Arduino – Programmierpraxis mit Feldvariablen

Eingabe von Werten in Feldvariable (1) – Ausgabe auf dem seriellen Monitor

Im letzten Abschnitt haben wir im Zusammenhang mit der Programmierung des Piezo-Lautsprechers sogenannte **Feldvariable (Arrays)** kennen gelernt. Solche Variablen ermöglichen es, mehrere zusammengehörige Werte des gleichen Datentyps mit einer einzigen Variablen zu speichern.

Wir haben **Feldvariablen** verwendet, um die Notenfrequenzen und die Notenlängen einer gesamten Melodie jeweils zusammen speichern zu können, und haben dabei die **Werte**, die **in** den **Feldvariablen gespeichert** werden sollen, gleich **bei der Deklaration** der Feldvariablen **festgelegt**. Der entsprechende Code hat zum Beispiel für die Feldvariable, die die Notenfrequenzen speichert, so ausgesehen:

```
int frequenzen[] = {297, 330, 352, 396};
```

Alternativ dazu können die in der Feldvariablen gespeicherten Werte auch erst nach der Variablendeklaration eingegeben werden. Dazu verwenden wir den Index, der die einzelnen Speicherbereiche, die zur Feldvariablen gehören, adressiert. Auch diese Variante wurde bereits angesprochen und sieht zum Beispiel so aus:

```
int frequenzen[4];      // Feldgröße bei Deklaration festlegen
frequenzen[0] = 297;    // Feldwerte über den Index zuweisen
frequenzen[1] = 330;
frequenzen[2] = 352;
frequenzen[3] = 396;
```

Diese Eingabevariante ist allerdings recht codeaufwändig – es sei denn, die zugewiesenen Werte lassen sich „automatisch erzeugen“, beispielsweise, weil sie sich mit Hilfe einer Formel berechnen lassen, oder weil sie mit Hilfe von Sensoren gemessen werden. In diesen Fällen können die **Zuweisungen mit Hilfe einer Programmschleife** codiert werden, zum Beispiel so:

```
int intArray[5];

void setup() {
  Serial.begin(9600); //für die Ausgabe
  int index;

  index = 0;
  while(index < 5){
    intArray[index] = 5 - index;
    index = index + 1;
  }
}
```

Neben der gelb unterlegten Programmschleife zur Dateneingabe findet sich ein weiterer neuer Befehl in diesem Code.

`Serial.begin(9600);` ermöglicht die Nutzung des sogenannten **seriellen Monitors** – das ist eine Anwendung, mit der Daten vom Arduino am Computerbildschirm ausgegeben werden können. Die Zahl in Klammer gibt an, mit welcher Geschwindigkeit dabei die Daten vom Arduino an den Computer übertragen werden – zumeist verwenden wir die Symbolrate 9600 und sagen dann, dass die Datenübertragung mit 9600 Baud (Kurzzeichen: Bd) erfolgt.

Die Ausgabe wird in diesem Beispiel im `loop`-Programmteil codiert:

```
void loop() {
  int index;

  index = 0;
  while(index < 5){
    Serial.print(" intArray[");
    Serial.print(index);
    Serial.print("]: ");
    Serial.println(intArray[index]);
    index = index + 1;
  }
  Serial.println("");
  delay(5000);
}
```

`Serial.print("Zeichenkette");` gibt den in Anführungszeichen stehenden Text genauso aus, wie er zwischen die Anführungszeichen geschrieben wurde;

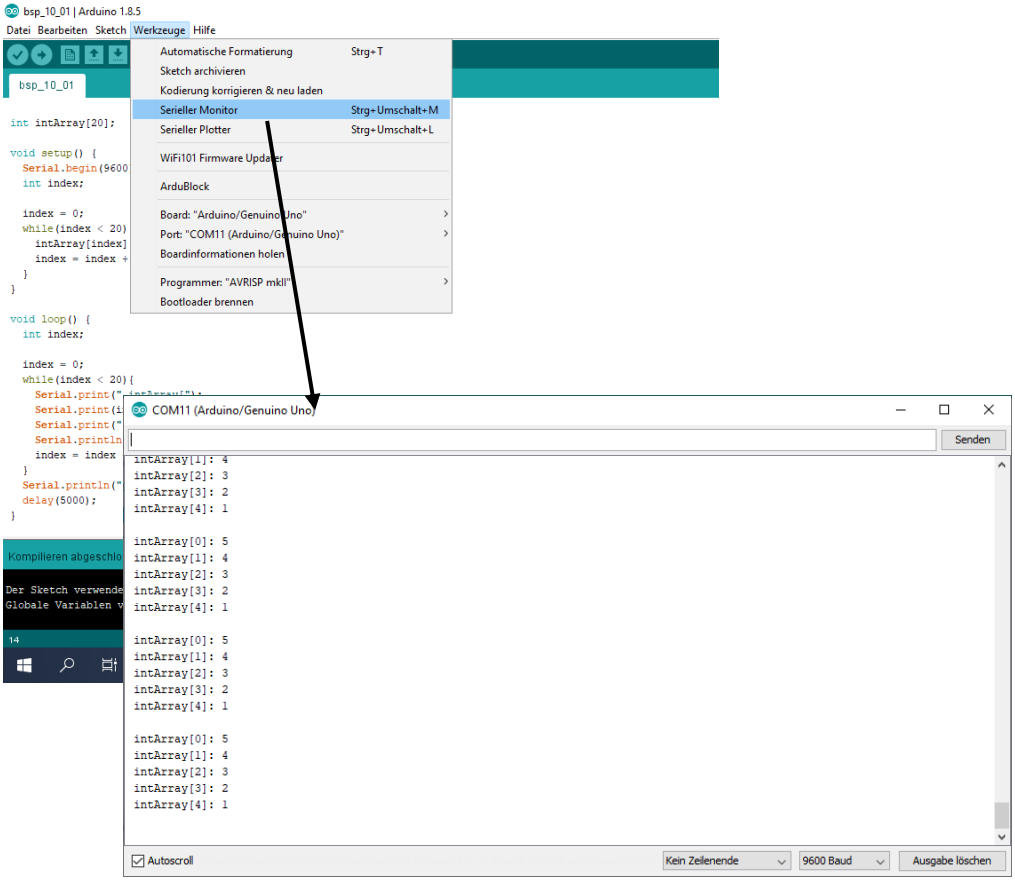
`Serial.print(variable);` hingegen gibt den Wert der Variablen aus, die als Parameter übergeben wird.

Während bei **`Serial.print`** die nächstfolgende Ausgabe in derselben Zeile erfolgt, erfolgt bei **`Serial.println`** ein Zeilenvorschub, d.h. die nächstfolgende Ausgabe erfolgt in der nächsten Zeile unterhalb.

Der oben angegebene Code der „Ausgabe-Programmschleife“ bewirkt daher nebenstehende Ausgabe am seriellen Monitor:

```
intArray[0]: 5
intArray[1]: 4
intArray[2]: 3
intArray[3]: 2
intArray[4]: 1
```

Zuvor (z.B. während das Programm auf das Arduino-Board hochgeladen wird) muss aber der serielle Monitor über den Menüpunkt »Werkzeuge – Serieller Monitor« (oder über die Tastenkombination »Strg + Umschalt + M«) noch angezeigt werden:



Analysieren von (Berechnungen in) Programmschleifen

Selbstverständlich können die in einer Feldvariablen gespeicherten Werte nicht nur gelesen und dann ausgegeben (oder, wie bei der Programmierung des Piezo-Lautsprechers, zum Ansteuern elektronischer Bauteile verwendet) werden. Mit diesen Werten kann auch gerechnet werden! Die folgenden Programmfragmente bieten erste Beispiele (basierend auf der zuvor deklarierten und befüllten Feldvariable namens **`intArray`**):

```
void loop() {
  int index;
  int newValue;

  index = 0;
  while(index < 4){
    newValue = intArray[index] + intArray[index + 1];
    Serial.print(" neuer Wert: ");
    Serial.println(newValue);
    index = index + 1;
  }
  Serial.println("");
  delay(5000);
}
```

```
void loop() {
  int index;

  index = 0;
  while(index < 5){
    intArray[index] = 2*intArray[index];
    index = index + 1;
  }
  index = 0;
  while(index < 5){
    Serial.print(" intArray[");
    Serial.print(index);
    Serial.print("]: ");
    Serial.println(intArray[index]);
    index = index + 1;
  }
  Serial.println("");
  delay(5000);
```

```
void loop() {
  int index;
  int summe;

  summe = 0;
  index = 0;
  while(index < 5){
    summe = summe + intArray[index];
    index = index + 1;
  }
  Serial.print(" Summe aller Zahlen: ");
  Serial.println(summe);
  Serial.println("");
  delay(5000);
}
```

Insbesondere beim letzten, dem nebenstehenden Beispiel, bei der Berechnung der Summe aller im Feld gespeicherten Werte, ist zunächst vielleicht nicht ganz klar, wie die Berechnung – Schritt für Schritt – abläuft. Nachfolgend sollen daher unter Verwendung unseres „naiven“ **Bildes vom Computerspeicher** die einzelnen **Berechnungsschritte innerhalb der Programmschleife veranschaulicht** werden.

Wir beginnen dabei unmittelbar vor der

```
summe = 0;
index = 0;
```

Programmschleife, d.h. nach der Ausführung der beiden Befehle und stellen die Situation im Speicher folgendermaßen dar – die für die Variablen **`summe`**, **`index`**, **`intArray`**, bzw. reservierten Speicherbereiche sind farblich hervorgehoben und bei den Speicherbereichen für die Feldvariable sind zusätzlich die Indizes angegeben:

0	229	244	23	178	50
178	297	255	352	396	181
0	0	196	213	230	62
255	[0] 5	[1] 4	[2] 3	[3] 2	[4] 1

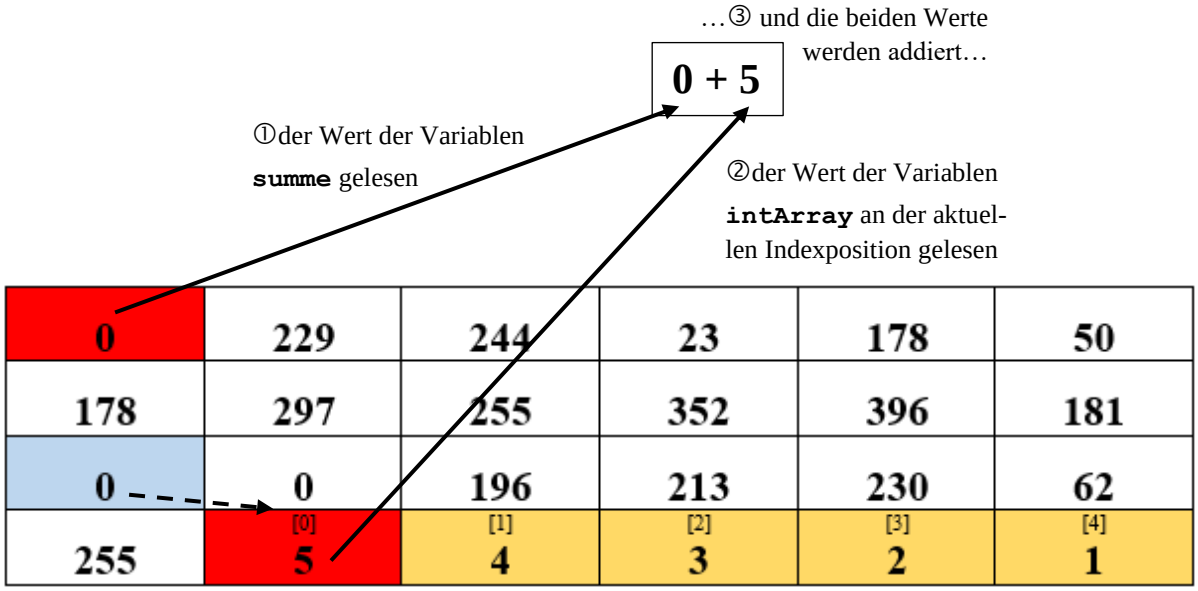
Die Variable **index** hat zunächst den Wert 0,

daher wird die Schleife

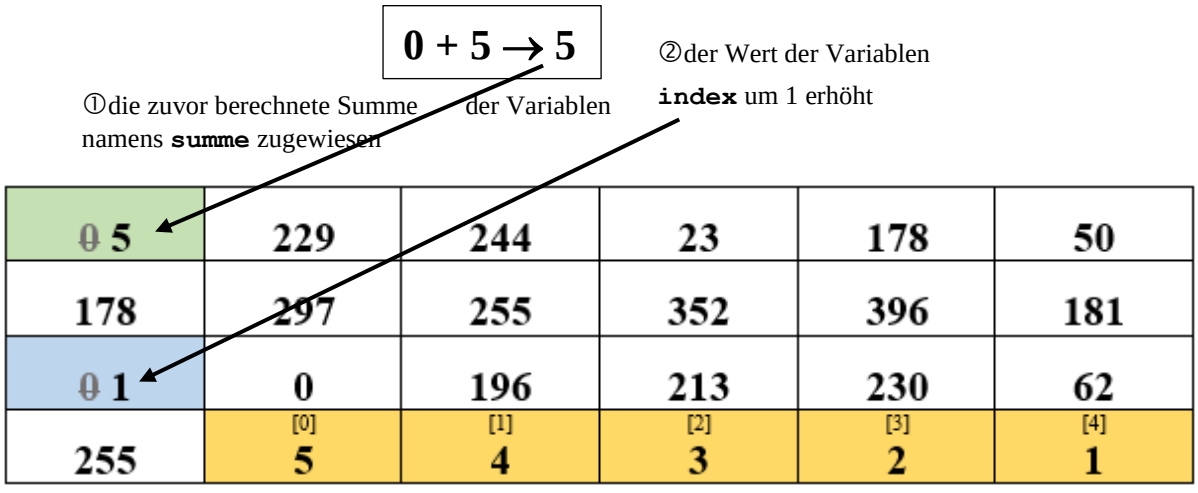
```
while(index < 5){
  summe = summe + intArray[index];
  index = index + 1;
}
```

ausgeführt:

Zuerst wird (`summe + intArray[index];`):



…sodann werden (① `summe = summe + intArray[index];` bzw. ② `index = index + 1;`):



ausgeführt.

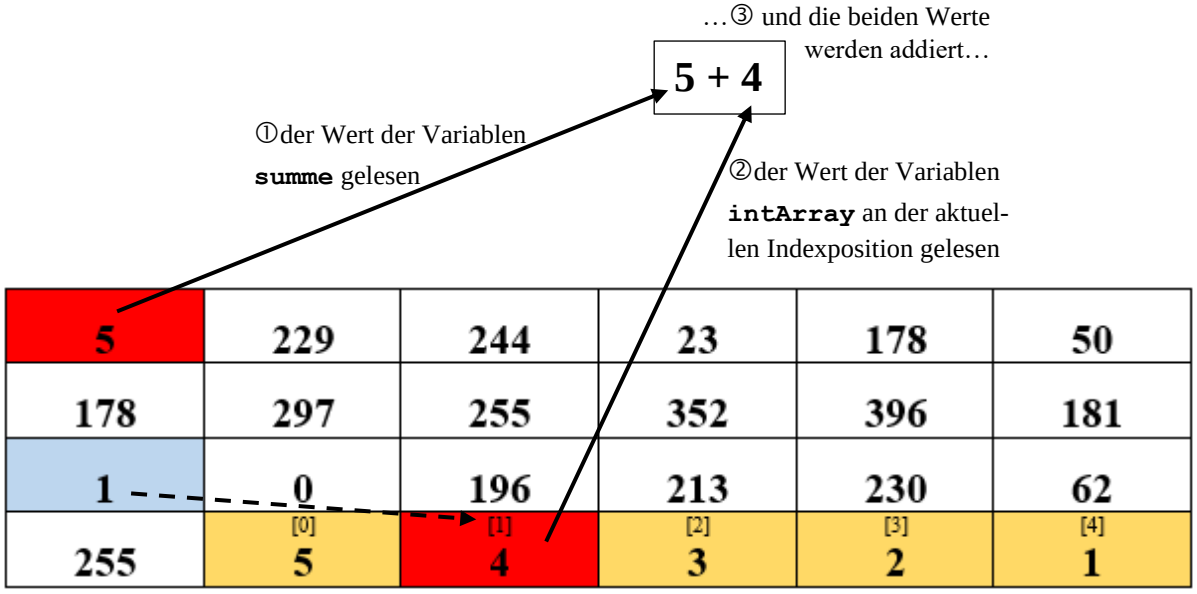
Die Variable **index** hat nun den Wert 1,

daher wird die Schleife

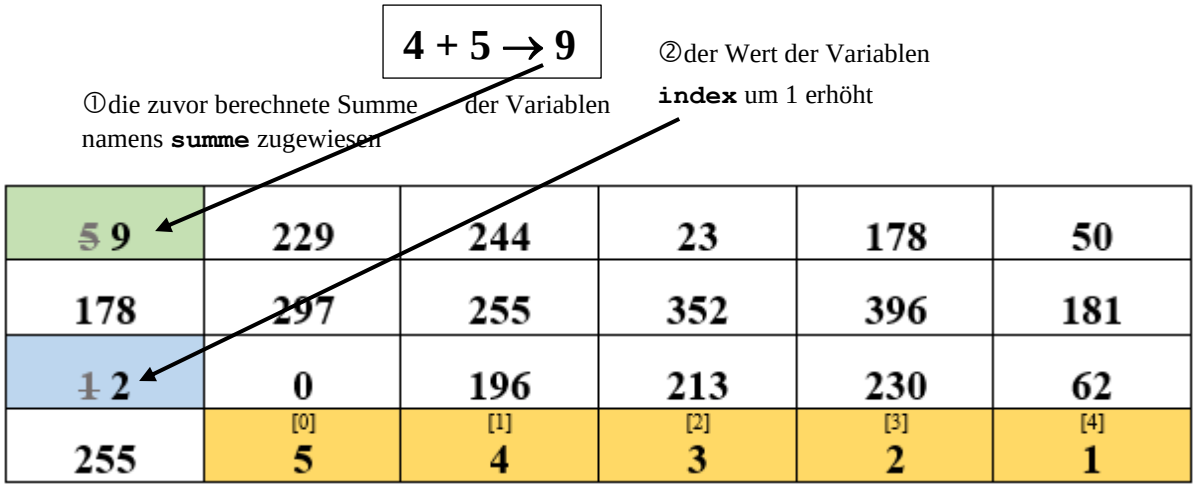
```
while(index < 5){
  summe = summe + intArray[index];
  index = index + 1;
}
```

ausgeführt:

Zuerst wird (`summe + intArray[index];`):



…sodann werden (① `summe = summe + intArray[index];` bzw. ② `index = index + 1;`):



ausgeführt.

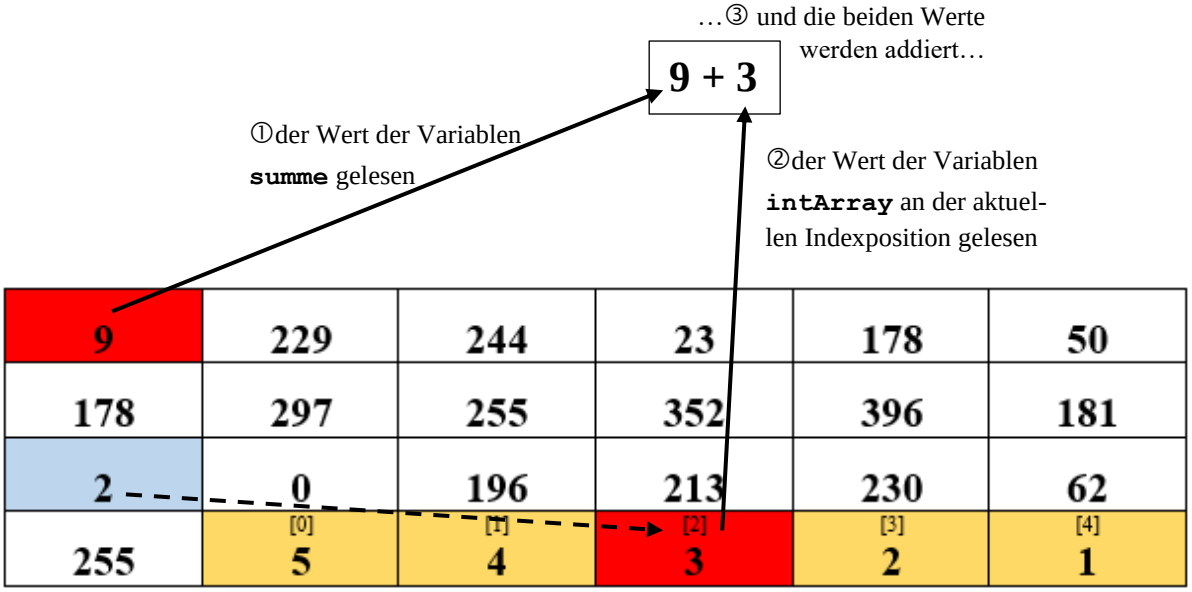
Die Variable **index** hat nun den Wert 2,

daher wird die Schleife

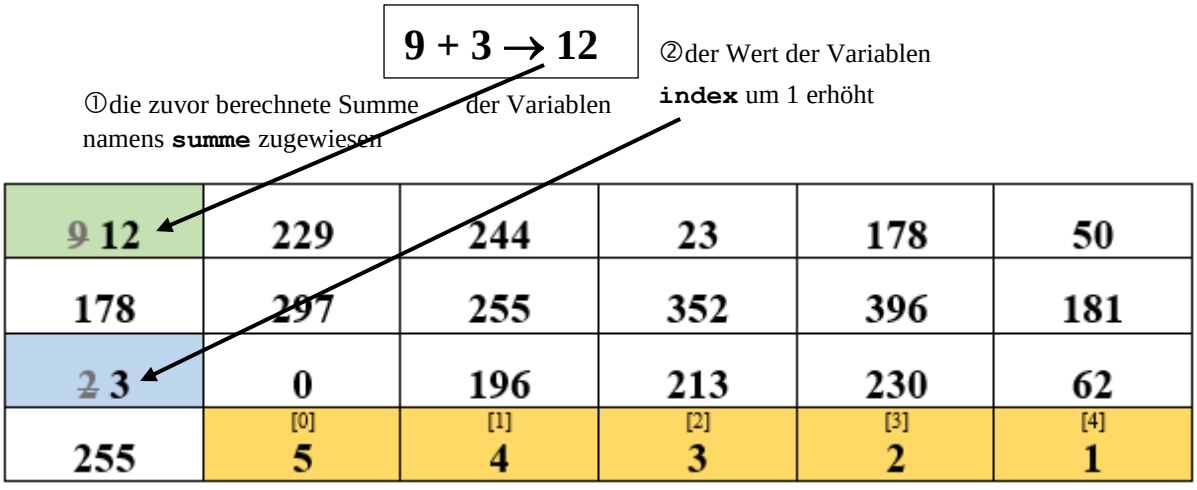
```
while(index < 5){
  summe = summe + intArray[index];
  index = index + 1;
}
```

ausgeführt:

Zuerst wird (`summe + intArray[index];`):



…sodann werden (① `summe = summe + intArray[index];` bzw. ② `index = index + 1;`):



ausgeführt.

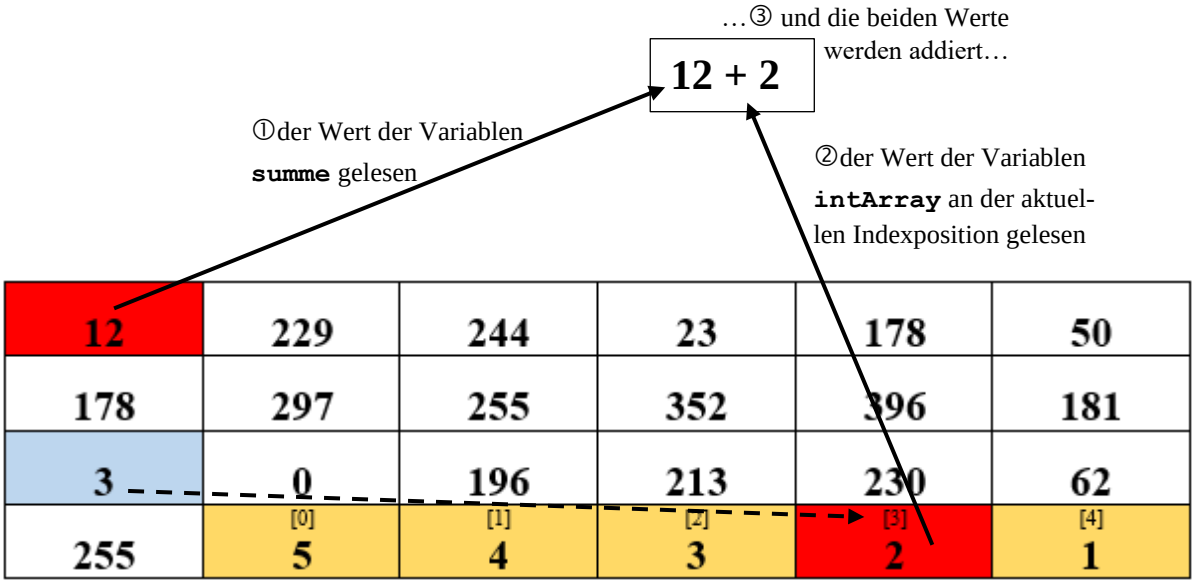
Die Variable **index** hat nun den Wert 3,

daher wird die Schleife

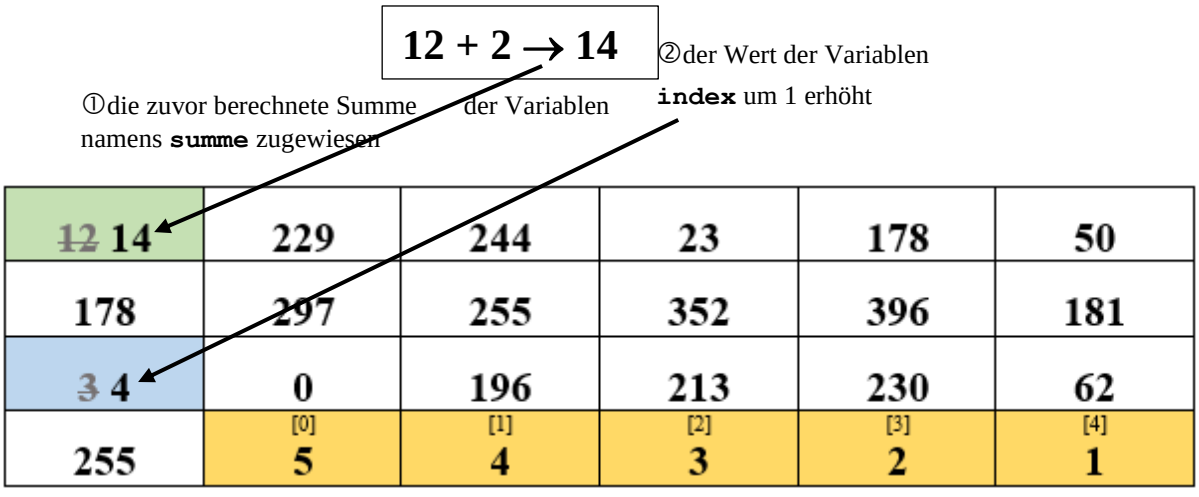
```
while(index < 5){
  summe = summe + intArray[index];
  index = index + 1;
}
```

ausgeführt:

Zuerst wird (`summe + intArray[index];`):



…sodann werden (① `summe = summe + intArray[index];` bzw. ② `index = index + 1;`):



ausgeführt.

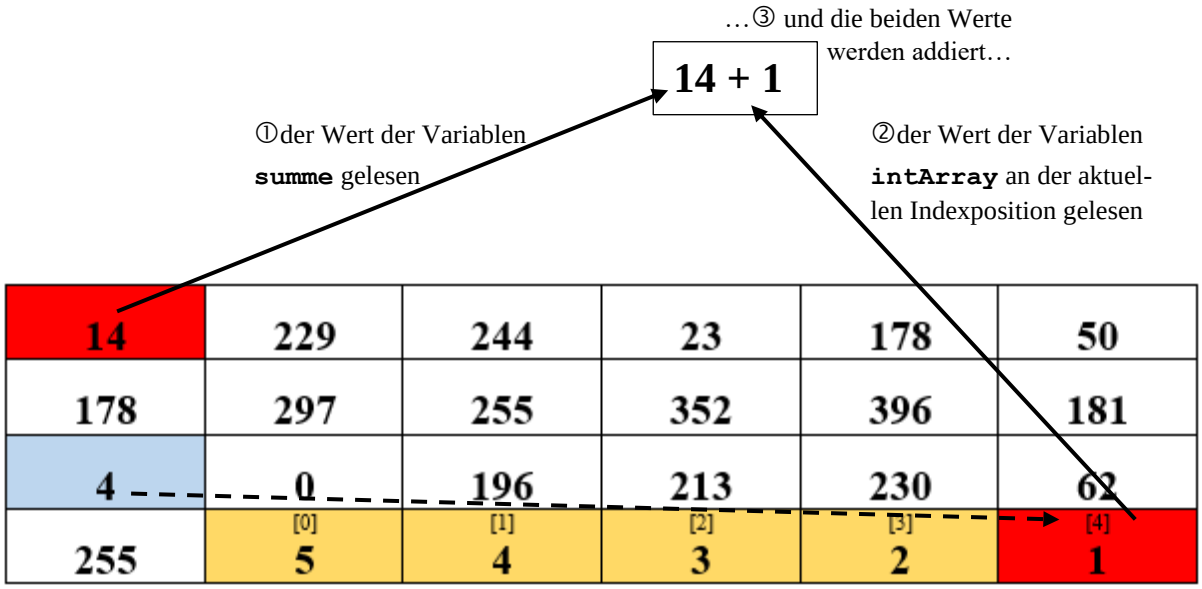
Die Variable **index** hat nun den Wert 4,

daher wird die Schleife

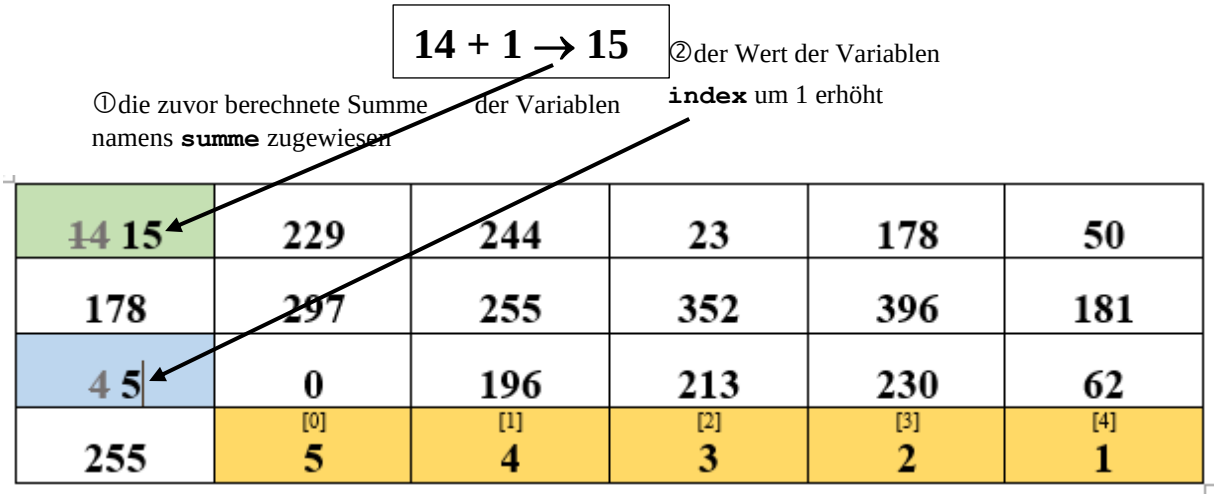
```
while(index < 5){
  summe = summe + intArray[index];
  index = index + 1;
}
```

ausgeführt:

Zuerst wird (`summe + intArray[index];`):



…sodann werden (① `summe = summe + intArray[index];` bzw. ② `index = index + 1;`):



ausgeführt. Die Variable **index** hat nun den Wert 5,

daher wird die Schleife

```
while(index < 5){
  summe = summe + intArray[index];
  index = index + 1;
}
```

nicht mehr ausgeführt:

Das Ergebnis der Summenberechnung ist 15 (= 5 + 4 + 3 + 2 + 1).

Du hast sicherlich gemerkt, dass das soeben praktizierte wiederholte **Zeichnen der Speicherinhalte und der Datenflüsse** ein gutes Bild von den Abläufen beim Durchlaufen einer Programmschleife vermittelt – allerdings ist dieses **Analyseverfahren** ziemlich aufwändig.

Zeitsparender lassen sich Programmschleifen durch **Tabellieren der Variablenwerte für jeden Schleifendurchlauf** analysieren. Das kann für obige Programmschleife zur Summenberechnung so aussehen (der relevante Programmcode wird zum „Mitschauen“ nebenstehend nochmals angegeben):

```
summe = 0;
index = 0;
while(index < 5){
  summe = summe + intArray[index];
  index = index + 1;
}
```

Durchlauf Nr.	index	intArray	summe
0	0	{ 5, 4, 3, 2, 1 }	0
1	**0** → 1	{ **5**, 4, 3, 2, 1 }	0 → 5 (= 0 + 5)
2	**1** → 2	{ 5, **4**, 3, 2, 1 }	5 → 9 (= 5 + 4)
3	**2** → 3	{ 5, 4, **3**, 2, 1 }	9 → 12 (= 9 + 3)
4	**3** → 4	{ 5, 4, 3, **2**, 1 }	12 → 14 (= 12 + 2)
5	**4** → 5	{ 5, 4, 3, 2, **1** }	14 → **15** (= 14 + 1)
…der Wert von **index** ist nicht mehr kleiner als 5, das Durchlaufen der Schleife wird beendet			

Wichtige Punkte bei dieser Art der Analyse von Programmschleifen sind …

- das Erfassen der **Werte aller relevanten Variablen auch VOR dem Durchlaufen** der Schleife. Dies ist in obiger Tabelle als (Schleifen-) Durchlauf Nr. 0 bezeichnet.

Für jede der relevanten Variablen wird dann **für jeden Schleifendurchlauf** vermerkt,

- **wie sich deren Wert verändert** (Wert zu Beginn des Schleifendurchlaufs → Wert am Ende des Schleifendurchlaufs, allenfalls mit Hinweis, wie der neue Wert der Variablen berechnet wird)
- bzw. – bei Feldvariablen – **auf welchen der gespeicherten Werte** aktuell **zugegriffen wird**.

Beachte: Auch wenn wir die beiden Analyseverfahren

- **Versinnbildlichung der Speicherinhalte und der Datenflüsse**
- **Tabellieren der relevanten Variablenwerte**

zum Verstehen von Berechnungen mit Feldvariablen in Programmschleifen kennen gelernt haben, lassen sich diese Verfahren **allgemein** zur **Analyse** von Berechnungen in **Programmschleifen** anwenden, auch wenn nur „einfache“ Variablen und keine Feldvariablen auftreten!

Feldvariable in selbst programmierten Befehlen … und Fragen

Die Berechnung der Summe aller in einer Feldvariablen gespeicherten Werte ist häufig durchzuführen. Daher ist es naheliegend, dafür einen eigenen „Befehl“ zu programmieren, der die Summe als „Antwort“ gibt. Um diesen „Befehl“ möglichst flexibel einsetzen zu können, sollten die Feldvariable, deren Werte addiert werden sollen, und die Größe dieser Feldvariablen als Daten „von außen“ (wir sagen auch: als **Parameter**) an den „Befehl“ übergeben werden.

Nebenstehend ist eine mögliche Codierung eines derartigen Programmblocks abgebildet.

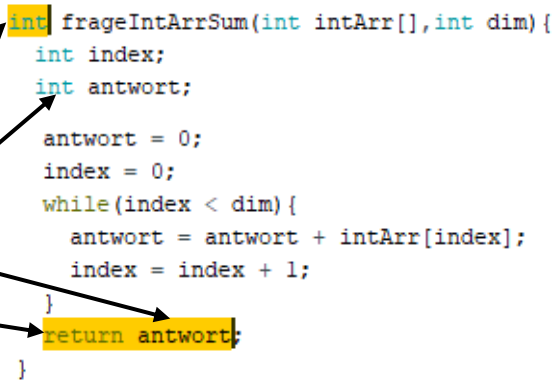
```
int frageIntArrSum(int intArr[],int dim){
  int index;
  int antwort;

  antwort = 0;
  index = 0;
  while(index < dim){
    antwort = antwort + intArr[index];
    index = index + 1;
  }
  return antwort;
}
```

An den Hervorhebungen im Code fällt sofort die neuartige Struktur dieses Programmblocks auf:

In der Überschrift des Programmblocks steht vor dem Namen des Programmblocks das Schlüsselwort `int` und nicht das Schlüsselwort `void`!

Am Ende des Programmblocks wird der Wert der im Programmblock vereinbarten Variablen **`antwort`** als „**Antwort**“ „zurückgegeben“ (`return`).



Vielleicht ist auch aufgefallen, dass für diesen Programmblock der Begriff „Befehl“ zuerst zwar verwendet wurde, aber stets mit Anführungszeichen versehen wurde, damit spürbar wird, dass hier etwas anderes codiert wird: Da dieser Programmblock eine Antwort gibt, ist es naheliegend, von einer **Frage** und nicht von einem **Befehl** zu sprechen.

- Der **Datentyp** `int` vor dem Namen der Frage zeigt an, dass die **Antwort** eine ganze Zahl ist, nämlich ein **Wert vom Datentyp Integer**…
- …daher muss auch die Variable, in der die **Antwort** gespeichert wird, im Programmblock als **Variable vom Datentyp Integer** vereinbart werden – selbstverständlich muss der Name dieser Variablen nicht unbedingt **`antwort`** sein: Dieser Name wurde in diesem Beispiel aber gewählt, um die Rolle dieser Variablen hervorzuheben – übrigens wurde ja auch der Name des Programmblocks aus genau diesem Grund so gewählt: **`frageIntArrSum`**.
- Der Befehl, der veranlasst, dass die Antwort, die in der als Programmblock codierten Frage berechnet wurde, (zurück-) gegeben wird, lautet **`return`**. Das ist leicht zu merken, bedeutet doch „to return“ im Englischen nichts anderes als „zurückgeben“.
- Wird einem Programmblock eine **Feldvariable** als Datum „von außen“, also **als Parameter**, übergeben, so sind **in der Fragen-Überschrift** hinter dem Namen der Feldvariablen eine öffnende und eine schließende **eckige Klammer** zu setzen, in obigem Beispiel: `int` **`intArr[]`**.

- Übrigens: Ein Frage-Programmblock kann selbstverständlich auch mit Parametern codiert werden, die keine Feldvariablen sind, und prinzipiell auch ganz ohne Parameter…

Wenn dieser Frage-Programmblock dann verwendet wird, um die Summe der in einem Feld gespeicherten Wert zu berechnen, muss dafür gesorgt werden, dass die **Antwort der Frage** von einer **passenden Variable "entgegengenommen"** wird. „Passende Variable" bedeutet in diesem Zusammenhang, dass der Datentyp der Variablen zu dem der Antwort passen soll, d.h. diese Variable ist auch als **Variable vom Datentyp Integer** zu vereinbaren.

Ein Beispiel für die Verwendung der Frage **`frageIntArrSum`** ist nebenstehend abgebildet.

```
void loop() {
  int index;
  int summe;

  summe = frageIntArrSum(intArray,5);
  Serial.print(" Summe aller Zahlen: ");
  Serial.println(summe);
  Serial.println("");
  delay(5000);
}
```

Beachte auch, dass bei der Verwendung des Frage-Programmblocks der Name der Feldvariablen OHNE eckige Klammern angegeben werden muss: **`summe = frageIntArrSum(intArray,5);`**

Unterscheide also: Ein selbst codierter Prorammblock stellt

- einen **Befehl** dar, wenn die Überschriftenzeile von der Form

  `void blockName(Parameterliste)`

  ist (vgl. nebenstehendes Beispiel für einen Befehl mit – unter anderem – einer Feldvariablen als Parameter).

```
void befehlPrintIntArray(int intArray[], int dim){
  int index;

  index = 0;
  while(index < dim){
    Serial.print("Wert an Indexposition ");
    Serial.print(index);
    Serial.print(": ");
    Serial.println(intArray[index]);
    index = index + 1;
  }
}
```

- eine **Frage** dar, wenn die Überschriftenzeile zum Beispiel von der Form

  `int blockName(Parameterliste)`

  ist (vgl. nebenstehendes Beispiel für eine Frage mit – unter anderem – einer Feldvariablen als Parameter).

  In diesem Fall muss der letzte Befehl innerhalb des Programmblocks der **`return`**-Befehl sein, der den Wert einer Variablen passenden Datentyps (hier: `int`) als **Antwort** „zurückgibt".

```
int frageIntArrSum(int intArr[],int dim){
  int index;
  int antwort;

  antwort = 0;
  index = 0;
  while(index < dim){
    antwort = antwort + intArr[index];
    index = index + 1;
  }
  return antwort;
}
```

  Diese Antwort muss in dem Programmteil, in dem der Frage-Programmblock aufgerufen wird, von einer Variablen des selben Datentyps entgegengenommen werden.

```
  int summe;

  summe = frageIntArrSum(intArray,5);
```

ACHTUNG!

Der Prozessor des Arduino Uno-Boards speichert in einer Variablen vom Datentyp `int` nur **ganze IZahlen im Bereich von – 32 768 bis 32 767.**

Wenn das Ergebnis einer Berechnung eine kleinere oder größere Zahl ist, wird „zyklisch weitergerechnet“. Das bedeutet: Wenn zum Beispiel bei einer Addition die Summe 32770 ist und diese Summe in der `int`-Variablen **`sum`** gespeichert werden soll, dann ist der **Wert**, der in der Variablen **`sum`** gespeichert wird, **NICHT 32770 SONDERN -32766!!** Nach 32767 kommt nämlich – 32768, und da nach – 32767, – 32766, – 32765 usw.

Es gibt aber einen weiteren Datentyp für ganze Zahlen, mit dem auch große ganze Zahlen (bis ca. zwei Milliarden) gespeichert werden können: `long int` (oder kurz: `long`).

Wenn also beispielsweise die Summe von vielen und/oder großen ganzzahligen Werten berechnet werde soll, ist sinnvoll, den Datentyp für große („lange“) ganze Zahlen zu verwenden, z.B. also:

```
long frageIntArrSum(int intArray[], int dim){
  int index;
  long antwort;

  index = 0;
  antwort = 0;
  while(index < dim){
    antwort = antwort + intArray[index];
    index = index + 1;
  }
  return antwort;
}
```

Natürlich muss dann auch die Variable, die diese Antwort engegennimmt, vom Datentyp `long` sein:

```
long summe;
...
summe = frageIntArrSum(intArray,5);
```

Hinweis:

Die Programmiersprache C für Arduino-Boards kennt noch einen weiteren ganzzahligen Datentyp namens `byte`. In einer Variablen dieses Datentyps können aber nur Zahlen im Bereich von 0 bis 255 gespeichert werden. Dennoch kann auch die Verwendung dieses Datentyps sinnvoll sein, wenn nur kleine ganze Zahlen benötigt werden: Eine Variable vom Datentyp `byte` benötigt weniger Speicherplatz als eine Variable eines anderen Datentyps für ganze Zahlen – und bei manchen Geräten, zum Beispiel bei manchen mobilen Anwendungen, ist der verfügbare Speicher begrenzt.

Bemerkung:

Neben ganzzahligen Datentypen gibt es in der Programmiersprache auch Datentypen für das Speichern von Kommazahlen – ein solcher Datentyp ist `float`. Bei der Wertzuweisung an Variable vom Datentyp `float` ist aber aufzupassen, wenn der zugewiesene Wert das Ergebnis einer Division ist und Dividend und Divisor ganzzahlig sind:

Der **Quotient zweier ganzer Zahlen ist** in der Programmiersprache C nämlich **stets** wieder **eine ganze Zahl**, die Nachkommastellen werden einfach abgeschnitten!

Im Codefragment

```
float kommaZahl;
int ganzeZahl = 12;
kommaZahl = ganzeZahl/5;
```

wird demnach in der Variablen namens **kommaZahl** der Wert 2 gespeichert, obwohl diese Variable eine Kommazahl speichern könnte. Da aber sowohl die Variable **ganzeZahl** als auch 5, die bei der Division auftretenden Operanden, ganzzahlig sind, wird der Nachkommaanteil „weggelassen".

Wenn dennoch das „korrekte" Ergebnis 2.4 in der Variablen **kommaZahl** gespeichert werden soll, muss zumindest ein Operand der Division für die Berechnung in eine Zahl vom Datentyp `float` umgewandelt werden, zum Beispiel so:

```
kommaZahl = ganzeZahl/5.0;
```

oder so:

```
kommaZahl = ganzeZahl/(float)5;
```

oder so:

```
kommaZahl = (float)ganzeZahl/5;
```

Eingabe von Werten in Feldvariable (2)

Für das Testen von Befehlen oder Fragen, die auf den in einer Feldvariablen gespeicherten Werten operieren, ist es häufig wünschenswert, die Feldvariable mit „schwer vorhersagbaren" Zahlen zu befüllen. Etwas ungenau spricht man in diesem Zusammenhang auch von „**Zufallszahlen**" – ungenau ist diese Bezeichnung deshalb, weil diese Zahlen nicht „zufällig" entstehen, sondern auch berechnet werden. Allerdings ist das Berechnungsverfahren, der **Algorithmus**, so kompliziert, dass die jeweils nächste **berechnete Zahl** schwer vorhergesagt werden kann und daher **zufällig erscheint**.

Das nebenstehende Programmfragment zeigt, wie solche schwer vorhersagbaren Zahlen für die Verwendung mit dem Arduino-Board berechnet werden können:

```
int intArray[5];

void setup() {
  Serial.begin(9600);
  int index;
  index = 0;
  randomSeed(analogRead(A1));
  while(index < 5)
    intArray[index] = random(1,101);
    index = index + 1;
  }
}
```

Mit dem Befehl `randomSeed` wird ein Startwert für die Berechnung erzeugt. Damit dieser Startwert möglichst „zufällig" ist, kann das Signal von einem analogen Port als Startwert dienen – im Beispiel wird das Signal von Port A1 verwendet, `randomSeed(analogRead(A1));`. An diesem Port sollte zu diesem Zeitpunkt natürlich kein Sensorsignal anliegen, ansonsten wäre der Startwert doch wieder vorhersagbar!

Ausgehend von diesem Startwert berechnet dann die Frage `random(1,101)` ganze Zahlen im Bereich von 1 bis 100 (!), die zufällig erscheinen. Fünf dieser schwer vorhersagbaren ganzen Zahlen werden in der Programmschleife dann mit `intArray[index] = random(1,101);` als Antworten in einer Feldvariablen gespeichert.