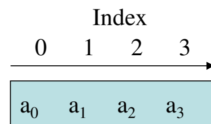
Algorithmen und Datenstrukturen

- Reihentyp
  - Benutzung von ein- und zweidimensionalen "Arrays" – Zeichenketten
- Modularisierung
  - Funktionen und Unterprogramme
- Satztyp
  - Aufbau von und Zugriff auf "Records"
- Spezielle Algorithmen
  - Sortieralgorithmen

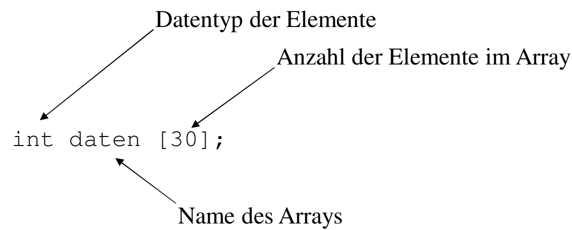
Arrays und Zeichenketten

- Beispiel: Ein Programm liest 100 Zahlen ein und gibt diese wieder in umgekehrter Reihenfolge aus.
- Mit dem derzeitigen Kenntnisstand benötigt man 100 Variablen, um die Zahlen einzeln einzulesen und wieder auszugeben.
- Es könnte noch nicht einmal eine Schleife verwendet werden, da es keinen Datentyp gibt 100 Zahlen aufzunehmen.
- Daher gibt es den "Reihentyp" Array (der oder das Array).
- Arrays sind Reihungen von Datenelementen gleichen Typs.
- Über einen Index kann auf jedes Datenelement unmittelbar zugegriffen werden.
- Der einfachste Fall ist eine eindimensionale, lineare Anordnung der Datenelemente.

Arrays



Notation für Arrays:




```
int daten [30];
```

Name des Arrays

Arrays

- Als Datentyp können alle bekannten Datentypen (char, short, unsigned long, float, …) verwendet werden.
- Das einzelne Elemente des Arrays wird über seinen Index (= Position im Array) angesprochen.
  ```
  daten [7] = 123;
  ```
- Es kann dann wie eine einzelne Variable des entsprechenden Typs verwendet werden.
- Die Nummerierung erfolgt immer beginnend mit 0.
- Das Array im obigen Beispiel hat also 30 Elemente, die von 0 bis 29 nummeriert sind.
- Das erste Element hat den Index 0, das zweite den Index 1, und das dreißigste hat den Index 29.

Arrays

- Die Verwendung falscher Indexwerte für den Zugriff auf Array-Elemente ist eine der Hauptfehlerquellen in C-Programmen. Der C-Compiler überprüft nicht, ob die im Programm verwendeten Indices im gültigen Bereich liegen.
- Beispiel: Die Anweisung `daten[50] = 2345;` passiert ohne Fehlermeldung den Compiler.
- Erst zur Laufzeit wird dann auf Datenbereiche außerhalb des Arrays zugegriffen, was in der Regel zu einem unkontrollierten Fehlverhalten des Programms führt.
- Es liegt also ausschließlich in der Verantwortung des Programmierers, dafür zu sorgen, dass Bereichsgrenzen nicht über- oder unterschritten werden. Insbesondere schreibender Zugriff außerhalb von zulässigen Grenzen kann zu schwerwiegenden Programmfehlern führen.
- Darstellung im Speicher !!!

Arrays

- Arrays können direkt bei der Definition mit Werten belegt werden. Die Werte werden in geschweiften Klammern und durch Kommata getrennt angegeben.

```
int daten[30] = { 3, 4, 5, 6, 7};
```

- Es ist unproblematisch, dass dabei unter Umständen nicht alle Felder besetzt werden. Der Compiler füllt das Array von vorn beginnend.
- Nicht angesprochene Felder bleiben uninitialisiert !!!

Arrays

- Verwendet werden die einzelnen Elemente eines Arrays wie eine Variable des entsprechenden Datentyps

```
int daten[30] = { 3, 4, 5, 6, 7 };
daten[5] = 9;
if ( 3 == daten[2] )
{
    daten[0] = daten[1];
}
else
{
    daten[6] = daten[1];
}
daten[8] = daten[8] + 1;    // Achtung !!!!
```


Arrays

- Die besondere Qualität von Arrays für die Programmierung liegt darin, dass die Indices für den Zugriff nicht fest vorgegebenen sein müssen. Sie können über Variablen zur Laufzeit berechnet werden:

```
int daten[30];
int index;

index = 3;
daten[index] = 4;
daten[2*index + 4] = daten[index-1] + 2;
```

- In Verbindung mit Zählschleifen (z.B. for-Schleifen) ergeben sich dann vielfältige Verarbeitungsmöglichkeiten für Arrays.
- Anwendung auf das Eingangsproblem (100 Zahlen einlesen und in umgekehrter Reihenfolge wieder ausgeben).

Arrays

```
int main ( )
  {
    int daten[100];
    int i;

    for ( i = 0; i < 100; i++ )
    {
      cout << "gib die " << i << ".te Zahl ein: ";
      cin >> daten[i];
    }
    for ( i = 99; i >= 0; i-- )
    {
      cout << "Die " << i << ".te Zahl ist ";
      cout << daten[i] << endl;
    }
    return 0;
  }
```

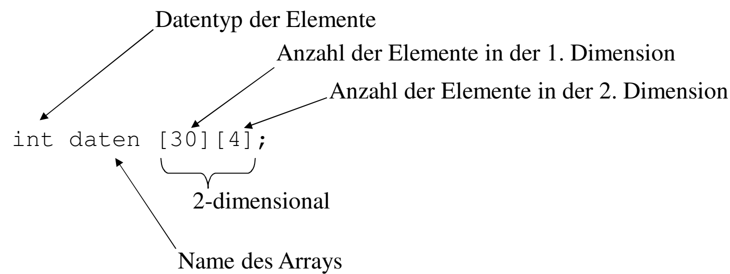

Arrays

- Die Verwendung von Arrays beschränkt sich nicht auf den eindimensionalen Fall.
- Auch 2, 3 und mehr Dimensionen sind möglich.
- Zunächst aber der zweidimensionale Fall:

Index	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}

Arrays

Bei der Definition muss jetzt natürlich angegeben werden, wie viele Dimensionen und welche "Ausdehnung" der Array in jeder Dimension haben soll:



Arrays

- Zum Zugriff auf konkrete Elemente des Arrays benötigt man jetzt natürlich für jede Dimension einen separaten Index:

```
int daten[3][4];
int i, k;
for ( i = 0; i < 3; i++ )
{
  for ( k = 0; k < 4; k++ )
  {
    daten[i][k] = i + k;
  }
}
```

- Das Beispiel besetzt den 2-dimensionalen Array `daten` in der folgenden Weise:

	Index k 0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

Arrays

- Auch 2-dimensionale Arrays können direkt bei der Definition mit Werten belegt werden.
- Im folgenden Beispiel wird der Array `daten` mit den gleichen Werten initialisiert, wie zuvor in der Doppelschleife berechnet.

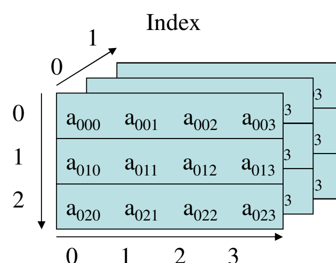
```
int daten[3][4] = {
                    { 0, 1, 2, 3 }
                    { 1, 2, 3, 4 }
                    { 2, 3, 4, 5 }
                  };
```

- Was die Bereichsgrenzen und insbesondere die Probleme mit Bereichsgrenzen betrifft, gilt jetzt in jeder Dimension das, was in den eindimensionalen Arrays festgestellt wurde.
- Die Gefahren und Probleme haben sich also verdoppelt.

Dirk Seeber, Informatik , Teil 4 13

Arrays

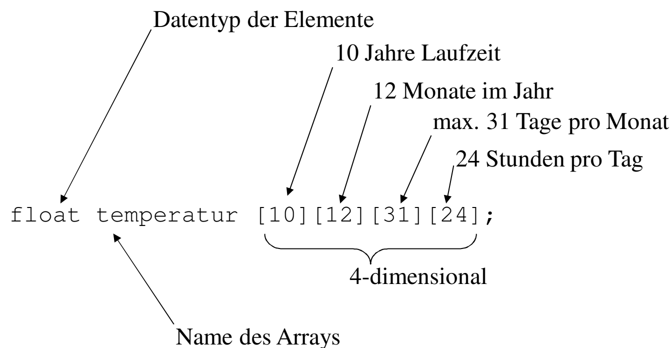
- Höherdimensionale Fälle (3, 4, 5, …) bringen nichts wirklich Neues mehr. Der 2-dimensionale Fall ist sinngemäß zu übertragen.
- Für den 3-dimensionalen Fall gibt es eine Veranschaulichung durch Anordnung der Elemente im Raum.



Dirk Seeber, Informatik , Teil 4 14

Arrays

- Obwohl es für Arrays von mehr als 3 Dimensionen keine "räumlichen" Veranschaulichung mehr gibt, sollte man nicht denken, dass solche Arrays in praktischen Anwendungen nicht vorkommen.
- Wenn man beispielsweise über einen Zeitraum von 10 Jahren stündlich Temperaturmesswerte erfassen und in einem Array speichern will, so kann man ein 4-dimensionales Array verwenden.



Arrays

- Die Temperatur am 24. Dezember im dritten Jahr der Temperaturaufzeichnung um acht Uhr morgens erhält man durch folgende Zugriff:
  `gesuchte_temp = temperatur[2][11][23][7];`

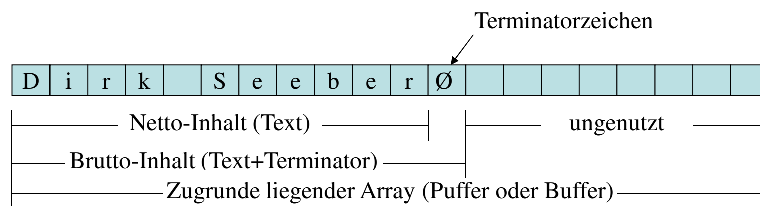
- Zu Bedenken ist allerdings, dass unter der Annahme, dass eine `double`-Zahl **8** Byte belegt, der Netto-Inhalt dieses Arrays 10 * 12 * 31 * 24 * 8 = 714.240 Byte beträgt.
- Das bedeutet, dass auf jeden Fall diese Bytes im Speicher belegt sind.

Zeichenketten

- Der Begriff Computer oder Rechner legt nahe, dass sich die Datenverarbeitung vornehmlich mit numerischen Problemen beschäftigt.
- Das war in den Anfängen auch richtig.
- Heute ist aber nur noch ein sehr kleiner Teil der Computer-anwendungen numerischer Natur.
- Von daher ist die Verarbeitung von Zeichen und Zeichenketten (Strings) von großer Bedeutung.
- Zum Glück haben wir eine Schriftsprache, die auf sehr wenige Grundelemente (Buchstaben, Zeichen) basiert. Andere Schriftsysteme (z.B. Japanisch oder Chinesisch) sind sehr viel komplexer angelegt, und entsprechend aufwändig ist dort die Speicherung und Verarbeitung von Texten.

Zeichenketten

- Zeichenketten (Strings) sind Reihungen von Zeichen.
- Es ist nahe liegend, einen String in einem Array abzulegen.
- Um den String Raum für Veränderungen zu lassen, ist der Array in der Regel um einiges größer als der eigentliche String.
- Um das Ende eines String zu markieren, wird der String durch ein Terminatorzeichen abgeschlossen. In C dient die Null (eine richtige Null, nicht das Zeichen "0") als Kennung für das String-Ende (EOS).



Zeichenketten

- Grundsätzlich ist ein String ein eindimensionaler Array von Zeichen (`char`) mit der zusätzlichen Eigenschaft, dass das letzte Zeichen `NULL` ist.
- Strings können also wie zuvor beschrieben im Kapitel über Arrays bearbeitet werden.
- Folgende Warnungen:
  - Der String befindet sich in einem Array fester Länge. Der Programmierer muss darauf achten, dass bei Manipulation des Strings (z.B. Anfügen von Buchstaben) die Grenzen des Arrays nicht überschritten werden.
  - Wegen des Terminators muss der Array mindestens ein Element mehr haben als der String Zeichen enthält.
  - Der String muss nach eventuellen Manipulationen immer konsistent sein. Insbesondere bedeutet das, dass das Terminatorzeichen korrekt positioniert werden muss.

Zeichenketten

- Um einen String verwenden zu können, legt man zunächst einen Array an, der groß genug sein muss, den zu erwartenden Text aufzunehmen:

```
int main ( )
{
  char vorname[21];  /* 20 Zeichen + Terminator */
  char nachname[21]; /* 20 Zeichen + Terminator */
  cout << "Bitte Nachname eingeben: ";
  cin >> nachname;
  cout << "Bitte Vorname eingeben: ";
  cin >> vorname;
  return 0;
}
```

- **ACHTUNG:** Es wird dabei nicht überprüft, ob der Array groß genug ist, um den String aufzunehmen.
- Werden im obigen Beispiel mehr als 20 Zeichen eingegeben, so wird rücksichtslos außerhalb des Arrays geschrieben.

Zeichenkette

- In C++ existiert eine weitere Möglichkeit Zeichenketten zu verarbeiten, die Standardklasse **`string`**.
- Strings sind hier dynamische Objekte, d.h. der erforderliche Speicherplatz wird automatisch allokiert/belegt.
- Es kommt nicht zu einer Speicherverletzung, falls der String zu groß wird
  (anders als z.B. char zeichenkette[20]).

Zeichenketten

- Um diesen „neuen Datentyp“ (Standardklasse) verwenden zu können, muss folgende Datei zusätzlich zur Datei *iostream* „inkludiert“ werden.

```
#include <string>
```

- Dann kann man diesen Datentyp **string** verwenden:

```
int main ( )
{
  string vorname, nachname, kuerzel, zusammen;
  cout << "Bitte Nachname eingeben: "; cin >> nachname;
  cout << "Bitte Vorname eingeben: ";  cin >> vorname;
  kuerzel = vorname[0];
  kuerzel += nachname[0];
  zusammen = nachname + ", " + vorname;
  cout << ">" << vorname << "<   >" << nachname;
  cout << "< " << kuerzel << " " << zusammen << endl;
  return 0;
}
```

Zeichenketten

- Auch in Zeichenketten werden ESCAPE-Sequenzen verwendet, um nicht druckbare bzw. mit einer Sonderbedeutung belegten Zeichen einzubauen:
- Beispiele:

```
cout << "\tAlles\tklar!\n";
cout << "Ein \"String\" in der Zeichenkette." << endl;
cout << "Dies ist ein Backslash: \\" << endl;
cout << "Dies ist ein \'A\': dies auch \101: und dies \x41");
```

- Beim Arbeiten mit Zeichenketten ist folgendes zu beachten:

```
– char weiter = 'A';

– #include <string>
  string zeichenkette;
  zeichenkette = "Dirk Seeber";
```

- Dies ist mehr als eine Spitzfindigkeit, da Zeichen bzw. Zeichenketten im Rechner unterschiedlich dargestellt werden und demnach auch unterschiedlich zu verarbeiten sind.

Zeichenkette

- Weiteres Beispiel
  – Operationen auf Zeichenkette als String:

```
#include <iostream>
#include <string>
using namespace std;
int main ( )
{
  long i;
  string einString = "hallo";
  // String zeichenweise ausgeben
  for ( i = 0; i < einString.length(); i++ )
  {
    cout << einString.at(i) << " ";
  }
  cout << endl;
  return 0;
}
```

Modularisierung

- Programme können mit wachsender Komplexität unübersichtlich und unverständlich werden.
- Die Frage ist daher:
  - Wie kann man umfangreichere Programme noch handhabbar halten?
- Die Antwort ist nahe liegend:
  - Man muss ein umfangreiches Programm in kleinere, jeweils noch überschaubare Einzelteile zerlegen und diese Einzelteile möglichst unabhängig voneinander entwickeln.
- Diese Vorgehensart bezeichnet man als **Modularisierung**.
- Modularisierung wird in C/C++ durch Unterprogramme und Funktionen unterstützt.

Funktionen und Unterprogramme

- Hinter dem Funktionsbegriff verbirgt sich das wesentliche Modularisierungskonzept von Programmiersprachen. Ein Programmteil nimmt die Dienstleistung eines anderen Programmteils in Anspruch, ohne mit Informationen darüber, **wie** der andere diesen Dienst ausführt, belastet zu sein. Das gerufene Programm führt die Dienstleistung aus, ohne störende Kenntnis darüber, **warum** oder **wofür** der Partner diese Dienstleistung benötigt.
- Klare Aufgabentrennung zwischen Auftraggeber und Auftragnehmer.
- Die Informationen, die zwischen beiden Parteien noch fließen muss, um die Dienstleistung korrekt zu erbringen, wird über die Schnittstelle ausgetauscht.

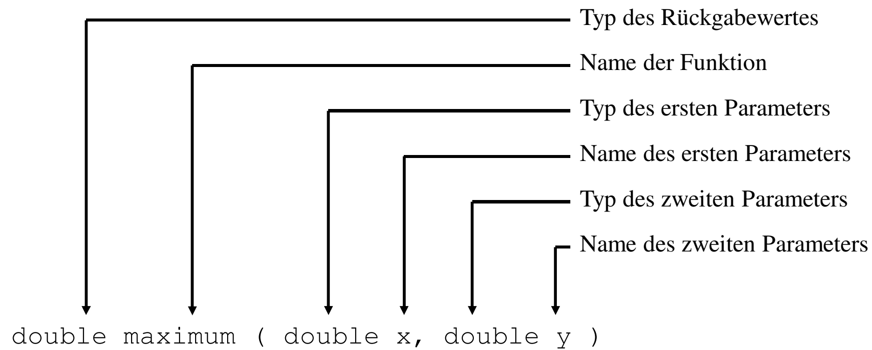
Funktionen und Unterprogramme

- Durch die Aufteilung zwischen Haupt- und Unterprogramm erhält man eine Trennung zwischen WIE und WARUM.
- Das Unterprogramm weiß, WIE etwas gemacht wird, aber nicht WARUM.
- Das Hauptprogramm weiß, WARUM etwas gemacht wird, aber nicht WIE.
- Man kann sich auf seine Aufgabe konzentrieren und ist nicht mit überflüssigem Wissen über die jeweils andere Seite belastet.
- Diese Technik ermöglicht es, größere Programme noch beherrschbar zu halten.
- Große Programme zu modularisieren, d.h. in kleine, überschaubare funktionale Einheiten aufzuteilen und mit geeigneten Schnittstellen zu versehen, ist eine zentrale Aufgabe des Programmdesigns.

Funktionen und Unterprogramme

- Der sichere Umgang mit dieser Technik ist eine der wichtigsten Fähigkeiten, die einen guten Software-Entwickler auszeichnen.
- Zu einer Funktion in C gehören folgende Dinge:
  - Ein **Funktionsprototyp**, der die Schnittstellenvereinbarung zwischen Haupt- und Unterprogramm darstellt.
  - die **Implementierung**, in der die Funktion konkret ausprogrammiert wird.
- Beispiel: Das Maximum zweier Zahlen muss innerhalb eines Programms mehrmals bestimmt werden. Diese Berechnung soll eine Funktion übernehmen. Auch die Schnittstelle ist klar
  - In die Funktion gehen zwei Gleitkommazahlen x und y hinein und als Ergebnis kommt die größere der beiden Zahlen wieder heraus.
- Der Name der Funktion ist maximum.
- Daraus folgt:

Funktionen und Unterprogramme



Dirk Seeber, Informatik , Teil 4 29

Funktionen und Unterprogramme

- Die Funktion wird realisiert, indem man an die Definition der Schnittstelle den Funktionskörper als Block anhängt.
- In diesem Block können die Parameter wie gewöhnliche Variablen des entsprechenden Typs benutzt werden.

```
double maximum ( double x, double y )
{
  double rueckgabe_wert;
  if ( x > y )
  {
    rueckgabe_wert = x;
  }
  else
  {
    rueckgabe_wert = y;
  }
  return rueckgabe_wert;
}
```

Dirk Seeber, Informatik , Teil 4 30

Funktionen und Unterprogramme

- Neu ist die `return`-Anweisung.
- Diese Anweisung bewirkt, dass der nachfolgende Ausdruck ausgewertet und als Funktionsergebnis (Returnwert) an das rufende Programm zurückgegeben wird.
- Der Typ des Rückgabewertes muss dabei natürlich dem in der Schnittstelle vereinbarten Typ entsprechen.
- Die Funktion hat genau einen "Ausstieg". Es gibt nur eine `return`-Anweisung.
- Bevor man eine Funktion erstmalig verwenden kann, muss ein Funktionsprototyp erstellt werden. Angewendet auf das obige Beispiel gilt (Beispiel Funktionen):

```
double maximum ( double, double );
```


Funktionen und Unterprogramme

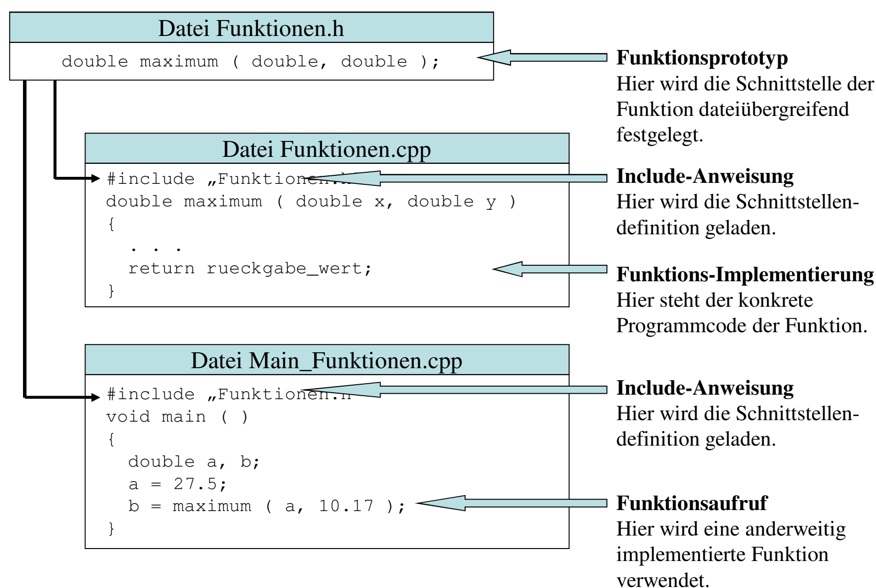
- Das bedeutet:
  - Irgendwo (extern) gibt es eine Funktion mit dem Namen maximum und der entsprechenden Schnittstelle.
- Eigentlich wiederholt man nur das, was im Kopf der Funktion sowieso schon steht.
- Warum macht man das so?
  - Angenommen man entwickelt zu zweit ein Programm. Dann ist es wenig sinnvoll den gesamten Quellcode in einer Datei zu haben, da immer nur einer zu einem Zeitpunkt daran programmieren könnte. Das Programm muss also auf mehrere Dateien aufgeteilt werden.
  - In einer Datei (Funktionen.cpp) wird das Unterprogramm maximum erstellt.
  - In einer Datei (Main_Funktionen.cpp) wird das Hauptprogramm erstellt.
  - Beide Dateien werden unabhängig voneinander durch den Compiler übersetzt. D.h. der Compiler hat beim Übersetzen keine Informationen über die Existenz der Funktion maximum und deren Schnittstelle.
  - Er kann daher nicht prüfen, ob die Funktion korrekt, d.h. mit der richtigen Parameterzahl und Parametertypen, verwendet wird.

Funktionen und Unterprogramme

- Warum macht man das so?
  - Jetzt kommt der Funktionsprototyp ins Spiel. Man erstellt eine dritte Datei (Funktionen.h) und trägt dort den Funktionsprototypen ein.
  - Dieser Funktionsprototyp ist die für alle Beteiligten verbindliche Festlegung der Schnittstelle.
  - Nun muss in den beiden Dateien (Funktionen.cpp und Main_Funktionen.cpp) die Anweisung #include "Funktionen.h" eingetragen werden, damit der Compiler in Funktionen.h nachsehen kann, welche Schnittstelle die Funktion maximum hat.
  - Der Compiler kann prüfen, ob die Funktion entsprechend der Schnittstellenspezifikation (in Funktionen.cpp) entwickelt wurde und ob die Funktion (in Main_Funktionen.cpp) korrekt verwendet wurde.
  - Das folgende Bild soll das verdeutlichen.

Dirk Seeber, Informatik , Teil 4 33

Funktionen und Unterprogramme



Dirk Seeber, Informatik , Teil 4 34

Funktionen und Unterprogramme

- Es ist nicht sinnvoll für jede Funktion und jeden Funktionsprototypen eine eigene Datei zu schreiben.
- Sondern man fasst alle zu einem Themenkomplex gehörenden Funktionen zusammen (z.B. maximum, minimum Fkt.) und erstellt dazu eine Header-Datei, die alle Funktionsprototypen enthält (z.B. math.h, string.h, . . .).
- Daran erkennt man die Bedeutung von Header-Dateien. Sie dienen dazu, Deklarationen (z.B. Funktionsprototypen), die von allgemeinem Interesse sind, aufzunehmen und allen Interessenten (Quellcode-Dateien) einheitlich zur Verfügung zu stellen.

Funktionen und Unterprogramme

- Beim Aufruf einer Funktion können beliebige Ausdrücke als Parameter mitgegeben werden.
- Wichtig ist nur, dass die Ausdrücke nach ihrer Auswertung den in der Schnittstelle geforderten Typ haben.
- Beispiel:

```
double a = 3.7;
double b = 1.5;
double c;
c = maximum ( a, b + 2 );
```

- Das Ergebnis eines Funktionsaufrufs kann überall dort verwendet werden, wo der Ergebnistyp zulässig ist. Zum Beispiel kann das Funktionsergebnis direkt in einer Formel verwendet werden.
- Beispiel:

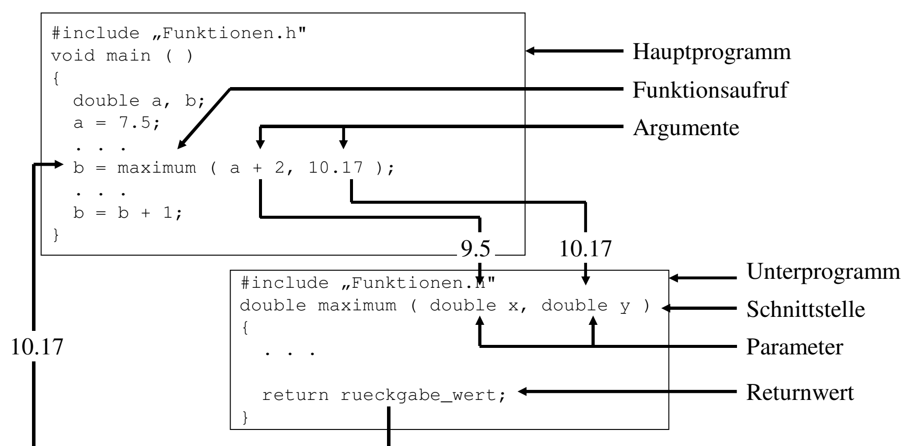
```
c = maximum ( a, b + 2 ) + 2.3;
```

Funktionen und Unterprogramme

- Die konkreten Argumente werden dem Unterprogramm in den Schnittstellenvariablen bereitgestellt. Hierbei handelt es sich um Variablen, die im Hauptprogramm nicht bekannt sind.
- Umgekehrt sind dem Unterprogramm auch keine Variablen des Hauptprogramms bekannt.
- das Unterprogramm arbeitet ausschließlich mit den Schnitt-stellenvariablen und gegebenenfalls weiteren interne Daten, ohne Zugriff auf die Daten des Hauptprogramms zu haben.
- Auch eine zufällige Namensgleichheit von Variablen im Haupt- und Unterprogramm ändert nichts an dieser Tatsache.

Funktionen und Unterprogramme

Die folgende Grafik zeigt noch einmal die wesentlichen Begriffe im Zusammenhang mit Funktionsaufrufen.



Funktionen und Unterprogramme

- Über die Schnittstelle hinaus haben rufende und gerufene Funktion keine gemeinsame Daten, es sei denn sie bedienen sich globaler Variablen bereitgestellt.
- Der Informationsaustausch über globale Variablen wird als **Seiteneffekt** bezeichnet, weil diese Information an der vereinbarten Schnittstelle vorbeifließen.
- Programmierung mit Seiteneffekten sollte aber auf das notwendige Minimum beschränkt werden.

Funktionen und Unterprogramme

- Eine Funktion kann Parameter und Rückgabewerte unterschiedlicher Typen haben

```
double potenz (double basis, unsigned int exponent)
{
  double ergebnis;
  for ( ergebnis = 1; exponent > 0; exponent-- )
  {
    ergebnis = ergebnis * basis;
  }
  return ergebnis;
}
```

- oder auch keine Parameter haben

```
char lies_zeichen( )
{
  char zeichen;
  cin >> zeichen;
  return zeichen;
}
```

Funktionen und Unterprogramme

- Eine Funktion, die kein Ergebnis liefert, erhält den Datentyp `void`.
- Eine `return`-Anweisung, dann allerdings ohne Rückgabewert, kann auch in solchen Funktionen verwendet werden, ist aber überflüssig.

```
void ausgabe(int i)
{
  cout << "Der Wert von i = " << i << endl;
  if ( i >= 0 )
  {
    cout << "Der Wert ist nicht negativ\n";
  }
  else
  {
    cout << "Der Wert ist negativ\n";
  }
  return;
}
```


Stack

- Wie funktionieren Unterprogramme?
- Um zu verstehen , wie der Rechner das macht, ist die Auseinandersetzung mit dem Stack eines Rechners notwendig.
- Ein **Stack** (Stapel) ist eine Datenstruktur, in der die Daten wie Teller auf einem Stapel in der Küche eines Restaurants verwaltet werden. (Tellerstapel !!!)
- Es gibt zwei Grundoperationen:
  - `push` lege einen Teller auf den Stapel
  - `pop` nimm einen Teller vom Stapel herunter
- Der zuletzt auf den Stapel abgelegten Teller wird dabei wieder als erster benutzt.
- Man spricht auch vom LIFO-Prinzip (last in first out).

Stack

- Konkret implementiert denkt man sich einen Stack als ausreichend großen Array mit einem Zeiger (Stackpointer), der jeweils auf die Spitze des Stacks zeigt.
- Die Operationen `push` und `pop` benutzen und verändern diesen Zeiger beim Datenzugriff.
- Wenn jetzt aus einem Hauptprogramm ein Unterprogramm gerufen wird, so legt das Laufzeitsystem zunächst die zur Parameterübergabe erforderlichen Argumentwerte auf den Stack. Bevor der endgültige Sprung ins Unterprogramm erfolgt, merkt sich das Laufzeitsystem ebenfalls auf dem Stack, an welcher Stelle es im Hauptprogramm nach der Rückkehr aus dem Unterprogramm weitergeht (Rücksprungadresse).

Stack

- Im Unterprogramm werden dann die benötigten Variablen als lokale Daten ebenfalls auf dem Stack angelegt.
- Das Unterprogramm kann jetzt auf seine Argumente und seine Variablen zugreifen, ohne irgendeine Rücksicht auf das Hauptprogramm oder gegebenenfalls noch weitere Unterprogramme nehmen zu müssen.
- Dieses Vorgehen wiederholt sich bei weiteren Unterprogrammaufrufen und es besteht dabei kein Unterschied, ob es sich um einen rekursiven oder normalen Unterprogrammaufruf handelt.
- Jedes Unterprogramm bekommt auf diese Weise seinen eigenen Datenbereich, den es ganz allein kennt und allein verwaltet.

Stack

- Hat ein Unterprogramm seine Pflicht getan, so wird der Rücksprung damit eingeleitet, dass alle lokalen Daten des Unterprogramms vom Stack entfernt werden.
- Dann wird mittels der jetzt wieder oben auf dem Stack liegenden Rücksprungadresse der Rücksprung ins übergeordnete Programm gefunden.
- Dort werden dann noch die Aufrufargumente vom Stack entfernt.
- Das übergeordnete Programm arbeitet dann wieder in seinem vertrauten Datenbereich, den das Unterprogramm nicht angetastet hat.

Funktionen und Unterprogramme

- Erstelle ein Unterprogramm, das die Werte von zwei Integer-Variablen im Hauptprogramm vertauscht.
- Erster Ansatz mit Testrahmen

```
void tausche (int a, int b)
{
  int t;
  t = a;
  a = b;
  b = t;
}
int main ()
{
  int x = 1; int y = 2;
  cout << "vorher: " << x << "  " << y << endl;
  tausche (x, y);
  cout << "nachher: " << x << "  " << y << endl;
  return 0;
}
```

Funktionen und Unterprogramme

- Das Ergebnis ist enttäuschend. Nicht ist passiert.

```
vorher: 1 2
nachher: 1 2
```

- Das war aber zu erwarten.
- Beim Aufruf des Unterprogramms werden die Werte von x und y in eigenständige, auf dem Stack liegende Variablen umkopiert.
- Ein Vertauschen dieser Variablen hat keinerlei Auswirkungen auf die ursprünglichen Variablen im Hauptprogramm.
- Man könnte die Variablen `x` und `y` des Hauptprogramms global anlegen und dann im Unterprogramm auf diese globalen Variablen zugreifen.

Funktionen und Unterprogramme

- Bei genauer Betrachtung erweist sich dies aber als keine echte Lösung der gestellten Aufgabe, da das Unterprogramm dann ja nur genau diese beiden Variablen vertauschen kann und nicht allgemein zur Vertauschung von Variablen eingesetzt werden könnte.
- Um das Problem wirklich zu lösen, übergibt man nicht die Werte, sondern die Adressen der Variablen an das Unterprogramm.
- Die Adresse identifiziert dann eindeutig die Stelle im Speicher, an der die Variable abgelegt ist.
- Zur Bestimmung der Adresse benutzt man den Adressoperator.

Funktionen und Unterprogramme

- Die Adresse einer Variablen erhält man, indem man dem Variablen-namen den **Adressoperator &** voranstellt.
- Um die Adressen von `x` und `y` an das Unterprogramm `tausche` zu übergeben, führt man im Testrahmen folgende Änderung durch:

```
int main ()
{
  int x = 1; int y = 2;

  cout << "vorher: " << x << "  " << y << endl;

  tausche (&x, &y);

  cout << "nachher: " << x << "  " << y << endl;

  return 0;
}
```


Funktionen und Unterprogramme

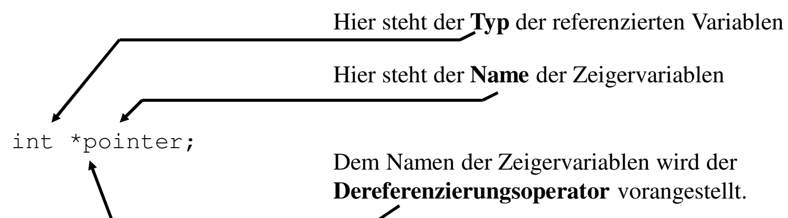
- Dadurch ändert sich die Schnittstelle der Funktion `tausche`.
- In den Parametern fließen jetzt nicht mehr die Zahlenwerte, sondern Adresswerte.
- Übertragen wird jetzt nicht mehr die Information, welche Werte die Variablen im Hauptprogramm haben, sondern wo im Speicher die Variablen stehen.
- Das Unterprogramm erhält somit als Parameter Variablen, in denen Adressen von Integer-Variablen stehen.
- So etwas nennt man einen Zeiger.
- Bevor mit der Behandlung des Beispiels fort gefahren werden kann, werden neue Begriffsbildungen benötigt.

Funktionen und Unterprogramme

- Eine Variable, in der die Adresse einer anderen Variablen gespeichert ist, nennt man eine **Zeigervariable** oder kurz **Zeiger** bzw. **Pointer**.
- Die Variable, deren Adresse im Zeiger gespeichert ist, bezeichnet man als die durch den Zeiger **referenzierte Variable**.
- Über einen Zeiger kann auf die Daten der referenzierten Variable zugegriffen werden. Dies nennt man **Indirektzugriff** oder auch **Dereferenzierung**.
- Zum Zugriff auf die referenzierte Variable stellt man der Zeiger-variablen den **Dereferenzierungsoperator *** voran. Ist p ein Zeiger, so ist *p der Wert der referenzierten Variablen.

Funktionen und Unterprogramme

- Zeigervariablen müssen, wie alle Variablen, vor ihrer ersten Verwendung definiert werden:



Leseanleitung: Wenn man auf die Variable `pointer` den Operator `*` anwendet, so erhält man `int`. `pointer` ist also ein Zeiger auf `int`.

Funktionen und Unterprogramme

- Der Typ der referenzierten Variablen muss bei der Definition des Zeigers angegeben werden.
- Insofern gibt es in C keine "Zeiger an sich", sondern immer nur "Zeiger auf etwas".
- In unserem Beispiel ist `pointer` ein Zeiger auf `int`, und nur in diesem Sinne kann der Zeiger verwendet werden.
- Über den Adressoperator wird die Verbindung zwischen dem Zeiger und der referenzierenden Variable hergestellt.
- Mit Hilfe des Dereferenzierungsoperators kann über den Zeiger auf die referenzierte Variable zugegriffen werden.
- Es ist immer darauf zu achten, dass ein Zeiger eine gültige Referenz enthält, bevor er verwendet wird. Die Verwendung von "wilden" Zeiger ist eine Hauptfehlerquellen in C-Programmen.

Funktionen und Unterprogramme

- Beispiele:

```
int x;
int *pointer;
```

Hier wird eine "gewöhnliche" Variable angelegt.

Hier wird ein Zeiger auf `int`, also eine Variable, die die Adresse einer `int`-Variablen aufnehmen kann, definiert.

```
pointer = &x;
```

Hier wird der Variablen `pointer` die Adresse der Variablen `x` zugewiesen. Damit ist `x` die von `pointer` referenzierte Variable.

```
*pointer = 1;
```

Hier wird der von `pointer` referenzierten Variable - also `x` - der Wert `1` zugewiesen.

```
*pointer = *pointer + 1;
```

Hier wird der Wert der von `pointer` referenzierten Variablen - also der Wert von `x` - um `1` erhöht.

Funktionen und Unterprogramme

- Mit diesen Informationen kann das Unterprogramm `tausche` vervollständigt werden.
- Dazu muss man an der Schnittstelle die korrekten Datentypen "Zeiger auf int" entgegennehmen und im Funktionskörper mit dem Dereferenzerungsoperator auf die Variablenwerte im Hauptprogramm durchgreifen.

```
void tausche (int *a, int *b)
{
  int t;
  t = *a;
  *a = *b;
  *b = t;
}
```

- Jetzt tritt der gewünschte Effekt ein:

```
vorher: 1 2
nachher: 2 1
```


Funktionen und Unterprogramme

- In C++ gibt es so genannte Referenzen.
- Eine Referenz ist ein konstanter Zeiger, der bei jeder Verwendung automatisch dereferenziert wird.
- Eine Referenz ist ein L-Value. D.h. sie kann also auf der rechten und der linken Seite einer Zuweisung verwendet werden.
- Beispiel: Funktion, die zwei Werte tauscht

```
void tausch_referenz( int& a, int& b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

Funktionen und Unterprogramme

- Bei der Verwendung einer Referenz muss dann nicht mehr explizit dereferenziert werden - der Compiler erledigt das automatisch.
- Beim Funktionsaufruf muss auch nicht mehr explizit der Adressoperator angewandt werden - auch das erledigt der Compiler.
- Beispiel: Funktion, die zwei Werte tauscht

```
int main()
{
    int x, y;
    x = 1;   y = 2;
    cout<<"vorher x:"<< x <<"   y:"<< y << endl;
    tausch_referenz( x, y );
    cout<<"nachher x:"<< x <<"   y:"<< y << endl;
    return 0;
}
```

Dirk Seeber, Informatik , Teil 4 57

Funktionen und Unterprogramme

- Natürlich beschränkt sich die Verwendung von Zeigern nicht auf "Zeiger auf int".
- Es können Zeiger auf jeden bekannten Datentyp eingerichtet werden.
  - `float *pf;           // Zeiger auf float`
  - `char *pc;            // Zeiger auf char`
  - `unsigned int *pu;    // Zeiger auf unsigned int`
- Auch ein "Zeiger auf einen Zeiger" ist nicht Außergewöhnlich
  - `int **z;        // Zeiger auf einen Zeiger auf int`

Dirk Seeber, Informatik , Teil 4 58

(Daten-) Strukturen

- Ein Array ist eine Zusammenfassung von Elementen gleichen Typs.
- Will man eine Zusammenstellung von Elementen unterschiedlichen Typs, dann sind Strukturen die Lösung.
- D.h. man benötigt Techniken, um elementare Daten (z.B. Zahlen oder Zeichen) zu **Datenstrukturen** (z.B. Studentendaten, Adressdaten) zusammenzufassen.
- Im Gegensatz zu Arrays müssen die aneinander gefügten Daten jedoch nicht den gleichen Typ haben, und sie werden nicht über einen Index, sondern über einen vom Programmierer gewählten Namen angesprochen.
- Die elementaren Datentypen (int, float, …) sind die Basis, aus dem man komplexere Datenstruktur zusammensetzt.

Datenstrukturen

- Datenstrukturen bringen eine deutliche Verbesserung in Richtung Komfort, Verständlichkeit, Erweiterbarkeit, Wiederverwertbarkeit, kurz Qualität des Programmcodes und sind daher für die Software-Entwicklung unentbehrlich.
- Datenstrukturen spielen für die Software-Entwicklung häufig eine weitaus wichtigere Rolle als Algorithmen, da Datenstrukturen in der Regel längerlebig sind als Algorithmen und daher eine zentrale Rolle im Design von Software-systemen bilden.

Datenstrukturen

- Als Beispiel ein Programm, das die Studenten einer Hochschule verwaltet.
- Die einem Studenten zugehörigen Daten wären hier
  - Vorname, Nachname, Matrikelnummer,
  - Adresse (Strasse, Postleitzahl, Wohnort)
  - sowie weitere Informationen über belegte Vorlesungen und Noten
- Die erforderlichen Datenstrukturen sind ein unmittelbares Abbild der in der Realität vorkommenden Daten und ihrer Beziehung untereinander und als solches weitaus stabiler als ein bestimmter Algorithmus, der etwa aus Einzelnoten eine Gesamtnote berechnet oder die Teilnehmer eine Klausur nach aufsteigenden Matrikelnummern sortiert/ordnet.

Datenstrukturen

- Ein Algorithmus kann in einem gut modularisierten Programm relativ einfach durch einen anderen Algorithmus ersetzt werden, ohne dass Auswirkungen auf andere Teile des Programms zu befürchten sind.
- Änderungen an einer Datenstruktur erfordern dagegen in der Regel Änderungen in allen Algorithmen, die auf der Datenstruktur arbeiten, und haben somit Auswirkung in unterschiedlichen Teilen des Programms.
- Die Festlegung einer Datenstruktur muss mit großer Sorgfalt getroffen werden, um zukünftige Änderungsaufwendungen so gering wie möglich zu halten.
- Dies ist besonders schwierig, da man häufig zum Zeitpunkt der Datenstruktur-Festlegung noch nicht weiß, welche Algorithmen auf der Datenstruktur arbeiten werden.

Datenstrukturen

- Obwohl man sich unter Daten und Algorithmen etwas gänzlich anderes vorstellt, sind die Abstrakionsmechanismen bei Daten und Algorithmen vergleichbar.
- Bei **Algorithmen** wurden bzw. werden Sequenz, Alternative, Iteration und Rekursion vorgestellt und diesen abstrakten Kontrollstrukturen jeweils konkrete C-Sprachelemente zugeordnet:

Kontrollabstraktion	**C-Sprachelement**
Sequenz	Block
Alternative	if … else, switch
Iteration	for, while, do … while
Rekursion	Rekursive Funktion

Datenstrukturen

- Bei den Datenstrukturen gibt es eine vergleichbare Abstraktion und deren Zuordnung:

Datenabstraktion	**C-Sprachelement**
Sequenz	struct
Alternative	union
Iteration	array
Rekursion	Rekursive Datenstruktur (Baum, Graph, Liste)

- Arrays (Zeichenketten) wurden bereits vorgestellt.
- Als nächstes Thema steht die Sequenz (struct) an.
- Die Alternative und rekursiven Datenstrukturen folgen dann zu einem späteren Zeitpunkt.

Datensequenzen (struct - records)

- In C werden Datenstrukturen wie folgt deklariert:

```
struct student
{
  string name;
  string vorname;
  long matrikelnummer;
  char geschlecht;
};
```

  – wobei:
    struct: Deklaration einer Datenstruktur
    student: Name der Datenstruktur
    1. Eintrag: Feld der Datenstruktur,
      jedes Feld hat einen Typ und einen Namen
    2. Eintrag: weitere Felder können folgen
- Von jetzt an ist student eine Datenstruktur mit 4 Feldern vom angegebenen Typ.

Dirk Seeber, Informatik , Teil 4 65

Datensequenzen (struct - records)

- Bei der Deklaration einer Datenstruktur werden (noch) keine Variablen angelegt (definiert).
- Obwohl es in C möglich ist, die Deklaration von Datenstrukturen direkt mit dem Definieren von Variablen zu verbinden, sollte im Sinne einer sauberen Trennung von Deklaration und Definition von diesen Möglichkeiten kein Gebrauch gemacht werden.
- In der Regel befinden sich die Deklarationen von Datenstrukturen in einem Headerfile, damit sie von unterschiedlichen Modulen einheitlich verwendet werden können.

Dirk Seeber, Informatik , Teil 4 66

Datensequenzen (struct - records)

- In der Quelldatei kann man dann Variablen vom Typ student anlegen:

```
struct student stud1;
struct student stud2;
```

- Auch solche Variablen können bei der Definition initialisiert werden, indem man, eingeschlossen in geschweifte Klammern, Werte für die einzelnen Felder der Struktur angibt:

```
struct student stud1 =
{ "Seeber", "Dirk", 123456, 'm' };

struct student stud2 =
{ "Meier", "Andrea", 234567, 'w' };
```

Dirk Seeber, Informatik , Teil 4 67

Datensequenzen (struct - records)

- Auf die einzelnen Felder einer Struktur kann mit dem "."-Operator zugegriffen werden:

```
stud1.name = "Seeber";
stud1.matrikelnummer = 123456;
stud1.geschlecht = 'm';
```

- Die Felder können dabei benutzt werden wie Variablen des zugehörigen Typs.

```
if ( stud2.geschlecht == 'm' )
{
  cout << "maennlich" << endl;
}
else
{
  cout << "weiblich" << endl;
}
```

Dirk Seeber, Informatik , Teil 4 68

Datensequenzen (struct - records)

- Strukturen können ihrerseits wieder Strukturen als Felder enthalten.
- Das wird veranschaulicht durch die neue Struktur s_adresse:

```
struct adresse
{
  string strasse;
  long plz;
  string ort;
};
```

- Diese Struktur wird als Feld in die Struktur student mit aufgenommen:

```
struct student
{
  string  name;
  string  vorname;
  long    matrikelnummer;
  adresse studAdresse;
  char    geschlecht;
};
```


Datensequenzen (struct - records)

- Bei der Initialisierung müssen die Initialwerte entsprechend der Schachtelung der Datenstruktur in geschweiften Klammern bereitgestellt werden::

```
struct student stud1 =
{ "Seeber", "Dirk", 123456,
  { "Wiesenstr.20", 12345, "Hamburg" },
  'm'
};

struct student stud2 =
{ "Meier", "Andrea", 234567,
  { "Rheinstr. 5", 64283, "Darmstadt" },
  'w'
};
```

Datensequenzen (struct - records)

- Der Zugriff auf die Datenfelder erfolgt jetzt längs des in der Deklaration vorgegebenen Zugriffspfades:

```
stud2.name = "Seeber";
stud2.matrikelnummer = 123456;
stud2.studAdresse.plz = 64283;
stud2.geschlecht = 'm';
stud1.studAdresse.plz = stud2.studAdresse.plz;
```

- Auch die Zuweisung von Teilstrukturen oder die Zuweisung ganzer Strukturen ist möglich:

```
struct adresse adr1 =
  { "Rheinstr. 5", 64283, "Darmstadt" };
struct student stud3, stud4;
stud4 = stud3;
```

- Voraussetzung dafür ist natürlich, dass die Typen links und rechts vom Zuweisungsoperator zueinander passen, d.h. von der gleichen Struktur sind.

Datensequenzen (struct - records)

- Datenstrukturen können auch als Parameter an Funktionen übergeben werden:

```
struct student datenreihe[100];
int anz_elemente;

anz_elemente = Einlesen( datenreihe );
```

- Datenstrukturen können auch Rückgabewerte von Funktionen sein:

```
struct student stud5;
int anz_elemente;

stud5 = Einlesen ( );
```

- Beispiel für einfache Datenstruktur: Bruch.

Compile Time Operator sizeof

- Mit dem unären Operator `sizeof` kann man die Größe von Variablen oder Typbezeichnern einer Variablen ermitteln.
- Abhängig vom Compiler können unterschiedliche Größen als Ergebnis festgestellt werden.
- Mit diesem Operator kann man Compiler unabhängigen Code implementieren.

Array

- Der Name des Arrays bildet einen Zeiger auf das erste Element.

```
int beispiel[10];
cout << beispiel << "    " << &beispiel << endl;
```

- Man kann in C/C++ nicht ganze Arrays an Funktionen übergeben.
- Die normale Übergabekonvention ist "call by value", die diese Array-Übergabe verhindert.
- Allerdings kann ein Zeiger auf das Array übergeben werden (call by reference).
- Der folgende Programmausschnitt übergibt die Adresse des Array:

```
int beispiel[10];
funktion1( beispiel );
```

Array

- Da hier nur Zeiger übergeben werden, kann man eindimensionale Arrays gleichwertig als Zeiger, als Array fester Größe oder als Array ohne Größenangabe übergeben.
- Programmausschnitte –
  Übergabe von eindimensionalen Array-Adressen:

```
// Zeiger
void funktion1( int *beispiel );
// Array mit fester Groesse
void funktion1( int beispiel[10] );
// Array mit unbestimmter Groesse
void funktion1( int beispiel[] );
```

- Hat man es an einer Schnittstelle mit mehrdimensionalen Arrays zu tun, so kann man immer nur den Bereich des ersten Index unbestimmt lassen, da die Bereiche der nachfolgenden Indices für die Berechnung des Zugriffs benötigt werden, z.B.:

```
void funktion2( int beispiel[][10] );
```

Dirk Seeber, Informatik , Teil 4 75

Funktionen und Unterprogramme

- Mit einem Funktionsaufruf verbindet man gemeinhin die Vorstellung, dass eine Funktion eine andere Funktion aufruft.
- Es besteht aber kein Grund auszuschließen, dass eine Funktion sich mittelbar (d.h. auf dem Umweg über eine andere Funktion) oder unmittelbar selbst aufruft.
- Man bezeichnet dies als **Rekursion**.
- Rekursion bedeutet, dass eine Funktion ihre Berechnungen unter Rückgriff auf sich selbst durchführt.
- Das erscheint zunächst paradox, ist aber eine sehr sinnvolle Programmiertechnik.
- Als Beispiel betrachtet man die Folge der Fakultäten - das Produkt der ersten n natürlichen Zahlen:

$$\prod_{k=1}^{n} k = 1*2*3*...*(n-1)*n = n!$$

Dirk Seeber, Informatik , Teil 4 76

Funktionen und Unterprogramme

- Programm:

```
/* Fakultaetsberechnung
*/
#include <iostream>
using namespace std;
int main ( )
{
  int i, n, fakul;
  cout << "Maximalzahl = ";
  cin >> n;
  cout << endl;
  fakul = 1;
  for ( i = 1; i <= n; i++ )
  {
    fakul = fakul * i;
    cout << i << "! = " << fakul << endl;
  }
  return 0;
}
```

- Beispiel für Fakultäten von 1 bis 8.
  Falls Integer = 2 Byte, ist dies die größte darstellbare Fakultät!

Funktionen und Unterprogramme

- Bildschirmausgabe:

```
maximalzahl = 8

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
```

- Es handelt sich bei der Folge der Fakultäten um eine derart stark anwachsende Zahlenfolge, dass schon für relativ kleine Werte von n ein Überlauf auftritt und den Ergebnissen des Programms nicht mehr zu trauen ist!

Funktionen und Unterprogramme

- Programm mit Funktionsaufruf (beginnend von n nach 1):

```
#include <iostream>
using namespace std;
long fakul_iterativ ( int );

int main ( )
{
  int n, i;
  long fakul;
  cout << "Maximalzahl = ";
  cin >> n;
  cout << endl;
  fakul = fakul_iterativ ( n );
  cout << "fakultaet von " << n << " = " << fakul << endl;
  return 0;
}

long fakul_iterativ ( int n )
{ int i;
  long fakul = 1;
  for ( i = 1; i <= n; i++ )
  {
    fakul = fakul * i;
    cout << i << "! = " << fakul << endl;
  }
  return fakul;
}
```


Funktionen und Unterprogramme

- Bildschirmausgabe:

```
maximalzahl = 8

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
Fakultaet von 8 = 40320
```

- Man nennt dies eine iterative (schrittweise) Berechnung.
- Es geht aber auch anders.

Funktionen und Unterprogramme

- Neue Formulierung der Formel:
  - Um das Produkt der ersten n natürlichen Zahlen zu berechnen, genügt es, das Produkt der ersten n-1 natürlichen Zahlen zu kennen und dieses mit n zu multiplizieren.
- Das ergibt folgende Notation:

$$n! = \begin{cases} 1 & \text{für } n = 1 \\ n * (n-1)! & \text{für } n > 1 \end{cases}$$

- In dieser Darstellung wird deutlich, dass eine Funktion zur Berechnung von Fakultäten durch einen Rückgriff auf sich selbst programmiert werden kann.

Funktionen und Unterprogramme

- Programm mit rekursiven Funktionsaufruf :

```
#include "Fakultaet.h"

int main ( )
{
  int n, i;
  long fakul;

  cout << "Maximalzahl = ";
  cin >> n;
  cout << endl;

  fakul = fakul_rekursiv ( n );

  cout << "fakulultaet von " << n << " = " ;
  cout << fakul << endl;
  return 0;
}
```

Funktionen und Unterprogramme

- Headerdatei Fakultaet.h :

```
#include <iostream>
using namespace std;
long fakul_rekursiv ( int );
```

- Quell (code) datei Rekursiv.cpp:

```
#include "fakultaet.h"
long fakul_rekursiv ( int n )
{
  long fakul;
  if ( n == 1 )
  {
    fakul = 1;
    cout << "n = " << n << "  fakul = " << fakul << endl;
  }
  else
  {
    fakul = n * fakul_rekursiv ( n - 1 );
    cout << n << "! = " << fakul << endl;
  }
  return fakul;
}
```


Funktionen und Unterprogramme

- Bildschirmausgabe:

```
maximalzahl = 8

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
Fakultaet von 8 = 40320
```

Funktionen und Unterprogramme

- Man beachte, dass der Parameter n bei jedem Rekursionsschritt um 1 vermindert wird und für n = = 1 kein weiterer Selbstaufruf mehr stattfindet.
- So wie man sich bei einer Schleife immer Gedanken über eine geeignete Abbruchbedingung machen muss, muss man sich auch bei der Rekursion immer Gedanken über einen Ausstieg machen, um sich nicht in einem endlosen rekursiven Abstieg zu verlieren.

Funktionen und Unterprogramme

- Von ihrem äußeren Verhalten her sind beide Implementierungen gleich. Beide haben die gleiche Schnittstelle und liefern die gleiche Ergebnisse.
- Man kann sich fragen, wofür denn Rekursion überhaupt sinnvoll ist, zumal theoretische Untersuchungen zeigen, dass Rekursion immer vermieden werden kann und iterative Algorithmen grundsätzlich effizienter arbeiten als ihre rekursiven Gegenstücke.
- Trotzdem sind rekursive Techniken von großem Nutzen in der Programmierung. Sie erlauben es oft, komplizierte Operationen verblüffend einfach zu implementieren.
- Rekursive Algorithmen werden gern verwendet, wenn man ein Problem durch einen geschickten Ansatz auf ein 'kleineres' Problem gleicher Struktur zurückführen kann.

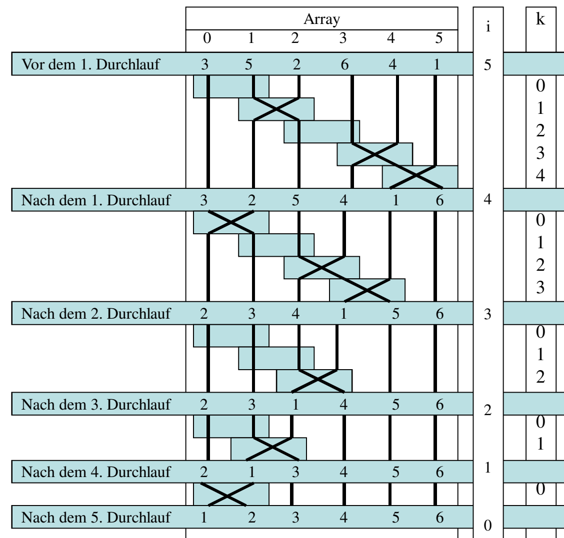
Sortierverfahren

- Eine klassische Aufgabe der Datenverarbeitung ist das Sortieren von Datensätzen. Eine algorithmische Lösung dieser Aufgabe hängt naturgemäß stark von Art und Umfang der Daten , von der Art der durchzuführenden Vergleiche sowie von den Zugriffsmöglichkeiten auf die Daten ab.
- Will man beispielsweise Strings alphabetisch sortieren, so ist unabhängig vom gewählten Sortierverfahren der Vergleich erheblich aufwendiger als bei der Sortierung von Zahlen.
- Daher werden die Rahmenbedingungen stark vereinfacht, um sich auf den algorithmischen Kern von Sortierverfahren zu konzentrieren.
- In der Terminologie von C gesprochen, heißt die Aufgabe, einen Array von n Integer-Zahlen in aufsteigender Reihenfolge zu sortieren.

Bubblesort - Austauschverfahren

- Der Name kommt daher, dass man sich vorstellt, dass die zu sortierenden Elemente im Array wie Luftblasen im Wasser aufsteigen.
- Das Verfahren läuft wie folgt ab:
  - Durchlaufe den Array in aufsteigender Reihenfolge! Betrachte dabei immer zwei benachbarte Elemente. Wenn zwei benachbarte Elemente in falscher Ordnung sind, dann vertausche sie! Nach diesem Durchlauf ist auf jeden Fall das größte Element am Ende des Arrays.
  - Wiederhole den obigen Verfahrensschritt solange, bis der Array vollständig sortiert ist! Dabei muss jeweils das letzte Element des vorherigen Durchlaufs nicht mehr betrachtet werden, da es schon seine endgültige Position gefunden hat!
- Die folgende Grafik veranschaulicht die einzelnen Durchläufe des Verfahrens am Beispiel eines Arrays mit 6 Elementen.

Bubblesort



Dirk Seeber, Informatik , Teil 4 89

Bubblesort - Austauschverfahren

- Im ersten Durchlauf startet man mit dem Vergleich der beiden ersten Zahlen (hier 3 und 5). Eine Vertauschung ist nicht erforderlich, da die beiden Zahlen in der korrekten Reihenfolge stehen.
- Im nächsten Verfahrensschritt betrachtet man die zweite und die dritte Zahl (hier 5 und 2). Diesmal ist eine Vertauschung erforderlich. usw.
- Die in den beiden rechten Spalten stehenden Zahlen stellen dabei bereits einen Bezug zu den Schleifenzählern des nachfolgenden C-Programms her.
- Man benötigt insgesamt 5 Durchläufe, um einen Array von 6 Elementen zu sortieren.
- Innerhalb des i-ten Durchlaufs finden dabei 6-i Vergleiche mit den gegebenenfalls erforderlichen Vertauschungen statt.

Dirk Seeber, Informatik , Teil 4 90

Bubblesort

- Vollständiges Programm - Funktion bubblesort

```
void bubblesort( int reihe[20], int anz );
{
  int i, k, temp;
  // Schleife für die n-1 Durchläufe
  for ( i = anz-1; i > 0; i-- )
  {
    // Schleife für die Einzelvergleiche
    for ( k = 0; k < i; k++ )
    {
      // Vergleich zweier benachbarter Elemente
      if ( reihe[k] > reihe[k+1] )
      {
        // Vertauschung zweier benachbarter Elemente
        temp = reihe[k];
        reihe[k] = reihe[k+1];
        reihe[k+1] = temp;
      }
    }
  }
}
```

Dirk Seeber, Informatik , Teil 4 91

Bubblesort

- Vollständiges Programm - Hauptprogramm

```
#include <iostream>
using namespace std;
void bubblesort( int [20], int );
int main ( )
{
  int daten[20] = {12,0,11,1,8,2,9,3,7,4,14,10,5,6,13};
  int i, n = 15;

  for ( i = 0; i < n; i++ ) {
    cout << daten[i] << "  ";
  }
  cout << endl;

  bubblesort( daten, n );

  for ( i = 0; i < n; i++ ) {
    cout << daten[i] << "  ";
  }
  cout << endl;
  return 0;
}
```

Dirk Seeber, Informatik , Teil 4 92

Bubblesort - Austauschverfahren

- Leistungsanalyse des Bubblesort
  - Dazu werden 1000 Elemente, die in einer zufälligen Reihenfolge abgelegt sind, mit dem Bubblesort sortiert.
  - Die erste for-Schleife wird 999 mal durchlaufen.
  - Die zweite for-Schleife wird auch insgesamt 999mal durchlaufen.
  - Die if-Abfrage wird 499500mal durchgeführt.
  - Das Tauschen der Elemente erfolgt ca. 239500mal.
- Der bubblesort besteht aus einem Kern, der in zwei Schleifen eingebettet ist.

Sortierverfahren

- Weitere Sortierverfahren:
  - Quicksort:
    Zerlege den Array in zwei Teile, wobei alle Elemente des ersten Teils kleiner oder gleich allen Elementen des zweiten Teils sind.
    Anschließend können die beiden Teile einzeln sortiert werden, Dazu werden sie jeweils wieder in zwei Teile zerlegt.
    Implementierung mittels Rekursion.
    Bessere Leistungsklasse als der Shellsort.
  - Heapsort:
    Sortieren in einer "Baumstruktur".
    Dieselbe Leistungsklasse wie der Quicksort