Datenbanken IIB

Kapitel A: Einführung in C++ für Java-Programmierer

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2015/16

http://www.informatik.uni-halle.de/~brass/dbi15/

Inhalt

Erstes Beispiel: “Hello, World!”

```
// Dies ist das klassische erste Programm.
// Es druckt den Text  Hello, world!

#include <iostream>
using namespace std;

int main()
{
        cout << "Hello, world!\n";
        return 0;
}
```

Hauptprogramm

- In C++ sind Funktionen/Prozeduren außerhalb von Klassen möglich. Mindestens `main` muss so definiert werden.

  C++ ist eine kompatible Erweiterung der Sprache C (nicht objektorientiert).

  C++ wird auch "Multi-Paradigmen-Sprache" genannt. Bei rein objektorientierter Programmierung nur `main()` und Operatoren außerhalb von Klassen.

- Der Rückgabetyp von `main()` ist `int` (in Java `void`):
  - Der Wert `0` bedeutet "erfolgreich beendet".
  - Kleine positive Zahlen dienen zur Codierung von Fehlern.

    In einem Shellscript kann man den Wert abfragen.
  - Negative Zahlen sollte man vermeiden.
- Variante mit Kommandozeilen-Parametern:

```
int main(int argc, char *argv[])
```

Include-Dateien

- Die Bezeichner/Symbole "`cout`" und "`<<`" sind in die Sprache C++ nicht eingebaut (Bibliothek zur Ein-/Ausgabe).

  << eigentlich schon, aber nur für Zahlen ("Shift"), nicht zur Ausgabe.

  In C++ kann man Operatoren für eigene Klassen/Datentypen überladen.

- Die Deklaration dieser Dinge steht in der Datei "`iostream`".
- Die Anweisung

  ```
  #include <iostream>
  ```

  teilt dem Compiler mit, dass er die Datei "`iostream`" an dieser Stelle lesen soll (formal macht das der Präprozessor).

- Anweisungen mit "`#`" richten sich an den C-Präprozessor.

  Der eigentliche Compiler bekommt die Ausgabe des Präprozessors als Eingabe. Der Präprozessor fügt die Include-Datei hier ein. Das "#" muss am Anfang der Zeile stehen, und die Anweisung kann nicht über mehrere Zeilen verteilt werden (nur mit "\" am Zeilenende). Sonst ist C++ wie Java formatfrei.

Namespaces

- Die Bezeichner in der Datei "`iostream`" sind im Namensraum "`std`" definiert (ähnlich zu "Package" in Java).

  Auf diese Art werden Symbole aus der Bibliothek von Symbolen aus Ihrem Programm getrennt. Die Bibliothek definiert sehr viele Symbole.
  Sie können schlecht alle kennen. Es könnte sonst aber zu Namenskonflikten kommen, wenn Sie zufällig eine Funktion gleichen Namens definieren.

- Außerhalb des Namespaces "`std`" müsste man schreiben:

  ```
  std::cout << "Hello, world!\n";
  ```

  Funktionen (<<) werden auch im Namespace ihrer Argumente gesucht.

- Mit der Anweisung

  ```
  using namespace std;
  ```

  werden alle Symbole aus dem Namensraum "`std`" auch ohne den Präfix "`std::`" verfügbar (ähnlich `import` in Java).

  Wenn in Ihrem Programm nicht ein Symbol gleichen Namens definiert ist.

Ausgaben

- `cout` ist der Standard-Ausgabestrom (Konsole/Terminal).

  Die Klasse von `cout` ist "`ostream`". Für Fehlermeldungen verwendet man `cerr`.

- `s << x` ist mit vielen Datentypen von `x` definiert und kann auch für eigene Klassen definiert werden.
- Es liefert den Strom `s` als Ergebnis und bindet von links, deswegen sind Ketten folgender Form typisch:

```
cout << "Der Wert von i ist: " << i << "\n";
```

- Man kann "Manipulatoren" ausgeben, um Einstellungen des Ausgabestroms zu verändern. Beispiel:

```
cout << fixed << setprecision(3);
```

  Dann: Drei Nachkommastellen für `double`-Werte.

  Oben "`#include <iomanip>`", damit diese Funktionen definiert sind.

  Weitere Manipulatoren z.B.: `hex`, `setw(5)` (nächste Feldbreite), `right`.

Compilierung (1)

- C++ wird direkt in Maschinensprache des jeweiligen Rechners übersetzt, nicht in Bytecode für eine virtuelle Maschine.

  Das gibt C++ einen Performance-Vorteil. Moderne JVMs versuchen, dies mit "just in time" Compilierung während der Ausführung auszugleichen. Sie haben dabei genauere Information über Hardware und den aktuellen Programmlauf als ein normaler Compiler. Wenn kleine Schleifen häufig ausgeführt werden, fällt der Overhead für die Compilierung nicht mehr ins Gewicht, ansonsten ist ein getrennter Compiler-Lauf (wie bei C++) günstiger. Bei Laufzeit-Vergleichen wähle man für C++ einen hohen Optimierungslevel.

- Unter Linux ist der GNU Compiler g++ Standard.

  ```
  g++ Hello.cpp
  ```

  Das ausführbare Programm steht anschließend in "`a.out`". Man kann es (unter Linux) ausführen mit "`./a.out`". Man muss das Verzeichnis mit angeben wenn es nicht Teil des Suchpfads für Kommandos ist (`$PATH`).

Compilierung (2)

- Einige Optionen des GNU-Compilers:
  - `-o Hello`: Ausgabedatei setzen (statt `a.out`).
  - `-Wall`: Warnungen anschalten (dringend empfohlen!).

    `-Wall` sind nicht ganz alle Warnungen. Ich verwende auch:
    `-Wextra -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wunreachable-code -pedantic -ansi`
  - `-g`: Informationen für Debugger gdb erzeugen.
  - `-O2`: Optimierungslevel 2.
  - `-c`: Nur compilieren (erfordert extra Linker-Lauf).

    Man macht dies, wenn ein Programm sich aus mehreren Quelldateien (Modulen) zusammensetzt, die getrennt übersetzt werden. Es entstehen dann Objektdateien mit der Endung "`.o`". Diese werden anschließend z.B. mit "`g++ A.o B.o -o C`" zu einem ausführbaren Programm (im Beispiel `C`) zusammengefügt (schneller als alles neu compilieren).

Compilierung (3)

- Für Windows-Rechner kann man z.B. “Microsoft Visual Studio Community 2015” kostenlos bekommen.

  Man muss sich registrieren, sonst läuft die Lizenz nach einiger Zeit aus.

  - Projekt-Typ: Leeres Projekt (für Kompatibilität mit Linux).

    Man wähle `File`→`New`→`Project`. Unter Templates muss “`Visual C++`” gewählt sein. Dann rechts “`Empty Project`”, unten Name und Verzeichnis wählen, und auf `OK` klicken. Im Verzeichnis wird automatisch ein Ordner mit dem Projektnamen angelegt. Existierende Quelldateien hinzufügen: Rechts-Klick auf das Projekt, `Add`→`Existing Item`.

  - Übersetzen: `Build`→`Build Solution` (`F7`).

  - Ausführen: `Debug`→`Start Without Debugging` (`Ctrl+F5`).

    Man muss am Ende des Programms auf eine Eingabe warten, sonst verschwindet das geöffnete Fenster sofort wieder. Alternativ mit PowerShell bzw. “Command Prompt” in Verzeichnis `Debug`, dort `exe`-Datei ausführen.

Kontrollstrukturen, Wahrheitswerte

- Variablendeklarationen, Zuweisungen und Kontrollstrukturen `if`, `while`, `for`, `do`, `switch` funktionieren in C++ wie in Java.

  `break` und `continue` auch, es gibt aber nicht die erweitere Form von `break` mit Label für die Schleife. Dafür gibt es ein allgemeines `goto` (ist aber verpönt).

- Es gibt aber eine automatische Typumwandlung von Zahlen und Zeigern (s.u.) nach `bool`: Die Zahl 0 und die Null-Referenz sind `false`, alles andere `true`.

  Man beachte, dass der Typ in C++ `bool` heisst, nicht `boolean` wie in Java.

- Das erlaubt manchmal kürzere Formulierungen, enthält aber das Risiko unerkannter Fehler: `if(i = 5) { ... }`

  Dies ist korrektes C++. Der Wert 5 wird `i` zugewiesen, und ist Wert der Zuweisung. Daher ist die Bedingung immer erfüllt. Warnungen anschalten!

- "Und": nur `&&` (nicht `&`). "Oder": nur `||` (nicht `|`).

Eingaben

- Beispiel:

```
int n = 0;
cout << "Bitte ganze Zahl eingeben: ";
cin >> n;
if(cin.fail()) {
    cerr << "Eingabe fehlerhaft.\n";
    return 1;
}
cout << n << "^2 = " << n*n << ".\n";
```

  Im Fehlerfall (z.B. Eingabe "abc") wird `n` kein Wert zugewiesen.

  Die falsche Eingabe bleibt im Eingabestrom stehen.

- `cin.get(c)` setzt die `char`-Variable `c` auf das nächste Zeichen des Eingabestroms.

  Dagegen überspringt "`cin >> c`" Whitespace (Leerzeichen, Zeilenumbrüche).

  Test auf Dateiende: `cin.eof()`. Test auf Fehler: `cin.fail()`. Beide liefern `bool`.

Uninitialisierte Variablen

- In C++ liegt es in der Verantwortung des Programmierers, uninitialisierte Variablen zu vermeiden.

  Wenn man den Wert einer Variablen abfragt, bevor man ihr einen Wert zugewiesen hat, bekommt man ein nicht (gut) vorhersehbaren Wert. Irgendein Bitmuster steht ja an der Speicherstelle. Wenn der Stack (zur Speicherung von Rücksprungadressen und lokalen Variablen) vorher schon größer war, steht da der Wert der letzten Variablen, die an dieser Stelle gespeichert wurde. Es kann aber z.B. sein, dass man einen Teil eines `double`-Wertes oder eine Rücksprungadresse jetzt als `int` interpretiert.

- Wenn man Warnungen einschaltet, bekommt man für möglicherweise nicht initialisierte lokale Variablen eine Warnung, wo Java einen Fehler meldet.

- Eine automatische Initialisierung von Arrays und Attributen gibt es aber nicht. Die kostet ja zusätzliche Laufzeit.

Zeichen, Wertebereiche

- In C++ ist der Datentyp `char` 8 Bit groß (ein Byte).

  Damit C++-Programme die Hardware optimal nutzen können, gibt es in C++ keine vorgeschriebenen Bitgrößen bzw. Wertemengen für die Datentypen. Eine Hardware, auf der `char` nicht 8 Bit sind, wäre aber sehr exotisch. Dagegen könnte auf kleinen Rechnern (Microcontrollern) ein `int` nur 16 Bit groß sein (normal sind 32 Bit wie in Java). Der Ausdruck `sizeof(T)` liefert die Speichergröße für Werte von Typ `T` in Einheiten vom Typ `char` (Bytes).

- Man kann `char` als kleine Zahlen verwenden, aber der Wertebereich kann `0..255` oder `-128..+127` sein.

  Wenn es wichtig ist, schreibe man "`signed char`" oder "`unsigned char`".

- C++ schreibt keine Zeichencodierung vor. Programme sollten mit jeder Codierung funktionieren (Betriebssystem-abhängig).

- Für internationale Zeichen gibt es den Typ `wchar_t`.

Typ-Definitionen

- Während in Java neue Typen nur als Klassen definiert werden können, kann man in C++ z.B. mit

  ```
  typedef int cent;
  ```

  den Typ `cent` als neuen Namen für `int` definieren.

  Der Wertebereich ist gleich, Zuweisungen sind in beiden Richtungen möglich, es ist also nur ein neuer Name. Da Typen auch komplizierter sein können als ein einzelner Basistyp (z.B. Zeiger, s.u.) kann es neben der besseren Dokumentation auch eine Abkürzung sein. Die Syntax ist wie die einer Variablendeklaration mit dem zusätzlichen Schlüsselwort `typedef`.

- Man kann dann Variablen dieses Typs deklarieren:

  ```
  cent preis;
  ```

  Manche Programmierer verwenden den Suffix "`_t`" für alle so definierten Typen (z.B. "`cent_t`"), um klarer zu machen, dass dieser Bezeichner für einen Typ steht (Stilfrage).

Aufzählungs-Typen

- Ein Aufzählungs-Typ ist ein Typ mit einer kleinen Anzahl von Werten, die explizit aufgezählt werden, z.B.

  ```
  enum wochentag { MONTAG, ..., SONNTAG };
  ```

  Dies deklariert den Datentyp `wochentag` und die Konstanten für die Werte.

  Man kann eine Variable dann z.B. deklarieren als "`wochentag w;`".

  Die Werte werden ohne Präfix für den Typ angesprochen, z.B. "`w = MONTAG;`".

- Während in Java die Elemente eines Aufzählungstyps Objekte sind, sind es in C++ ganze Zahlen.

  Wenn man nicht explizit Zahlwerte vorgibt, bekommt die erste Konstante den Wert 0 und jede folgende eins mehr als die vorhergehende Konstante.

- Eine Typ-Umwandlung nach `int` ist automatisch.

- Eine Typ-Umwandlung von `int` muss explizit verlangt werden, z.B.: `w = wochentag(w+1);`

Operator-Tabelle (sehr ähnlich Java)

18	`::`	Gültigkeitsbereich
17	`++` (Postfix), `.`, `->`, `[]`, `f()`, ...	Postfix-Operatoren
16	`-` (unär), `!`, `*` (deref), `++` (Präfix), ...	Präfix-Operatoren
15	`.*`, `->*`	Zeiger auf Komp.
14	`*`, `/`, `%`	Multiplikation etc.
13	`+`, `-`	Addition, Subtraktion
12	`<<`, `>>`	Shift etc.
11	`<`, `<=`, `>`, `>=`	kleiner etc.
10	`==`, `!=`	gleich, verschieden
9	`&`	Bit-und
8	`^`	Bit-xor
7	`|`	Bit-oder
6	`&&`	und
5	`||`	oder
4	`?:`	Bedingter Ausdruck
3	`=`, `+=`, `-=`, `*=`, `/=`, ...	Zuweisungen
2	`throw`	Exception auslösen
1	`,`	Sequenz

Schlüsselworte (1)

```
and          compl        export       namespace
and_eq       const        extern       new
asm          const_cast   false        not
auto         continue     float        not_eq
bitand       default      for          operator
bitor        delete       friend       or
bool         do           goto         or_eq
break        double       if           private
case         dynamic_cast inline       protected
catch        else         int          public
char         enum         long         register
class        explicit     mutable      ...
```

Schlüsselworte (2)

```
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while
xor
xor_eq
```

Inhalt

Arrays (1)

- Ein Array wird in der folgenden Form deklariert:

```
int arr[10];
```

- Man muss also gleich eine Größe wählen.

  Der Speicherplatz wird dann auch reserviert (ohne zusätzlichen Aufruf von `new`). Für lokale Variablen erfolgt die Speicherplatzreservierung auf dem Stack (Problem bei sehr großen Arrays). Bei Attributen wird das Array im Objekt angelegt (nicht als getrenntes Objekt im Heap wie bei Java).

- Die Klammern `[...]` müssen hinter der Variable stehen, nicht hinter dem Typ.

  Die Intuition ist, dass man auf der rechten Seite einen (einfachen) Ausdruck angibt, der den Typ auf der linken Seite liefern würde. "`int[10] arr;`" wäre ein Syntaxfehler. "`int i, arr[10];`" ist möglich.

- Das Array wird nicht automatisch initialisiert.

Arrays (2)

- Wie bei Java erstreckt sich der Indexbereich für ein Array der Größe n von 0 bis $n-1$.
- Es gibt keine automatische Prüfung der Arraygrenzen.

  Diese würde ja zusätzliche Laufzeit kosten. Wenn man das will, kann man eine entsprechende Klasse definieren (sogar mit dem Operator `[]`).
- Wenn man einen Indexwert außerhalb der Grenzen einsetzt, greift man auf Speicherstellen zu, die einer anderen Variablen entsprechen (oder z.B. eine Rücksprungadresse enthalten).

  Intern wird die Speicheradresse des i-ten Array-Elements berechnet als "Basisadresse des Arrays + $i *$ Speichergröße eines Array Elements".
  Wenn man negative oder zu große Werte für i einsetzt, landet man außerhalb des Speicherbereichs, der für das Array reserviert ist.
  Hacker nutzen das gerne, wenn der Programmierer nicht aufgepasst hat (sogenannte "Buffer Overflows" bei sehr großen Eingaben).

Zeiger/Pointer (1)

- In C++ kann man explizit mit Zeigern/Pointern arbeiten, also Variablen, die Hauptspeicheradressen enthalten.

  Auch Referenzen in Java sind Zeiger auf Objekte (intern also eine Hauptspeicheradresse). In C++ kann man aber die Adresse jeder Variablen bestimmen und mit Zeigern auch rechnen (z.B. innerhalb eines Arrays auf das nächste Element weiterschalten).

- Deklaration eines Zeigers `p` auf eine Variable vom Typ `int`:

```
int *p;
```

- Der `*`-Operator dient zur "Dereferenzierung", um also vom Zeiger wieder auf die Variable überzugehen, auf die der Zeiger zeigt.

  Wenn `p` ein "Zeiger auf `int`" ist, ist `*p` ein `int`. wieder passt die Intuition, dass man in der Deklaration rechts einen Ausdruck schreibt, der den Typ links liefert. Mehrere Zeiger deklariert man daher so: "`int *p, *q;`".

Zeiger/Pointer (2)

- Der Operator "&" dient zu Adressbestimmung (er ist also die Umkehrfunktion zu "*"):

```
int i = 3;
int *p;
p = &i;
```

- Nach Ausführung dieses Programmstücks zeigt `p` also auf die Variable `i`, die den Wert 3 enthält.
- Nach Ausführung von

```
*p = 5;
```

enthält `i` den Wert 5.

`*p` hat den Typ `int` und ist genauer gesagt ein "lvalue", kann also bei einer Zuweisung links stehen (Variable vom Typ `int` statt Wert vom Typ `int`). Der unäre Operator `*` hat sehr hohe Priorität, daher kann `*p` ohne Klammern in arithmetischen Ausdrücken benutzt werden, wo sonst `i` stehen würde.

Zeiger und Arrays (1)

- Der Name eines Arrays steht für seine Basisadresse:

```
int arr[10];
int *p;
p = arr;
```

- Anschließend zeigt `p` auf `arr[0]`.

  Man hätte also genauso gut "`p = &(arr[0]);`" schreiben können.
  In C/C++ gibt es also eine automatische Umwandlung von einem Array-Typ in den entsprechenden Zeigertyp. Der Unterschied zwischen einer Array-Deklaration und einer Zeiger-Deklaration ist nur, dass bei der Array-Deklaration Speicherplatz reserviert wird, auf das der Zeiger zeigt, und man diese Adresse nicht ändern kann (der Zeiger also konstant ist).

- Man kann eine ganze Zahl auf einen Zeiger addieren. Er zeigt dann entsprechend viele Array-Elemente weiter. D.h. `p+1` zeigt auf `arr[1]`.

Zeiger und Arrays (2)

- Man kann in C/C++ also mit Zeigern/Adressen rechnen.

  Intern wird nicht einfach die entsprechende Anzahl von Bytes auf die Adresse addiert, sondern dieser Wert noch mit der Größe des Basisdatentyps multipliziert. Im Beispiel ist `p + 1` die Adresse in `p` plus 4 Byte (bei 32 Bit `int`).

- Für einen Zeiger `p` auf einen Basis-Datentyp sind folgende Ausdrücke äquivalent:

  `p[i]` und `*(p+i)`

  Deswegen kann man sogar `i[p]` schreiben (ist aber schlechter Stil).

- Man kann zwei Zeiger (auf den gleichen Datentyp) von einander subtrahieren und bekommt dann den Abstand in Array-Elementen.

  D.h. die Differenz der entsprechenden Index-Werte. Die Differenz der Hauptspeicher-Adressen wird also durch die Größe des Basis-Datentyps geteilt.

Zeiger und Arrays (3)

- Eine häufige Operation ist das Inkrementieren eines Zeigers:

  ```
  p++;
  ```

  Dann zeigt `p` nicht mehr auf `arr[0]`, sondern auf `arr[1]`.

- Die folgende Schleife setzt jedes Array-Element auf `0`:

  ```
  p = arr;
  for(int i = 10; i-- > 0; )
      *p++ = 0;
  ```

  Damit wird die Multiplikation, die beim Array-Zugriff zur Adress-Berechnung nötig wäre, auf eine Addition zurückgeführt ($*4$ ist aber als Shift schnell).

- Die Sprache garantiert, dass ein Zeiger auf eine Position vor oder hinter einem Array (ohne Überlauf) möglich ist. Dereferenzieren darf man einen solchen Zeiger aber nicht.

- Vergleiche (`==`, `<`, ...) sind auch für Zeiger möglich.

Null-Zeiger

- Man kann die Zahl `0` an einen Zeiger zuweisen.

  Das geht nur für die Konstante `0`. Andere `int`-Ausdrücke benötigen eine explizite Typ-Umwandlung (selbst wenn sie zur Laufzeit 0 ergeben).

- Das Ergebnis ist ein Null-Zeiger, verschieden von jedem gültigen Zeiger auf eine Variable im Hauptspeicher.

  Eine explizite Typumwandlung wird manchmal empfohlen (leichter zu finden, falls 0 doch gültige Adresse). C++ 11 hat das Schlüsselwort `nullptr`.

- In Bedingungen ist die Null-Referenz falsch, alle anderen Zeiger-Werte `true`.

  Man braucht also keinen expliziten Vergleich mit dem Nullzeiger zu schreiben.

- Wenn `p` ein Null-Zeiger ist, und man greift auf `*p` zu, wird das Programm mit "segmentation fault" oder "access violation" abgebrochen (Laufzeitfehler).

Typ-Umwandlungen

- Typ-Umwandlungen sind in der (von C ererbten) Notation möglich, die auch in Java verwendet wird:

```
char *q = (char *) p;
```

- Man sollte aber die Art der Typ-Umwandlung angeben:
  - `static_cast<T>(p)`: Normale Typ-Umwandlungen.

    Der Compiler weiss, dass die Typen in Beziehung stehen.

    Z.B. `int`→`short`.
  - `dynamic_cast<T>(p)`: Typ-Umwandlung von Oberklasse zu Unterklasse mit Laufzeit-Prüfung (s.u.).
  - `reinterpret_cast<T>(p)`: Fragwürdige Umwandlungen.

    Dazu würde eine Umwandlung von `int *` nach `char *` gehören:

    Der Compiler lässt es zu, wenn es der Programmierer unbedingt will.
  - `const_cast<T>(p)`: Entfernung der `const`-Einschränkung.

Untypisierte Zeiger

- Es gibt in C/C++ den Typ `void *` für Zeiger, die auf keinen festgelegten Typ zeigen.

  Es gibt ja keine Variablen vom Typ `void`, insofern ist dieser Typ von jedem normalen Zeigertyp zu unterscheiden. Werte vom Typ `void *` sind also einfach Hauptspeicher-Adressen.

- Eine Umwandlung von einem beliebigen Zeiger-Typ in den Typ `void *` ist automatisch.

  Man darf also den Typ vergessen, auf den ein Zeiger zeigt, und einer Variablen vom Typ `void *` einen beliebigen Zeiger zuweisen.

- Die umgekehrte Richtung (von `void *` in einen konkreten Zeigertyp) geht nur mit einer expliziten Typ-Umwandlung.

Dynamische Speicheranforderung (1)

- Wenn man ein Array braucht, dessen Größe von Eingaben abhängt, muss man zur Laufzeit Speicher anfordern.
- Wenn man z.B. ein Array von `n` Zeichen braucht, kann man dies folgendermaßen anfordern:

```
char *arr = new char[n];
```

  Wie oben erklärt, kann man auch für Zeiger die []-Notation verwenden.

- Der Speicher muss später wieder freigegeben werden mit

```
delete[] arr;
```

- C++ hat keine automatische Garbage-Collection.

  Garbage-Collection ist aufwändig und führt zu Speicherfreigaben zu unbekannten Zeiten. In C++ ist der Programmierer dafür verantwortlich. Man braucht aber weniger dynamische Speicherverwaltung, weil Objekte auch als lokale Variablen auf dem Stack angelegt werden können (s.u.).

Dynamische Speicheranforderung (2)

- Wenn man Speicher dynamisch anfordert, aber nicht wieder freigibt, spricht man von einem “Speicherleck”.
- Wenn das Programm lange läuft (z.B. Server-Prozess) und dies immer wieder geschieht, wird das Programm erst immer langsamer (virtuelle Speicherverwaltung), und schließlich mit einer `bad_alloc` Exception beendet.

  Mit `set_new_handler` kann man entscheiden, was geschehen soll.
- In der Standard-Bibliothek `<cstdlib>` gibt es auch

  ```
  void *std::malloc(size_t size)
  ```

  zur Anforderung eines Speicherbereichs von `size` Bytes.

  Der Typ `size_t` ist ganzzahlig und groß genug, dass die mit `size(T)` berechnete Speichergröße jedes Typs `T` darin ausgedrückt werden kann.

  Im Fehlerfall liefert `malloc` einen Null-Zeiger (keine Exception).

  Mit `malloc` angeforderter Speicher ist mit `free(void *p)` wieder freizugeben.

Zeiger und const

- Man kann für jede Variable erklären, dass Zuweisungen (außer der Initialisierung) ausgeschlossen sind:

```
const int i = 10;
```

  In Java würde man statt `const` "`final`" schreiben.

- Für Zeiger muss man unterscheiden zwischen
  - Zeiger änderbar, aber auf "read-only" Speicherbereich:

```
const char *p;
```

    Eine Zuweisung an `p` ist möglich (z.B. `p++`), eine Zuweisung an `*p` nicht.
    Alternative Schreibweise (äquivalent): `char const *p;`

  - Zeiger fest, Zuweisung über Zeiger möglich:

```
char * const p = &c;
```

    In diesem Fall ist `p` nicht änderbar und muss deswegen auch gleich initialisiert werden. Eine Zuweisung an `*p` ist möglich.

Referenzen

- C hatte (wie Java) nur "Call by Value".

  Man kann natürlich Zeiger übergeben. In Java werden Objekte immer als Referenz übergeben, in C++ werden sie kopiert (wenn Parameter von Klassentyp).

- C++ hat auch Referenzen, d.h. implizite Zeiger, die immer automatisch dereferenziert werden.

  ```
  void set_zero(int &r) { r = 0; }
  ```

  Nun würde ein Aufruf wie `set_zero(n)` die Variable `n` ändern (auf 0 setzen). Stilistisch fragwürdig.

  Referenzen sind dafür gedacht, das Kopieren von Objekten bei der Parameterübergabe zu vermeiden, ohne jedes Mal "`&obj`" schreiben zu müssen. Das ist besonders für die Überladung eingebauter Operatoren für eigene Klassen nötig.

- Konstante Referenzen "`const MyClass &obj`" schließen eine Änderung des Objektes in der Methode aus.

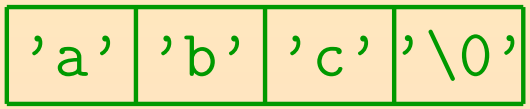
Inhalt

C Strings (1)

- Strings (Zeichenketten) werden in C als Arrays von Zeichen dargestellt, die mit einem Null-Zeichen abgeschlossen sind.

  Es gibt in C also keinen speziellen (zusätzlichen) Typ für Zeichenketten.

- `"abc"` ist im Speicher also als vier Bytes dargestellt:



- Mindestens für String-Konstanten (Datentyp-Literale) ist das auch in C++ so.

- Als Programmierer kann man sich entscheiden, Strings wie in C mit Zeichen-Arrays und Zeigern zu verarbeiten, oder mit der Bibliotheks-Klasse `string` (s.u.).

  Die Klasse verwendet dynamische Speicherverwaltung und ist vermutlich etwas langsamer. Dafür gibt es keine "Buffer Overflows".

C Strings (2)

- Folgende Zuweisung ist möglich:

```
const char *p = "abc";
```

  String-Konstanten können im ROM abgelegt sein, und der Compiler kann gleiche Strings nur einmal speichern. Daher ist wichtig, dass man die Zeichen einer String-Konstanten nicht ändert (Zeiger auf `const`).
  Mit "`typedef const char *str_t`" kann man einen Typnamen einführen, und anschließend einfach "`str_t p;`" schreiben.

- `p` zeigt nun auf das erste Zeichen des Strings (`'a'`) und kann mit `p++` weiter geschaltet werden.

- Da das Null-Zeichen als logisch falsch zählt, würde folgende Schleife alle Zeichen einzeln ausgeben:

```
for(const char *p = "abc"; *p; p++)
    cout << *p;
```

C Strings (3)

- Es ist eine häufige Quelle von Fehlern, dass man das zusätzlich nötige Array-Element für die Ende-Markierung `'\0'` vergisst.
- Wenn man z.B. Namen bis zur Länge 20 als C-String in ein Zeichenarray speichern will, muss das Array die Größe 21 haben.
- Selbstverständlich sollten solche Größen leicht änderbar und daher als Konstante definiert sein.
- Übliche Konvention ist, dass die Bytes hinter dem Null-Zeichen nicht abgefragt werden.

  Falls in das Array der Größe 21 also ein Name mit 4 Zeichen gespeichert wird, brauchen nur die ersten 5 Array-Elemente initialisiert zu sein. Im Rest des Arrays kann beliebiger Müll stehen.

C Strings (4)

- Bei Eingaben sind unbedingt die Array-Grenzen zu beachten.
- Die Eingabe von C-Strings mit >> ist unsicher:
  In C/C++ wird an eine Funktion nur die Anfangsadresse des Arrays übergeben (es gibt kein "`.length`").
  Der Operator >> weiß also nicht, wie lang das Array ist.
- Bei der Funktion `getline` kann man die Größe des Arrays mit übergeben, diese Variante ist sicher:

```
char input[80];
cin.getline(input, 80);
```

Im Unterschied zu >> werden Leerzeichen nicht übersprungen. Man kann als drittes Argument ein Zeichen übergeben, bei dem `getline` aufhören soll. Wenn man nichts angibt, ist das `'\n'`. Dieses Zeichen wird aus der Eingabe weggelesen, aber nicht in den String geschrieben.

Arrays und Zuweisung (1)

- Man kann Arrays (und damit Strings) nicht mit einem Befehl zuweisen oder vergleichen.

  In C sollte es in der Regel möglich sein, elementare Operationen der Sprache (wie Zuweisungen) durch einen Maschinenbefehl auszuführen. Wenn etwas viele Maschinenbefehle benötigt, sollte der Programmierer es auch merken (indem er eine Schleife programmieren muss etc.).

- Selbstverständlich kann ein Array/String kopiert werden, indem die Elemente einzeln zugewiesen werden.

- Für C-Strings gibt es eine Funktion `strcpy` (deklariert in <cstring>) zum Kopieren der Zeichen-Arrays:
  `strcpy(⟨Ziel⟩, ⟨Quelle⟩);`

  Aber man kann eben nicht direkt eine Zuweisung aufschreiben. Wenn man einen Funktionsaufruf macht, ist dem Programmierer bewußt, dass dies keine elementare Operation mehr ist. Vorsicht Buffer-Overflow!

Arrays und Zuweisung (2)

- Entsprechend muss man für den Test auf Gleichheit zwei Strings Zeichen für Zeichen in einer Schleife vergleichen.
- Es gibt dafür aber eine Bibliotheksfunktion `strcmp`.

  Sie liefert 0 (logisch falsch!), wenn die Zeichenketten gleich sind. Ein negativer Wert bedeutet, dass die erste Zeichenkette lexikographisch vor der zweiten kommt, bei positivem Wert ist es umgekehrt. (Allg.: Differenz der Zeichencodes an erster Position mit Unterschied.)
- In C++ wurde die Sprachphilosophie hinsichtlich elementarer Operationen geändert: Für Klassen kann man Operatoren wie `=` und `==` mit einer beliebigen Prozedur hinterlegen.

  Dann funktionieren Zuweisung/Vergleich, auch wenn diese Klassen komplexe Datenstrukturen inklusive Arrays enthalten. Manche Prozeduren werden sogar ganz ohne expliziten Aufruf ausgeführt (Konstruktoren, Destruktoren, Typumwandlungen).

Klasse `string` in C++ (1)

- Die Standardbibliothek von C++ enthält aber auch eine Klasse `string`, die den Umgang mit Strings vereinfacht (gegen einen gewissen Effizienzverlust).
- Um sie nutzen zu können, benötigt man

```
#include <string>
```

- Insbesondere kümmert sich diese Klasse automatisch um die Speicherverwaltung, fordert also intern ein hinreichend großes Array von Zeichen an.
- Damit funktioniert auch `>>` wieder sicher.
- Eine Variable "`s`" vom Typ `string` (ein Objekt dieser Klasse) kann man anlegen mit

```
string s;
```

Klasse `string` in C++ (2)

- Wie erwartet kann man einen String mit `>>` einlesen und mit `<<` ausgeben.
- Die aktuelle Länge des Strings `s` kann man abfragen mit
  `s.length()` oder äquivalent `s.size()`
- Auf einzelne Zeichen kann man zugreifen wie bei einem Array mit
  `s[i]`
- Wie bei einem Array werden die Index-Grenzen hier nicht geprüft. Will man das (sehr zu empfehlen, s.o.), muss man folgendes verwenden:
  `s.at(i)`

  Greift man außerhalb der Grenzen auf den String zu, wird eine Exception ausgelöst, die normalerweise das Programm beendet.

Klasse `string` in C++ (3)

- Die Zuweisung ist für Objekte der Klasse `string` definiert, man kann auch einen C-String zuweisen:

```
s = "abc";
```

- Zur Initialisierung verwendet man besser die Syntax

```
string s("abc");
```

- Entsprechend sind Vergleiche für `string`-Objekte möglich, auch Vergleiche mit C-Strings, z.B.

```
if(s == "abc") ...
```

  Auf mindestens einer Seite von == muss aber ein `string`-Objekt stehen (bei C-Strings auf beiden Seiten werden wie in Java die Adressen verglichen).

- Mit der `string`-Klasse kann man Strings auch konkatenieren (aneinanderhängen), z.B. `s += "def";`

Inhalt

Klassen und getrennte Übersetzung (1)

Datei "date.h":

```
class Date {
    public:
        Date(int d, int m, int y);
        int get_day()   { return day; }
        int get_month() { return month; }
        int get_year()  { return year; }
        int day_of_week(); // 1: Mon, ...
    private:
        int day;
        int month;
        int year;
};
```

Klassen und getrennte Übersetzung (2)

- Klassendeklarationen sehen ganz ähnlich wie in Java aus, aber:
  - Am Ende (hinter der "`}`") muss ein "`;`" stehen.
  - Die Schlüsselworte "`private`" und "`public`" werden von einem "`:`" gefolgt und erstrecken sich auf alle folgenden Komponenten der Klasse.

    Bis zu einer neuen Setzung des Zugriffsschutzes. In Java muss man den Zugriffsschutz dagegen für jede Komponente einzeln festlegen.
  - Der Default-Zugriffsschutz bei Klassen ist "`private`".

    In Java ist es dagegen "Package", was für kleine Programme ähnlich zu "`public`" ist. In C++ kann man statt "`class`" auch "`struct`" schreiben. Der einzige Unterschied ist, dass der Default dann "`public`" ist.

Klassen und getrennte Übersetzung (3)

- Wie in Java bestehen größere Programme meist aus vielen Klassen, die einzeln übersetzt werden.
- Aber ein C++-Compiler übersetzt direkt in Maschinensprache, und das Dateiformat für Objektdateien (`.o`) ist recht alt und nicht speziell für C++ gemacht (enthält keine Typ-Infos).
- Daher teilt man die Definition einer Klasse `C` meist in zwei Quelldateien auf:
  - `c.h`: Deklaration der Klasse (Informationen, die der Compiler braucht, eine Verwendung der Klasse zu übersetzen).

    Typischerweise findet sich hier nur der Kopf der Methoden.
  - `c.cpp`: Definition der Methoden der Klasse (die der Compiler nur ein Mal übersetzen muss, nicht für jede Benutzung).

    Hier wird für die Methoden auch der Rumpf angegeben.

Klassen und getrennte Übersetzung (4)

- Der Compiler muss die `.h`-Datei ("Header-Datei") sehen, bevor man die in ihr deklarierte Klasse verwenden kann.

  Sonst ist der Name der Klasse ein undefinierter Bezeichner.

- Deswegen enthält jede Quelldatei, die die Klasse `C` verwendet, oben den Befehl:

```
#include "c.h"
```

  Würde man `#include <c.h>` schreiben, würde nur in Bibliotheksverzeichnissen gesucht. Für eigene Header-Dateien ist also `"..."` wichtig.
  Man kann einen Suchpfad für Include-Dateien setzen, bei g++ mit `-I`.

- Bei wechselseitigen Referenzen kann man eine "forward" Deklaration verwenden:

```
class C;
```

  Das sagt dem Compiler nur, dass es so eine Klasse gibt. Man kann dann Zeiger of Objekte der Klasse deklarieren.

Klassen und getrennte Übersetzung (5)

- Die Klassendeklaration in der `.h`-Datei enthält:
  - Die Köpfe der Methoden:
    Wichtig für Typ-Prüfung (ggf. Umwandlung) beim Aufruf.
  - Die Deklaration der Attribute:
    Wichtig für die Bestimmung der Speichergröße.

    Man kann Variablen für Objekte anlegen oder Objekte mit `new` erzeugen.
  - Außerdem die Deklaration statischer Komponenten (s.u.).
- Man kann für kurze, häufig aufgerufene Methoden auch die Rumpf in die Klassendeklaration schreiben:
  - Dann hat der Compiler die Möglichkeit, anstelle eines Methodenaufrufs den Rumpf einzufügen.

    Wie die Expansion eines Makros. Man spart so den Overhead für den Methodenaufruf. "`inline`" (vor Rückgabetyp) macht dies noch klarer.

Klassen und getrennte Übersetzung (6)

Datei "date.cpp":

```
#include "date.h"

// Konstruktor:
Date::Date(int d, int m, int y) {
        day   = d; // oder: this->day = d;
        month = m;
        year  = y;
}

int Date::day_of_week() {
        ...
}
```

Klassen und getrennte Übersetzung (7)

- Wenn man die Methoden (mit Rumpf) außerhalb der Klasse definiert, muss man mit dem Präfix "`Date::`" klar machen, dass die Methode zu der Klasse "`Date`" gehört.

  Verschiedene Klassen können ja Methoden gleichen Namens haben. Der Name "`date.cpp`" der Quelldatei bedeutet für den Compiler nichts (es ist nur Konvention, sie so zu nennen). Man hätte die Datei auch "`xyz.cpp`" nennen können, und darin Methoden der Klasse "`Date`" definieren.

- Natürlich muss die Klassendefinition bekannt sein, wenn man Rümpfe von Methoden der Klasse schreibt. Deswegen der `#include`-Befehl oben.

  Man würde z.B. eine Fehlermeldung bekommen, wenn Anzahl oder Typen der Parameter einer Methode nicht übereinstimmen. Selbstverständlich dürfen Methoden der Klasse auf `private`-Komponenten zugreifen (wie in Java von allen Objekten der Klasse, nicht nur dem aktuellen).

Verwendung von Objekten (1)

Datei “main.cpp”:

```
#include <iostream>
using namespace std;
#include "date.h"

int main() {
        Date d(27, 10, 2015);
        cout << d.get_year() << "\n";

        Date *p = new Date(28, 10, 2015);
        cout << p->day_of_week() << "\n";
        delete p;

        return 0;
}
```

Verwendung von Objekten (2)

- Objekte können angelegt werden (u.a.) als
  - Lokale Variablen:
    Hier steht der Name der Variable für das Objekt selbst.

    D.h. der Speicherplatz für das Objekt wird auf dem Stack reserviert.
    Das Objekt wird automatisch freigegeben (und der Destruktor aufgerufen, s.u.), sobald sich die Methode beendet.
  - Im Heap mittels `new`:
    Die lokale Variable enthält dann nur den Pointer.

    In diesem Fall ist man selbst dafür verantwortlich, dass der Speicherplatz wieder freigegeben wird (mittels `delete`). Auch dann wird ein Destruktor ausgeführt (s.u.).
    Selbstverständlich holt sich das Betriebssystem allen Speicherplatz zurück, wenn der Prozess endet (sich das Programm beendet).
    Dann werden allerdings keine Destruktoren mehr ausgeführt.
    Java hat nur die Lösung mit `new`, der Pointer ist dann implizit.

Verwendung von Objekten (3)

- Wenn man ein Objekt hat, greift man auf Komponenten mit "`.`" zu (wie in Java).
- Wenn man einen Zeiger auf ein Objekt hat, greift man auf Komponenten mit "`->`" zu.

  Tatsächlich ist "`p->a`" nur eine Abkürzung für "`(*p).a`".
- In C++ arbeitet man häufig mit Zeigern auf Objekte.

  Wie in Java implizit auch. Man möchte ja nicht das ganze Objekt kopieren, wenn man es z.B. an eine Methode übergibt. In C++ wäre dieses Kopieren allerdings möglich, lohnt sich aber nur für ganz kleine Objekte.
- Natürlich kann man von "`main`" aus nicht auf private Komponenten der Klasse "`Date`" zugreifen, da "`main`" nicht Teil der Klasse "`Date`" ist.

  "`main`" ist eine globale Funktion (steht außerhalb von Klassen).

Konstruktoren

- Die Sprache C++ garantiert, dass für jedes Objekt ein Konstruktor aufgerufen wird.

  Der Programmierer ist selbst verantwortlich dafür, dass der Konstruktor dann auch Attribute von einfachem Typ wie `int` initialisiert. Für objekt-wertige Attribute ist dagegen wieder garantiert, dass ein Konstruktor aufgerufen wird. Bei Arrays von Objekten wird ein Konstruktor (ohne Argumente) aufgerufen für jede Komponente des Arrays.

- Wie in Java kann eine Klasse mehrere Konstruktoren haben, die sich in Anzahl oder Typ der Argumente unterscheiden.

  Wie in Java wird automatisch ein leerer Default-Konstruktor (ohne Argumente) angelegt, wenn kein Konstruktor für eine Klasse definiert ist. Sobald mindestens ein Konstruktor deklariert ist, geschieht dies nicht mehr.

- Die Deklaration "`Date d;`" ist nur erlaubt, wenn es einen Konstruktor ohne Argumente gibt.

Komponenten-Objekte (1)

- Beispiel: Klasse `Person` hat Attribut vom Typ `Date` (Geburtsdatum):

```
class Person {
        char fname[20]; // first/Christian name
        char lname[20]; // last/family name
        Date birthdate;
public:
        Person(const char *fn, const char *ln,
                        int d, int m, int y):
                birthdate(d, m, y)
        { ... } // copy fn, ln to fname, lname

        void print() {
                cout << fname << " " << lname;
        }
};
```

Komponenten-Objekte (2)

- Objekte können auch als Komponenten von anderen Objekten angelegt werden.
- Dann wird der Speicherplatz für das `Date`-Objekt innerhalb des `Person`-Objektes reserviert.

  So wie ein `int`-Attribut z.B. 4 Byte groß ist, ist ein `Date`-Attribut eben 12 Byte groß (weil es drei `int`-Komponenten hat). Man kann die Größen mit dem `sizeof`-Operator abfragen, z.B. "`cout << sizeof(Date);`".
  In Java wäre das `Date`-Objekt dagegen getrennt vom `Person`-Objekt gespeichert, dies würde nur einen Zeiger enthalten.
- Für den Konstruktor-Aufruf von Komponenten schreibt man nach der Parameterliste "`:`" und dann die Namen der Komponenten mit Konstruktor-Argumentwerten.

  Bei mehreren Komponenten trennt man diese Initialisierungen mit "`,`". Falls gewünscht, kann man auch Attribute von einfachem Typ so initialisieren.

Destruktoren (1)

- Wenn ein Objekt gelöscht wird, z.B. weil der Block verlassen wird, in dem es als lokale Variable angelegt wurde, ruft C++ automatisch eine Destruktor-Methode auf.

  Wenn man keine deklariert, legt der Compiler eine an, die nichts tut (außer ggf. Destruktoren von Komponenten aufzurufen). Im Gegensatz zu `finalize()` in Java ist in C++ klar definiert, wann der Destruktor aufgerufen wird.

- Destruktoren sind dadurch gekennzeichnet, dass ihr Name der Klassenname mit vorangestellter Tilde "~" ist.

  Sie sind ja das Komplement des Konstrukors. Wie bei Konstruktoren gibt es keinen Ergebnistyp.

- Falls das Objekt Resourcen belegt hat (z.B. eine Datei geöffnet hat), muss der Destruktor diese Resourcen wieder freigeben (die Datei schließen).

Destruktoren (2)

- Wenn das Objekt z.B. eine kompliziertere Datenstruktur verwaltet (verkettete Liste etc.), sollte der Destruktor die komplette Datenstruktur löschen.

  Hier ist die Resource der dynamisch angeforderte Speicher für die Listenknoten.

- Falls Objekte wechselseitig verzeigert sind, könnte ein Destruktoraufruf den Zeiger auf der anderen Seite löschen (auf 0 setzen).

- Im Zusammenhang mit Assertions (Zusicherungen, s.u.) kann man Destruktoren verwenden, um ein gelöschtes Objekt unbrauchbar zu machen.

  Es könnte ja möglicherweise noch Zeiger auf das Objekt geben, und falls der Compiler bzw. das Laufzeitsystem den Speicherplatz nicht gleich neu verwendet, würden Methodenaufrufe über diese Zeiger zufällig noch funktionieren. Der Fehler sollte aber sicher und früh bemerkt werden.

const-Methoden

- Man kann explizit angeben, dass eine Methode den Zustand des Objektes nicht verändert:

```
class Date {
    public:
        Date(int d, int m, int y);
        int get_day()   const { return day; }
        int get_month() const { return month; }
        int get_year()  const { return year; }
        int day_of_week() const;
    private:
        ...
};
```

Der `this`-Zeiger hat dann den Typ "`const Date *`".

- Wenn man `day_of_week` später definiert, muss `const` noch einmal mit angegeben werden.

Statische Komponenten (1)

- `static` funktioniert in C++ wie in Java.

  Statische Attribute existieren nur ein Mal für die ganze Klasse, statt ein Mal pro Objekt. Statische Methoden werden ohne Objekt aufgerufen (außerhalb der Klasse mit der Syntax "`Klasse::Methode(...)`") und haben keinen Zugriff auf ein aktuelles Objekt (kein "`this`").

- Wenn man den Rumpf einer statischen Methode außerhalb der Klasse in der "cpp"-Datei definiert, darf man das Schlüsselwort "`static`" nicht nochmals schreiben.

  Vermutlich weil dies in C (und in C++ für globale Funktionen) die Bedeutung hat, dass die Funktion nur innerhalb der einen Quelldatei bekannt ist.

- Statische Attribute können (wie auch normale Attribute) nicht in der Klassendeklaration initialisiert werden (siehe nächste Folie).

  Normale Attribute können ausschließlich im Konstruktor initialisiert werden.

Statische Komponenten (2)

- Statische Attribute müssen außerhalb der Klasse nochmals definiert werden (dort ggf. Initialisierung):

```
int MyClass::MyStaticVar = 0;
```

Die Deklaration in der Klasse sagt dem Compiler nur, dass es so eine Variable gibt. Diese Definition legt nun die Variable wirklich an (reserviert Speicherplatz). Die statische Variable soll es ja nur ein Mal geben (in der einen Objekt-Datei), nicht in jeder Objekt-Datei, bei deren erzeugung die Klassendeklaration mittels "`#include`" gesehen wurde. Wenn der Compiler die Klassendeklaration sieht, weiss er nicht, ob er gerade die Objektdatei für die Klasse oder für ein anderes Modul erzeugt, das diese Klasse nur verwendet.

- Konstanten werden dagegen nur in der Klasse definiert:

```
static const int MAX_NAME_SIZE = 20;
```

Hier wird der Wert bei jeder Verwendung eincompiliert. Es wird kein Speicherplatz für eine Variable gebraucht. Daher "Initialisierung" in der Klassendeklaration und keine zusätzliche Definition in cpp-Datei.

Implizite Methoden (1)

- Der Compiler definiert einige Methoden automatisch, wenn man dies nicht selbst macht:

```
class C { int n; };
```

ist äquivalent zu

```
class C {
private:
      int n;
public:
      C() {};         // Default Konstruktor
      C(const C& x) // Copy Konstruktor
            { n = x.n; }
      ~C() {}; // Destruktor
      C& operator=(const C& x) // Zuweisung
};          { n = x.n; }
```

Implizite Methoden (2)

- Manchmal soll es von einer Klasse keine Objekte geben (nur statische Methoden und Variablen).
- Ein Trick, die Erzeugung von Objekten zu verhindern, ist, eine Konstruktor-Methode (z.B. ohne Parameter) zu deklarieren, aber als "`private:`".

  Ein Aufruf von außen würde dann zu einer Fehlermeldung führen.

  Ohne Objekte sind auch Copy-Konstruktor und Zuweisung nicht anwendbar.
- Außerdem gibt man keinen Rumpf (Implementierung) für den Konstruktor an (nur Deklaration).

  Dann würde ein Aufruf aus Methoden der Klasse selbst zu einem Fehler beim Linken führen.
- Ebenso kann man Copy-Konstruktor/Zuweisung ausschliessen.

  Wenn die Objekte Resourcen wie dynamisch angeforderten Speicher verwalten.

Geschachteltes Include (1)

- Oft benötigt eine Klasse andere Klassen, Konstanten oder Datentypen.
- Z.B. kann `Person.h` mit der Deklaration der Klasse `Person` nur übersetzt werden, wenn der Compiler die Deklaration der verwendeten Klasse `Date` schon gesehen hat.
- Daher sollte man oben in die Datei `Person.h` den Befehl

  ```
  #include "Date.h"
  ```

  schreiben.

  Nun muss der Benutzer der Klasse `Person` nicht mehr daran denken, vor `Person.h` auch `Date.h` einzulesen — das geht automatisch.
- Allerdings kann es dann dazu kommen, dass eine `.h`-Datei mehrfach bei einem Compilerlauf eingelesen wird.

  Z.B. könnte das Programm auch eine Datei `Calendar.h` einlesen, die ihrerseits auch `Date.h` einbindet.

Geschachteltes Include (2)

- Bei reinen Deklarationen schadet es nichts, wenn sie mehrfach gelesen werden, aber bei Definitionen von Klassen oder Konstanten ist es ein Fehler.
- Mit folgendem Rahmen aus Präprozessor-Befehlen kann man die doppelte Verarbeitung einer Include-Datei vermeiden:
  - Zu Anfang der Datei `Date.h`:

```
#ifndef DATE_INCLUDED
#define DATE_INCLUDED
```

  - Ganz unten: `#endif`

    Falls `DATE_INCLUDED` noch nicht definiert ist, dann definiere es, und lies den Rest der Datei. Sonst wird alles bis zum `#endif` übersprungen (bedingte Compilierung). Der Rahmen lässt den eigentlichen Inhalt der Datei also nur einmal durch. Wie man das leere Makro `DATE_INCLUDED` nennt, ist egal, es darf nur sonst nicht benutzt sein (z.B. auch: `DATE_H`).

Include-Optimierung

- Wenn eine Klasse `X` nur Zeiger auf Objekte einer Klasse `Y` enthält, muss die Datei `X.h` nicht unbedingt `Y.h` includen.

  Für die Anwender der Klasse `X` (die `Y` selbst nicht brauchen), geht das Compilieren etwas schneller (besonders, wenn `Y.h` selbst vieles included).

- Es reicht in diesem Fall eine Deklaration der Klasse in `X.h`:

```
class Y;
```

- Für Zeiger ist die Größe der Objekte der Klasse `Y` unwichtig.

  Natürlich kann man dann auch nicht direkt in der Klassendeklaration von `X` in `X.h` Methodenrümpfe angeben, die Methoden von `Y` aufrufen. Größere Methodenrümpfe sollten aber ohnehin eher in `X.cpp` stehen.

- Die Methoden der Klasse `X` werden natürlich Methoden der Klasse `Y` aufrufen, daher braucht man die Include-Anweisung dann spätestens in `X.cpp`.

Compilierung (1)

- Bei kleinen Programmen aus wenigen Quelldateien kann man sie jedes Mal vollständig übersetzen:

  ```
  g++ date.cpp main.cpp -o prog
  ```

  Man ruft den Compiler nur für die `.cpp`-Dateien auf. Die `.h`-Dateien werden dabei verwendet (durch die `#include`-Befehle). Für die Klassen-Deklarationen selbst wird aber kein Code erzeugt, nur für Benutzungen der Klasse.

- Bei größeren Programmen übersetzt man die Quelldateien einzeln (für schnellere Compilierung bei Änderungen, s.u.):
  - `g++ -c date.cpp` → `date.o`
  - `g++ -c main.cpp` → `main.o`
  - `g++ date.o main.o -o prog` → `prog`

    Dieser letzte Befehl compiliert nichts, sondern ruft den Linker auf: Er bindet die Objektdateien `date.o` und `main.o` (und ggf. Bibliotheken) zusammen zu `prog` (ausführbares Programm).

Compilierung (2)

- Wenn `main.cpp` geändert wird, muss nur dies neu übersetzt werden. Entsprechend für `date.cpp`.

  Anschließend muss natürlich immer der Linker aufgerufen werden, aber dieser arbeitet schnell (Linken ist viel einfacher als Compilieren).

- Wenn dagegen `date.h` geändert wird, müssen sowohl `date.cpp` als auch `main.cpp` neu übersetzt werden.

  Beide enthalten ja einen `#include`-Befehl für diese Datei, sind also von einer Änderung potentiell betroffen. Vergisst man eine Neucompilierung, ist die Typsicherheit nicht gegeben (beliebige Fehler/Abstürze möglich).

- Eine Entwicklungsumgebung wie Visual Studio übersetzt automatisch nur die jeweils nötigen Quelldateien neu.

  Zur Sicherheit sollte man gelegentlich "`Clean ...`" im "`Build`"-Menu auswählen, um alle Objektdateien zu löschen und anschließend eine vollständig neue Übersetzung zu machen.

Make (1)

- Unter Linux verwendet man `make` mit einem `Makefile`:

```
# Beispiel fuer Makefile mit Makros.
PROG    = myprog
OBJECTS = main.o date.o

all: $(PROG)
main.o: main.cpp date.h
        g++ -c main.cpp
date.o: date.cpp date.h
        g++ -c date.cpp
$(PROG): $(OBJECTS)
        g++ -o $(PROG) $(OBJECTS)
clean:
        rm -f $(PROG) $(OBJECTS)
```

Make (2)

- Ein `Makefile` besteht aus einer Liste von Abhängigkeitsregeln und Makro-Definitionen.
- Eine Regel besteht aus
  - Ziel A,

    Häufig ist dies eine Datei, die erstellt werden soll. Es kann aber auch einfach ein Name für eine Aktion sein, die man durchführen will.

  - Dateien $B_1, \ldots, B_n$, von denen A abhängt (Subziele),
  - Kommandos $C_1, \ldots, C_k$ zur Erstellung von A.

$$A: \; B_1 \; B_2 \; \ldots \; B_n$$
$$C_1$$
$$\vdots$$
$$C_k$$

Make (3)

Syntaktische Feinheiten:

- Das Ziel muss in der ersten Spalte beginnen.

  Es sind keine Leerzeichen davor erlaubt.

- Die Liste der Dateien, von denen das Ziel abhängig ist, erstreckt sich nur bis zum Zeilenende.

  Wenn man sie in der nächsten Zeile fortsetzen will, muss man die aktuelle Zeile mit "\" beenden.

- Die Kommandos müssen mit ein oder mehreren Tabulator-Zeichen eingerückt werden.

  Alternativ: Kennzeichnung der Kommandos durch vorangestelltes ";".
  Stellt man dem Kommando ein "@" voran, wird es bei der Ausführung nicht ausgegeben (man kann Ausgaben dann explizit mit dem `echo`-Befehl von Linux durchführen). Jedes Kommando wird in einer eigenen Shell ausgeführt. Ein `cd` wirkt also nicht zu nächsten Zeile (ggf. mit \ fortsetzen).

Make (4)

- Um das Ziel A zu erstellen, erstellt `make` zunächst rekursiv die Ziele $B_1, \ldots, B_n$.
- Dann prüft `make`, ob es die Datei A gibt:
  - Falls nein: Die Kommandos $C_1, \ldots, C_k$ werden ausgeführt, um A zu erstellen.
  - Falls ja, werden Datum und Uhrzeit der letzten Änderung von $B_1, \ldots, B_n$ mit dem entsprechenden Zeitstempel von A verglichen. Wurde ein B_i nach A geändert, werden die Kommandos $C_1, \ldots, C_k$ ausgeführt.

    Sonst (A existiert und ist jünger als alle B_i) ist A also aktuell.

    Die Kommandos $C_1, \ldots, C_k$ werden dann nicht ausgeführt.
- Der Aufruf "`make`" erzeugt das erste Ziel im `Makefile`.

  Ansonsten muss man ein Ziel explizit angeben, z.B. "`make clean`".

Inhalt

Subklassen (1)

- Beispiel: Subklasse `Student` von `Person` hat ein zusätzliches Attribut "`matrnr`" und überschreibt die Methode "`print`":

```
class Student : public Person {
        long matrikelnr;
public:
        Student(const char *fn, const char *ln,
                int d, int m, int y, long nr):
                Person(fn, ln, d, m, y)
        {
                matrikelnr = nr;
        }
        void print() {
                Person::print(); // like "super."
                cout << ": " << matrnr;
        }
};
```

Subklassen (2)

- Wie in Java erben Subklassen Attribute und Methoden von der Oberklasse.

  In C++ kann eine Subklasse aber mehrere Oberklassen haben ("multiple Inheritance"). Sie erbt dann Attribute und Methoden aller Oberklassen. Dafür hat C++ keine Interfaces.

- Statt "`extends`" schreibt man "`: public`" zwischen Unterklasse und Oberklasse.

  Wenn man "`private`" schreibt, werden alle ererbten Attribute und Methoden `private`. Dann dient die Oberklasse nur zur Implementierung der Unterklasse, das Substitutionsprinzip würde nicht mehr gelten. Das ist exotisch.

- Wie in Java muss ein Konstruktor der Unterklasse einen Konstruktor der Oberklasse aufrufen, wenn die Oberklasse keinen parameterlosen Konstruktor hat.

  Die Syntax ist etwas anders: So wie bei den Komponentenobjekten.

Subklassen (3)

- Wie in Java kann Programmcode aus der Unterklasse nicht auf `private` Attribute/Methoden der Oberklasse zugreifen.

  Im Beispiel kann die Methode `print` nicht direkt `fname` und `lname` ausgeben. Sie muss dafür die Methode `print` aus der Oberklasse aufrufen. Das geht nicht mit `super` wie in Java, sondern indem man die gewünschte Version mit "`Klasse::`" eindeutig macht.

- Wie in Java kennzeichnet "`protected`" Attribute/Methoden, die von Unterklassen aus zugreifbar sein sollen, aber nicht von fremden Klassen aus.

- In C++ gibt es keine Klasse `Object` als Wurzel der Vererbungshierarchie.

  Wenn eine Klasse nicht explizit als Subklasse deklariert ist, hat sie keine Oberklasse. Wenn man will, kann man sich natürlich selbst eine Oberklasse für alle eigenen Klassen deklarieren.

Subklassen (4)

- Wie in Java ist der "Upcast" automatisch (Umwandlung von Unterklasse in Oberklasse):

```
Student *s = new Student("Lisa", "Weiss",
                         6, 3, 1995, 1234);
Person  *p = s;
p->print();
```

  Man kann in C++ auch Objekte zuweisen, nicht nur Zeiger auf Objekte. Dann wird der `Student`-spezifische Teil weggeschnitten ("slicing").

- Im Gegensatz zu Java wird hier die Implementierung von `print` aus der Klasse `Person` aufgerufen.

  Der Compiler weiss nicht mehr, dass es sich eigentlich um ein `Student`-Objekt handelt. Teilweise ist das auch, was man will, da man den Studenten hier als Person behandelt. Andernfalls: Siehe nächste Folie.

Subklassen (5)

- Wenn man eine Methode in der obersten Klasse, in der sie definiert ist, als "`virtual`" kennzeichnet, findet die dynamische Bindung wie in Java statt:

```
class Person {
        public:
                virtual void print() { ... }
```

  Dann wird im Objekt ein Zeiger auf die aufzurufende Methode gespeichert (meist indirekt über eine "virtual function table"). Man zahlt also einen Preis: Das Objekt wird etwas größer und der Methodenaufruf dauert etwas länger.

- Wenn man in der Subklasse die Methode `print` überschreibt, wird dort das Schlüsselwort `virtual` nicht erneut angegeben.
- Ruft man nun die Methode `print` für ein `Student`-Objekt auf, auch über einen `Person`-Zeiger, wird die Implementierung aus `Student` verwendet.

Subklassen (6)

- Virtuelle Methoden heißen auch “polymorph”.

  Polymorphe Klasse: Klasse mit mindestens einer virtuellen Methode.

- Falls man ein Objekt mit `delete` über einen Zeiger auf eine Oberklasse löscht, muss der Destruktor `virtual` sein.

- Wie in Java: Downcast nur mit expliziter Typumwandlung:

```
Student *x = static_cast<Student *>(p);
```

  In diesem Beispiel ist die Variable `p` deklariert als `Person*` (statischer Typ, den der Compiler kennt). Zur Laufzeit muss die Variable natürlich einen Zeiger auf ein Objekt der Unterklasse `Student` enthalten (dynamischer Typ).

- Im Gegensatz zu Java findet keine Prüfung zur Laufzeit statt.

  Wenn man auf die Makrikelnummer zugreift, aber das Objekt ist tatsächlich nur eine `Person`, wirkt das wie ein Zugriff auf ein Array außerhalb des Indexbereiches: Man bekommt den Inhalt der Speicherstelle, die sich rechnerisch ergeben würde. Besonders übel bei schreibenden Zugriffen.

Subklassen (7)

- Wenn `Person` eine polymorphe Klasse ist, kann man auch einen Downcast mit Prüfung machen:

```
Student *x = dynamic_cast<Student *>(p);
```

- Falls `p` auf ein `Student`-Objekt zeigt, wird es `x` zugewiesen. (Ebenso, wenn `p` auf eine Subklasse von `Student` zeigt.) Ansonsten wird `x` auf einen Nullzeiger gesetzt.

  Es gibt keine Exception. Dieses Konstrukt ist auch für Fälle gedacht, in denen man in Java mit `instanceof` den dynamischen Typ von `p` abfragen würde. In C++ gibt es kein `instanceof`.

- Abstrakte Methoden werden so deklariert:

```
virtual int my_abstr_meth(int n) = 0;
```

  Es gibt kein Schlüsselwort "`abstract`". Eine Klasse, die mindestens eine solche "pure virtual function" enthält, ist automatisch abstrakt.

Inhalt

Definition von Templates (1)

- Definition des Templates für einfache Listenknoten:

```
template<class T> class List {
    private:
        T elem;
        List* next;
    public:
        List(T e) { elem = e; next = 0; }
        void set_next(List* l) { next = l; }
        T get_elem() { return elem; }
        List* get_next() { return next; }
};
```

Syntaktisch besteht ein Klassentemplate aus einer normalen Klassendeklaration, der das Schlüsselwort "`template`" und dann in spitzen Klammern die Parameter vorangestellt sind.

Definition von Templates (2)

- Typparameter werden in der Parameterliste durch das Schlüsselwort "`class`" gekennzeichnet.

  Man kann dafür aber später beliebige Typen einsetzen, nicht nur Klassen. Z.B. auch den Typ `int`, was in Java nicht möglich wäre. Insofern passt das Schlüsselwort nicht ganz. Inzwischen kann man alternativ "`typename`" schreiben (kein semantischer Unterschied, eventuell Probleme bei älteren Compilern).

- In der Klassendeklaration kann der Typparameter `T` wie ein normaler Typ verwendet werden.

- Im Innern der Klassendeklaration ist es egal, ob man "`List`" oder "`List<T>`" schreibt.

  Einzige Ausnahme ist die Definition der Template-Klasse selbst.
  In "`template<class T> class List {`" darf man nur "`List`" schreiben.
  Dort definiert man ja gerade das Template.

Definition von Templates (3)

- Falls man in der Klassendeklaration den Rumpf für eine Funktion nicht angegeben hat, muss man die Definition über ein weiteres Template nachholen:

```
template<class T>
    void List<T>::set_next(List* l)
        { next = l; }
```

  In der Festlegung der Klasse (links vom `::`) bei der nachträglichen Definition einer Funktion ist der Parameter nötig (man muss `List<T>` schreiben).

- Der Compiler muss allerdings wissen, für welche konkreten Element-Typen `T` Code erzeugt werden soll. Daher steht häufig die ganze Template-Definition in einer `.h`-Datei.

  Wenn man (wie im obigen Beispiel) einen Teil der Template-Definition in eine cpp-Datei schreibt, muss man hier explizit angeben, welche Instanzen man braucht, z.B. "`template class List<int>;`".

Template-Instanziierung (1)

- Man instanziiert das Template, indem man einen konkreten Typ für den Typ-Parameter einsetzt:

```
int main()
{   List<int>* tmp = new List<int>(3);
    List<int>* first = tmp;
    List<int>* last = tmp;
    tmp = new List<int>(5); // Element anhängen
    last->set_next(tmp);
    last = tmp;
    for(List<int>*p = first; p; p=p->get_next())
        cout << p->get_elem() << "\n";
    return 0;
}
```

Wenn man für einen Template-Parameter einen Typ einsetzt, der selbst durch Instanziierung eines Templates entsteht, muss man "> >" schreiben, nicht >>.

Template-Instanziierung (2)

- In der Template-Deklaration können für Werte des Typ-Parameters zunächst beliebige Operatoren, Methoden, Funktionen benutzt werden.
- Bei der Instanziierung kann es dann zu einem Fehler kommen, wenn ein Typ eingesetzt wird, der diese Funktionen etc. nicht hat.

  Es ist eine Schwäche von C++, dass man Anforderungen an Template-Parameter in der Template-Deklaration nicht explizit angeben kann.
- Man kann sich den Template-Mechanismus also wie eine Art Macro vorstellen.

  Bei der Template-Definition wird im wesentlichen nur der Text abgespeichert. Bei der Instanziierung werden die Parameter ausgefüllt und erst dann findet die eigentliche Compilierung und Prüfung statt.

Template-Instanziierung (3)

- Da es etwas mühsam ist, den Parameter immer explizit anzugeben, kann man mit `typedef` einen Namen für diese Instanz der Templateklasse einführen:

```
typedef List<int> intlist;
```

`typedef` erzeugt in C++ keinen neuen Typ (nur Abkürzung).

- Wenn man Deklaration des Templates `list.h` und Definition längerer Methoden `list.cpp` trennen will, kann man für jede nötige Instanziierung, z.B. `T = int`,
  - eine Datei "`intlist.h`" anlegen mit zwei Zeilen:

```
#include "list.h"
typedef List<int> intlist;
```

  - außerdem eine Datei "`intlist.cpp`":

```
#include "list.cpp"
template class List<int>;
```

Templates: Ausblick

- Es sind nicht nur Typ-Parameter möglich, sondern z.B. auch ganze Zahlen (Konstanten) und Adressen von globalen Objekten und Funktionen.

```
template<class T, int Max> class Stack {
        T elems[Max];
        int num_elems;
        ...
```

- Templates sind nicht nur für Klassen möglich, sondern auch für Funktionen:

```
template<class T> T maximum(T n, T m)
    { if(n > m) return n; else return m; }
```

Während man bei der Verwendung von Klassen-Templates Werte für die Parameter explizit angeben muss, bestimmt der Compiler bei Funktionstemplates wie `max` den Parameter `T` aus dem Aufruf (soweit als möglich, ggf. angeben).