Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 2: Programme mit Zuweisungen und Kontrollstrukturen

- Deklarationen, Variablen, Zuweisungen
- Basis-Datentypen
- Bedingungen
- Schleifen
- Fehlersuche- und vermeidung

Inhalt

1. Variablen, Datentypen
2. Programme mit Ein-/Ausgaben, Zuweisungen
3. Bedingungen
4. Schleifen
5. Fehlersuche und -vermeidung (gdb)

Variablen: Motivation (1)

- Ein Programm soll normalerweise nicht nur eine feste Ausgabe machen, oder nur eine einzige feste Berechnung durchfuehren.
- Stattdessen soll es für viele verschiedene Eingabewerte jeweils einen Ausgabewert berechnen.
- Wenn man einen Wert einliest, wird er im Hauptspeicher abgelegt.
- Variablen sind benannte Hauptspeicherbereiche, die man sich zur Aufnahme eines bestimmten Wertes (z.B. der Eingabe) reserviert hat.

Variablen: Motivation (2)

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

int main()
{
        int n; // Deklaration der Variablen n

        cout << " Bitte eine ganze Zahl eingeben: ";
        cin >> n;
        cout << n << " zum Quadrat ist "
                << n*n << ".\n";
        return 0;
}
```

Variablen: Deklaration

- Im obigen Beispiel wird eine Variable mit Namen "`n`" und Datentyp "`int`" deklariert:

```
int n;
```

- Etwas vereinfacht sehen Variablendeklarationen in C/C++ immer so aus:

$$\langle\text{Datentyp}\rangle\ \langle\text{Bezeichner}\rangle\texttt{;}$$

Es gibt verschiedene Notationen, um Syntax (korrekte Zeichenfolgen) zu erklären/definieren (siehe Kapitel 4 und 5). Hier schreiben wir syntaktische Kategorien (wie "Subjekt") in spitzen Klammern $\langle\ldots\rangle$. Wofür sie stehen, sollte an anderer Stelle definiert sein. Das Semikolon ist dagegen genau so zu schreiben, wie es dort steht.

Datentyp `int` (1)

- Wenn der Compiler eine Deklaration sieht, reserviert er einen Hauptspeicherbereich, der für einen Wert des Datentyps ausreichend groß ist.
- `int` steht für "integer", ganze Zahl, Element von $\mathbb{Z} = \{\ldots, -2, -1, 0, +1, +2, \ldots\}$.
- Z.B. wäre 1.5 keine ganze Zahl.

  Dafür würde man den Typ `float` oder `double` verwenden, s.u.
- Bei heutigen Rechnern sind `int`-Werte typischerweise 32-Bit lang. Der Compiler muß also 4 Bytes reservieren.

Datentyp `int` (2)

- In 32 Bit können aber nur $2^{32} = 4\,294\,967\,296$ verschiedene Werte dargestellt werden.
- Daher sind `int`-Werte meist eingeschränkt auf: $-2\,147\,483\,648$ (-2^{31}) bis $+2\,147\,483\,647$ ($2^{31}-1$).
- Überschreitet eine Rechenoperation diesen Bereich, können (je nach Compiler) zwei Dinge passieren:
  - ◇ Man bekommt den falschen Wert, z.B. die Addition zweier großer positiver Zahlen wird negativ.
  - ◇ Das Programm wird mit einem Laufzeitfehler (Exception: Overflow/Überlauf) abgebrochen.

Datentyp `int` (3)

- Die erste Lösung (unbemerktes Weiterrechnen mit verkehrtem Wert) ist häufiger.

  Das kann natürlich schlimme Konsequenzen haben. Ein unbehandelter Programmabbruch aber auch.
- Insofern unterscheidet sich die Rechnerarithmetik ziemlich deutlich von der in der Mathematik üblichen Arithmetik:

```
1000000 zum Quadrat ist -727379968.
```

- Gute Programmierer stellen sicher, daß man in so einem Fall wenigstens eine Fehlermeldung bekommt.

Datentyp `int` (4)

- Wenn man so große Zahlen braucht, kann man sich das Rechnen mit beliebig langen Zahlen selbst programmieren (es gibt auch fertige Bibliotheken).

  Dann ist natürlich die Addition oder Multiplikation kein einzelner Maschinenbefehl mehr. Die CPU kann direkt nur mit 32-Bit Zahlen (oder 64-Bit) rechnen.

- Der für `int`-Werte mögliche Bereich kann von CPU zu CPU bzw. genauer von Compiler zu Compiler verschieden sein.

  Früher waren 16-Bit CPUs häufig. Dort war `int` typischerweise auf den Bereich von −32 768 bis +32 767 beschränkt, bei Maschinen mit Einerkomplement-Darstellung sogar auf −32 767 bis +32 767.

Datentyp `int` (5)

- Man kann den Bereich für den Typ `int` abfragen:
  - ◇ Alte (von C geerbte) Methode:

```
#include <limits.h> // Oder: <climits>
...
cout << "Maximales int: " << INT_MAX;
```

  - ◇ Neue (C++) Methode:

```
#include <limits>
...
cout << "Maximales int: "
        << numeric_limits<int>::max;
```

- Entsprechend Minimum.

Datentyp `short int`

- Wenn ein kleiner Bereich ausreicht, und man gerne Speicherplatz sparen möchte, kann man den Typ "`short int`" verwenden.
- Dies ist typischerweise eine 16-Bit Zahl, d.h. der Compiler reserviert nur 2 Byte.
- Der garantierte Bereich ist: −32 767 bis +32 767.

  Die Konstanten aus `<limits.h>` heißen `SHRT_MIN` und `SHRT_MAX`.

- Statt "`short int`" kann man auch einfach "`short`" schreiben.

Datentyp `long int`

- Der Datentyp `int` ist optimal an die CPU angepasst, auf einer alten 16-Bit CPU also äquivalent zu `short`.
- Wenn man sicher sein will, daß man mindestens eine 32-Bit Zahl hat, muß man "`long int`" schreiben.
- Der garantierte Bereich ist dann −2 147 483 647 bis +2 147 483 647.

  Die Konstanten aus `<limits.h>` heißen `LONG_MIN` und `LONG_MAX`.

- Statt "`long int`" kann man auch einfach "`long`" schreiben.

unsigned Datentypen (1)

- Wenn man keine negative Zahlen braucht, aber das sonst für das Vorzeichen benutzte Bit für noch größere Zahlen verwenden will, kann man auch folgende Datentypen verwenden:
  - ◇ `unsigned short int` bzw. kurz `unsigned short`: Bereich von 0 bis 65 535 (oder größer)

    Die Konstante aus `<limits.h>` heißt `USHRT_MAX`.
  - ◇ `unsigned long int` bzw. kurz `unsigned long`: Bereich von 0 bis 4 294 967 295 (oder größer)
  - ◇ `unsigned int` oder kurz `unsigned`: heute meist wie `unsigned long`.

unsigned Datentypen (2)

- Als Anfänger verzichtet man besser auf `unsigned`-Typen: Wenn sie mit vorzeichenbehaften Zahlen in einer Berechnung gemischt werden, kann es zu unerwarteten Effekten kommen.

  Das Problem ist, dass beim Vergleich von vorzeichenbehafteten Werten und `unsigned`-Werten der vorzeichenbehaftete Wert als `unsigned` interpretiert wird, wodurch z.B. `-1` als sehr große positive Zahl erscheint. Die `unsigned`-Typen sind nicht dafür gedacht, um zu dokumentieren, dass eine Variable nur nicht-negative Werte annehmen kann. Das müsste man mit `assert` machen (wird später erläutert). Der Zweck der `unsigned`-Typen ist, das letzte Bit auszunutzen, das normalerweise für das Vorzeichen benötigt wird. Es ist auch üblich, `unsigned` zu verwenden, wenn man eigentlich gar keine Zahlen braucht, sondern nur das reine Bitmuster im Speicher (z.B. zur Repräsentation von Teilmengen einer kleinen Menge mit einem Bit pro Element).

Datentyp char (1)

- Der Datentyp `char` (von "character") dient zur Repräsentation von Zeichen.
- Da C/C++ aber recht nah an der Hardware ist, handelt es sich einfach um Bytes, und man darf auch mit ihnen rechnen (wie mit kleinen Integers).
- Als Konstanten des Typs `char` kann man sowohl kleine Zahlen (z.B. `93`) als auch Zeichen (z.B. `'a'`) verwenden.

  Die Umwandlung von Zeichen in Zahlen ist systemabhängig.

Datentyp char (2)

- Der Wertebereich des Typs `char` ist je nach CPU bzw. Compiler normalerweise
  - ◇ 0 bis 255 (`unsigned char`)
  - ◇ −128 bis +127 (`signed char`)

  Wenn der genaue Bereich wichtig ist, kann man die Schlüsselworte `signed`/`unsigned` explizit verwenden.

  Wenn ein Programm Zeichen ausgibt, gibt es eigentlich nur Bitmuster (Bytes) aus. Wie diese Bytes dargestellt werden, hängt am Terminalprogramm, dem Editor, dem Medientyp bei Webseiten, u.s.w. Der ASCII-Zeichensatz verwendet den Wertebereich 0-127 und ist sehr portabel, aber die deutschen Umlaute fehlen darin. ISO Latin 1 (8859-1) ist ein häufig verwendete Zeichenkodierung, die ASCII erweitert. Ein "ä" in ISO Latin 1 hat den Code 228. In UTF-8 wird ein "ä" als Folge von zwei Bytes dargestellt: 195, 164.

Datentyp `char` (3)

- Zur Aussprache: Man kann "`char`" wie den Anfang von "character" aussprechen ("kär"), oder wie eine Mischung von "child" und "car" ("tschar").

  Die zweite Aussprache ist deswegen legal, weil es ein englisches Wort "char" gibt, mit der Bedeutung "verkohlen", oder Rotforelle, Putzen.

- Für den Unicode-Zeichensatz, der auch chinesische etc. Schriftzeichen darstellen kann, reicht ein Byte nicht aus. Daher gibt es auch den Typ `wchar_t`.

  "wide character". Der Grund für den Suffix "`_t`" ist, daß dieser Typ in C nachträglich über eine Include-Datei `wchar.h` definiert wurde. In C++ ist er eingebaut. Typischerweise entspricht er `unsigned short`.

Datentypen `float`, `double` (1)

- Für Zahlen mit Nachkommastellen verwendet man (normalerweise) die Datentypen
  - ◇ `float`: typisch 4 Byte ("floating point number")
  - ◇ `double`: typisch 8 Byte ("double precision")
  - ◇ `long double`: typisch 10 Byte.
- Es handelt sich dabei um Gleitkommazahlen, d.h. sie werden intern mit Mantisse und Exponent repräsentiert, z.B. `1.04E5` $= 1.04 * 10^5 = 10400$.

  Tatsächlich verwendet man typischerweise Exponenten zur Basis 2. Statt "Gleitkomma" kann man auch "Fließkomma" sagen. Man beachte, dass Werte mit "`.`" notiert werden, und nicht mit Komma.

Datentypen `float`, `double` (2)

- Durch die Repräsentation mit Mantisse und Exponent ("wissenschaftliche Notation") sind sehr große und sehr kleine Zahlen darstellbar.
- "Gleitkomma": Das Komma kann sich an beliebiger Stelle der dargestellten Zahl befinden, die Kommaposition kann sich im Laufe der Berechnung ändern.
- Im Gegensatz dazu rechnet man bei Festkomma-Zahlen mit einer festen Anzahl von Stellen nach dem Komma (z.B. 2).

  C/C++ haben keine Festkommazahlen außer den Integer-Typen. Man kann sich das natürlich bei Bedarf selbst programmieren.

Datentypen `float`, `double` (3)

- Zum Beispiel hat der Typ `float` normalerweise 6 signifikante Dezimalstellen.
- Man kann damit z.B. folgende Zahl darstellen:

$$0.0000123456 = 1.23456 * 10^{-6}$$

- Folgende Zahl kann man dagegen nicht darstellen:

$$1.0000123456$$

  Dies würde 11 signifikante Stellen benötigen.
- Da man nur 6 signifikante Stellen hat, würde diese Zahl gerundet zu: 1.00001

Datentypen `float`, `double` (4)

- Bei Gleitkommazahlen kann es zu Rundungsfehlern kommen, deren Effekt bei komplizierten Berechnungen undurchschaubar ist.
- Übliche mathematische Gesetze gelten nicht, z.B.:

$$(A + B) + C \;=\; A + (B + C)$$

(Assoziativgesetz). Verletzt für:

◇ $A = +1000000$

◇ $B = -1000000$

◇ $C = 0.0001$

Datentypen `float`, `double` (5)

- Wenn sich Rundungsfehler (wie im Beispiel) fortpflanzen, ist es möglich, daß das Ergebnis absolut nichts mehr bedeutet.

  Also ein mehr oder weniger zufälliger Wert ist, der weit entfernt vom mathematisch korrekten Ergebnis ist.

- Geldbeträge würde man z.B. mit `int` in Cent repräsentieren, und nicht mit `float`.
- Es gibt Bibliotheken, die mit Intervallen rechnen, so daß man am Ende wenigstens merkt, wenn das Ergebnis sehr ungenau geworden ist.

Datentypen `float`, `double` (6)

Datentyp `float`:

- Nach dem IEEE Standard 754 werden für 32-Bit Gleitkommazahlen 24 Bit für die Mantisse verwendet, 8 Bit für den Exponenten (-125 bis $+128$), und 1 Bit für das Vorzeichen.

  Das sind eigentlich 33 Bit. Da das erste Bit der Mantisse aber immer 1 ist, wird es nicht explizit abgespeichert, man braucht also in Wirklichkeit nur 23 Bit für die Mantisse. Dies betrifft aber nur die sogenannten "normalisierten Zahlen". Der Standard behandelt auch nicht normalisierte Zahlen, $+\infty$, $-\infty$ und einen Fehlerwert ("not a number"). Dies erklärt, warum einige theoretisch mögliche Werte für den Exponenten nicht vorkommen. Der Wert des Exponenten e wird übrigens immer als $e+127$ in den Bits 23 bis 30 abgespeichert (Bit 31 ist das Vorzeichen, Bit 0 bis 22 die Mantisse).

Datentypen `float`, `double` (7)

Datentyp `float`, Fortsetzung:

- (Etwas mehr als) sechs signifikante Dezimalstellen.

  Konstante `FLT_DIG` in `float.h`, bzw. `numeric_limits<float>::digits10` in `limits`. Binäre Länge der Mantisse: `FLT_MANT_DIG` in `float.h`, bzw. `numeric_limits<float>::digits` in `limits`.

- Kleinster positiver Wert: $1.17549 * 10^{-38}$

  Konstante `FLT_MIN` in `float.h`. Exponent-Bereich: `FLT_MIN_10_EXP` und `FLT_MAX_10_EXP` (dezimal), bzw. `FLT_MIN_EXP` und `FLT_MAX_EXP` (binär). In `limits` gibt es Komponenten `min`, `max`, `min_exponent`, `max_exponent`, `min_exponent10`, `max_exponent10`.

- Größter Wert: $3.40282 * 10^{38}$

  Konstante `FLT_MAX` in `float.h`, Komponente `max` in `limits`.

Datentypen `float`, `double` (8)

Datentyp `double`:

- 8 Byte (53 Bit Mantisse, 11 Bit Exponent).

  Der Exponent läuft im Bereich -1021 bis $+1024$.
- 15 signifikate Dezimalstellen.

  Die Konstanten in `float.h` beginnen mit `DBL_`.
- $2.22507385850720*10^{-308}$: kleinster positiver Wert.
- $1.79769313486232 * 10^{308}$: größter Wert.
- In C wurden ursprünglich alle Rechnungen mit doppelter Genauigkeit (`double`) ausgeführt.

  Es ist also der normale Gleitkomma-Typ.

Datentyp `bool`

- Der Datentyp `bool` dient zur Repräsentation von Wahrheitswerten.

  Benannt nach George Boole, 1815-1864.
- Er hat nur zwei mögliche Werte:
  - ◇ `true`: wahr (1)
  - ◇ `false`: falsch (0)
- Man erhält einen booleschen Wert z.B. als Ergebnis eines Vergleichs:
  - ◇ $1 < 2$ ist wahr,
  - ◇ $3 > 4$ ist falsch.

Operator `sizeof`

- Mit dem Operator `sizeof` kann man den für einen Datentyp bzw. eine Variable nötigen Speicherplatz bestimmen (in Byte/`char`-Einheiten).
- `sizeof(char)` ist definitionsgemäß 1.
- `sizeof(int)` ist heute meist 4, es könnte aber z.B. auch 2 sein.
- Wenn die Variable `n` als `int` deklariert ist, liefert `sizeof(n)` den gleichen Wert wie `sizeof(int)`.

Inhalt

Beispiel (noch einmal)

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

int main()
{
        int n; // Deklaration der Variablen n

        cout << " Bitte eine ganze Zahl eingeben: ";
        cin >> n;
        cout << n << " zum Quadrat ist "
                << n*n << ".\n";
        return 0;
}
```


Variablen, Initialisierung (1)

- Die Deklaration der Variablen `n` bewirkt, daß der Compiler einen bestimmten Speicherbereich (typischerweise 4 Byte) für diese Variable reserviert.

  Wenn Sie wollen, können Sie sich die Hauptspeicheradresse mit
  `cout << (unsigned long) &n;`
  ausgeben lassen.

- Dieser Speicherbereich kann alle möglichen Werte des Datentyps `int` aufnehmen (z.B. $-2\,147\,483\,648$ bis $+2\,147\,483\,647$).

Variablen, Initialisierung (2)

- Obwohl man sich Variablen wie einen Kasten vorstellen kann, in den man Werte hineinlegen kann, gibt es doch Unterschiede:
  - ◇ Der Kasten ist niemals leer. Wenn man hineinschaut, findet man immer einen Wert (die Bits können ja nur 0 oder 1 sein, nicht “leer”).
  - ◇ Wenn man einen neuen Wert hineinlegt, verschwindet damit automatisch der alte Wert (er wird “überschrieben”).

Variablen, Initialisierung (3)

- Im Beispiel-Programm wird durch die Anweisung

  ```
  cin >> n;
  ```

  ein Wert eingelesen und in die Variable geschrieben.

  Der Wert wird von der Standard-Eingabe gelesen (normalerweise ist das die Tastaur).

- Das erste Mal, wenn ein Wert in die Variable geschrieben wird, heißt Initialisierung der Variablen.
- Es ist ein Fehler, die Initialisierung einer Variablen zu vergessen, also einen Wert auszulesen, ohne vorher einen Wert hineingeschrieben zu haben.

Variablen, Initialisierung (4)

- Im Beispiel könnte man die entscheidende Anweisung löschen oder "herauskommentieren" (der Compiler ignoriert sie dann):

```
// cin >> n;
```

Natürlich wartet das Programm dann nicht mehr auf eine Benutzer-Eingabe.

- In diesem Fall wird ein relativ zufälliger Wert aus der Variablen ausgelesen (was immer vorher an der Adresse im Hauptspeicher gestanden hat), z.B.

```
-858993460 zum Quadrat ist 687194768.
```


Variablen, Initialisierung (5)

- Das passiert auch, wenn der Benutzer eine ungültige Eingabe macht, also z.B. "`abc`" eingibt.
- Der Operator >> macht in diesem Fall folgendes:
  - ◇ Er ändert den Wert der Variablen nicht (sie bleibt also uninitialisiert).
  - ◇ Er liefert 0 (so kann man den Fehler erkennen).

    Im obigen Beispiel wird der Rückgabewert gar nicht abgefragt.
  - ◇ Er läßt die falschen Zeichen in der Eingabe.

    Es würde also nichts bringen, einen neuen Einlese-Versuch zu starten. Man muß den Eingabepuffer zuerst mit `cin.sync();` leeren, und Fehlerflags mit `cin.clear();` zurücksetzen.

Ausgabesyntax (1)

- Im Beispiel werden mehrere Werte (Zeichenketten und Zahlen) in einer Anweisung ausgegeben:

```
cout << n << " zum Quadrat ist "
        << n*n << ".\n";
```

In C/C++ dürfen Anweisungen beliebig über mehrere Zeilen verteilt werden. Das Ende wird jeweils mit einem Semikolon markiert.

- Dies ist äquivalent zu:

```
cout << n;
cout << " zum Quadrat ist ";
cout << n*n;
cout << ".\n";
```


Ausgabesyntax (2)

- Der Grund ist, daß << implizit von links geklammert wird, und jeweils sein linkes Argument (`cout`) zurückliefert:

```
((((cout << n) << " zum Quadrat ist ")
        << n*n) << ".\n");
```

- Zuerst wird also

```
cout << n
```

ausgeführt und dann durch `cout` ersetzt.

- u.s.w.

Zuweisungen (1)

- Man braucht Variablen nicht nur, wenn man Werte einlesen will.
- Häufig ist die direkte Berechung aus den eingelesenen Werten relativ kompliziert, und man möchte sich daher Zwischenergebnisse unter einem Namen merken.
- Daher kann man eine Variable auch mit einer Zuweisung auf einen Wert setzen, z.B. speichert

  `a = 1;`

  den Wert 1 in die Variable `a`.

Zuweisungen (2)

- Allgemein haben Zuweisungen die Form

  ⟨Variable⟩ = ⟨Wert⟩;

- Der Wert kann dabei auch berechnet sein, z.B.

  `a = 1 + 1;  // Setzt a auf 2.`

- Tatsächlich funktioniert auch folgendes:

  `a = a + 1;  // Erhöht Wert von a um 1.`

- Hier wird der aktuelle Wert von `a` genommen, 1 aufaddiert, und das Ergebnis in `a` zurück gespeichert (es überschreibt daher den bisherigen Wert von `a`).

Zuweisungen (3)

- Spätestens bei

  `a = a + 1;`

  bekommen Mathematiker Bauchschmerzen: `a` kann niemals gleich `a+1` sein.

- Eigentlich muß man diese Zuweisung so verstehen:

  $a_{neu} = a_{alt} + 1;$

- Die Zuweisung ist nicht der Vergleichsoperator, der wird `==` geschrieben (s.u.).

  Z.B. in Pascal wird die Zuweisung `:=` geschrieben, das war dem C-Erfinder aber zu lang für eine so häufige Operation.

Zuweisungen (4)

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

int main()
{
        int a; // Deklaration der Variablen a

        a = 1; // Zuweisung
        a = a + 1;
        a = a * a;
        cout << "a ist jetzt " << a << "!\n";
        return 0;
}
```

Beispiel: Ausführung (1)

- Es soll jetzt noch einmal Schritt für Schritt erklärt werden, wie das Beispiel-Programm ausgeführt wird.
- Es ist eine wichtige Fähigkeit eines Programmiers, die Ausführung eines Programms in Gedanken oder auf dem Papier zu simulieren.
- Man schlüpft dabei in die Rolle der CPU.
- Wie erwähnt, hat die CPU einen "Instruction Pointer" / "Program Counter", der die Adresse des aktuell abzuarbeitenden Maschinenbefehls enthält.

Beispiel: Ausführung (2)

- Als Programmierer arbeitet man natürlich nicht auf der Ebene der Maschinenbefehle, sondern merkt sich stattdessen immer, welches die nächste auszuführende Anweisung des C++-Programms ist.

  Meist entsprechen die Anweisungen den Zeilen. Gelegentlich erstreckt sich eine Anweisung über mehrere Zeilen, manchmal muß man auch innerhalb einer Anweisung mehrere Schritte betrachten.

- Auf den Folien, die die Ausführung des Programms visualisieren, ist die nächste auszuführende Anweisung mit "$\Rightarrow$" gekennzeichnet.

Beispiel: Ausführung (3)

- Solange man keine Kontrollstrukturen verwendet, wird einfach eine Anweisung nach der anderen abgearbeitet (in der Reihenfolge, in der sie aufgeschrieben sind).

  So wie der "Instruction Pointer" / "Program Counter" in der CPU einfach hochgezählt wird, wenn es sich nicht gerade um einen Sprungbefehl handelt.

  Genauer führen auch Funktionsaufrufe (z.B. mit <<) zu einem vorübergehenden Transfer der Kontrolle, aber je nach Abstraktionsebene kann man den Funktionsaufruf auch als elementaren Schritt betrachten. Gerade Bibliotheksfunktionen wird man einfach als Erweiterung der Sprache um neue atomare Befehle verstehen.

Beispiel: Ausführung (4)

- Außerdem hat der Rechner noch Hauptspeicher, in den Daten geschrieben, und aus dem Daten ausgelesen werden.
- Auf C++-Ebene entspricht dem eine Tabelle mit den aktuellen Werten der Variablen.

  Natürlich steht vieles mehr im Hauptspeicher, z.B. das Programm, aber das wird bei der Simulation als fest gegeben angenommen.

- Schließlich muss man ggf. noch über den Zustand der Ein-/Ausgabe Buch führen.

  Was wurde bisher eingelesen oder ausgedruckt? Z.B. markiert man das nächsten einzulesenden Zeichens der Eingabe mit $\uparrow$.

Beispiel: Ausführung (5)

- Für reine Deklarationen wird kein Code erzeugt.
  - ◇ Sie bewirken zur Compilezeit eine Speicherreservierung,
  - ◇ werden aber zur Laufzeit im wesentlichen übersprungen (sind nicht ausführbar).

    Für Experten: Beim Prozeduraufruf gibt es einen Vorspann, der Register so setzt, dass für die lokalen Variablen Platz reserviert wird. Insofern gibt es doch etwas Programmcode (Maschinenbefehle), der für die Deklaration ausgeführt wird. Wenn man mehrere Variablen deklariert, wird aber in einem Schritt ein entsprechend größerer Speicherbereich reserviert. Man kann diesen Programmcode also nicht einer bestimmten Deklaration zuordnen. Anders ist die Deklaration mit Initialisierung oder Konstruktoraufruf: Dann gibt es natürlich Programmcode für die spezielle Deklaration.

Beispiel: Ausführung (6)

a:	(unbekannt)
Ausgabe:	(leer)

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

int main()
{
        int a; // Deklaration der Variablen a

     ⇒  a = 1; // Zuweisung
        a = a + 1;
        a = a * a;
        cout << "a ist jetzt " << a << "!\n";
        return 0;
}
```


Beispiel: Ausführung (7)

a:	1
Ausgabe:	(leer)

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

int main()
{
        int a; // Deklaration der Variablen a

        a = 1; // Zuweisung
     ⇒  a = a + 1;
        a = a * a;
        cout << "a ist jetzt " << a << "!\n";
        return 0;
}
```


Beispiel: Ausführung (8)

a:	2
Ausgabe:	(leer)

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

int main()
{
        int a; // Deklaration der Variablen a

        a = 1; // Zuweisung
        a = a + 1;
     ⇒  a = a * a;
        cout << "a ist jetzt " << a << "!\n";
        return 0;
}
```

Beispiel: Ausführung (9)

a:	4
Ausgabe:	(leer)

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

int main()
{
        int a; // Deklaration der Variablen a

        a = 1; // Zuweisung
        a = a + 1;
        a = a * a;
     ⇒  cout << "a ist jetzt " << a << "!\n";
        return 0;
}
```


Beispiel: Ausführung (10)

a:	4
Ausgabe:	a ist jetzt 4!

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

int main()
{
        int a; // Deklaration der Variablen a

        a = 1; // Zuweisung
        a = a + 1;
        a = a * a;
        cout << "a ist jetzt " << a << "!\n";
     ⇒  return 0;
}
```


Aufgabe

Was gibt dieses Programm aus?

```
int main()
{
        int i;
        int n;

        i = 27;
        n = i - 20;
        i = 5;
        cout << i * n << "\n";
        return 0;
}
```


Inhalt

Bedingungen (1)

- Bisher werden die Anweisungen in einem Programm einfach von oben nach unten der Reihe nach ausgeführt.
- So kann man natürlich noch keine sehr anspruchsvollen Programme schreiben.
- Es ist auch möglich, Anweisungen nur auszuführen, wenn eine Bedingung erfüllt ist.

Bedingungen (2)

```
...
int main()
{
        int antwort;

        cout << "Was ist 1+1? ";
        cin >> antwort;
        if(antwort == 2)
                cout << "Ja, richtig!\n";
        else
                cout << "Leider falsch.\n";

        return 0;
}
```


Bedingungen (3)

- Bedingte Anweisungen sehen in C/C++ so aus:

```
if ( ⟨Bedingung⟩ )
        ⟨Anweisung⟩
else
        ⟨Anweisung⟩
```

- Die erste Anweisung wird ausgeführt, wenn die Bedingung erfüllt ist.

  "if" bedeutet "wenn", "falls".

- Die zweite Anweisung wird ausgeführt, wenn die Bedingung nicht erfüllt ist.

  "else" bedeutet "sonst".

Bedingungen (4)

- Die Zeilenumbrüche und Einrückungen sind in C++ (und allen anderen "formatfreien" Sprachen) nicht wichtig.
- Man könnte es z.B. auch so schreiben:

```
if(⟨Bedingung⟩) ⟨Anweisung⟩
else ⟨Anweisung⟩
```

- Es ist aber gut, einen bestimmten Einrückungsstil konsequent anzuwenden, weil das die Programme für den Menschen lesbarer macht.

  Dem Compiler sind solche Dinge völlig egal.

Bedingungen (5)

- Den `else`-Teil kann man auch weglassen, wenn man nur im positiven Fall (Bedingung ist erfüllt) eine Anweisung ausführen möchte.
- Beispiel:

```
// Berechnung des Absolutwertes von x:
if(x < 0)
        x = -x;
```

- Falls mehr als eine Anweisung von `if` oder `else` abhängen, muß man sie in `{...}` einschließen.

Beispiel siehe nächste Folie.

Bedingungen (6)

```
int main()
{
        int antwort;
        cout << "Was ist 1+1? ";
        cin >> antwort;
        if(antwort == 2)
                cout << "Ja, richtig!\n";
        else {
                cout << "Leider falsch.\n";
                cout << "Richtige Antwort: 2.\n";
        }
        return 0;
}
```


Bedingungen (7)

- Formal sind
  - ◇ `if(`⟨Bedingung⟩`)` ⟨Anweisung⟩
  - ◇ `if(`⟨Bedingung⟩`)` ⟨Anweisung⟩ `else` ⟨Anweisung⟩
  - ◇ `{` ⟨Anweisung⟩ ...`}`

  selbst wieder Anweisungen.
- Man kann also z.B. `else if`-Ketten wie im nächsten Beispiel bilden.

  Hier ist die von `else` abhängige Anweisung wieder eine `if ... else`-Anweisung. Wenn man dagegen im `if`-Teil eine bedingte Anweisung verwenden will, sollte man sie besser in `{...}` einschließen (s.u.).

Bedingungen (8)

```
int main()
{
        int n;
        cout << " Bitte ganze Zahl eingeben: ";
        cin >> n;
        if(n > 0)
                cout << "Die Zahl ist positiv.\n";
        else if(n == 0)
                cout << "Die Zahl ist Null.\n";
        else
                cout << "Die Zahl ist negativ.\n";
        return 0;
}
```

Inhalt

Schleifen (1)

- Bisher wird jede Anweisung im Programm maximal ein Mal ausgeführt.
- In den meisten Programmen müssen aber bestimmte Anweisungen wiederholt ausgeführt werden.
- Hierfür gibt es verschiedene Konstrukte in C/C++, das grundlegendste ist wohl die `while`-Schleife:

  `while(`⟨Bedingung⟩`)`
  ⟨Anweisung⟩
- Hier wird zunächst die Bedingung getestet.
- Falls sie erfüllt ist, wird die Anweisung ausgeführt.

Schleifen (2)

- Dann wird wieder die Bedingung getestet.
- Ist sie immernoch erfüllt, wird die Anweisung erneut ausgeführt.
- Und so weiter (bis die Bedingung hoffentlich irgendwann nicht mehr erfüllt ist).
- Die Anweisung muß also (u.a.) den Wert einer Variable ändern, die in der Bedingung verwendet wird.

  Und zwar in eine Richtung, die schließlich dazu führt, daß die Bedingung nicht mehr erfüllt ist.

Schleifen (3)

- Falls die Schleifenbedingung immer erfüllt bleibt, erhält man eine Endlosschleife.

  Die CPU arbeitet hart, aber es geschieht nichts mehr (oder eine endlose Ausgabe rauscht vorbei, man wird immer wieder zu einer Eingabe aufgefordert ohne das Programm verlassen zu können, etc.).
- Man kann Programme normalerweise mit `Crtl`+`C` abbrechen.

  Unter Windows kann man sich mit `Crtl`+`Alt`+`Delete` die Prozesse anzeigen lassen und das Programm abbrechen. Unter UNIX kann man mit `ps` oder `ps -ef` sich die Prozesse anzeigen lassen, und dann mit `kill` ⟨`Prozessnummer`⟩ abbrechen, notfalls mit `kill -9` ⟨`Prozessnummer`⟩.

Schleifen (4)

```
...
int main()
{
        int i;

        i = 1;
        while(i <= 10) {
                cout << i << " zum Quadrat ist "
                        << i * i << "\n";
                i = i + 1;
        }
        return 0;
}
```


Schleifen (5)

- Das Muster im obigen Programm ist sehr typisch:
  - ◇ Es gibt eine Laufvariable, im Beispiel `i`.
  - ◇ Diese wird zuerst initialisiert:

    `i = 1;`
  - ◇ In der Bedingung wird getestet, ob die Laufvariable schon eine gewisse Grenze erreicht hat:

    `while(i <= 10)`
  - ◇ Am Ende der Schleife wird die Laufvariable auf den nächsten Wert weitergeschaltet:

    `i = i + 1;`

Schleifen (6)

- Weil das Muster so typisch ist, gibt es in C/C++ noch die `for`-Schleife, bei der die ganze Schleifenkontrolle im Kopf der Schleife zusammengefasst ist:

  `for(` ⟨Initialisierung⟩; ⟨Bedingung⟩; ⟨Erhöhung⟩)
  ⟨Anweisung⟩

- Daher ist die `for`-Schleife häufig übersichtlicher als die `while`-Schleife.

  Wenn das Programm diesem Muster entspricht und die drei Teile relativ einfach/kurz sind.

Schleifen (7)

- Tatsächlich wäre die `for`-Schleife aber nicht nötig. Sie ist definitionsgemäß äquivalent zu

```
{ ⟨Initialisierung⟩;
  while(⟨Bedingung⟩) {
      ⟨Anweisung⟩
      ⟨Erhöhung⟩;
  }
}
```

- Wenn die vielen Möglichkeiten am Anfang verwirren, beschränke man sich auf die `while`-Schleife.

  Man kann auch sinnvolle Programme in einer C++-Teilmenge schreiben. Manche Professoren behandeln die `for`-Schleife nicht.

Schleifen (8)

```
...
int main()
{
        int i;

        for(i = 1; i <= 10; i = i+1) {
                cout << i << " zum Quadrat ist "
                        << i * i << "\n";
        }
        return 0;
}
```


Schleifen (9)

- Man hört öfters, das Studenten von "If-Schleifen" reden. Das ist falsch!
- Die "If-Anweisung" (das "If-Statement") ist keine Schleife, sondern gehört zu den bedingten Anweisungen.

  Eine Schleife ist etwas, wo die gleiche Anweisung wiederholt ausgeführt wird, eben in einer Art Kreis.

- Es gibt in C++ die `while`-Schleife, die `for`-Schleife, und die `do`-Schleife. Mehr nicht.

Aufgabe

- Schreiben Sie ein Programm, das testet, ob eine eingegebene Zahl eine Primzahl ist.

  D.h. nur durch 1 und sich selbst teilbar. Primzahlen sind z.B. 2, 3, 5, 7, 11, 13, 17, 19.

- Den Teilbarkeitstest können Sie mit dem Modulo-Operator `%` ausführen: `a % b` liefert den Rest, der übrig bleibt, wenn man `a` durch `b` teilt.

  Z.B. ist `7 % 3 == 1`.

- Sie können eine Prozedur wie `main` auch vorzeitig durch eine `return`-Anweisung beenden.

  Sie muß nicht immer ganz am Ende stehen.

Inhalt

Fehlervermeidung (1)

- Programme funktionieren oft nicht sofort, wenn man sie eingetippt hat.
- Zunächst gibt der Compiler Fehlermeldungen aus, wenn der Programmtext kein gültiges C++ ist.
- Nachdem man diese Fehler korrigiert hat, und das Programm durch den Compiler läuft, erhält man ein ausführbares Programm.
- Wenn man das Programm dann ausprobiert, tut es öfters nicht sofort das, was die Aufgabe verlangt.

Fehlervermeidung (2)

- Es gibt bestimmte häufig auftretende Fehler, die durch Testen schwer zu finden sind, aber leicht vom Compiler durch Analyse des Programmtextes.
- Dazu gehört z.B. die Verwechselung von = und ==:

```
if(n = -1) cout << "n ist minus 1"; // Fehler!
```

Dies ist (leider) gültiges C++, verhält sich aber ganz anders als erwartet: Der Wert `-1` wird in die Variable `n` gespeichert (`=` ist der Zuweisungsoperator!), anschließend wird der zugewiesene Wert (`-1`) als Wahrheitswert interpretiert: Alles, was von 0 verschieden ist, gilt als wahr! Damit wird der Text ausgegeben, unabhängig davon, was `n` vor der `if`-Anweisung war. Hinterher ist es sicher `-1`. Dieses Verhalten erklärt sich aus dem minimalistischen Ansatz von C (es gab keinen booleschen Datentyp) und dem Hang zu kompakten Formulierungen.

Fehlervermeidung (3)

- Der Compiler muss den obigen Befehl akzeptieren, weil er nach dem C++-Sprachstandard zulässig ist.
- Da der Fehler immer wieder vorkommt, haben viele Compiler, auch der `g++`, einen Test dafür eingebaut.

  Ebenso wie für viele andere Situationen, die formal korrektes C++ sind, aber oftmals nicht beabsichtigt.
- Wenn Sie den Compiler mit den Optionen `-Wall` und `-Wextra` starten, so bekommen Sie entsprechende Warnungen angezeigt.

```
g++ -Wall -Wextra -o hello hello.cpp
```


Fehlervermeidung (4)

- Benutzen Sie diese Optionen immer!
- Beheben Sie die Ursachen für alle Warnungen!

  Auch wenn das Programm zu funktionieren scheint, können die Warnungen Sie auf Probleme hinweisen, die erst in anderen Umgebungen zu Tage treten.
- Sie müssen die Warnungen verstehen, bevor Sie ihre Gründe beseitigen.

  Manche Warnungen sind von der Art: „Sie haben X geschrieben, was Sie möglicherweise nicht beabsichtigt haben. Falls Sie es dennoch so meinten, schreiben Sie lieber deutlicher Y." Es wäre falsch, in so einem Falle einfach X durch Y zu ersetzen, denn es ist sehr wahrscheinlich, dass Sie weder X noch Y meinten.

Fehlervermeidung (5)

- Ein weiterer tückischer Fehler sind fehlende oder falsch gesetzte geschweifte Klammern und dazu inkonsistente Einrückungen.
- Beispiel:

```
if (x > 0)
    cout << "x hat den Wert " << x << "\n";
    cout << "x ist positiv.\n";
```

- Die zweite Ausgabe gehört nicht zu der `if`-Struktur und wird daher immer ausgeführt.

  Wenn man keine geschweiften Klammern `{...}` setzt, ist nur eine Anweisung vom `if` abhängig.

Fehlervermeidung (6)

- Bei Einrückungsfehlern gibt es keine Warnungen.

  Die Art der Einrückung ist in gewissen Grenzen eine Geschmacksfrage. Obige Einrückung ist natürlich irreführend.

- Man kann sein Programm aber mit einem "Pretty-Printer" formatieren lassen, so dass die Einrückung zur Struktur der Befehle passt. Ein solches Programm ist "`indent`".
- Wenn man sich seinen Programmtext nach der automatischen Einrückung anschaut, springt der Fehler sofort ins Auge.

Fehlervermeidung (7)

- Beispiel-Aufruf:

  `indent hello.cpp`

  Die Datei `hello.cpp` wird dadurch verändert, die Originalversion in `hello.cpp~` gespeichert. Ruf man `indent` dann noch einmal auf, ist die Sicherungskopie auch überschrieben. Deswegen kopiere man sich die Quelldatei `hello.cpp` besser explizit, bevor man mit `indent` experimentiert. Das Programm hat viele Optionen, mit denen man die Formatierung steuern kann, siehe "`man indent`" oder "`info indent`". Ich persönlich finde "`indent -kr -i8 hello.cpp`" nicht schlecht (die Anwendung von Templates sieht aber komisch aus), bleibe aber doch bei meinem manuellen Stil. Das ist eine Geschmacksfrage. Sicher ist aber, dass man mit `indent` Fehler der obigen Art leicht erkennen kann.

- Bei manchen Editoren kann man `indent` in einem Menüpunkt mit Tastenkürzel integrieren.

Fehlervermeidung (8)

Anleitung zur Integration von `indent` in Editoren:

- `gedit` (Version 2.30):

  "Bearbeiten/Einstellungen/Plugins/Externe Werkzeuge" ankreuzen, dann "Plugin konfigurieren". Links unter der Werkzeugliste befindet sich ein Symbol zum Erstellen eines neuen Menüpunktes. Unter `/bin/sh` schreiben Sie `indent`, suchen Sie sich ein Tastenkürzel aus und wählen "Eingabe: Momentan geöffnetes Dokument", "Ausgabe: Momentan geöffnetes Dokument ersetzen".

- `nedit` (Version 5.1):

  "Preferences/Default Settings/Customize Menus/Shell Menu". In der Liste "New" wählen. "Menu Entry: indent @C@C++", "Command Input: document", "Command Output: same document", "Output replaces input", "Shell command to execute: indent". Dann "Ok" und "Preferences/Save defaults".

Fehlervermeidung (9)

Weiteres Beispiel:

- Hier wurde in der Zahl ein Komma (deutsch) statt eines Dezimalpunktes (amerikanisch) verwendet:

```
float x;
x = 3,5;
cout << x << '\n';
```

- Unglücklicherweise ist auch dies korrektes C++, aber man wird sich wundern, dass das Programmstück "3" druckt, und nicht "3.5".

  Noch schlimmer ist es, wenn man sich nicht wundert. Dann bleibt der Fehler im Programm stehen. Wenn man die Warnungen nicht angestellt hat, wird das Programmstück kommentarlos übersetzt.

Fehlervermeidung (10)

Weiteres Beispiel, Forts.:

- Wenn man die Warnungen angestellt hat, bekommt man folgende Meldung:

```
test.cpp: In function int main():
test.cpp:9: warning: right-hand operand of comma
                     has no effect
```

- Der Komma-Operator wird erst in Kapitel 6 eingeführt, aber man bekommt jedenfalls einen Hinweis, dass mit dem Komma etwas nicht stimmt.

  Bei Bedarf kann man einen Tutor fragen, oder die Fehlermeldung im Forum posten. Falls Sie Meldung und Programmergebnis etwas mehr verstehen wollen: Intern wird die Anweisung wie `(x = 3), 5` behandelt.

Fehlervermeidung (11)

Weiteres Beispiel, Forts.:

- `indent` würde hier auch helfen, denn es formatiert die Zeile so: `x = 3, 5;`
- Weil Zahlkonstanten keine Leerzeichen enthalten können, ist visuell klar, dass etwas nicht stimmt.

  `indent` fügt Leerzeichen, Tabulatoren und Zeilenumbrüche nur so ein (oder löscht sie), dass die Bedeutung des Programms nicht verändert wird. Die Hilfe besteht darin, dass man hinter die Bedeutung dessen, was man geschrieben hat, deutlicher sehen kann.

- Bei einem Editor mit Syntax-Unterstützung könnte auch auffallen, dass das Komma in einer anderen Farbe angezeigt wird als die Zahlen.

Fehlersuche (1)

- Die obigen Fehler kann man finden, ohne das Programm ein einziges Mal auszuführen.

  Daher die Überschrift "Fehlervermeidung". Natürlich handelt es sich auch um Fehler, die beseitigt werden, aber normalerweise geht das schnell. Die Beseitigung von Fehlern, die erst bei der Ausführung des Programms erkannt werden, dauert meist etwas länger. Noch schlimmer sind Fehler, die man beim Testen übersieht (weil sie nur bei manchen Eingaben auftreten, und man diese Eingaben nicht ausprobiert hat). Die im Abschnitt "Fehlervermeidung" diskutierten Techniken sind unabhängig vom Testen und speziellen Eingaben. Sie können daher helfen, einige echte Fehler im Endprodukt zu vermeiden.

- Nun probiert man das Programm mit verschiedenen Testeingaben aus. Wenn es sich nicht so verhält, wie man beabsichtigt hat, muß man Fehler suchen.

Fehlersuche (2)

- Man muß versuchen, zu verstehen, warum sich das Programm so verhält, wie man beobachtet, und nicht so, wie man beabsichtigt hat.

  Wie immer ist eine genaue Aufklärung des Fehlers Voraussetzung dafür, dass man ihn wirklich beseitigen kann. Wenn man nur an den Symptomen herumdoktert, erhält man vielleicht das richtige Verhalten für eine spezielle Eingabe, aber reißt häufig neue Löcher für andere Eingaben auf. Das kann beliebig lange dauern und sehr frustrierend sein. Investieren Sie lieber etwas Zeit, um den Fehler wirklich zu verstehen, als mit Programmänderungen herumzuprobieren (Testausgaben sind dagegen nützlich, siehe nächste Folie).

- Oft kann man die Ausführung der kritischen Stelle im Kopf simulieren, und das Verhalten erklären.

Fehlersuche (3)

- Eine übliche Methode ist, zusätzliche Ausgaben in das Programm einzubauen,
  - ◇ um zu sehen, ob bestimmte Zeilen im Programm überhaupt erreicht werden, oder auch,
  - ◇ um die Werte von Variablen zu kontrollieren.
- So kann man aktiv etwas unternehmen, um den Fehler einzukreisen.

  Wichtig ist natürlich, dass Sie in Ihrem Kopf ein klares Konzept haben, wie das Programm funktionieren soll. Sonst können Sie die Abweichung davon ja gar nicht an einer Stelle festmachen.

- Anschließend werden die Ausgaben wieder gelöscht.

Fehlersuche (4)

- Es macht allerdings Mühe, Ausgaben in das Programm einzubauen.

  Man muß es jeweils neu compilieren, und mit einer Runde von Änderungen für Testausgaben ist es selten getan. Außerdem kann sich das Programm in ungünstigen Fällen durch die eingefügten Ausgabeanweisungen anders verhalten, als ohne sie (der Unterschied könnte größer sein, als nur die zusätzliche Ausgabe).

- Professioneller ist die Nutzung eines Debuggers. Das ist ein Programm, das die Fehlersuche in anderen Programmen unterstützt, z.B. indem man
  - ◇ das Programm in Einzelschritten ausführen kann,
  - ◇ die Inhalte von Variablen inspizieren kann.

Benutzung des `gdb` (1)

- Man ruft den Debugger `gdb` unter Linux auf mit

  `gdb ⟨Programm⟩`

- Falls folgende Meldung erscheint:

  ```
  Reading symbols from ⟨Programm⟩...
  (no debugging symbols found)
  ```

  wurde das Programm nicht mit der Option `-ggdb` compiliert. `gdb` ist dann nur eingeschränkt nutzbar.

  Normalerweise enthält die Programmdatei nur die Maschinenbefehle, eventuell noch einige Information für den Linker. Der Debugger muß aber wissen, welche Maschinenbefehle welcher Zeile im Quellprogramm entsprechen, und an welcher Hauptspeicher-Adresse welche Variable steht.

Benutzung des gdb (2)

- Man setzt zunächst einen “Breakpoint” an eine Stelle im Programm, ab der man die Ausführung genauer anschauen will.
- Man kann den Namen einer Funktion angeben, z.B.

  `(gdb) break main`

  Die Ausführung wird beim Betreten dieser Funktion unterbrochen.
- Man kann auch eine Zeilennummer angeben:

  `(gdb) break 1` oder: `(gdb) break test.cpp:1`

  Falls für die Zeile kein ausführbarer Code erzeugt wurde (weil dort z.B. ein Kommentar steht), wird der Breakpoint auf die erste folgende Zeile gesetzt, für die Maschinenbefehle erzeugt wurden.

Benutzung des gdb (3)

- Das Setzen des Breakpoints wird mit einer Meldung bestätigt, die die genaue Position angibt:

```
(gdb) break main
Breakpoint 1 at 0x804866d: file test.cpp, line 8.
```

  `804866d` ist die Hauptspeicheradresse des Maschinenbefehls, bei dem die Programmausführung unterbrochen wird. (Der Präfix `0x` zeigt an, dass die folgende Zahl in hexadezimaler Notation, d.h. zur Basis 16, geschrieben wird. Mit etwas Erfahrung kann man die einzelnen Bits dann leichter erkennen als in der üblichen Dezimalschreibweise. Das ist hier aber nicht wichtig.)
- Man kann mehrere Breakpoints setzen, der Befehl “`info break`” zeigt die aktuelle Liste.

Benutzung des gdb (4)

- Dann startet man die Ausführung des Programms:

  `(gdb) run`
- Die Ausführung stoppt mit der Meldung, dass ein Breakpoint erreicht wurde:

```
Breakpoint 1, main () at test.cpp:8
8         a = 1; // Zuweisung
```

- Es wird dann auch die zugehörige Quellcode-Zeile angezeigt (aus Beispiel Folie 2-40).

  Der Breakpoint ist vor der Zeile, die Anweisung wurde also noch nicht ausgeführt.

Benutzung des gdb (5)

- Man kann sich nun Werte von Variablen (allgemein Ausdrücken) anzeigen lassen, z.B.

  `(gdb) print a` oder kurz: `(gdb) p a`
- Im Beispiel ist `a` noch nicht initialisiert:

  `$1 = -1209466892`

  Der aktuelle Befehl, die Zuweisung, ist noch nicht ausgeführt. Eine Zeile weiter würde natürlich der Wert `1` ausgegeben. Die angezeigten Werte werden durchnumeriert, man kann später den Wert mit `print $1` noch einmal anzeigen. Das Ausgabeformat entspricht dem Datentyp des Ausdrucks, man kann aber auch ein Format wählen, z.B. “`(gdb) print /x a`” (hexadezimale Ausgabe).

Benutzung des gdb (6)

- Einzelschritt-Ausführung ist mit folgenden Befehl möglich:

  `(gdb) step` oder kurz: `(gdb) s`

  Die Maschinenbefehle des Programms werden ausgeführt bis zu einem Befehl, der einer anderen Zeile im C++-Programm zugeordnet ist.

- Der obige Befehl würde auch innerhalb von aufgerufenen Funktionen anhalten ("step into").

  Das ist mindestens für Bibliotheksfunktionen wie << uninteressant. Oft gibt es dafür auch keine Debugging-Informationen, was es noch verwirrender macht. Wenn es passiert ist, läßt "`finish`" die Funktion bis zum Ende laufen, und zeigt den Rückgabewert an (der Ausgabestrom sieht ziemlich gewaltig aus).

Benutzung des gdb (7)

- Einzelschritt-Abarbeitung mit der Ausführung von Funktionsaufrufen in einem Schritt ("step over"):

  `(gdb) next` oder kurz: `(gdb) n`

- Wenn man eine Schleife in einem Stück ausführen will, ist folgende Variante nützlich:

  `(gdb) until` oder kurz: `(gdb) u`

  Dies hält erst bei der nächsten Zeile an, die im Quellprogramm unterhalb der aktuellen Zeile steht, Rücksprünge für Schleifen werden also ohne Unterbrechung ausgeführt (Funktionsaufrufe auch).

- Drückt man "Enter" / "Return" bei `gdb`, wird der letzte Befehl wiederholt (sehr praktisch für `s`, `n`).

Benutzung des gdb (8)

- Das Programm bis zum nächsten Breakpoint (oder Ende) weiter laufen lassen:

  `(gdb) continue` oder kurz: `(gdb) c`

- Ein "Watchpoint" ("data breakpoint") unterbricht die Ausführung des Programms immer, wenn sich der Wert einer Variable / eines Ausdrucks ändert:

  `(gdb) watch a`

  Weil es sich um eine lokale Variable der Funktion `main` handelt, kann man diesen Watchpoint erst setzen, wenn die Funktion `main` erreicht ist. Man setzt also einen Breakpoint auf `main`, und dort dann einen Watchpoint auf `a`, und läßt das Programm dann mit `c` weiter laufen.

Benutzung des gdb (9)

- Man kann auch Variablen/Ausdrücke festlegen, deren Werte jedes Mal ausgegeben werden, wenn die Ausführung des Programms unterbrochen wird:

  `display a`

  Man muß wieder einen Breakpoint auf `main` setzen, und das Programm mit `run` dorthin laufen lassen, bevor man den `display`-Befehl eingeben kann, sonst erhält man die Fehlermeldung "No symbol `"a"` in current context."

  Mit dem `display`-Befehl erspart man sich, ggf. nach jedem Einzelschritt "`print a`" einzugeben. Der Befehl "`info display`" listet alle automatisch anzuzeigenden Variablen/Ausdrücke auf. Diese Werte sind durchnumeriert. Wenn "`display a`" der erste solche Befehl war, kann man mit "`undisplay 1`" die Anzeige beenden.

Benutzung des gdb (10)

- Folgender Befehl zeigt an, wo man sich gerade im Programm befindet, und wie (über welche Funktionsaufrufe) man dorthin gekommen ist:

  `(gdb) backtrace` oder kurz: `(gdb) bt`

- Im Moment sieht man nur die Funktion `main`, aber später enthalten Programme viele Funktionen.

  Dann ist dies ein sehr nützlicher Befehl. Für das "Post Mortem" Debugging ist es sogar der wichtigste Befehl. Wenn man z.B. durch 0 teilt, löst die CPU einen "Hardwarealarm" aus, und das Programm wird beendet. In diesem Fall wird der Hauptspeicherinhalt (für dieses Programm) in der Datei "`core`" gesichert. Diese Informationen kann man auch mit dem `gdb` analysieren. Mit "`bt full`" werden auch die Werte der lokalen Variablen angezeigt.

Benutzung des gdb (11)

- Man beendet den `gdb` mit dem Befehl

  `(gdb) quit`

  Falls das Programm noch nicht zu Ende gelaufen ist, bekommt man eine Warnung. (dann mit "`y`" bestätigen, dass wirklich beenden).

- Man bekommt Hilfe zu einem Befehl mit

  `(gdb) help` ⟨Befehl⟩

  Ohne Argumente gibt `help` eine Liste von Kategorien aus, zu denen man Befehlslisten bekommen kann.

- Der `gdb` ist ein umfangreiches Programm mit vielen Befehlen (hier nur kleiner Ausschnitt).

  Anleitung: [http://sourceware.org/gdb/current/onlinedocs/gdb/].