Vorkurs C++ Programmierung

Arrays
Funktionen

Rückblick

- Operatoren
  - logische Verknüpfungen
  - Zusammengesetzte Operatoren („Zuweisungsoperatoren“)
- Kontrollstrukturen
  - Bedingte Anweisungen (**`if`**-Abfrage, **`switch`**-Konstrukt)
  - Schleifen: **`while`**, **`do...while`**, **`for`**-Schleife
- Blöcke, Geltungsbereiche, Programmierstil

Heute

- Arrays als Zusammenfassung von Daten gleichen Typs
- Funktionen als Zusammenfassung wiederkehrender Anweisungen

Arrays

Ziel: Zusammenfassung von Daten gleichen Typs

- Eine Reihe von Temperatur-Messwerten für ein Jahr:

```
float messwerte [] = {5.0, 4.8, 6.4, 12.8,
                      14.4, 17.2, 16.7, 16.3,
                      12.6, 8.9, 4.3, 2.1};
```

(...Durchschnittstemperaturen in Siegen 2007 ☺
Quelle: http://www.uni-siegen.de/fb10/fwu/ww/wetterstation/)

- Deklaration mit Initialwerten:

```
type name [] = {val_1, val_2, ..., val_n};

int zahlen [] = {1, 4, 6, 3, 2, 7, 8, 2};
```

Arrays

- Keine Initialwerte nötig!
  - Werte bei Initialisierung meist noch nicht bekannt
  - Dann: Größe angeben (→ Speicher)
  - Deklaration:

    ```
    float messwerte [12];
    ```
  - Initialwerte sind in diesem Fall nicht definiert

- Deklaration ohne Initialwerte reserviert Speicherplatz für ***12*** Variablen vom Typ ***float***, die unter dem Namen ***messwerte*** erreichbar sind
  - 12 * 32 bit = 48 Byte

Arrays

- Verwendung
  - Zugriff auf Elemente: `[]`-Operator (lesend oder schreibend)
  - Indexzählweise: Von 0 (erstes Element) bis Länge-1 (letztes Element)!

- Beispiel (mit 7 Initialwerten):

```
int exp [] = {1, 2, 4, 8, 16, 32, 64};
std::cout << "Die ersten 2er-Exponenten: ";
for (int i = 0; i < 7; i++) //Indizes von 0 bis 6
{
  std::cout << exp[i] << " ";
}
```


Arrays

- Verwendung
  - Zugriff auf Elemente: `[]`-Operator (lesend oder schreibend)
  - Indexzählweise: Von 0 (erstes Element) bis Länge-1 (letztes Element)!

- Beispiel ohne Initialwerte, Werte werden erst in der Schleife geschrieben (Pseudocode):

```
float messwerte [12];
for (int i = 0; i < 12; i++) //Indizes von 0 bis 11
{
  //Berechne z.B. durchschnittliche Temperaturen für
  //jeden Monat...
  messwerte[i] = Monatsdurchschnitt im Monat i
}
//Jetzt steht in jedem Arrayelement ein Wert
```


Arrays

```
int quadrate [10];
std::cout << "Quadratzahlen von 1 bis 10: ";

for (int i = 0; i < 10; i++) //Indizes von 0 bis 9
{
  quadrate[i] = (i+1) * (i+1);
  std::cout << quadrate[i] << " ";
}
```

- Alternative?

```
for (int i = 1; i <= 10; i++) //Indizes von 1 bis 10
{
  quadrate[i] = i * i;
  std::cout << quadrate[i] << " ";
}
```

Fehler bei i == 10!

quadrate[10] existiert nicht.

Arrays

- Arrays liegen im Speicher in einem Block vor.
  Beispiel int-array
  `int exp0f2[] = {1, 2, 4, 8, 16, 32, 64};`

- Indexgrenzen `[0]` und `[count-1]` müssen unbedingt eingehalten werden, sonst findet ein *Speicherzugriffsfehler* statt.
- `exp0f2[7]` ist nicht zulässig. Der Fehler wird aber nicht vom Compiler abgefangen, sondern führt zu einem Laufzeitfehler!

Einschub... Fehlerarten

- „`exp0f2[7]` ist nicht zulässig. Der Fehler wird aber nicht vom Compiler abgefangen, sondern führt zu einem Laufzeitfehler!“
- 2 Arten von Fehlern können auftreten:
  - Syntaktische Fehler (Syntax → betrifft die „Sprache“)
    - Diese Fehler meldet der Compiler („Compilerfehler“)
    - Ziemlich einfach zu korrigieren
  - Semantische Fehler (Semantik → betrifft die „Bedeutung“)
    - Diese Fehler treten erst bei der Programmausführung auf („Laufzeitfehler“, „Ausnahme“, „Exception“)
    - Können schwer zu finden sein
- Ein Programm kann syntaktisch fehlerfrei sein und dennoch Laufzeitfehler produzieren
  - z.B. Division durch 0

Mehrdimensionale Arrays

- Anordnung von Daten gleichen Typs in mehreren Dimensionen
- Deklaration:
  - Wie beim eindimensionalen Array, aber pro Dimension ein `[]`-Paar
  - Elementanzahl in der ersten Dimension kann weggelassen werden
  - Deklaration mit Initialwert funktioniert auch hier

```
int table[5][2] = {{1,3},{5,7},{9,0},{2,4},{6,8}};
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 2; j++) {
      std::cout << "table[" << i << "][" << j << "] = "
           << table[i][j] << std::endl;
    }
}
```


Mehrdimensionale Arrays

- Anordnung von Daten gleichen Typs in mehreren Dimensionen, Beispiel: Matrizen
  Eine 3x4 Matrix A:

$$A = \begin{pmatrix} 1 & 4 & 3 & 8 \\ 33 & 42 & 1 & 17 \\ 4 & 40 & 21 & 23 \end{pmatrix} \qquad \text{allg.} \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

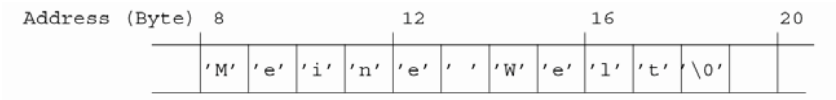
```
int matrix[3][4] = {{1,4,3,8},{33,42,1,17},{4,40,21,23}};
```

Hilfreich: Schreibweise anpassen

```
int matrix[3][4] = { { 1,  4,  3,  8},
                     {33, 42,  1, 17},
                     { 4, 40, 21, 23} };
```

Arrays aus Zeichen

- Darstellung von Zeichenketten (*Strings*)
  Deklaration: **`char name[size];`**
  Mit Initialwert: **`char name[] = "Meine Welt";`**
- Zeichenketten werden intern als Arrays von Zeichen dargestellt
  Bsp.: Interne Speicherung des Strings „Meine Welt“



- Werden mit Anführungszeichen definiert
- Der Null-Character (**`'\0'`**) wird automatisch angehängt, um das Ende des Strings zu markieren.
  → **`size`** = Anzahl der Zeichen + 1

Funktionen

- Bisher haben wir unsere Anweisungen immer in die **`main`**-Funktion geschrieben...
- Jetzt: Eigene Definition von Funktionen
- Zusammenfassung von Anweisungen unter einem Namen
- Abstraktion und Wiederverwendung von Programmteilen

- Beispiel Sommerschlussverkauf: Die Preise sollen in der ersten Woche um 20 Prozent gesenkt werden, in der zweiten Woche noch einmal um 30 Prozent, danach vielleicht noch ein weiteres mal...

Funktionen

- Beispiel: Preise ändern um x Prozent

```
//Eine Funktionsdefinition:

float aenderePreis(float alterPreis, float prozent)
{
   float erhoehung = prozent / 100.0f;
   float neuerPreis = alterPreis + erhoehung * alterPreis;
   return neuerPreis;
}
```

- Eingegeben wird ein Preis und der prozentuale Anteil, um den der Preis geändert werden soll.
- Ausgegeben wird der neue Preis.

Funktionen

- **Syntax:**

```
Rückgabetyp Funktionsname (Typ1 param1, ..., TypN paramN)
{
     Funktionsrumpf: Anweisungen
}
```

- **Beispiel:**

```
int berechneQuadrat( int eineZahl )
{
   return (eineZahl * eineZahl);
}
```

- Übergabe unterschiedlicher Parameter mit jedem Aufruf
- ***`Typ1`*** bis ***`TypN`*** sind die Typen der entsprechenden Parameter
- ***`Rückgabetyp`*** ist der Rückgabe-Typ der Funktion, also der Typ des Ergebnisses, das an der Aufrufstelle verwendet werden kann.

Wohin mit den Funktionen? Bisher so...

```
#include <iostream>

int main ( int argc, char* argv )
{
  int result;
  result = 1;
  int n = 10;
  for (int i = n; i >= 1; i-- )
  {
      result = result * i;
  }
  std::cout << "Fakultaet von " << n << ": " << result;

  return 0;
}
```


Wohin mit den Funktionen? Jetzt so...

```
#include <iostream>

int faculty(int number)
{
  int result = 1;
  for (int i = number; i > 1; i--)
  {
      result *= i;
  }
  return result;
}
```

Funktionsdefinition

```
int main ( int argc, char* argv )
{
  int ergebnis = faculty(10);
  std::cout << ergebnis;
  return 0;
}
```

Aufruf mit Zuweisung

Mehr Beispiele

```
float add ( float a, float b)
{
  return a + b;
}
```

Parameterübergabe

Funktionsaufruf durch:

```
float fErgebnis = add(3.24, 123.53);
```

```
int maximum ( int a, int b)
{
  if (a > b)
  {
    return a;
  }
  else { return b; }
}
```

Parameterübergabe

Funktionsaufruf durch:

```
int nbiggerNumber = maximum(3,4);
```


Funktionen...

- Funktionen müssen genau wie Variablen deklariert werden bevor sie benutzt werden können
- Eine Funktion ist eindeutig durch ihre ***Signatur***, die aus ihrem Namen, den Typen ihrer Argumente und deren Reihenfolge besteht, beschrieben.
- Doppelte Signaturen dürfen nicht vorkommen!

```
int berechneEtwas ( int zahl1, float zahl2 ) {...}
float berechneEtwas ( int x, float y) {...}
```

Konflikt: Gleicher Name, gleiche Argumenttypen, gleiche Reihenfolge

`Return-Type` gehört nicht zur Signatur

- Mit der **`return`**-Anweisung wird das Ergebnis der Funktion zurückgegeben

Funktionen...

- Funktionen müssen genau wie Variablen deklariert werden bevor sie benutzt werden können
- Eine Funktion ist eindeutig durch ihre ***Signatur***, die aus ihrem Namen, den Typen ihrer Argumente und deren Reihenfolge besteht, beschrieben.
- Doppelte Signaturen dürfen nicht vorkommen!

```
int berechneEtwas ( int zahl1, float zahl2 ) {...}
float berechneEtwas ( int x, float y) {...}
int berechneEtwas ( float zahl1, int zahl2 ) {...}
```
OK!

Return-Type gehört nicht zur Signatur

- Mit der **`return`**-Anweisung wird das Ergebnis der Funktion zurückgegeben

Rückgabewert

- Rückgabewert macht nur bei einer Zuweisung an der Aufrufstelle Sinn:

```
int berechneQuadrat( int eineZahl )
{
     return (eineZahl * eineZahl);
}
```

Aufruf:

```
int ergebnis = berechneQuadrat(5);
```

→ **`ergebnis`** bekommt den Wert 5 zugewiesen

- Braucht eine Funktion keinen Rückgabewert, so wird ihr Rückgabewert als **`void`** deklariert:

```
void eineFunktion (bool bVar)
{
   Anweisungen. Z.B. eine reine Textausgabe in Abh. von bVar
   //hier entweder keine Rückgabe mittels return oder:
   return; //leeres return
}
```


Funktionsaufrufe

- Es gibt zwei verschiedene Arten der Parameterübergabe:
  - „Call by Value“
  - „Call by Reference“

  Bisher: Nur Call by Value
- „intern“ wird der Variablenwert beim Aufruf in eine typgleiche Variable kopiert
- bei umfangreichen Datentypen entsteht großer Kopieraufwand
- eine Veränderung innerhalb der Funktion hat keine Auswirkung nach außen, d.h. der Wert der Variablen *im Aufruf* bleibt unverändert!

Beispiel für Call-By-Value Problematik

```
#include <iostream>

void quadrat (int i)
{
  i = i * i;
}

int main (int argc, char* argv[])
{
  int s = 2;
  quadrat(s);
  std::cout << s << std::endl;
  return 0;
}
```
Ausgabe: 2

Begründung:

Es wird nicht wirklich s, sondern eine **Kopie des Wertes** an die Funktion übergeben! Die Kopie wird geändert, s bleibt aber wie vorher!

Beispiel für Call-By-Value Problematik

```
#include <iostream>

int quadrat (int i)
{
  i = i * i;
  return i;
}

int main (int argc, char* argv[])
{
  int s = 2;
  s = quadrat(s);
  std::cout << s << std::endl;
  return 0;
}
```

Ausgabe: 4

Mögliche Lösung:

Funktion mit Rückgabewert, dann eine Zuweisung an der Aufrufstelle.

Richtig gut ist diese Lösung noch immer nicht, denn nach wie vor wird eine Kopie erzeugt.

Ergebnis wird wieder zurück kopiert.

Beispiel für Call-By-Value Problematik

```
#include <iostream>

Datei komprimiereDatei (Datei d)
{
  //Komprimiere Datei d,
  //liefere komprimierteDatei
  //zurück
  return komprimierteDatei;
}

int main (int argc, char* argv[])
{
  //...
  Datei d = eineGrosseDatei;
  Datei kleineDatei = komprimiereDatei(d);
  return 0;
}
```

Sehr schlecht!

Eine Datei z.B. wird viel größer sein als eine einfache **`int`**-Variable (32 bit). Diese Daten müssen alle kopiert werden.

Kostet Zeit und Speicher!

...und was passiert erst, wenn wir 50 Dateien nacheinander komprimieren wollen?

Funktionen: Parameter-Übergabe

Die Lösung: Eine andere Art der Parameterübergabe, bei welcher kein Kopiervorgang mehr stattfindet!

„Call by Reference“: Parameter ist Referenz- oder Zeigertyp!

- Eine Veränderung des referenzierten Wertes innerhalb der Funktion hat eine Auswirkung nach außen
- Der Wert der referenzierten Variablen wird geändert!
- Zeiger (Pointer) sind ein sehr wichtiges Konzept in der Sprache C/C++.
- ***Um Funktionen sinnvoll nutzen zu können, müssen wir noch zwei wichtige Konzepte verstehen: Zeiger und Referenzen***

...deshalb gibt es dafür eine eigene Theorie-Einheit!

Zusammenfassung

- Arrays als Zusammenfassung von Daten gleichen Typs
  - Deklaration, Initialisierung, Zugriff per **`[]`**-Operator, Indexgrenzen, Mehrdimensionale Arrays
- Funktionen als Zusammenfassung von Anweisungen, die man wiederholt braucht
  - Struktur:
    `Rückgabetyp Funktionsname (Parameterliste)`
    `{ //Funktionsrumpf }`
  - Signatur muss eindeutig sein
  - Parameterliste kann leer sein (trotzdem ein Klammernpaar!)
  - Rückgabetyp kann **`void`** sein

Zusammenfassung

- Parameterübergabe:
  - Call-by-Value-Problematik (Kopieraufwand, keine Auswirkung nach außen)
  - Call-by-Reference als Lösung → Nächste Stunde!