Objektorientierte Programmierung
(Winter 2006/2007)

# Kapitel 2: Programme mit Zuweisungen und Kontrollstrukturen

- Deklarationen, Variablen, Zuweisungen
- Basis-Datentypen
- Bedingungen
- Schleifen





## 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:

  ⟨Datentyp⟩ ⟨Bezeichner⟩;

  Wir schreiben syntaktische Kategorien in spitzen Klammern ⟨. . .⟩. Wofür sie stehen, sollte an anderer Stelle definiert sein. Siehe Kapitel über formale Syntax.

# 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

- 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`.

# 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.
- 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
  - ◇ `double`: typisch 8 Byte ("double precision")
  - ◇ `long double`: typisch 10 Byte.
- Es handelt sich dabei um Fließkommazahlen, d.h. sie werden intern mit Mantisse und Exponent repräsentiert, z.B. $1.04 * 10^5 = 10400$.

  Tatsächlich verwendet man typischerweise Exponenten zur Basis 2.





# Datentypen `float`, `double` (2)

- Durch die Repräsentation mit Mantisse und Exponent ("wissenschaftliche Notation") sind sehr große und sehr kleine Zahlen darstellbar.
- "Fließkomma": 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 Fließkommazahlen 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 Fließkommazahlen 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 Fließkomma-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).

# 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

  $\langle$Variable$\rangle$ = $\langle$Wert$\rangle$;
- 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;
}
```

# 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;
}
```

# 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⟩;
  }
}
```





# Schleifen (8)

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

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

# 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.