Programmiermethoden in der Mathematik WS 02/03
Zeiger (Pointer) und dynamische Speicherverwaltung

- Jede auf die bisher bekannte Weise deklarierte Variable und Funktion in einem Programm ist an einer Stelle im Speicher abgelegt, die zur Compilezeit eindeutig festgelegt wird (sie ist *statisch*). In C++ kann man auf diese **Adresse** zugreifen. Damit ist es möglich
  - hardwarenah zu programmieren
  - (in C, das kein *call by reference* bei Funktionen kennt) Variablenwerte in Funktionen zu verändern, indem man die Adressen der Variablen übergibt
- Für eine statische Variable

  ```
  double x;
  ```

  erhält man mit dem *Referenz- oder Adressenoperator* `&`

  ```
  xp = &x;
  ```

  ihre **Adresse**. Man nennt `xp` den **Zeiger** oder engl. **Pointer** auf `x`. Wir wissen, dass in C++ jede Variable deklariert werden muss, bevor sie benutzt werden kann. Welchen Typ hat nun ein Zeiger und wie wird er deklariert? Dazu dient der sog. *Inhaltsoperator* `*`, im Beispiel oben:

  ```
  double* xp;   oder   double *xp;   oder   double * xp;
  ```

  Das kann man lesen als: "`xp` ist ein Zeiger auf ein `double`" bzw. "Inhalt von `xp` ist ein `double`". Zu beachten ist, das sich der Operator `*` nur auf die folgende Variable bezieht: Nach der Deklaration

  ```
  double* x,y;   oder   double *x,y;   oder   double * x,y;
  ```

  ist nur `x` ein Zeiger auf ein `double`, `y` ist dagegen selbst eine `double`-Variable. Daher ist die jeweils mittlere Schreibweise die klarste. Nach der Deklaration ist ein Zeiger wie jede Variable in C++ zunächst undefiniert, er zeigt *nirgendwohin* und hat den Wert `NULL` oder `nil`. Ihn kann man auch benutzen, wenn man einen Zeiger nirgendwohin zeigen lassen will, weil man ihn nicht mehr benutzen will:

  ```
  double *xp;
  xp = NULL;
  ```

- **Pointerarithmetik:** In C/C++ kann man mit Zeigern rechnen, man kann z.B. einen Zeiger erhöhen:

  ```
  double x, *xp;
  xp=&x;           // oder: *xp=x;
  xp++;
  ```

  Normalerweise macht das wenig Sinn, da nicht bekannt ist, was hinter `x`, also an der Adresse `xp+1` im Speicher liegt. Anders ist das bei Feldern: Nach

  ```
  double x[n], *xp;
  xp=&x[0];       // oder: *xp=x[0];
  xp++;
  ```

  zeigt `xp` auf die nächste Komponente `x[1]`. Damit kann man eine Schleife

  ```
  double x[n];
  for(i=0;...;i++)  x[i]=...;
  ```

  auch schreiben als

  ```
  double x[n], *xp;
  xp=&x[0];
  while (...)
  {
      *xp=...;
      xp++;
  }
  ```

  Bei mehrdimensionalen Arrays kann das vorteilhaft sein, denn mehrdimensionale Arrays werden im Speicher hintereinander, d.h. als eindimensionales Feld abgelegt. Dabei zählt der letzte Index zuerst. Sind die Schleifen geschickt angeordnet (vgl. unter Arrays), so macht das der Compiler aber ohnehin.

- Ein Array an sich (ohne Indizierung, bezeichnet mit `x` oder `x[]`) ist in C++ eigentlich nichts anderes als ein Zeiger auf seine erste Komponente. Übergibt man es an eine Funktion, so wird nur der Zeiger weitergereicht: Nach der Variablendeklaration

  ```
  double x[n];
  ```

  wird bei dem Aufruf

  ```
  f(x);
  ```

  nur der Zeiger auf `x` (bzw. `x[0]`), übergeben. Zugriffe der Form

  ```
  x[i];
  ```

  werden immer, also auch innnerhalb der Funktion über Zeigerarithmetik (s.o.) realisiert. Daher sind nach dem Aufruf von `f` innnerhalb der Funktion durchgeführte Änderungen am Feld `x` auch außerhalb sichtbar. Ein *call by reference* ist daher für Arrays nicht sinnvoll und führt zu einer Compiler-Fehlermeldung. Daher sieht ein Funktionskopf bzw. die Signatur einer Funktion,die ein Array als Parameter hat, wie folgt aus:

  `void f(double *x);` oder `void f(double x[]);`

- Eine wesentliche Anwendung von Pointern ist, dass man sie als Komponenten in Strukturen verwenden kann, denn sie können dabei wieder auf eine Struktur desselben Typs zeigen:

  ```
  struct T
  {
      double x;
      T *tp;
  };
  ```

  Damit kann man geordnete Listen erstellen, in denen ein Element immer auf seinen Nachfolger zeigt. Durch entsprechende Zeigermanipulationen kann man solche Listen jederzeit beliebig umordnen. Dabei gibt es noch folgende alternative Bezeichnung:

  ```
  T* S;
  S->x    ist dasselbe wie    (*S).x
  ```

- Ebenfalls wichtig sind Zeiger bei der **dynamischen Speicherverwaltung**. Damit kann man z.B. Arrays variabler Länge benutzen, bei denen erst zur Laufzeit des Programms bekannt ist, wieviele Komponenten tatsächlich benötigt werden. Solche Variablen heißen *dynamisch*. Das Anfordern von Speicher geschieht mit dem Befehl `new`:

  ```
  double* x;           // Pointer deklarieren
  n=...;               // ben"otigten Speicherplatz festlegen
  x = new double[n];   // Speicherplatz anfordern
  ```

  oder zusammen

  ```
  double* x = new double[n];
  ```

  Mehrdimensionale Felder ergeben sich durch die Deklaration eines Feldes von Pointern

  ```
  double** x;              // Pointer auf Pointer auf doubles
  x = new double*[n];      // Speicherplatz f"ur n Pointer auf doubles
  ```

  oder zusammen

  ```
  double** x = new double*[n];
  ```

  und dann

  ```
  for(i=0;i<n;i++) x[i]=new double[m]; // n mal Speicherplatz f"ur je m doubles
  ```

  Der Zugriff auf dynamische Variablen erfolgt genauso wie auf statische, bei einem Array also mit

  `x[i]=...;` bzw. `x[i][j]=...;`

  Bei dynamischen Variablen wird also *erst zur Laufzeit des Programms* Speicherplatz angefordert. Er kann und sollte später auch wieder freigegeben werden, und zwar durch

  ```
  delete [] x;
  ```

  bei Feldern wie im obigen Beispiel, bei skalaren Variablen einfach durch

  ```
  delete x;
  ```