

## 8.9 Arrays

---

**Ziel:** Zusammenfassen von Daten gleichen Typs

**Deklaration:**

```
type name [count];
```

**Deklaration mit Initialwert:**

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

**Beschreibung:**

- Ein Array ist eine eindimensionale Sequenz von Daten gleichen Typs
- Deklaration ohne Initialwert alloziert Platz für *count* Variablen vom Typ *type*, die unter dem Namen *name* erreichbar sind
- Bei Deklaration mit Initialwert kann *count* ausgelassen werden, da sich die Anzahl aus den Initialwerten *val\_1* bis *val\_n* ergibt



## 8.9 Arrays ...

### Verwendung:

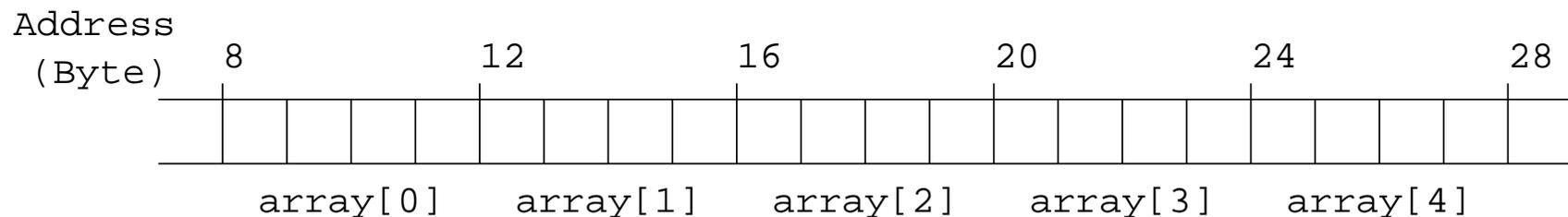
- Mit Hilfe des []-Operators wird auf die einzelnen Elemente des Arrays (lesend oder schreibend) zugegriffen

**Indezählweise** beginnend bei 0, maximal bis *count-1*

### Beispiel:

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

**Speicherung:** Arrays liegen im Speicher „en bloc“ vor; Beispiel int-Array:



## 8.9 Arrays ...

---

### Mehrdimensionale Arrays

**Ziel:** Anordnung von Daten gleichen Typs in mehreren Dimensionen

#### **Deklaration:**

- Grundsätzlich gleich wie beim eindimensionalen Array, jedoch pro Dimension wird ein [] hinzugefügt
- Deklaration mit Initialwert funktioniert auch hier

#### **Beispiel:** Zweidimensionales Array

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

## 8.9 Arrays ...

### Arrays aus Zeichen

**Ziel:** Darstellung von Zeichenketten (**Strings**)

**Deklaration:** `char name[];`

**Deklaration mit Initialwert:** `char name[] = "string";`

**Beschreibung:**

- Zeichenketten werden intern als Arrays von UTF8-Zeichen dargestellt
- Zeichenketten müssen **null-terminiert** sein, d.h. das letzte Zeichen muss der Null-Character ('`\0`') sein

**Beispiel:** Interne Speicherung von Meine Welt

Address (Byte)	8	12	16	20								
	'M'	'e'	'i'	'n'	'e'	' '	'W'	'e'	'l'	't'	'\0'	

**Bemerkungen:** ○ Konstante Zeichenketten werden mit Anführungszeichen definiert und werden automatisch null-terminiert

- Die null-terminierte String-Darstellung ist für Textverarbeitung (vergl., suchen, ersetzen oder einfügen) unhandlich ( $\Rightarrow$  string-Klasse)



## 8.10 Referenzen und Zeiger

---

### Referenzen (References)

**Ziel:** Verweis auf die Adresse einer Variable im Speicher

**Deklaration:**  $type\& refName = varName;$

### **Beschreibung:**

- Definition einer Referenz-Variable auf eine (bereits initialisierte) Variable gleichen Typs im Speicher
- $refName$  ist der Name der Referenz,  $varName$  der Name der referenzierten Variable
- $type$  ist der Typ der Variable  $varName$
- Die Referenz  $refName$  verweist **unveränderlich** auf die Variable  $varName$
- Über die Referenz kann der Wert von  $varName$  verändert werden

**Beachte:** Es ist nicht möglich eine Referenz zu deklarieren, ohne direkt eine referenzierte Variable anzugeben!





### Beispiel:

```
// Deklaration
int k      = 10;
int i      = 5;
int copy_k = k; // setze copy_k auf den Wert von k: copy_k = 5
int& r_i   = i; // erstelle Referenz auf i

// Zuweisung
copy_k     = 20; // setze copy_k = 20 (k bleibt 10)
r_i        = 20; // r_i bleibt unverändert. i wird auf 20 gesetzt
```

### Bemerkung:

- Oftmals werden Präfixe – wie hier `r_` für Referenz – vor Variablennamen gestellt um den Typ der Variable deutlich zu machen (derartige Konventionen werden hier nur teilweise eingesetzt)



## 8.10 Referenzen und Zeiger ...

### Zeiger (Pointer)

**Ziel:** Explizites Speichern von Variablen-Adressen

**Deklaration:** `type* ptrName;` bzw. `type* ptrName = &varName;`

**Beschreibung:**

- Deklaration einer Zeigervariablen `ptrName`, ggf. mit Initialisierung
- `ptrName` verweist auf Variable vom Typ `type`

**Referenz-Operator** `&` holt die Adresse einer Variable

**Dereferenz Operator** `*` holt den Wert einer Adresse zu einem Zeiger

**Beispiel:** Adresse einer Variable speichern

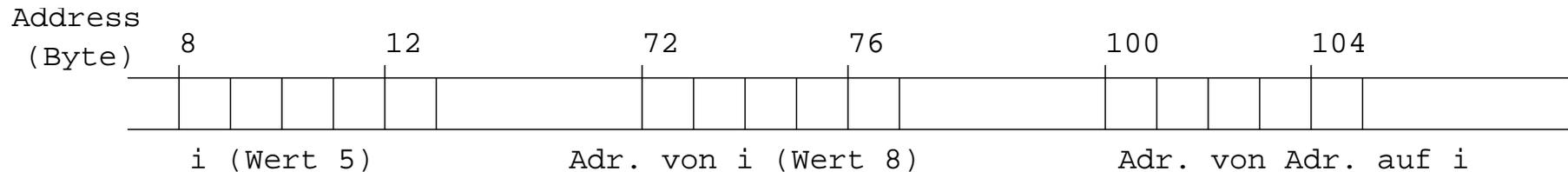
```
int    i      = 5;      // Integer Variable (Wert 5)
int*   p_i    = &i;    // Adresse von i
int**  p_p_i  = &p_i; // Adresse von p_i
std::cout << "i:          "      << i      << "\n"
          << "Adresse von i: "    << p_i    << "\n"
          << "Wert von i via Zeiger: " << *p_i  << "\n"
          << "Adresse des Zeigers "  << p_p_i << "\n";
```





## 8.10 Referenzen und Zeiger ...

### Speicher-Beispiel:



**Bemerkung:** Einem Zeiger kann jederzeit die Adresse einer anderen Variablen gleichen Typs zugewiesen werden

### Unerlaubter Speicherzugriff: Zugriff auf ungültige Speicheradresse

- Im Idealfall bricht das Programm mit einer (meist wenig aussagekräftigen) Fehlermeldung ab
- Im schlechten Fall läuft das Programm weiter und erzeugt nicht nachvollziehbare Ergebnisse

### Beispiel eines ungültigen Speicherzugriffs

```
int    i      = 5;    // Integer Variable (Wert 5)
int*   p_i    = &i;  // Adresse von i
```

```
p_i   += 4711;    // Veränderung des Zeigers auf irgendeine Adresse
*p_i  = 123456;  // !!! Zuweisung auf irgendeiner Adresse !!!
```





### Null-Zeiger (NULL):

- Der Wert `NULL=0x0` kann jedem Zeiger zugewiesen werden
- Zeigt an, dass der entsprechende Zeiger keine gültige Adresse darstellt
- Bei konsequenter Verwendung des Null-Zeigers kann die Prüfung, ob ein Zeiger auf einen gültigen Wert verweist durch einen Vergleich mit `NULL` realisiert werden

### Typ eines Zeigers:

- Prinzipiell ist ein Zeiger nichts anderes als ein `int`, der als Speicheradresse interpretiert wird
- Die Größe eines Zeigers entspricht der eines `int`, also 32 Bit bei 32-Bit Prozessoren und 64 Bit bei 64-Bit Prozessoren



### Wozu Referenzen / Pointer?

#### Referenzen:

- Referenzen haben, genau wie Zeiger, eine feste Größe von 32 bzw. 64 Bit
- Bei Typen, die mehr Speicher belegen, ist es ggf. sinnvoller eine Referenz zu verwenden als die Variable zu kopieren  
⇒ wichtig im Kontext von Funktionsaufrufen

#### Zeiger:

- Zeiger auf Variablen können genau wie Referenzen im Kontext von Funktionen anstatt kopierten Variablen verwendet werden
- Zeiger werden auch zur Allokation und Deallokation (Freigabe) von dynamischem Speicher benötigt

## 8.10 Referenzen und Zeiger ...

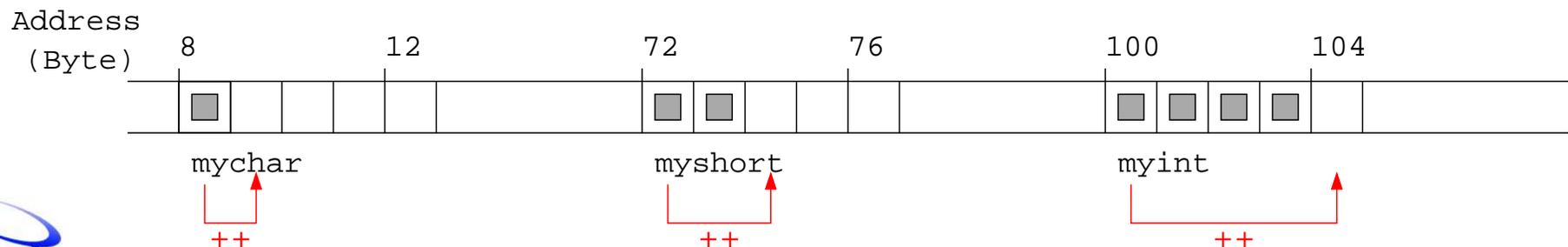
### Zeiger-Arithmetik

**Ziel:** Direkte Operationen auf Zeigern (nicht auf deren Inhalten)

### **Bechreibung:**

- Arithmetische Operationen wie ++, -- oder +2 können auf Zeiger angewandt werden
- Die Umsetzung dieser Operationen unterscheidet sich jedoch von „normaler“ Arithmetik
- Zeiger-Arithmetik basiert immer auf der Größe des adressierten Typs in Byte  
`sizeof(char) == 1, sizeof(short) == 2, sizeof(int) == 4`
- Ein Inkrement von 1 erhöht die Adresse um die Anzahl Bytes des Typs

### **Schaubild:**





## 8.10 Referenzen und Zeiger ...

---

### Beispiel: Zeiger-Arithmetik

```
// Array mit 6 Höhenwerten
double heightValue[6] = {1.1, 3.1, 2.7, 8.5, 9.1, 2.1};
double* p_height = &(a[0]); // Zeiger auf erstes Element

// Ausgabe der Höhenwerte
for ( int i = 0; i < 6; i++, p_height ++ ) {
    std::cout << "Höhe " << i << ": " << *p_height << std::endl;
}
```



**Bisher:** Geltungsbereich von Variablen

- global, falls ausserhalb von {}-Blöcken
- innerhalb des aktuellen {}-Blocks

Diese Variablen werden als **automatic**-Variablen bezeichnet (kurz: `auto`)

**Speicherung** der `auto`-Variablen auf einem **Stack**

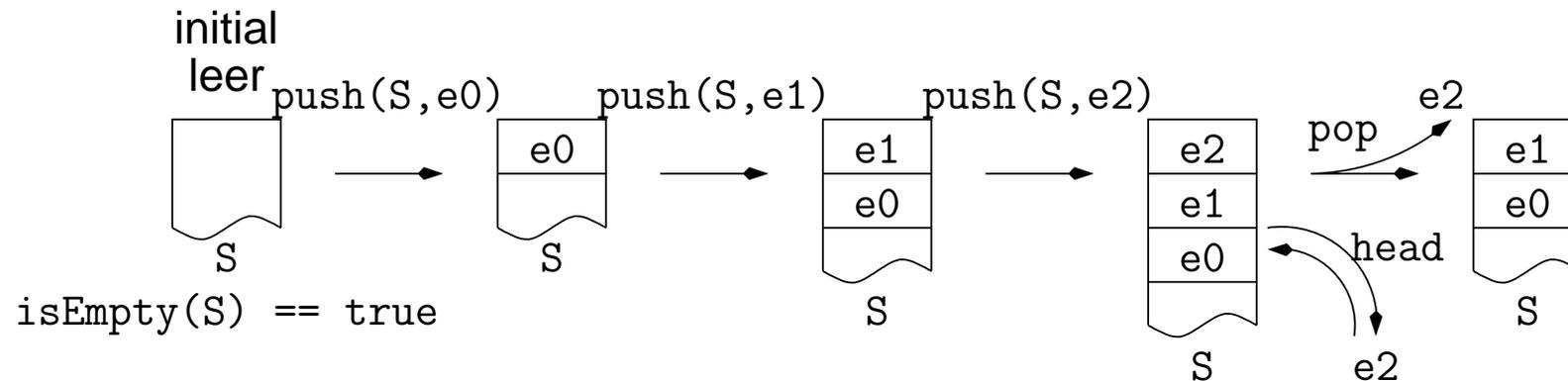
**Stack-Konzept:** **Stapelspeicher** nach dem **Last-In-First-Out (LIFO)** Prinzip:

- **push**: neues Element auf den Stapel ablegen (Variablen-Deklaration)
- **pop**: oberstes Stapel-Element holen & vom Stapel entfernen (Blockende)
- initial ist ein Stack immer leer
- Hilfsfunktionen eines Stacks:
  - `head`: Holt oberstes Element ohne es zu entfernen
  - `isEmpty`: Prüft, ob Stack leer ist (Ergebnis: boolsch)
- Umsetzung eines Stack: später



## Stack

### Beispiel zum Stack-Konzept



### Stack im Kontext der Variablenverwaltung:

```
{
  int    i=0;           // push(i)
  float  f=1;           // push(f)

  if ( f > 0 ) {
    float ff;           // push(ff)
    ff = f + 1;
  }
  // pop() (entfernt ff)
  // 2 x pop() (entfernt i, f)
```



### Speicherklasse `static`

**Problem:** Werte einer `auto`-Variablen geht am Blockende verloren

**static:** ○ Fester Speicherplatz für Variablen bis Programmende

- Wert der Variablen bleibt auch nach Blockende erhalten
- Sichtbarkeit wie bei `auto`-Variablen
- Allokation und Initialisierung nur beim ersten Aufruf!

**Beispiel:** Erhöhung nach jeder Abarbeitung des Blocks

```
{
    static float f=0;           // f erstmalig mit 0 initialisieren

    f ++;                      // f um eines erhöhen
    std::cout << "f = " << f << endl;
}                               // f bleibt erhalten
...                             // hier: f unbekannt, aber vorhanden!
```

Ausgabe bei mehrfachem Aufruf des Blocks:

```
f = 1
f = 2
f = 3
```





### Dynamischer Speicher

**Grundsätzlich** Programme realisierbar, die nur mit `auto`, `static` auskommen, aber

- feste Arraylängen  $\Rightarrow$  Obergrenze von „Objekten“ festgelegt
- Programm belegt Speicherplatz auf allen Fälle!

**Dynamischer Speicherung:** Speicher wird

- bereitgestellt, sobald im Programmablauf tatsächlich benötigt (***Allokation***)
- freigegeben, sobald er nicht mehr benötigt wird (***Deallokation***)

**Freier Speicher (Free Store, Heap):** Speicherbereich, der zur dynamischen Speicherung verwendet werden kann

**Bemerkung:** Verwaltung dynamischen Speichers ist für große Programm nicht trivial!



### Dynamischer Speicher (Forts.)

**Operatoren:** Bereitstellung (`new`) und Freigabe (`delete`)

`new`: ○ Optionale Angabe der Anzahl bereitzustellender Variablen (sonst 1)

- Rückgabe eines Zeigers auf den allokierten Speicher; wenn kein Speicherplatz vorhanden, Rückgabe des NULL-Zeigers

`delete`: ○ Angabe des Zeigers auf allokierten Speicher

- Syntax: Zusätzlich [], wenn ein Array allokiert wird

### **Beispiel:**

```
int    *p_zahl = NULL;
float  *p_ff   = NULL;

p_zahl  = new int[128];    // allokiert 128 ints
p_ff    = new float;      // allokiert 1 float

delete [] p_zahl;         // Freigabe der 128 ints
p_zahl = NULL;           // Konvention: Zeiger im Folgenden ungültig
delete p_ff;             // Freigabe des float
p_ff = NULL;            // Konvention: Zeiger im Folgenden ungültig
```



**Ziel:** Zusammenfassen von Anweisungen unter einem Namen zur Abstraktion und Wiederverwendung von Programmteilen

**Syntax:**

```
Return-Type Function-Name ( Type_1 Argument_1, ..., Type_n Argument_n )  
{  
    Code  
}
```

**Beschreibung:**

- Eine Funktion ist eine wieder verwertbare Zusammenstellung von Anweisungen
- Durch die Verwendung der Parameter *Argument\_1* bis *Argument\_n* können der Funktion bei jedem Aufruf andere Werte übergeben werden
- *Type\_1* bis *Type\_n* sind die Typen der entsprechenden Parameter
- *Return-Type* ist der Rückgabe-Typ der Funktion, also der Typ des Ergebnisses



- 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.  
*Return-Type* gehört nicht zur Signatur
- Innerhalb eines Namensraums muss eine Signatur eindeutig sein
- Mit der `return`-Anweisung wird das Ergebnis der Funktion zurückgegeben
- Hat eine Funktion keinen Rückgabewert, so wird ihr Rückgabewert als `void` deklariert
- Hat eine Funktion keine Parameter, so kann ihre Parameterliste ebenfalls als `void` deklariert werden (sie kann aber auch einfach ausgelassen werden)

## 8.12 Funktionen ...

---

### Beispiele:

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

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

**Funktionsaufruf**, z.B.: `unsigned int i = faculty( 5 ); // i = 120`

Bei Funktionsaufrufen wird für die zu übergebenden Variablen das ***Stack-Konzept*** verwendet

## 8.12 Funktionen ...

---

### Parameter-Übergabe

**Per Reference:** Übergabe einer Referenz- oder einer Zeigervariablen

- es wird eine **Verweis** zu der Variablen im Funktionsaufruf erstellt
- eine Veränderung innerhalb der Funktion hat eine Wirkung „nach außen“, d.h. der Wert der referenzierten Variablen wird geändert
- Größe der zu übergebenden Daten immer gleich (`int`)
- Arrays werden automatisch per reference übergeben
- Referenz-Parameter können keine Konstanten sein

**Per Value:** Übergabe einer Kopie

- es wird der Wert der Variablen beim Aufruf in einen typgleiche Variable **kopiert**
- eine Veränderung innerhalb der Funktion hat keine Auswirkung „nach außen“, d.h. der Wert der referenzierten Variablen bleibt unverändert
- bei großen Datentypen entsteht großer Kopieraufwand



## 8.12 Funktionen ...

---

### Beispiel:

```
void func_a( int i ) { // per value Übergabe
    i += 3;
}

void func_b( int& i ) { // per reference Übergabe
    i += 3;
}

void func_c( int* i ) { // per reference Übergabe mit Zeiger
    *i += 3;
}

int a = 2;
func_a( a );           // a bleibt 2
func_b( a );           // a wird auf 5 gesetzt
func_c(&a );           // a wird auf 8 gesetzt

func_b( 5 );           // das geht nicht (Compilerfehler)
func_c( 7 );           // ungültige Umwandlung (Compilerfehler)
```



## 8.12 Funktionen ...

---

### Default-Wert für Parameter

**Ziel:** Verwendung optionaler Parameter bzw. von Standardwerten für Funktionsparameter

### **Syntax:**

```
Return-Type  
Name ( Type_1 Arg_1, ..., Type_k Arg_k,  
        Type_{k+1} Arg_{k+1} = Val_{k+1}, ..., Type_m Arg_m = Val_m) {  
    Code  
}
```

**Parameter mit Default-Wert** stehen immer **rechts** von Parametern ohne Default-Wert

**Beim Funktionsaufruf** können Parameter mit Default-Wert **von rechts beginnend** weggelassen werden

⇒ nur so ist eine eindeutige Interpretation des Funktionsaufrufs möglich!

## 8.12 Funktionen ...

---

### Beispiel zur Verwendung von Default-

```
// Standardverhalten: i bleibt unverändert
void increase( int& i, int amount = 0, float factor = 1 ) {
    i = factor * ( i + amount );
}
```

```
int a = 3;
a = increase( a, 5 ); // a <- 1 * ( 3 + 5 )
```

**Bemerkung:** Im obigen Beispiel wird die Angabe eines optionalen Parameters immer dem Variablen `amount` zugeordnet

### Das Parameter-Attribut `const`

**Ziel:** Anzeige, dass ein per-reference-Parameter innerhalb der Funktion nicht geändert wird

**Syntax,** falls *Type<sub>i</sub>* ein Referenz- oder Zeigertyp ist: *Return-Type Name* ( ..., `const Typei Argumenti, ... ) {`

Code

## 8.12 Funktionen ...

---

### Überladen von Funktionen

**Problem:** Funktionen, die dieselbe Funktionalität für verschiedene Parametertypen umsetzen, sollten gleich benannt werden können

**Überladene Funktionen** besitzen den gleichen Namen, aber eine unterschiedliche **Parameterliste** (Typen und Reihenfolge der Parameter)  
Es ist verboten, dass sich alleine der Rückgabetyt unterscheidet!

**Compiler** wählt selbständig die passende Funktion aus

⇒ Achtung: Unter Umständen werden Parameter implizit umgewandelt

### **Beispiel:**

```
int    add ( int    a, int    b ) { return a+b; } // Variante 1
float  add ( float  a, float  b ) { return a+b; } // Variante 2
...
int    i1 = 2, i2 = 4;
float  f1 = 1.1, f2 = 7.9;
int    resInt = add( i1, i2 ); // Compiler wählt Variante 1
float  resFlt = add( f1, f2 ); // Compiler wählt Variante 2
double resDb1 = add( f1, i2 ); // ??? (sollte vermieden werden!)
```



## 8.12 Funktionen ...

---

### Prototypen

**Erinnerung:** Funktionen müssen *deklariert* werden, bevor sie genutzt werden können

**Problem:** Was tun, wenn sich zwei Funktionen gegenseitig aufrufen?

**Bislang:** Deklaration der Funktion durch deren *Definition*, d.h. gleichzeitige Angabe des *Funktionsheaders* (Rückgabewert und Signatur) und des *Funktionsrumpfs*

**Prototyp (=Funktionsheader):** Reine Deklaration einer Funktion möglich

**Beispiel:**

```
bool odd ( int a ); // Prototyp für odd
bool even( int a ); // Prototyp für even
...
bool b = odd( 3 );
...
bool even( int a ) { return !odd(a); } // Definition von even
bool odd ( int a ) { return a % 2; } // Definition von odd
```



## 8.12 Funktionen ...

---

### Die main-Funktion

**main-Funktion** definiert den eindeutigen *Einstiegspunkt* für die Ausführung des Programms

⇒ muss exakt einmal auftreten!

**Deklaration von main:** `int main (int argc, char* argv[])`

- Die Benennung der Parameter ist beliebig, sie werden i.a. aber wie oben benannt
- *argv* ist ein Array von char-Arrays (Zeichenketten)

*argc*, *argv* geben Anzahl und Inhalt der dem Programm beim Aufruf übergebenen Parameter wieder

Die Parameter können auch weggelassen werden, wenn sie nicht genutzt werden sollen: `int main()`

**Nutzung** der Parameter-Inhalte geschieht i.a. durch Umwandlung der Zeichenketten in entsprechende Typen (siehe dazu `string`-Klasse)