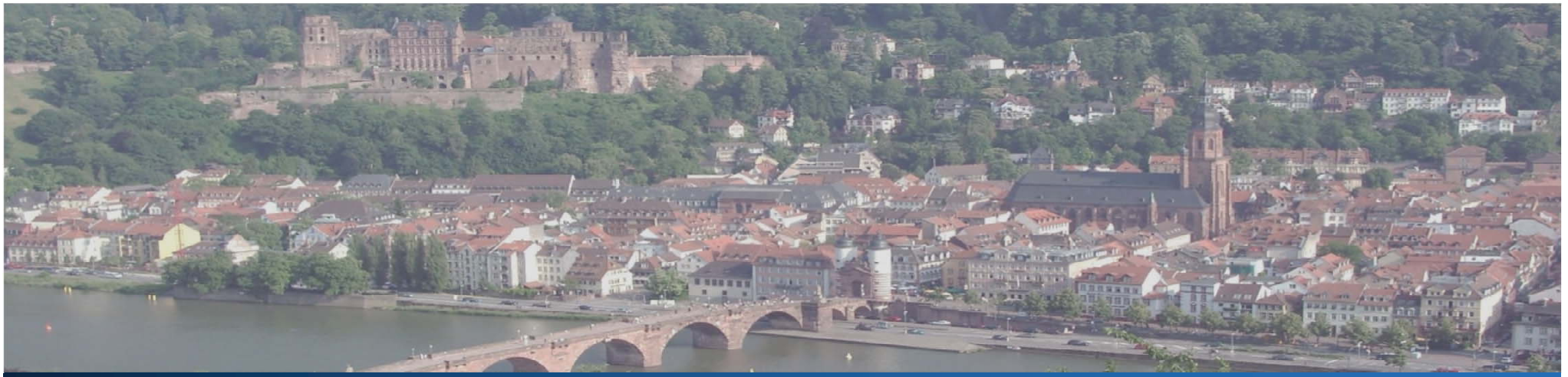
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Benutzerdefinierte Datentypen

- Aufzählungstyp
- Felder
- Zeichen und Zeichenketten
- Typedef
- Das Acht-Damen-Problem
- Zusammengesetzte Datentypen

Benutzerdefinierte Datentypen

Die bisherigen Programme haben nur mit Zahlen (unterschiedlichen Typs) gearbeitet. „Richtige“ Programme bearbeiten allgemeinere Daten, z. B.

- Zuteilung der Studenten auf Übungsgruppen,
- Flugreservierungssystem,
- Textverarbeitungsprogramm, Zeichenprogramm, . . .

Bemerkung: Im Sinne der Berechenbarkeit ist das keine Einschränkung, denn auf beliebig großen Bändern (Turing-Maschine), in beliebig tief verschachtelten Funktionen (Lambda-Kalkül) oder in beliebig großen Zahlen (FC++ mit langen Zahlen) lassen sich beliebige Daten kodieren.
Da dies aber sehr umständlich und ineffizient ist, erlauben praktisch alle Programmiersprachen dem Programmierer die Definition neuer Datentypen.

Aufzählungstyp

Dies ist ein Datentyp, der aus endlich vielen Werten besteht. Jedem Wert ist ein Name zugeordnet.

Beispiel:

```
enum color { white, black, red, green, blue, yellow };
...
color bgcolor = white;
color fgcolor = black;
```

Syntax: **(Aufzählungstyp)**

<Enum> ::= enum <Identifikator>
{ <Identifikator> [, <Identifikator>] } ;

Programm: (Vollständiges Beispiel [enum.cc])

```
#include "fcpp.hh"

enum Zustand { neu, gebraucht, alt, kaputt };

void druckeZustand( Zustand x ) {
  if ( x == neu )       print( "neu" );
  if ( x == gebraucht ) print( "gebraucht" );
  if ( x == alt )       print( "alt" );
  if ( x == kaputt )    print( "kaputt" );
}

int main() { druckeZustand( alt ); }
```

Felder

Wir lernen nun einen ersten Mechanismus kennen, um aus einem bestehenden Datentyp, wie `int` oder `float`, einen neuen Datentyp zu erschaffen: das Feld (engl.: Array).

Definition: Ein Feld besteht aus einer *festen Anzahl* von Elementen eines Grundtyps. Die Elemente sind angeordnet, d. h. mit einer Numerierung (Index) versehen. Die Numerierung ist fortlaufend und beginnt bei 0.

Bemerkung: In der Mathematik entspricht dies dem (kartesischen) Produkt von Mengen.

Felder

Beispiel: Das mathematische Objekt eines Vektors $x = (x_0, x_1, x_2)^T \in \mathbb{R}^3$ wird in C++ durch

```
double x[3];
```

dargestellt. Auf die Komponenten greift man wie folgt zu:

```
x[0] = 1.0;   // Zugriff auf das erste Feldelement
x[1] = x[0]; // das zweite
x[2] = x[1]; // und das letzte
```

D. h. die Größen `x[k]` verhalten sich wie jede andere Variable vom Typ `double`.

Syntax: **(Felddefinition)**

<FeldDef> ::= <Typ> <Name: > [<Anzahl>]

Erzeugt ein Feld mit dem Namen <Name: >, das <Anzahl> Elemente des Typs <Typ> enthält.

Bemerkung: Eine Felddefinition darf wie eine Variablendefinition verwendet werden.

Felder

Achtung: Bei der hier beschriebenen Felddefinition muss die Größe des Feldes zur **Übersetzungszeit bekannt sein**! Folgendes geht also **nicht**:

```
void f( int n )
{
  char s[n];
  ...
}
```

aber immerhin

```
const int n = 8;
char s[ 3*(n+1) ];
```

Vorsicht: Der GNU-C-Compiler erlaubt Felder variabler Größe als Spracherweiterung! Sie müssen die Optionen `-ansi -pedantic` verwenden, um für obiges Programm eine Fehlermeldung zu erhalten.

Sieb des Eratosthenes

Als Anwendung des Feldes betrachten wir eine Methode zur Erzeugung einer Liste von Primzahlen, die Sieb des Eratosthenes[15] genannt wird.

Idee: Wir nehmen eine Liste der natürlichen Zahlen größer 1 und streichen alle Vielfachen von $2, 3, 4, \ldots$. Alle Zahlen, die durch diesen Prozess $nicht$ erreicht werden, sind die gesuchten Primzahlen.

Bemerkung:

- Es genügt, nur die Vielfachen der Primzahlen zu nehmen (Primfaktorzerlegung).
- Um nachzuweisen, dass $N \in \mathbb{N}$ prim ist, reicht es, $k \nmid N$ (k ist kein Teiler von N) für alle Zahlen $k \in \mathbb{N}$ mit $k \leq \sqrt{N}$ zu testen.

[15] Eratosthenes von Kyrene, ca. 276–194 v. Chr., außergewöhnlich vielseitiger griechischer Gelehrter. Methode war damals schon bekannt, Eratosthenes hat nur den Begriff „Sieb“ geprägt.

Programm: **(Sieb des Eratosthenes [eratosthenes.cc])**

```
#include "fcpp.hh"

int main()
{
  const int n = 500000;
  bool prim[n];

  // Initialisierung
  prim[0] = false;
  prim[1] = false;
  for ( int i=2; i<n; i=i+1 )
    prim[i] = true;

  // Sieb
  for ( int i=2; i<=sqrt((double) n); i=i+1 )
```

```
    if ( prim[i] )
      for ( int j=2*i; j<n; j=j+i )
        prim[j] = false;

  // Ausgabe
  int m = 0;
  for ( int i=0; i<n; i=i+1 )
    if ( prim[i] )
      m = m+1;
  print( "Anzahl Primzahlen:" );
  print( m );

  return 0;
}
```

Bemerkung: Der Aufwand des Algorithmus lässt sich wie folgt abschätzen:

1. Der Aufwand der Initialisierung ist $\Theta(n)$.

2. Unter der Annahme einer „konstanten Primzahldichte“ erhalten wir

$$\text{Aufwand}(n) \leq C \sum_{k=2}^{\sqrt{n}} \frac{n}{k} = Cn \sum_{k=2}^{\sqrt{n}} \frac{1}{k} \leq Cn \int_1^{\sqrt{n}} \frac{dx}{x} = Cn \log \sqrt{n} = \frac{C}{2} n \log n$$

3. $O(n \log n)$ ist bereits eine fast optimale Abschätzung, da der Aufwand ja auch $\Omega(n)$ ist. Man kann die Ordnungsabschätzung daher nicht wesentlich verbessern, selbst wenn man zahlentheoretisches Wissen über die Primzahldichte hinzuziehen würde.

Bemerkung: Die Beziehung zur funktionalen Programmierung ist etwa folgende:

- Mit großen Feldern operierende Algorithmen kann man nur schlecht rein funktional darstellen.
- Dies ist vor allem eine Effizienzfrage, weil oft kleine Veränderungen großer Felder verlangt werden. Bei funktionaler Programmierung müsste man ein neues Feld erzeugen und als Rückgabewert verwenden.
- Algorithmen wie das Sieb des Eratosthenes formuliert man daher funktional auf andere Weise (Datenströme, Streams), was interessant ist, allerdings manchmal auch recht komplex wird. (Allerdings ist dieser Programmierstil auf neuen Prozessoren wie Grafikkarten interessant).

Zeichen und Zeichenketten

Datentyp `char`

- Zur Verarbeitung von einzelnen Zeichen gibt es den Datentyp `char`, der genau ein Zeichen aufnehmen kann:

```
char c = '%';
```

- Die Initialisierung benutzt die einfachen Anführungsstriche.

- Der Datentyp `char` ist kompatibel mit `int` (Zeichen entsprechen Zahlen im Bereich $-128 \ldots 127$). Man kann daher sogar mit ihm rechnen:

```
char c1 = 'a';
char c2 = 'b';
char c3 = c1+c2;
```

Normalerweise sollte man diese Eigenschaft aber nicht brauchen!

Zeichen und Zeichenketten

ASCII

Die den Zahlen $0 \ldots 127$ zugeordneten Zeichen nennt man den American Standard Code for Information Interchange oder kurz ASCII. Den druckbaren Zeichen entsprechen die Werte $32 \ldots 127$.

Programm: **(ASCII.cc)**

```
#include "fcpp.hh"

int main()
{
  for ( int i=32; i<=127; i=i+1 )
    print( i, (char) i, 0 );
}
```

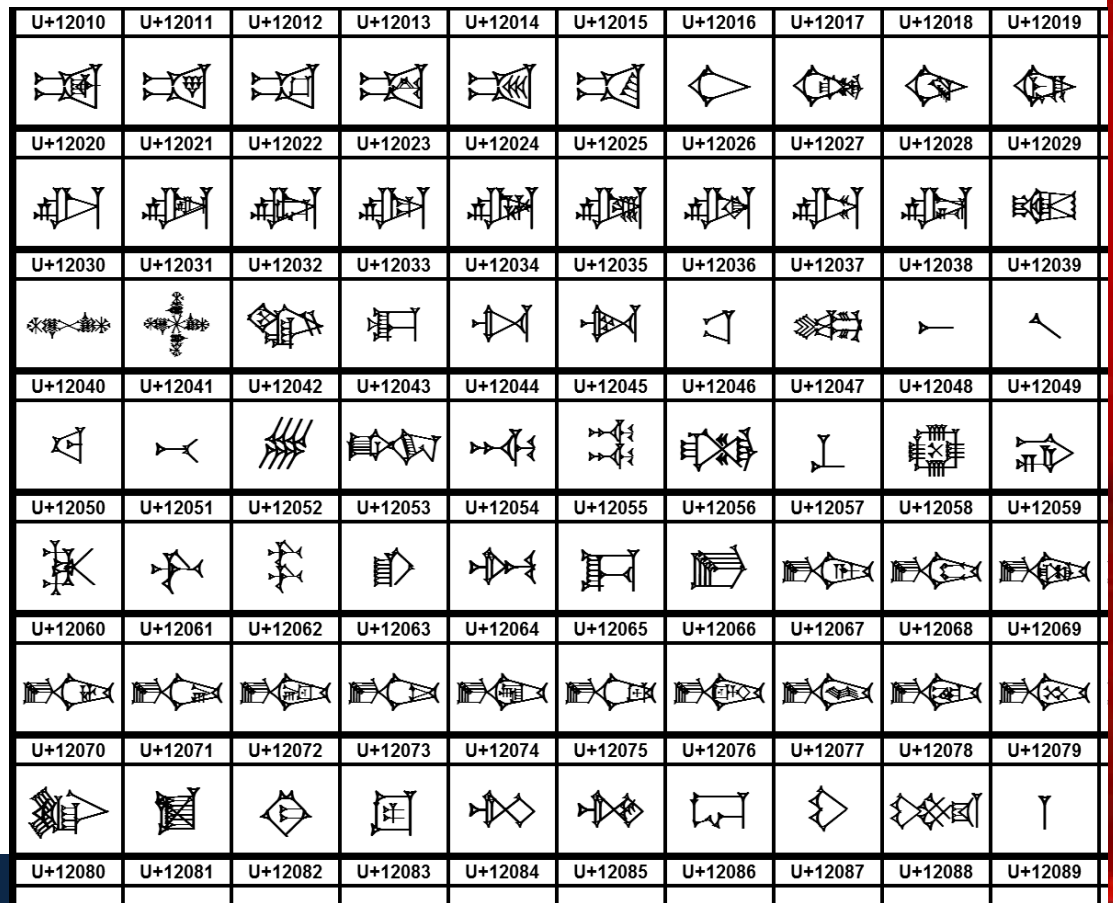
Zeichen und Zeichenketten

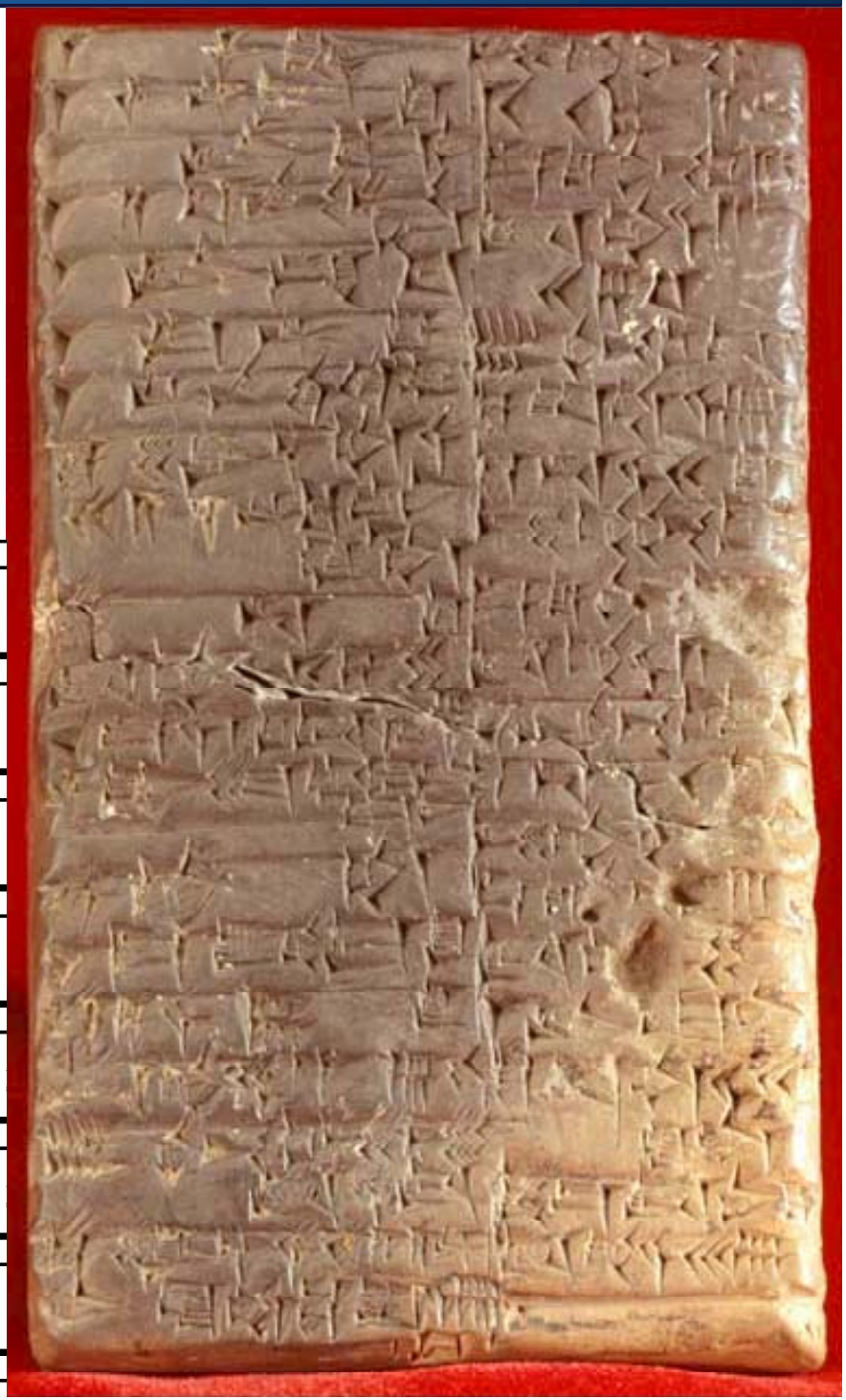
Bemerkung:

- Das dritte Argument von `print` ist der (ignorierte) Rückgabewert.
- Die Zeichen $0, \ldots, 31$ dienen Steuerzwecken wie Zeilenende, Papiervorschub, Piepston, etc.
- Für die negativen Werte $-128, \ldots, -1$ (entspricht $128, \ldots, 255$ bei vorzeichenlosen Zahlen) gibt es verschiedene Belegungstabellen (ISO 8859-n), mit denen man Zeichensätze und Sonderzeichen anderer Sprachen abdeckt.
- Noch komplizierter wird die Situation, wenn man Zeichensätze für Sprachen mit sehr vielen Zeichen (Chinesisch, Japanisch, etc) benötigt, oder wenn man mehrere Sprachen gleichzeitig behandeln will.
  Stichwort: Unicode.

UNICODE

Aktuell >135k Zeichen!





Zeichenketten

Zeichenketten realisiert man in C am einfachsten mittels einem `char`-Feld. Konstante Zeichenketten kann man mit doppelten Anführungsstrichen auch direkt im Programm eingeben.

Beispiel: Initialisierung eines `char`-Felds:

```
char c[10] = "Hallo";
```

Bemerkung: Das Feld muss groß genug sein, um die Zeichenkette samt einem Endezeichen (in C das Zeichen mit ASCII-Code 0) aufnehmen zu können. Diese feste Größe ist oft sehr unhandlich, und viele Sicherheitsprobleme entstehen aus der Verwendung von zu kurzen `char`-Feldern von unachtsamen C-Programmierern!

Programm: (Zeichenketten im C-Stil [Cstring.cc])

```
#include "fcpp.hh"

int main()
{
  char name[32] = "Alan Turing";

  for ( int i=0; name[i]!=0; i=i+1 )
    print( name[i] );          // einzelne Zeichen
  print( name );               // ganze Zeichenkette
}
```

String

In C++ gibt es einen Datentyp `string`, der sich besser zur Verarbeitung von Zeichenketten eignet als bloße `char`-Felder:

Programm: (Zeichenketten im C++-Stil [CCstring.cc])

```
#include "fcpp.hh"
#include <string>

int main()
{
   std::string vorname = "Alan";
   std::string nachname = "Turing";
   std::string name = vorname + " " + nachname;
   print( name );
}
```

Dies erfordert das einbinden der Header-Datei `string` mit dem **#include** Befehl.

Typedef

Mittels der `typedef`-Anweisung kann man einem bestehenden Datentyp einen neuen Namen geben.

Beispiel:

```
typedef int MyInteger;
```

Damit hat der Datentyp `int` auch den Namen `MyInteger` erhalten.

Bemerkung: `MyInteger` ist kein neuer Datentyp. Er darf synonym mit `int` verwendet werden:

```
MyInteger x = 4;    // ein MyInteger
int y = 3;          // ein int

x = y;              // Zuweisung OK, Typen identisch
```

Anwendung: Verschiedene Computerarchitekturen (Rechner/Compiler) verwenden unterschiedliche Größen etwa von `int`-Zahlen. Soll nun ein Programm portabel auf verschiedenen Architekturen laufen, so kann man es an kritischen Stellen mit `MyInteger` schreiben. `MyInteger` kann dann an einer Stelle architekturabhängig definiert werden.

Beispiel: Auch Feldtypen kann man einen neuen Namen geben:

```
typedef double Punkt3d[3];
```

Dann kann man bequem schreiben:

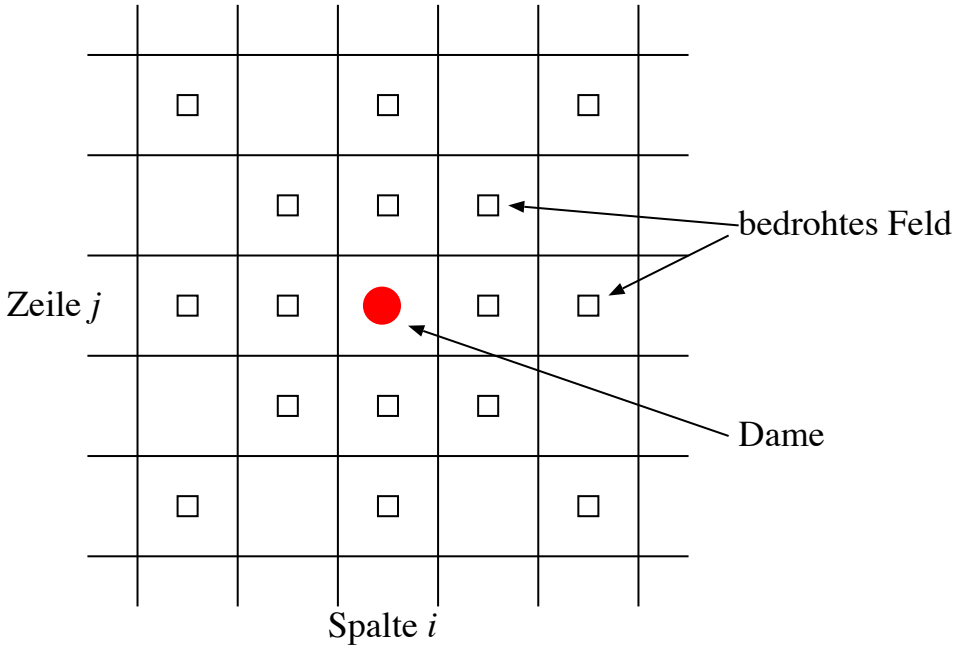
```
Punkt3d a, b;
a[0] = 0.0; a[1] = 1.0; a[2] = 2.0;
b[0] = 0.0; b[1] = 1.0; b[2] = 2.0;
```

Bemerkung: Ein Tipp zur Syntax: Man stelle sich eine Felddefinition vor und schreibt `typedef` davor.

Das Acht-Damen-Problem

Problem: Wie kann man acht Damen so auf einem Schachbrett positionieren, dass sie sich nicht gegenseitig schlagen können?

Zugmöglichkeiten der Dame: horizontal, vertikal, diagonal



Bemerkung:

- Ist daher die Dame an der Stelle (i, j), so bedroht sie alle (i', j') mit
  - $i = i'$ oder $j = j'$
  - $(i - i') = (j - j')$ oder $(i - i') = -(j - j')$
- Bei jeder Lösung steht in jeder Zeile/Spalte des Bretts genau eine Dame.

Idee:

- Man baut die Lösungen sukzessive auf, indem man erst in der ersten Zeile eine Dame platziert, dann in der zweiten, usw.
- Die Platzierung der ersten n Damen kann man durch ein `int`-Feld der Länge n beschreiben, wobei jede Komponente die Spaltenposition der Dame enthält.

Programm: (Acht-Damen-Problem [queens.cc])

```
#include "fcpp.hh"
#include <string>

const int board_size = 8;           // globale Konstante
typedef int columns[board_size];    // neuer Datentyp "columns"

bool good_position( int new_row, columns queen_cols, int new_col )
{
  for ( int row=0; row<new_row; row=row+1 )
    if ( ( queen_cols[row] == new_col ) ||
         ( new_row-row == abs( queen_cols[row]-new_col ) ) )
      return false;
  return true;
}

void display_board( columns queen_cols )
{
```

```
    for ( int r=0; r<board_size; r=r+1 )
    {
      std::string s( "" );
      for ( int c=0; c<board_size; c=c+1 )
        if ( c != queen_cols[r] )
          s = s + ".";
        else
          s = s + "D";
      print( s );
    }
    print( "␣" );
}

int queen_configs( int row, columns queen_cols )
{
    if ( row == board_size )
    {
      display_board( queen_cols );
      return 1;
```

```
    }
    else
    {
      int nr_configs = 0;
      for ( int col=0; col<board_size; col=col+1 )
        if ( good_position( row, queen_cols, col ) )
        {
          queen_cols[row] = col;
          nr_configs = nr_configs +
                       queen_configs( row+1, queen_cols );
        }
      return nr_configs;
    }
}

int main()
{
  columns queen_cols;
  print( "Anzahl Loesungen" );
  print( queen_configs( 0, queen_cols ) );
  return 0;
}
```

Bemerkung: Dieses Programm benutzt ein weiteres neues Element:

- Es wurde eine globale Konstante `board_size` verwendet. Diese kann innerhalb aller Funktionen benutzt werden.
- Auch die Typdefinition kann innerhalb aller Funktionen benutzt werden.
- Ein Feld wird als Argument an eine Funktion übergeben. Solche Argumente werden **nicht** kopiert (!). Dazu später mehr. Solange man Argumente nur liest oder nur neue Feldelemente beschreibt (wie hier) kann man sich auch vorstellen das Argument würde kopiert werden.
- Daher bis auf weiteres: Vorsicht bei der Benutzung von Feldern als Argumente von Funktionen!

Bemerkung:

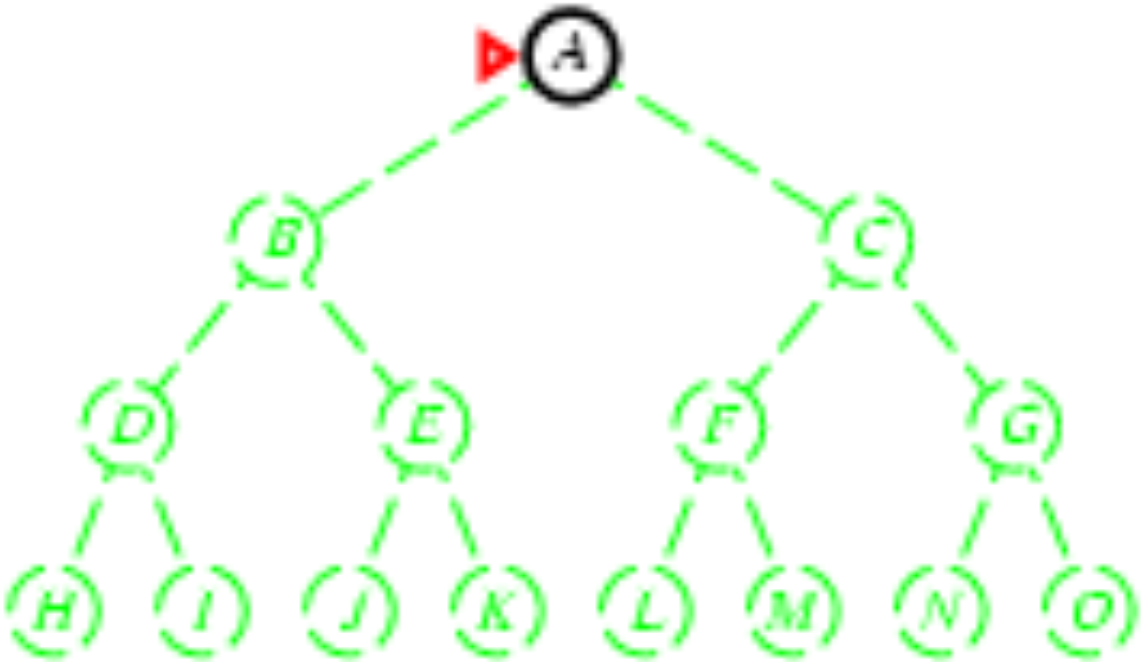
- Dieses Verfahren des Ausprobierens verschiedener Möglichkeiten durch eine sogenannte Tiefensuche in einem Baum ist als Backtracking bekannt.

- Für $n = 8$ gibt es 92 Lösungen. Eine davon ist

```
. . . . . . . D
. . . D . . . .
D . . . . . . .
. . D . . . . .
. . . . . D . .
. D . . . . . .
. . . . . . D .
. . . . D . . .
```

Depth-first search

- Expand deepest unexpanded node



Wiederholung

- Feld (array) als wesentliches Konstruktionselement für neue Datentypen
- Zeichenketten sind Felder über dem Typ **char**. C-Variante und C++-Variante.
- **typedef** zur Einführung neuer Namen für Datentypen (kein neuer Datentyp!)
- Acht-Damen-Problem als Beispiel für die Anwendung eines Felds:
  - Bisheriger Zustand der Berechnung wird in einem Feld zusammengefasst
  - Zustandsmenge wächst im Laufe der Rechnung

Benutzerdefinierte Datentypen

- Aufzählungstyp
- Felder
- Zeichen und Zeichenketten
- Typedef
- Das Acht-Damen-Problem
- Zusammengesetzte Datentypen

Zusammengesetze Datentypen

Bei zusammengesetzten Datentypen kann man eine beliebige Anzahl möglicherweise verschiedener (sogar zusammengesetzter) Datentypen zu einem neuen Datentyp kombinieren. Diese Art Datentypen nennt man Strukturen.

Beispiel: Aus zwei `int`-Zahlen erhalten wir die Struktur `Rational`:

```
struct Rational {  // Schluesselwort struct
  int zaehler;     // eine Liste von
  int nenner;      // Variablendefinitionen
};                 // Semikolon nicht vergessen
```

Dieser Datentyp kann nun wie folgt verwendet werden

```
Rational p;          // Definition einer Variablen
p.zaehler = 3;       // Initialisierung der Komponenten
p.nenner = 4;
```

Syntax: (Zusammengesetzter Datentyp)

$$\begin{array}{lll} \langle \text{StructDef} \rangle & ::= & \underline{\text{struct}}\ \langle \text{Name} \rangle\ \underline{\{}\ \{\ \langle \text{Komponente} \rangle\ \underline{;}\ \}^{+}\ \underline{\}}\ \underline{;} \\ \langle \text{Komponente} \rangle & ::= & \langle \text{VarDef} \rangle \mid \langle \text{FeldDef} \rangle \mid \ldots \end{array}$$

Eine Komponente ist entweder eine Variablendefinition ohne Initialisierung oder eine Felddefinition. Dabei kann der Typ der Komponente selbst zusammengesetzt sein.

Bemerkung: Strukturen sind ein Spezialfall von sehr viel mächtigeren Klassen, die später im OO-Teil behandelt werden.

Bemerkung: Im Gegensatz zu Feldern kann man mit Strukturen (zusammengesetzten Daten) gut funktional arbeiten, siehe etwa Abelson&Sussman: *Structure and Interpretation of Computer Programs*.

Anwendung: Rationale Zahlen

Programm: (Rationale Zahlen, die erste [Rational2.cc])

```
#include "fcpp.hh"

struct Rational {
  int zaehler;
  int nenner;
};

// Abstraktion: Konstruktor und Selektoren

Rational erzeuge_rat( int z, int n )
{
  Rational t;
  t.zaehler = z;
  t.nenner = n;
  return t;
```

```
}

int zaehler( Rational q )
{
  return q.zaehler;
}

int nenner( Rational q )
{
  return q.nenner;
}

// Arithmetische Operationen

Rational add_rat( Rational p, Rational q )
{
  return erzeuge_rat( zaehler(p)*nenner(q) +
                      zaehler(q)*nenner(p),
                      nenner(p)*nenner(q) );
```

```
}

Rational sub_rat( Rational p, Rational q )
{
  return erzeuge_rat( zaehler(p)*nenner(q) -
                      zaehler(q)*nenner(p),
                      nenner(p)*nenner(q) );
}

Rational mul_rat( Rational p, Rational q )
{
  return erzeuge_rat( zaehler(p)*zaehler(q),
                      nenner(p)*nenner(q) );
}

Rational div_rat( Rational p, Rational q )
{
  return erzeuge_rat( zaehler(p)*nenner(q),
                      nenner(p)*zaehler(q) );
```

```
}

void drucke_rat( Rational p )
{
  print( zaehler(p), "/", nenner(p), 0 );
}

int main()
{
  Rational p = erzeuge_rat( 3, 4 );
  Rational q = erzeuge_rat( 5, 3 );
  drucke_rat( p ); drucke_rat( q );

  // p*q + p - p*p
  Rational r = sub_rat( add_rat( mul_rat( p, q ), p ),
                        mul_rat( p, p ) );
  drucke_rat( r );
  return 0;
}
```

Bemerkung: Man beachte die Abstraktionsschicht, die wir durch den Konstruktor `erzeuge_rat` und die Selektoren `zaehler` und `nenner` eingeführt haben. Diese Schicht stellt die sogenannte Schnittstelle dar, über die unser Datentyp verwendet werden soll.

Problem: Noch ist keine Kürzung im Programm eingebaut. So liefert das obige Programm $1104/768$ anstatt $23/16$.

Abhilfe: Normalisierung im Konstruktor:

```
Rational erzeuge_rat( int z, int n )
{
  int g;
  Rational t;

  if ( n < 0 ) { n = -n; z = -z; }
  g = ggT( std::abs( z ), n );
  t.zaehler = z / g;
  t.nenner = n / g;
  return t;
}
```

Bemerkung: Ohne die Verwendung des Konstruktors hätten wir in allen arithmetischen Funktionen Änderungen durchführen müssen, um das Ergebnis in Normalform zu bringen.

Komplexe Zahlen

Analog lassen sich komplexe Zahlen einführen:

Programm: (Komplexe Zahlen, Version 1 [Complex2.cc])

```
#include "fcpp.hh"

struct Complex
{
  float real;
  float imag;
};

Complex erzeuge_complex( float re, float im )
{
  Complex t;
  t.real = re; t.imag = im;
  return t;
}
```

```
float real( Complex q ) { return q.real; }
float imag( Complex q ) { return q.imag; }

Complex add_complex( Complex p, Complex q )
{
  return erzeuge_complex( real(p) + real(q),
                          imag(p) + imag(q) );
}

// etc

void drucke_complex( Complex p )
{
  print( real(p), "+i*", imag(p), 0 );
}

int main()
{
```

```
    Complex p = erzeuge_complex( 3.0, 4.0 );
    Complex q = erzeuge_complex( 5.0, 3.0 );
    drucke_complex( p );
    drucke_complex( q );
    drucke_complex( add_complex( p, q ) );
}
```

Bemerkung: Hier wäre bei Verwendung der Funktionen `real` und `imag` zum Beispiel auch eine Änderung der internen Darstellung zu Betrag/Argument ohne Änderung der Schnittstelle möglich.

Gemischtzahlige Arithmetik

Problem: Was ist, wenn man mit komplexen und rationalen Zahlen gleichzeitig rechnen will?

Antwort: Eine Möglichkeit, die bereits die Sprache C bietet, ist die folgende:

1. Führe eine sogenannte variante Struktur (Schlüsselwort `union`) ein, die entweder eine rationale oder eine komplexe Zahl enthalten kann $\rightsquigarrow$ neuer Datentyp `Combination`

2. Füge auch eine Kennzeichnung hinzu, um was für eine Zahl (rational/komplex) es sich tatsächlich handelt $\rightsquigarrow$ neuer Datentyp `Mixed`.

3. Funktionen wie `add_mixed` prüfen die Kennzeichnung, konvertieren bei Bedarf und rufen dann `add_rat` bzw. `add_complex` auf.

```
struct Rational { int n; int d; };

struct Complex { float re; float im; };

union Combination   // entweder oder!
{
   Rational p;
   Complex c;
};

enum Kind { rational, complex };

struct Mixed          // gemischte Zahl
{
   Kind a;            // welche bist Du?
   Combination com;   // benutze je nach Art
};
```

Bemerkung: Diese Lösung hat etliche Probleme:

- Umständlich und unsicher (überschreiben)
- Das Hinzufügen weiterer Zahlentypen macht eine Änderung von bestehenden Funktionen nötig
- Typprüfungen zur Laufzeit $\rightarrow$ keine optimale Effizienz
- Speicherplatzbedarf der größten Komponente
- Hätten gerne: Infix-Notation mit unseren Zahlen

Bemerkung: Einige dieser Probleme werden wir mit den objektorientierten Erweiterungen von C++ vermeiden. Man sollte sich allerdings auch klar machen, dass das Problem von Arithmetik mit verschiedenen Zahltypen und eventuell auch verschiedenen Genauigkeiten tatsächlich extrem komplex ist. Eine vollkommene Lösung darf man daher nicht erwarten.