und **`free`**) mehr vorkommen, sondern nur mehr die unten erwähnten Konstrukte, weil sich die *explizite Speicherverwaltung* als *unbeherrschbare Quelle von zahlreichen und gefährlichen Programmierfehlern* herausgestellt hat:

"never use* `new`*, use* `make_shared` *or* `make_unique`*"

Die erste Frage ist natürlich, warum man in C++ *keine automatische Garbage Collection* wie in Java, C#, Python usw. einführt:

- Der erste Grund ist, dass eine Garbage Collection in einer Sprache, die dem Programmierer so viele Freiheiten und Möglichkeiten wie C/C++ gibt, *gar nicht hundertprozentig möglich* ist.
- Der zweite Grund ist, dass eine Garbage Collection *deutlich ineffizienter*

als eine explizite Speicherverwaltung ist.

- Und der dritte Grund ist, dass in einer Sprache mit Garbage Collection *nicht wirklich vorhersehbar und kontrollierbar* ist, *wann* dynamischer Speicher *freigegeben* wird, und wann daher die *Destruktoren* dynamischer Objekte laufen. Diese Kontrolle möchte man in C++ aber behalten.

  Aus demselben Grund ist eine explizite Speicherverwaltung *unter beschränkten Platzverhältnissen* von Vorteil: Sie gibt nicht mehr benötigte Daten *sofort* frei, während solche Daten in Systemen mit Garbage Collection potentiell noch lange nach ihrem "Tod" Speicher belegen.

Aber auch mit RAII ist eine Speicherverwaltung deutlich komplizierter zu realisieren als z.B. eine Verwaltung offener Files oder anderer Ressourcen:

- Aus unserer Programmiererfahrung wissen wir, dass es *nicht nur einen Pointer* auf einen dynamisch angelegten Speicherblock geben kann, sondern oft *mehrere Pointer auf ein und denselben Block* zeigen (genau daher kommt ja die Gefahr, dass ein Block über einen Pointer schon freigegeben wird, während später über einen anderen Pointer nochmals darauf zugegriffen wird, oder dass ein Block versehentlich zwei Mal freigegeben wird):

  Der Block darf nicht schon dann freigegeben werden, wenn der erste dieser Pointer verschwindet, sondern *erst dann*, wenn der *letzte* Pointer auf ihn weg ist.

- Und es gibt auch Fälle, bei denen *"ewig"* Pointer auf dynamisch angelegte Blöcke zeigen (und damit eine *Freigabe verhindern*), obwohl auf die Blöcke eigentlich *gar nicht mehr zugegriffen werden kann* und sie daher freigegeben werden sollten. Ein Beispiel ist eine *zyklische Liste*, auf die kein Pointer von außen mehr zeigt.

Der *"alte" Mechanismus* (*vor* C++11) für C++-RAII-Speicherverwaltung, nämlich **`auto_ptr`**, erwies sich daher als in der Praxis *nicht wirklich optimal*. Er wurde in C++11 als *"veraltert"* ("deprecated") gekennzeichnet (sollte also *nicht mehr verwendet* werden!) und durch die drei neuen Klassen **`shared_ptr`**, **`unique_ptr`** und **`weak_ptr`** ersetzt.

`shared_ptr`

`shared_ptr` sind *universelle Verwaltungsobjekte für dynamisch angelegte Speicherblöcke*.

- Ein **`shared_ptr`** ***"verpackt" einen normalen Pointer*** und verhält sich in zentralen Punkten wie ein Pointer: Man kann mit **`*`** auf das "Ding" *zugreifen*, auf das er zeigt, und mit **`->`** auf einzelne Member dieses "Dings". Mit **`get()`** kann man sich auch wieder den "nackten" Pointer aus einem **`shared_ptr`** holen.

  Auch die *Vergleichsoperatoren* zwischen zwei **`shared_ptr`** funktionieren, und man kann einen **`shared_ptr`** mit dem **`nullptr`** vergleichen (aber nicht mit einem normalen Pointer).

  Man sollte allerdings *nur* einen *Pointer auf einen dynamisch angelegten Speicherblock* in einem **`shared_ptr`** verpacken, *keinen* Pointer auf *globale oder lokale* Objekte, und auch *keinen* Pointer, die z.B. *mitten in einen Block* zeigt (für solche Fälle gibt es Sonderformen des **`shared_ptr`**): **`shared_ptr`** ruft ja irgendwann **`delete`** für seinen Pointer auf.

  Erst seit C++17 darf man auch einen Pointer auf ein *dynamisch angelegtes Array*

in einem **shared_ptr** verpacken, und **shared_ptr** unterstützt den Operator **[]** zum Zugriff auf die Elemente dieses Arrays.

- Ein **shared_ptr**-Objekt hat im Unterschied zu einem normalen Pointer *zwei "Spezialfälle"*: Es kann explizit den **nullptr** enthalten, oder es kann *noch gar keinen Pointer* enthalten (das passiert z.B. wenn ein **shared_ptr** mit dem *Standardkonstruktor* ohne Initialisierungswert angelegt wird), d.h. *für keinen Speicherblock zuständig* sein. In beiden Fällen gilt der **shared_ptr** in *Bedingungen* so wie ein **nullptr** als **false**.
- **shared_ptr**-Objekte können *zugewiesen* und (per Copy-Konstruktor, "by Value"-Übergabe usw.) *kopiert* werden. Dadurch können *mehrere* **shared_ptr** entstehen, die *denselben Pointer enthalten* bzw. *auf denselben Block* zeigen.

  Deshalb auch der Name **shared_ptr**: Mehrere **shared_ptr** *teilen sich gleichberechtigt* den Datenblock und die Verantwortung dafür.
- *Jeder* **shared_ptr** ist ein ***"Owner"*** des dynamischen Datenblocks, auf den er gerade zeigt:
  - Einerseits hält er diesen Datenblock ***"am Leben"***: Der dynamisch angelegte Speicherblock wird erst dann *freigegeben*, wenn der *letzte* **shared_ptr** auf ihn *verschwindet*.
  - Andererseits trägt er aber auch die *Verantwortung für das Freigeben* der Daten, falls er der letzte **shared_ptr** darauf ist.

Technisch funktioniert das, indem **shared_ptr** bei jeder Zuweisung, Kopie usw., und natürlich auch bei jedem Destruktor, einen *internen Zähler* aktualisiert, der angibt, wie viele **shared_ptr** gerade auf diesen Block zeigen. Geht dieser Zähler *von* **1** *auf* **0**, wird der Block *freigegeben* (mit der Methode **use_count** kann man auch abfragen, *wie viele* **shared_ptr** gerade *auf denselben Block* wie der aktuelle **shared_ptr** zeigen). Technisch gesprochen realisieren **shared_ptr** also eine *Garbage Collection nach dem "Reference Counting"-Prinzip*.

Damit dieser Mechanismus funktioniert, gibt es aber eine *ganz wichtige Voraussetzung*:

Jeder "normale" Pointer darf nur ein einziges Mal
in einen* shared_ptr *(oder einen* unique_ptr*) verwandelt werden!

Weitere **shared_ptr** auf *denselben Block* bzw. mit demselben internen Pointer dürfen *nur durch Kopie, Zuweisung usw.* des ursprünglichen **shared_ptr** erzeugt werden, *niemals* dadurch, dass man *aus dem normalen Pointer ein zweites Mal* einen **shared_ptr** erzeugt. In diesem Falle würde nämlich *intern ein zweiter interner Use-Counter* für denselben Speicherblock entstehen und *unabhängig vom urprünglichen Zähler zählen*, was die gesamte interne Verwaltung ins Chaos führt.

Dies gilt *auch für das Objekt selbst*, auf das der **shared_ptr** zeigt: Es wäre ein *schwerer Fehler*, in einer Methode dieses Objektes ausgehend von **this** *nochmals* einen **shared_ptr** *auf sich selbst* zu erzeugen. Wenn ein Objekt (eventuell mehrfach) **shared_ptr** auf sich selbst generieren soll (was ja prinzipiell möglich und sinnvoll ist), so muss dafür der spezielle Mechanismus **shared_from_this** verwendet werden.

Ursprünglich erzeugte man **shared_ptr** vor allem durch einen *Konstruktor-Aufruf mit dem "normalen" Pointer*, also z.B.

```
myClass *p = new myClass(arg1, arg2);
shared_ptr<myClass> sptr(p);
```

(**arg1** und **arg2** sind hier Argumente des Konstruktors von **myClass**)
oder kürzer

```
shared_ptr<myClass> sptr(new myClass(arg1, arg2));
```

Dies erwies sich aber als *fehleranfällig*, erstens im Hinblick auf die oben genannte Regel (weil man ja versehentlich nochmals einen **shared_ptr** aus **p** erzeugen könnte), und zweites im Hinblick auf eventuell an der falschen Stelle auftretende Fehler oder Exceptions (die dann doch bewirken könnten, dass das **myClass**-Objekt im Speicher übrigbleibt, *ohne* dass ein Pointer darauf oder ein **shared_ptr** dafür existiert). Außerdem ist es nicht effizient, weil *zwei Mal separat Speicher allokiert* wird.

Daher wurde ein neues *Template* **make_shared** eingeführt, dass das *Anlegen und Initialisieren* des Datenobjektes und des darauf zeigenden **shared_ptr** *"in einem Rutsch"* erledigt, *ohne* dass man den "normalen" Pointer jemals zu Gesicht bekommt:

```
auto sptr = make_shared<myClass>(arg1, arg2);
```

Daher die oben genannte Regel "***never use* new*, use* make_shared *or* make_unique"**: Die "normalen" Pointer auf dynamisch angelegte Objekte und Arrays sollten in einem modernen Programm *gar nicht mehr sichtbar* sein, nur die **shared_ptr**.

unique_ptr

Ein **shared_ptr** ist *ziemlich aufwändig*. Bei einer simplen Zuweisung **sp1 = sp2;** muss beispielsweise der Counter von dem Block, auf den **sp1** bisher zeigte, um **1** reduziert werden. Dann muss geprüft werden, ob der Counter dadurch **0** wurde, und wenn ja, muss der alte Block von **sp1** freigegeben werden.
Schließlich muss das "Innenleben" von **sp2** in **sp1** kopiert werden, und der Counter des nun gemeinsamen Blockes muss um **1** erhöht werden.

In vielen Fällen weiß der Programmierer aber schon beim Design seiner Anwendung, dass *auf einen Speicherblock konzeptionell nie zwei Pointer* zeigen sollten, sondern immer nur einer. Technisch folgt daraus, dass es *immer nur einen einzigen Owner* bzw. nur *ein* für den Block zuständiges *Verwaltungsobjekt* geben darf.

Für solche Fälle sollte man **unique_ptr** statt **shared_ptr** verwenden:

- **unique_ptr** ist intern *deutlich effizienter* als **shared_ptr** (es führt z.B. *keinen Counter* über die Anzahl der Pointer auf den Block).
- Die Verwendung eines **unique_ptr** ist ziemlich ähnlich zu einem **shared_ptr**: Er kann mit **make_unique** erzeugt werden (oder mit einem Konstruktor mit dem normalen Pointer), und er kennt *** -> [] get()** und Vergleiche.
- Damit die intendierte und notwendige Bedingung *"es gibt nur einen Owner"* auch eingehalten wird, gibt es aber *keinen normalen Copy-Konstruktor* und *keinen normalen Zuweisungs-Operator*: Es gibt nur einen Move-Konstruktur und eine Move-Zuweisung (siehe Kapitel C++11) die dem ursprünglichen **unique_ptr** den Speicherblock wegnehmen und ihn an den neuen **unique_ptr** übertragen.

  Damit ist technisch sichergestellt, dass es *pro Speicherblock nur einen* **unique_ptr** gibt, der darauf zeigt. Das bedeutet z.B. auch, dass man **unique_ptr**

nicht als *"by Value"*-Parameter an Funktionen und Methoden übergeben kann, sondern *nur* *"by Reference"*.

`weak_ptr`

Ein **`weak_ptr`** speichert einen Pointer auf einen Speicherblock,
der von einem **`shared_ptr`** verwaltet wird,
ohne dass der **`weak_ptr`** ein *Owner* dieses Speicherblockes wird:

- Ein **`weak_ptr`** ist *nicht "lebenserhaltend"*: Wenn alle **`shared_ptr`** verschwinden, die auf den betreffenden Block zeigen, dann wird der Speicherblock *freigegeben*, auch wenn noch ein oder mehrere **`weak_ptr`** *auf ihn zeigen*.
- Umgekehrt bewirkt das Verschwinden eines **`weak_ptr`** *niemals*, dass der dazugehörige Block *freigegeben* wird (denn wenn es noch **`shared_ptr`** darauf gibt, dann darf der Block noch nicht freigegeben werden, und wenn es keine **`shared_ptr`** mehr darauf gibt, ist der Block ohnehin schon weg).
- Rein technisch bedeutet das, dass ein **`weak_ptr`** den Pointer-Counter des dazugehörigen Blockes *weder erhöht noch erniedrigt*.

Ein **`weak_ptr`** wird also verwendet, wenn man sich einen Pointer auf Daten merken will, *ohne* die Daten dadurch zwangsweise *am Leben zu erhalten*:
Die Daten *können und dürfen verschwinden*, auch wenn noch **`weak_ptr`** darauf zeigen.
Man behält die Daten sozusagen *als Unbeteiligter* im Auge.

Ein typisches Beispiel ist der *"Ringschluss"* (der Pointer *vom letzten auf das erste Element*) bei einer *zyklisch verketteten Liste*: Einerseits muss es (eben über diesen Pointer) möglich sein, eine zyklische Liste "im Kreis herum" zu durchlaufen. Andererseits würde die Liste *nie mehr freigegeben* werden, wenn alle Pointer im Kreis **`shared_ptr`** wären, weil dadurch die Counter aller Elemente der Liste mindestens **`1`** wären.
Ist der "Ringschluss" aber ein **`weak_ptr`**, so geht der Counter des ersten Elementes auf **`0`**, *sobald der Head-Pointer der Liste verschwindet*. In diesem Moment wird das erste Element (und in der Folge die gesamte Liste) *freigegeben*.
Der "Ringschluss" allein erhält die Liste also nicht am Leben.

Da es die Daten, auf die ein **`weak_ptr`** zeigt, vielleicht gar nicht mehr gibt, unterstützt ein **`weak_ptr`** *keine direkten Zugriffe auf die Daten* (kein **`* -> [] get()`**):
Um zugreifen können, muss man vom **`weak_ptr`** durch Aufruf von **`lock()`** einen (oft temporären!) **`shared_ptr`** auf die Daten *erzeugen*. Nur über diesen **`shared_ptr`** kann man dann *zugreifen*, und er garantiert auch, dass die Daten *nicht verschwinden*, solange man zugreift. Sind die Daten inzwischen schon freigegeben worden, liefert **`lock()`** einen "leeren" **`shared_ptr`**. Mit **`expired()`** kann man prüfen, ob die Daten verschwunden sind, ohne einen **`shared_ptr`** zu erzeugen.

Anhang A: Unbehandelte Konstrukte

Folgende Punkte haben wir nicht oder nicht in ausreichender Tiefe behandelt:

- Geschachtelte und lokale **`class`**-Deklarationen, **`struct`** in **`class`**, ...:

  Man kann innerhalb eines **`class`** weitere Klassen deklarieren (*"nested class"*). Eine innere Klasse hat vollen Zugriff auf die äußere Klasse, aber unterliegt zum Rest des Programmes hin den gleichen Namens-, Sichtbarkeits- und Zugriffsregeln wie andere Member und Methoden der äußeren Klasse.
- Namensräume und andere **`using`**-Deklarationen
- **`explicit`**-Deklarationen zum Unterbinden impliziter Konvertierungen
- Explizite Zugriffsrechte für geerbte Member und Methoden
- Member-Pointer, **`->*`** und **`.*`**
- Fehlerbehandlung bei **`new`**
- Definition eigener **`new`** und **`delete`**
- C++ I/O, I/O-Fehlerbehandlung, Stringstreams (siehe Schüler-Unterlagen!)
- C++ **`string`**-Klasse (siehe Schüler-Unterlagen!)
- C++ Standard Library STL im Detail, Boost
- Template-Spezialisierungen
- Templates mit Template-Argumenten,
  Fortgeschrittene Template-Programmierung, ...
- RTTI (RunTime Type Information), **`typeid`**
- Neue Features aus dem kommenden C++-Standard (C++ 11) (viele, siehe Anhang!)

Anhang B: C++-11, der neue Standard

Compilerunterstützung (Stand Ende 2013):

Folgende freie Compiler unterstützten C++11 *vollständig*:

- LLVM / clang (Mac, Linux optional) seit Version 3.3
- GCC (Linux, DevCpp, ...) seit Version 4.8.1 (in DevCpp seit Version 5.6.0 / Jan 2014)

ICC (Intel) und MSVC (Visual Studio) unterstützen den Standard knapp zur Hälfte (Stand Sommer 2014), aber bei weitem nicht vollständig.
Alle anderen bedeutenden C++-Compiler unterstützen nur kleine Teile des Standards.

Liste der Neuerungen

- **`nullptr`**:

  Statt **`NULL`**, macht weniger Probleme bei Überladung, Templates, ...

- **`{}` in Deklarationen mit Initialisierung, Konstruktor-Aufrufen**, ...:

  Es wurde auch **`{}`** statt **`()`** erlaubt,
  das **`=`** in Deklarationen mit Initialwerten kann man weglassen:

  ```
  int i{5};
  ```

  ... `new Schueler{"Max Muster", 0123456}` ...

  `{}` gilt auch in Initialisierungslisten von Konstruktoren: ... `, member{wert},` ...

  Neuer Mechanismus: Auch als Initialisierung von Containertyp-Variablen:

  ```
  vector<int> data{1, 2, 3, 4, 5};
  ```

  Es können auch selbst Konstruktoren bzw. Funktionen definiert werden,
  die so eine *Liste von Initial-Werten als Parameter* bekommen.

- **Typ "`auto`"**:

  Automatische Typ-Herleitung bei initialisierten Variablen:
  Die Variable bekommt den Typ, den der Initialisierungswert hat.

  Vor allem bei Iterator-Variablen in **`for`**-Schliefen, die komplizierte Typen haben!

  ```
  auto iter = myVector.begin();
  ```

  In Zusammenhang damit neu bei Container-Datentypen:

  `cbegin()` und **`cend()`** zusätzlich zu **`begin()`** und **`end()`** :
  Liefern **`const`**-Iteratoren statt normale Iteratoren
  (u.a. damit **`auto`** den Iterator **`const`** macht).

  "**`auto`**" hatte früher eine ganz andere (seit langem funktionslose) Bedeutung.
  Diese wurde ersatzlos entfernt, "alte" **`auto`** einfach wegstreichen!

- **Range-`for`-Schleife:**

  Im "alten" C++ gab es eine Funktion **`for_each`** in der STL (Header **`<algorithm>`**), die mit einem **`begin`**- und einem **`end`**-Iterator sowie einer unären Funktion aufgerufen wurde und diese Funktion auf jedes Element zwischen **`begin`** und **`end`** angewendet hat.

  Jetzt gibt es eine neue **`for`**-Variante, die das noch einfacher macht, z.B.:

  ```
  for (auto &elem : vect) { cout << elem << ' '; }
  ```
  (z.B. für `vector<double> vect;`)

  ```
  for (int i : { 2, 3, 5, 7, 11, 13, 17, 19 }) { … i … }
  ```

  Ein solches **`for`** funktioniert für:

  - *Echte Arrays* (mit **bekannter** Größe, *keine* Funktions-Parameter!)
  - Listen von Werten (gleichen Typs!) in **`{ }`**
  - Container-Werten: Alles, was Iteratoren **`begin()`** und **`end()`** hat

  Achtung:

  - Die Schleifenvariable ist *kein Iterator*, sondern *der Wert selbst* (das jeweilige Element des Containers).
  - Bei Containern: *Referenz* vor der Variable nicht vergessen!
    `auto &elem` bzw. bei nur-Lese-Elementen **`const auto &elem`**
  - Auch bei Arrays gilt:
    Schleifen-Variable ist *nicht Referenz*
    ==› *Kopie* des jeweiligen Array-Elementes (Änderung wirkungslos!)
    Schleifen-Variable ist *Referenz*
    ==› Das *Array-Element selbst* (*änderbar!*)

- **Default-Member-Initialisierung:**

  Initialwert-Zuweisung in der Member-Deklaration
  (von *nicht* statischen und *nicht* konstanten Membern)
  ==› Das Member wird beim Erzeugen des Objektes *mit diesem Wert initialisiert*, wenn das Member *nicht in der Initialisierungs-Liste des Konstruktors* steht (d.h. keinen expliziten Wert bekommt).

  ```
  string name = "Max Mustermann";
  ```

- **Vererben von Konstruktoren:**

  ```
  class Abgel : public BaseClass {
    using BaseClass::BaseClass;
  };
  ```

  `Abgel` bekommt *alle* Konstruktoren von **`BaseClass`**.

  In **`Abgel`** neu dazugekommene Member bekommen ihren *Default-Wert* (siehe Punkt "Default-Member-Initialisierung").
  Wenn sie keinen Defaultwert haben, bleiben sie uninitialisiert.

- **Konstruktor-Delegation:**

  Ein Konstruktor kann in der Initialisierungs-Liste einen anderen Konstruktor *derselben* Klasse aufrufen (dann darf aber sonst *nichts* in der Init-Liste stehen), z.B. mit bestimmten Parametern, um ihm die Initialisierungsarbeit zu übertragen.

- **`operator=` usw. kann mit `= delete` verboten / entfernt werden:**

  ```
  MyClass &operator=(const MyClass &right) = delete;
  ```

  ==› Es gibt in dieser Klasse keinen **`operator=`** für diesen Parameter-Typ.
  ==› Auch der automatisch erzeugte **`operator=`** wird in dieser Klasse *nicht* erzeugt!

  `= delete` ist auch für *Copy-Konstruktor*, *Standard-Konstruktor*, ... verwendbar!

  Ebenso kann man damit einzelne Template-Spezialisierungen verbieten:
  "Das Template gibt es für alle Typen, nur für diesen bestimmten Typ nicht".

- **Typ "`decltype`":**

  `decltype(`*variable*`)`**
  oder
  `decltype(`*rechnung*`)`**
  oder
  `decltype(`*funktionsaufruf*`)`**
  liefert den Typ, den das Ergebnis in den **`( )`** hat.

  Man kann mit **`decltype(...)`** Variablen, Returntypen, Parameter, ... deklarieren.

- **Rvalue-Referenzen:**

  Referenzen auf **Rvalues**, d.h. auf ***temporäre Objekte***
  (z.B. Rechen-Zwischenergebnisse, "by value"-Returnwerte,
  direkte Konstruktor-Aufrufe, Konstanten, ...).

  Kennzeichen eines Rvalues:
  Es ist nicht möglich / nicht sinnvoll, auf Rvalues etwas zuzuweisen (sie zu ändern), weil Rvalues unmittelbar nach Ende der Berechnung "entsorgt" werden.

  Bei Parameter-Deklarationen in Funktions-Prototypen: **`&&`** statt **`&`**
  ==› Der Parameter ist eine *Referenz auf einen Rvalue*.
  ==› Die Funktion bzw. Methode "passt" dann und nur dann,
  wenn das entsprechende *Argument im Aufruf ein Rvalue* ist.

  Das **`&&`** gibt es auch für das "**`this`**" bei Methoden-Aufrufen:

  ```
  class myClass {
    int myFunc(...) &; // wird für "normale" Objekte aufgerufen
    int myFunc(...) &&; // wird für "temporäre" Objekte aufger.
  };
  ```

- **"Verschiebe-Konstruktor" und "Verschiebe-Zuweisung":**

  Problem:

  In C++ wird sehr oft der Inhalt eines Objektes aufwendig kopiert (z.B. bei Objekten, die Pointer auf dynamisch angelegte Daten enthalten), obwohl das Original danach nicht mehr benötigt bzw. unmittelbar darauf gelöscht wird (z.B. bei temporären Objekten), oder obwohl das Original kurz darauf mit anderen Werten überschrieben wird.

  Beispiel:

```
template<class T>
void swap(T &a, T &b)
{
  T tmp(a);
  a = b;
  b = tmp;
}
```

  Obwohl es nach dem Aufruf von **swap** nicht mehr Objekte gibt als vorher (nämlich zwei), wird hier drei Mal eine Kopie eines kompletten Objekt-Inhaltes erstellt (im Copy-Konstruktor und in den beiden Zuweisungen), und drei Mal wird ein kompletter Objekt-Inhalt gelöscht (in den beiden Zuweisungen die alten Inhalte von **a** und **b** und bei der **}** der Inhalt von **tmp**). Enthalten die Objekte umfangreiche dynamische Datenstrukturen, ist das ein großer (und unnötiger!) Aufwand (viele **new** und **delete**): Eigentlich würde es reichen, die bestehenden dynamischen Daten unverändert vom alten Objekt ins neue zu verschieben (d.h. nur die Pointer darauf zu kopieren statt die Inhalte) und danach im Ausgangsobjekt als ungültig zu markieren (z.B. durch Nullpointer zu ersetzen), damit im alten Objekt nicht versehentlich nochmals auf die Daten zugegriffen wird, obwohl sie schon einem anderen Objekt gehören, und vor allem, damit sie nicht vom Destruktor gelöscht werden, wenn das alte Objekt freigegeben wird.

  Lösung:

  - Dafür wurde der "Verschiebe-Konstruktor" und die "Verschiebe-Zuweisung" eingeführt: Der Aufruf sieht aus wie bei Copy-Konstruktor und **operator=**, nur mit **&&**-Parameter statt **&**-Parameter (und *ohne* **const** beim Parameter).

    myClass::myClass(myClass &&orig) und
    myClass &myClass::operator=(myClass &&rhs)

    Der Zweck ist auch derselbe, nur dass intern eben ins Zielobjekt "verschoben" statt kopiert wird (das muss man selber entsprechend programmieren, z.B. indem man einfach die Pointer kopiert) und das Original danach irgendwie als "leer" oder "ungültig" markiert wird (auch das muss man selber programmieren, indem man z.B. alle Pointer-Member auf Nullpointer setzt, genau deswegen ist der Parameter für das Original nicht **const**). Der ursprüngliche Inhalt wird also dem alten Objekt "weggenommen" und ins neue Objekt "eingepflanzt".

  - Wenn das Original bzw. die rechte Seite vom = ein Rvalue ist (also z.B. ein Funktions-Returnwert, ein temporäres Objekt, ...), ruft der Compiler *automatisch* den Verschiebe-Konstruktor statt dem Copy-Konstruktor bzw.

das Verschiebe-= statt dem normalen = auf, falls sie vorhanden sind:

Er kann sich ja sicher sein, dass das (temporäre) Original-Objekt unmittelbar danach sowieso verschwindet.

- Wenn das Original kein Rvalue ist, aber man genau weiß, dass das Original nachher nicht mehr verwendet oder gleich mit einem neuen Wert überschrieben wird, kann man den Compiler mit **move** dazu bringen, den Verschiebe-Konstruktor bzw. die Verschiebe-Zuweisung zu verwenden:

  Die neue vordefinierte Funktion **move(***variable***)** macht aus einem "echten" Objekt (der Variable), das ja kein Rvalue ist, ein temporäres Objekt (einen Rvalue), ohne etwas zu kopieren oder am Objekt zu ändern.

  Auf den Wert der Variable darf nachher nicht mehr lesend zugegriffen werden, da ihr Inhalt ja von der anschließenden Verschiebe-Operation auf "leer" gesetzt wird (d.h. die Variable muss anschließend entweder gelöscht werden bzw. ihr Gültigkeitsbereich muss enden, oder es muss ein neuer Wert in der Variable gespeichert werden).

Beispiel:

```
template<class T>
void swap(T &a, T &b)
{
  T tmp(move(a));
  a = move(b);
  b = move(tmp);
}
```

Hier werden nur die beiden Original-Inhalte von **a** und **b** 3 Mal verschoben:
Es werden keine dynamischen Daten neu angelegt, kopiert oder gelöscht
==› Bei komplexen Objekten viel effizienter!

Und wenn der Typ **T** keinen eigenen Verschiebe-Konstruktor bzw. keine Verschiebe-Zuweisung hat, funktioniert alles wie bisher
(mit dem normalen Copy-Konstruktor bzw. Zuweisungsoperator).

- **Lambda-Ausdrücke:**

  Ein Lambda ist eine Funktion ohne Namen (anonymer Funktionscode).
  Sie kann z.B. überall dort hingeschrieben werden,
  wo ein Function Pointer o.ä. als Argument erwartet wird.
  Man kann sie auch in einem Function Pointer speichern.

  Einfachste Form: **[](*argumente*){ *Code* }**

  In **[]** steht, auf welche Variablen des umgebenden Codes man zugreifen kann (im einfachsten Fall: Klammer leer ==› Kein Zugriff auf umgebende Variablen).

  Der Returntyp wird automatisch ermittelt.

Beispiel:

```
for_each(&(arr[0]), &(arr[size]),
        [](Elem &x){ cout << x << " "; });
```

for_each ist eine vordefinierte Funktion aus dem Header **<algorithm>**.
Sie hat als drittes Argument eine Funktion mit einem Parameter und ruft diese Funktion der Reihe nach für alle Elemente eines Containers zwischen zwei Iteratoren bzw. für alle Elemente eines Arrays zwischen zwei Pointern auf.
In diesem Beispiel ist die Funktion ein Lambda. Es gibt den Parameter aus.
Es werden also alle Elemente des Arrays **arr** ausgegeben.

- **Funktionsprototypen und Kopfzeilen mit Returntyp hinten:**

  Vorne steht als Returntyp **auto**, hinter der **)** der Parameter steht **–>** ***returntyp***

  ```
  auto myfunc(int i, int j) -> int
  ```

- **Iteratoren begin und end als Funktionen:**

  Bisher waren **begin** und **end** Methoden der Container-Klassen.
  Man schrieb also **v.begin()** und **v.end()** .

  Jetzt sind sie zusätzlich als überladene globale Funktionen definiert.
  Man kann also auch **begin(v)** und **end(v)** schreiben.

- **using ähnlich typedef:**

  Statt **typedef vector<double> messw_t;**
  kann man auch schreiben **using messw_t = vector<double>;**

  Vorteil gegenüber **typedef**:

  using erlaubt es, bei einem Template einige, aber nicht alle Typ-Parameter zu fixieren:

  ```
  template <typename A, typename B, typename C>
  class myClass { ... };

  template <typename T>
  using mySpecialClass = myClass<vector<int>, T, int>;
  // A und C fixiert, B bleibt Parameter

  mySpecialClass<string> myObj;
  // myObj ist myClass<vector<int>, string, int>
  ```

- **constexpr:**

  Wie **const**, aber stärker: Der Wert muss schon *zur Compile-Zeit berechenbar* sein, sonst meldet der Compiler einen Fehler.

  - Bei **Variablen**: **constexpr int myConst = …;**

    … muss zur Compilezeit berechenbar sein.
    myConst ist sicher eine "echte" Konstante (darf als Template-Parameter, Array-Größe in Deklarationen, als **case**-Fall usw. verwendet werden).

  - Bei **Funktionen**:
    constexpr int abs(int n) { return (n < 0) ? –n : n; }

    Wenn die Funktion mit Compilezeit-konstanten Parametern aufgerufen wird, dann muss das Ergebnis zur Compilezeit berechenbar sein.

(Die Funktion besteht nur aus **`return`** mit einer berechenbaren Rechnung).

- Bei **Konstruktoren**: **`constexpr myClass(...) : ... { ... }`**

  Wenn der Konstruktor mit Compilezeit-konstanten Parametern aufgerufen wird, dann muss das Objekt zur Compilezeit anleg- und initialisierbar sein.

Fortgeschrittenes Konzept: "Literale" Typen:
Klassen mit bestimmten Eigenschaften, erlauben **`constexpr`**-Methoden.
==› In **`constexpr`**-Initialisierungen verwendbar.

- **Das Template "`function`":**

  Es gibt in der STL ein neues, vordefiniertes Template "**`function`**"
  (aus dem Header **`<functional>`**) für Funktions-Objekte
  (das können sein: C Funktionspointer, Methoden, **`operator()`**, Lambda's)

  Beispiel: **`function<double(double x, double y)> f;`**

  ==› **`f`** ist ein Funktionsobjekt (oder Funktionspointer)
  für eine Funktion mit 2 **`double`**-Argumenten und einem **`double`**-Returnwert.

  `function` wird vor allem in Parameter-Deklarationen verwendet:
  Jede Funktion / Methode der STL, die einen Funktions-Parameter hat
  (z.B. für Vergleiche oder zum Filtern von Container-Elementen),
  kann für diesen Parameter alles übergeben bekommen, was auf **`function`** passt:
  Funktionspointer, Lambda, **`operator()`**, Funktions-Returnwert z.B. von **`bind`**, ...

- **Methoden-Schlüsselwort "`override`":**

  Kann im Prototyp / der Kopfzeile einer Methode nach den Parameter-**`()`** stehen:
  ==› Die Methode *muss* eine Methode der Basisklasse überschreiben,
  sonst kommt ein Compilerfehler.

  Auch für virtuelle Destruktoren!

  Zweck: Schützt vor Denk- und Typ-Fehlern, wenn man eigentlich eine Methode *überschreiben wollte,* aber *versehentlich* (z.B. auf Grund nicht passender **`const`**) eine *neue Methode* definiert hat.

- **Methoden-Schlüsselwort "`final`":**

  Kann im Prototyp / der Kopfzeile einer Methode nach den Parameter-**`()`** stehen:
  ==› Die Methode darf nicht mehr überschrieben werden.

  Auch im Klassenkopf möglich: **`class ... final`**
  ==› Von dieser Klasse kann nichts mehr abgeleitet werden.

  Schützt vor versehentlichem Überschreiben oder Ableiten und hilft dem Compiler, effizienteren Code zu erzeugen (direkte Methoden-Aufrufe statt virtuell-Mechanismus).

- **Methoden-Schlüsselwort "`explicit`":**

  Bisher nur bei Konstruktoren erlaubt, jetzt auch bei *Typumwandlungs-Operatoren*:
  "Nur explizit aufgerufen verwendbar, nicht für implizite Typumwandlungen!"

- **Funktions- und Methoden-Schlüsselwort "`noexcept`":**

  ==› *"Diese Funktion wirft sicher nichts"*

- Unterschied zwischen **noexcept** und **throw()**:
  - **noexcept**
    lässt den Exception-Implementierungs-Overhead fast komplett weg.

    ==› Kommt doch eine Exception, wird das Programm sofort beendet.
  - **throw()**
    (eigentlich selbe Bedeutung: "Da sollte nichts geflogen kommen")
    prüft auf unerwartet geworfene Exceptions (viel mehr Overhead).

    ==› Kommt doch eine Exception:
    Aufruf der vordefinierten Funktion "**unexpected()**",
    das Programm könnte danach geordnet weitermachen.

  throw(...) zur Angabe der Exceptions, die aus einer Funktion heraus kommen können, ist ab C++-11 deprecated (veraltert, sollte man nicht mehr verwenden).
- **Prüfungen zur Compile-Zeit:**

  static_assert(*bedingung*, "*Fehlertext*");

  Die Bedingung muss Compilezeit-berechenbar sein.
  Ist sie falsch, liefert der Compiler eine Fehlermeldung.
- **"Variadic Templates":**

  Templates mit variabel vielen Template-Parametern.
- **Erweiterungen bei enum's:**
  - **enum class myEnum { C1, C2, C3 };**
    statt
    enum myEnum { C1, C2, C3 };

    ==› Die Enum-Konstanten **C1** usw. sind keine globalen Namen mehr, sondern lassen sich nur mit **myEnum::C1** usw. ansprechen.

    ==› *Gleichnamige Konstanten* in verschiedenen **enum**'s *kollidieren nicht mehr*!
  - z.B.: **enum class myEnum : unsigned long { C1, C2, C3};**

    ==› "Nimm einen **unsigned long** zur Speicherung dieses **enum**'s!"
    Bisher hatte der Compiler mehr oder weniger freie Wahl
    bzw. nahm den kleinstmöglichen Typ, in den alle Werte hineinpassten.
- **Unicode-Strings:**
  - **u8"blabla"**

    ==› Speichere **"blabla"** als UTF-8-String
    (Typ: normaler **string** oder **char ***)
  - **u"blabla"**

    ==› Speichere **"blabla"** als UTF-16-String
    (Typ: **u16string** oder **char16_t ***)

- **`U"blabla"`**

  ==› Speichere **`"blabla"`** als UTF-32-String
  (Typ: **`u32string`** oder **`char32_t *`**)

- Unicode-Konstanten in Hex angeben:
  `\u`*`hhhh`* (16 bit) oder **`\U`*`hhhhhhhh`*** (32 bit)

- **Roh-Strings:**

  `R"(blabla)"`

  ==› Der String **`"blabla"`**, aber ohne Sonderbedeutung für **`\`**, **`"`**, …
  (d.h. **`\`** im String bleibt als solches erhalten)

  Achtung: Die **`( )`** gehören zum Konstrukt dazu und sind nicht im resultierenden String enthalten!

Einige Neuerungen in Randbereichen habe ich weggelassen…

Erweiterungen der STL (vordefinierte Library):

- **Container "`unordered_map`" und "`unordered_set`":**

  Wie "**`map`**" und "**`set`**", aber nicht sortiert (es gibt auch die "**`multi`**"-Varianten).

  Interne Speicherung als *Hashtable* statt Binärbaum
  (schneller beim Suchen / Einfügen / Löschen) .

  Iterieren über alle Elemente ist möglich, aber nur in "irgendeiner" Reihenfolge.

- **Container "`forward_list`":**

  Einfach verkettete Liste (normale "**`list`**" ist doppelt verkettet).

- **Viele neue Container-Funktionen.**

- **`tuple`**:
  Template-Typ für eine *fixe Anzahl von Werten verschiedenen Typs*
  (Verallgemeinerung des bisherigen Templates **`pair<T1, T2>`** für zwei Werte).

  Zugriff mit

  - **`get<i>(tupleWert)`**

    ==› liefert die **`i`**-te Komponente (**`i`** muss eine Compilezeit-Konstante sein!)

  - **`tie(v1, v2, ...) = tupleWert;`**

    ==› Speichert die Komponenten des Tupels
    der Reihe nach in den Variablen **`v1`**, **`v2`**, …

  Anlegen eines Tupels mit **`{…}`**-Initialisierung, **`make_tuple(…)`** oder **`tie(…)`**.

- **Neue vordefinierte Header:**

  U.a. für *Regular Expressions*,
  für *Wahrscheinlichkeitsverteilungen* (Binomial, Poisson, Normal, Chi2, …)

- **Multithreading: `thread` oder `async`**

  Weiters:
  Neue Speicherklasse "**`thread_local`**" für Variablen,

die *einmal pro Thread* angelegt werden sollen.

Zur Synchronisation: **`mutex`**, ...