3.1. Arduino Programmeren voor Beginners – Deel 3: Werken met Informatie

Arduino Programmeren voor Beginners

In deze derde les van Arduino Programmeren voor Beginners, gaan we praten over informatie of te wel “data” en hoe we daar meer kunnen werken. We gaan kijken naar soorten informatie, hoe we informatie kunnen opslaan in “variabelen” en “constanten”. Het afhandelen van informatie is natuurlijk bij uitstek waar computers voor gemaakt zijn.

Inhoudsopgave

3.2. DataTypes: Data voor Arduino Programmeren

Elk programma werkt op de een of andere manier met data – kenmerkend voor computers. Informatie, of te wel “data”, wordt weergegeven, ingelezen, uitgevoerd, veranderd, gesorteerd, etc. etc. – Wat je ook doet met een computer, er wordt data verwerkt.

Waarom Data Types?

Informatie, data dus, kennen we in verschillende vormen en formaten, en iedere “variant” zou de computer op een andere manier moeten behandelen. Ik zal proberen dat uit te leggen.

Laten we beginnen met twee basis types van “data”, als voorbeeld: Tekst en Nummers.
Als voorbeeld waarom deze twee verschillende data typen anders behandelt moeten worden, zou je kunnen denken aan het volgende:
Je kunt twee teksten niet met elkaar vermenigvuldigen, maar twee nummer wel.
Je kunt twee stukken tekst achter elkaar plakken, maar bij nummer eindig je dan mogelijkerwijs met onzin.

Als voorbeeld dus “abc” × “def” gaat nergens over, maar “abc” + “def” zou “abcdef” kunnen worden. Hetzelfde met nummers; 3 × 4 levert 12 en 3 + 4 levert 7 (en dus niet “34“)!

Omdat tekst ook nummers kan bevatten, wordt het e.e.a. nog ingewikkelder voor een computer.

Het wordt nog lastiger als we weten hoe tekst en hoe nummers in de computer worden opgeslagen. We moeten daarvoor even teruggaan naar de uitleg van “bit” en “byte” in het voorgaande deel. We weten dat een byte uit 8 bits bestaat en voldoende ruimte biedt om letters op te slaan. Dus als we tekst opslaan dan gebruiken we een byte per letter.

Als we met nummers werken, dan veranderd dit. Een byte kan namelijk een nummer van 0 … 255 opslaan.
Met het nummer 34 in gedachten:
Als we “34” als tekst opslaan, dan hebben we dus 2 “letters” en gebruiken we dus minimaal 2 bytes.
Als we het nummer 34 opslaan, dan zien we dat dit al in 1 byte past!

Dit zijn natuurlijk eenvoudige voorbeelden, en in werkelijkheid kunnen deze voorbeelden nog veel complexer. Dit zijn natuurlijk eenvoudige voorbeelden, maar ik denk dat je begrijpt dat de computer verteld moet worden met wat voor soort data hij moet werken. Dit is nodig om te weten wat voor soort handeling er verricht moet worden en hoe hij de data moet opslaan

Daarom wordt er in de meeste programmeertalen de zo genaamde “data types” worden gedefinieerd zodat de computer weet hoe hij met de data moet omgaan.

3.3. Data Types definiëren hoe data behandelt en opgeslagen wordt.

Data Types voor de Arduino

Je hebt misschien al gezien dat ik, rond tekst, dubbele aanhalingstekens plaats (“). Dat is de manier waarom in de programmeertaal C (de taal die voor Arduino Programmeren gebruikt wordt) de gebruikelijke manier om tekst aan te geven. Een dergelijke reeks karakters (tekst) noemt men een “string“, en een “string” is een van die zogenaamde “data types”.

De taal C kent een aantal verschillende “Data Types”, en ik laat ze hieronder zien zoals de Arduino ze kent – maak je niet ongerust als je niet meteen begrijpt wat het allemaal betekend. Naar mate je ze meer en meer gaat gebruiken wordt het e.e.a. duidelijker en zul je ze ook gemakkelijker onthouden.

Data Types

Data Type	Toepassing / Type waarden	Geheugen verbruik
array	Een array kan gezien worden als een reeks waarden welke aangesproken kunnen worden met behulp van een zogenaamd index nummer.	Wisselend
boolean	Een boolean kan slecht 1 van deze twee waarden aannemen: true (waar) of false (niet waar)	1 byte
byte of unsigned char	Een geheel nummer tussen 0 … 255	1 byte
char	Een enkel teken (of character – zie onze character lijst)	1 byte
double	Een nummer met cijfers achter de komma, met dubbele precisie (op de meeste Arduino’s hetzelfde als een float)	4 bytes
float	Een nummer met cijfers achter de komma, bereik van -3.4028235E+38 … -3.4028235E+38 (niet super nauwkeurig!)	4 bytes
int or short	Geheel nummer tussen -32,768 … 32,767	2 bytes
long	Geheel nummer tussen -2,147,483,648 … 2,147,483,647	4 bytes
string (char array)	Een array van Char(acters)	Wisselend
String (object)	Een string object, kost iets meer geheugen maar biedt wat extra handigheidjes	Wisselend
unsigned int of word	Een geheel nummer tussen 0 … 65,535	2 bytes
unsigned long	Een geheel nummer tussen 0 … 4,294,967,295	4 bytes
void	“niks” (alleen gebruikt voor functie definities)	N.V.T.

N.b. : De data types waar je “wisselend” ziet staan in de geheugen verbruik kolom, groeien naar behoeft (en nemen dus meer of minder geheugen in beslag).

N.b. : Berekeningen met cijfers met nummers achter de komma is beduidend langzamer dan berekening met gehele nummers.

Zoals je ziet; niet ieder data type, neemt even veel geheugen in beslag. Men probeert data (informatie) zo efficiënt mogelijk op te slaan. Soms kan dat resulteren in onverwachte resultaten als je bijvoorbeeld een data type kiest welke niet alle data kan bevatten. Bijvoorbeeld een “byte” (0-255) voor het opslaan van het nummer 300.

Arduino’s hebben maar een beperkte geheugencapaciteit, ga er dus zuinig mee om!

Nu dat we een hele lijst hebben gezien, gaan we kijken naar de meest gangbare typen data: booleaans, int, char, string, float en array.
Array zullen we later bekijken, laten we eerst eens gaan kijken naar de eenvoudigere typen.

Als voorbeeld beginnen we met de boolean waarden. Een boolean kan slechts èèn van de volgende twee waarden aannemen: “true” (waar) of “false” (onwaar of niet waar). Je kunt dat ook lezen als Ja/Nee, Aan/Uit, etc. en wordt heel vaak gebruikt bij het vergelijken van informatie. Bijvoorbeeld “is het licht aan?” – we vergelijken de informatie (licht aan) met een mogelijk antwoord (Ja of Nee). Dit gaat straks dus handig worden bij het aan- of uitzetten van lampjes, of het lezen van schakelaar of sensoren.

Bedenk dat “true” en “false” vast zijn gedefinieerd in de Arduino (IDE) en altijd worden getypt met kleine letters.

De boolean waarden “true” en “false” worden opgeslagen als nummers.
Initieel zul je hier niet veel mee doen, maar het is belangrijk om te weten dat true = 1 en false = 0 .

Int (integer) is een geheel nummer, dus een nummer zonder cijfers achter de komma. Dit data type is een van de makkelijkste typen o mee te werken en worden dus vaak voor berekeningen gebruikt zoals optellen, aftellen, etc. of bijvoorbeeld voor pinnummers enzo.

Een char (character is het Engelse woord voor “letter”) wordt gebruikt voor letters, maar ook cijfers en bijzonder tekens – eigenlijk elke “letter” die we in een string kunnen plaatsen – wat ons bij een complexer data type brengt: de string.

Er zijn 2 string soorten: string (array van chars) en String (object).
Haal ze niet door elkaar want ze zijn echt andere beestjes!

Arduino kent 2 soorten van strings.

De ene is een string en is wat eenvoudig, namelijk een array van characters (een reeks van char), en wordt met een kleine “s” geschreven.

De andere String is een stukje meer complex omdat het een zogenaamd “object” is – dit string type wordt met een hoofdletter “S” geschreven. We praten later meer over objecten, maar voor het moment is het goed te weten dat objecten vaak met handige functies komen.

We hebben dus 3 soorten basis data typen: nummers (int, long, float, double, etc.), tekst (char en strings) en Booleaans (true/false).
De Array types zijn een groepering, reeks of lijst van deze “eenvoudigere” data typen.

3.4. Wat zijn Variabelen?

We hebben even snel gekeken wat voor soorten data er zijn – en daar zouden we nog wel een paar dagen over door kunnen praten. Maar wat doen we nu met deze zogenaamde “data typen”? Nou, we gaan ze gebruiken in wat ze “variabelen” noemen.

Variabelen zijn zeg maar “namen” die wijzen naar een specifiek stukje geheugen -zeg maar het “adres” waar de data staat in het geheugen.
Om nu te weten wat te verwachten, moeten we computer (Arduino) dus niet alleen vertellen waar het staat, maar ook wat er staat. Zoals je al eerder zag; tekst en nummers werken niet hetzelfde, en verschillende data typen kunnen een verschillende hoeveelheid geheugen gebruiken.

Deze “namen” gebruiken we in ons programma om te refereren naar informatie of data.

Variabelen zijn een soort “”aanduiding” of “plaat houder” voor de data die we in ons programma gebruiken (meer correct: de geheugenlocatie waar onze data gevonden kan worden).

Klink lekker onduidelijk? Geen probleem, laten we eens naar wat voorbeelden kijken.

Hier onder wat voorbeelden hoe we variabelen kunnen gebruiken (dit is niet helemaal correct notatie voor de taal C!) – we definiëren wat variabelen en doen er wat berekeningen mee:

```
1A = 4
2B = 12
3C = 4 + 12   // = 16
4D = A + B    // = 4 + 12 = 16
5C = 4 + A    // = 4 + 4 = 8
6A = A + A    // = 4 + 4 = 8
```

In de eerste regel maken we een variabele met de naam “A” welke we de waarde 4 geven als “data”. De tweede regel is vergelijkbaar maar nu definiëren we de variabele “B”, met de waarde 12.

Wanneer we de variabele “A” aanroepen, dan wijzen we eigenlijk naar diens waarde … 4.
Hetzelfde geldt voor de variabele “B”, met een waarde van 12.

We kunnen in regel 3 zien dat we er ook mee kunnen rekenen: We maken de variabele “C” en geven het de waarde van 4 en 12 opgeteld (16).

Omdat de variabele de een waarde heeft kunnen we daar ook mee gaan rekenen zoals we in regel 4 zien waar we [de waarde van] “A” en “B” optellen.
We tellen de de waarden van de variabele “A” (4) en de variabele “B” (12) op, wat resulteert in 16, en die waarde wijzen we toe aan de variabele D.

De 5de regel laat zien dat we variabelen en nummers door elkaar kunnen gebruiken door de variabele “C” een nieuwe waarde tegen, wat het resultaat is van het optellen van 4 en de waarde van de variabele “A”. De “oude” waarde van de variabele “C” wordt hierdoor dus overschreven!

Regel 6 maakt het nog gekker. We tellen de waarde van de variabele “A” en de waarde van dezelfde variabele “A” op (wat dus 4+4 is) en slaan dit weer op in weer dezelfde variabele “A” (8). Eerst telt de computer dus de waarden op, en vervolgens slaat die het resultaat op in de variabele, welke nu de waarde “8” heeft.

Dus … variabelen zijn variable – ze kunnen veranderen.

Tot dusver gebruikte ik simpele letters voor de variabelen, maar je kunt er beter zinvolle namen voor gaan gebruiken waardoor het programma voor ons mensen beter te lezen is.

Variabelen dienen namen te hebben die zinvol zijn en bijdragen aan de leesbaarheid van jouw programma…

Een voorbeeld:

```
1ZakGeld = 4
2SpaarGeld = 12
3AlMijnGeld = ZakGeld + Spaargeld
```

Je ziet meteen wat we hier aan het doen zijn – stukken beter leesbaar dan alleen maar een “A” of een “B”.

Let echter wel op dat de namen van variabelen hoofdletter gevoelig zijn. “ZakGeld” is dus niet hetzelfde als “zakGeld”, “Zakgeld”, “zakgeld” of “ZAKGELD”!

Variabelen, en andere namen in code, zijn over het algemeen hoofdletter gevoelig!

We moeten echter nog een paar extra regels in de gaten houden. We mogen namelijk alleen maar letters, nummers en een zogenaamde underscore (liggend streepje: _) gebruiken, maar niet speciale tekens of spaties!

Namen van variabelen:

Beginnen ALTIJD met een letter (a, b, …, z, A, B, … , Z) of een underscore (_)

mag letters bevatten

mag underscore(s) bevatten

mag numbers bevatten

MAG NOOIT speciale tekens, symbolen of spaties bevatten

Laten we eens een voorbeeld programma maken.
We gaan in dit “programma” alleen maar gehelen getallen gebruiken – kun je al raden welk data type we nodig zouden kunnen hebben?

Daarvoor moeten we eerst wat meer weten. Stel we gaan ons eigen geld tellen en we hebben het niet al te breed, dan zou het kunnen zijn dat een int genoeg is (€32.768).

Omdat we alles maar 1 keer willen doen, zullen we onze “code” maar even in de “setup()” zetten.
Ter herinnering: “setup()” wordt maar 1x doorlopen, “loop()” wordt eindeloos herhaald.

Maar omdat we resultaten willen zien moeten we dus eerst de praat snelheid even instellen tussen de computer en de Arduino – zodat de “Seriële Monitor” functioneert, zoals we dat eerder gedaan hebben.

Daarna gaan we 3 variabelen definiëren: ZakGeld, SpaarGeld en AlMijnGeld en we maken deze gehele nummers, en voor het voorbeeld kiezen we dus “int”.
Zie je hoe we iedere regel (statement) in onderstaande code met een punt-komma afsluiten? Niet vergeten he!?

In de volgende stap gaan we de variabelen een waarde geven.

Uiteindelijk sturen we de resultaten naar de “Seriële monitor”.
Hiervoor gebruiken we weer “Serial.print” zodat we kunnen zien wat de naam van de variabelen kunnen zien.
En vervolgens gebruiken we “Serial.println” om de waarde van de variabele weer te geven. Hier zien we een klein verschil tussen “Serial.print” en “Serial.println” – die laatste begint namelijk een nieuwe regel na het printen van de informatie. De extra “ln” (Engels: LINE) zorgt hiervoor.

```
1  void setup() {
2    // snelheid voor de seriële monitor:
3    Serial.begin(9600);
4
5    // variabelen definiëren als "int"
6    int ZakGeld;
7    int SpaarGeld;
8    int AlMijnGeld;
9
10   // variabelen waarden geven
11   ZakGeld = 4;
12   SpaarGeld = 12;
13   AlMijnGeld = ZakGeld + SpaarGeld;
14
15   // variabelen weergeven
16   Serial.print("ZakGeld = ");
17   Serial.println(ZakGeld);
18
19   Serial.print("SpaarGeld = ");
20   Serial.println(SpaarGeld);
21
22   Serial.print("AlMijnGeld = ");
23   Serial.println(AlMijnGeld);
24 }
25
26 void loop() {
27 // leave empty for now
28 }
```

Kopieer deze code en plak het in de Arduino IDE waarbij het eventuele bestaande code (tekst) geheel vervangt.
Klik vervolgens op de “Compileer en Upload” knop (het kan zijn dat de Arduino IDE vraagt of je deze code wilt opslaan – gewoon doen en op “Save” klikken).
Na wat geknipper start de Arduino het programma en zien we het volgende in het Seriële Monitor venster:

ZakGeld = 4
SpaarGeld = 12
AlMijnGeld = 16

3.5. Wat zijn Constanten?

Constanten zijn “namen” (vergelijkbaar met zoals we namen in variabelen gebruikt hebben) die ook een waarde hebben, maar welke nooit veranderen in een programma.
Ze worden vaak gebruikt als een bepaalde vaste waarde wordt gebruikt of als een waarde vaak gebruikt wordt in een programma, maar in de toekomst zou je deze waarden willen veranderen zonder dat je ieder voorkomen van deze waarde in jouw programma moet terug gaan zoeken.

Een Constante lijkt op een variabele, maar dan met een Vaste Waarde.

Namen van Constanten volgende dezelfde regels als voor variabelen.

Je zou kunnen zeggen dat we 3 soorten constanten hebben.

We hebben de constanten die in de Arduino IDE of Arduino Bibliotheken zijn gedefinieerd, en diegene die we in ons programma definiëren (b.v. “true” en “false”).

Ehm, je zei toch dat we er “3” hadden – ik zie er maar 2 … !?
Dat klopt helemaal en dat komt omdat we 2 manieren hebben waarop we onze eigen constanten kunnen definiëren.

We kunnen constanten definiëren met het trefwoord “const“,
of we kunnen een zogenaamde “compiler aanwijzing” (compiler directive) gebruiken: “#define“.

Het grootste verschil is wel het gebied (scope) waarin de constante werkt of beschikbaar is – ik laat je zo het verschil zien.

Gebruik “const” als je deze alleen in een bepaald deel van het programma wilt zien, of als je denkt dat je deze later gaat omzetten naar een variabele.

Gebruik “#define” voor de meeste andere gevallen.

const

Het definiëren van een constant wordt als volgt gedaan voor “const”:

```
const int AantalLichten = 5;
```

Dit lijkt veel op de manier waarop we een variabele definiëren, misschien kun jij je dit nog herinneren:

int AlMijnGeld;

Een “const” constante gedraagt net als een variabele …
Je kunt alleen z’n waarde niet veranderen.

We hebben er gewoon het woord “const” voor geplakt, zodat we weten dat deze waarde nooit veranderen, en we wijzen het meteen een waarde toe met “ = 5 “.

Dus niet vergeten: Je kunt de waarde dus NIET veranderen als het programma draait.
Als we dit in de “setup()” doen, dan zal deze constante alleen maar bekend zijn in “setup()” – je ziet ‘m dus niet in “loop()”.

#define

Een alternatieve manier van constanten gebruiken is met de define compiler instructie – wat aan het begin van onze code gedaan wordt:

#define AantalLichten 5

We zien hier een aantal belangrijk verschillen.

Als eerste, starten we deze regel met een hash (hekje of pound, of te wel dit symbool: #). We hebben dat nog niet eerder gezien en het geeft de compiler (vertaler) specifieke opdrachten die uitgevoerd worden voor de vertaling start. In principe zegt het “voor je gaat vertalen, vervang alle AantalLichten in de code met het nummer 5“. Dus in tegenstelling van de “const” methode, wijst deze dus niet naar een geheugenlocatie!
Er zijn nog meer compiler instructies maar we laten het even bij deze.

In tegenstelling tot gewone instructies zien we ook dat de punt-komma ontbreekt (;) – wat alleen correct is voor compiler instructies en commentaar.

En als laatste type we gewoon “5” in plaats van ” = 5 ” – het is-gelijk symbool is hier niet van toepassing.

Als je een constante op deze manier definieert, dan is deze over het gehele programma bekend en bereikbaar.

Wat is Scope (bereik of werkgebied)?

We hebben het net even vermeld: Scope – wat zoiets wil zeggen als bereik of (werk)gebied.
Dit is van toepassing voor zowel variabelen, constanten en zelfs functies. Maar dus niet van toepassing voor “#define”!

Wanneer we dus over de “scope” van een variabele of constante praten, dan bedoelen we dus eigenlijk “dat deel van onze code waar deze constante of variabele bestaat en zichtbaar is”.

N.b.: Vanaf dit punt zal ik praten variabelen, maar de “const” constanten gedragen zich identiek aan variabelen – met het verschil natuurlijk dat je de constanten niet kunt wijzigen.

De Scope van een variabele geeft aan “waar” deze gezien en gebruikt kan worden in de code van ons programma.

Hierdoor kennen we twee hoofdgroepen van variabelen: lokale variabelen en globale variabelen.

Globale variabelen zijn OVERAL in jouw programma te zien. Om dat voor elkaar te krijgen moeten ze wel op een speciale plaats gedefinieerd worden. Vaak zijn dit waarden die op veel plaatsen in het programma gebruikt worden.

Lokale variabelen echter hebben maar een beperkt gebied waarin ze beschikbaar zijn en bestaan. Dit wordt vaak gedaan als een waarde alleen in bijvoorbeeld 1 functie gebruikt wordt.

Dus wat zijn nou die gebieden waar we het steeds over hebben?

Laten we eens naar een “leeg: Arduino programma kijken:

```
1  void setup() {
2    // put your setup code here, to run once:
3
4  }
5  void loop() {
6    // put your main code here, to run repeatedly:
7
8  }
```

We zien 2 gebieden: het blok “setup()” en “loop()” – beiden omsloten door accolades. Maar er is nog een 3de gebied: het gehele programma.

Laten we eens een globale variabele definiëren, om dit te illustreren:

```
1  int GlobaleVariable; // GlobaleVariable kan hier gebruikt worden
2
3  void setup() {
4    // put your setup code here, to run once:
5    // GlobaleVariable kan hier gebruikt worden
6  }
7
8  void loop() {
9    // put your main code here, to run repeatedly:
10   // GlobaleVariable kan hier gebruikt worden
11 }
```

En nu een voorbeeld van een lokale variabele:

```
1  // LokaleVariabele kan hier NIET gebruikt worden
2
3  void setup() {
4    // put your setup code here, to run once:
5    int LokaleVariabele;
6    // LokaleVariabele kan hier WEL gebruikt worden
7  }
8
9  void loop() {
10   // put your main code here, to run repeatedly:
11   // LokaleVariabele kan hier NIET gebruikt worden
12 }
```

Dit was misschien niet de meest uitgebreide uitleg, maar ik vermoed dat je het wel snapt.

We kunnen echter meerdere lokale variabelen opzetten die hetzelfde heten, in verschillende blokken natuurlijk.
Als ik “LokaleVariabele” definieer in de functie “setup()” EN in de functie “loop()”, dan zijn beiden een lokale variable, ook al zijn de namen hetzelfde. Echter, als we de waarde van “LokaleVariabele” veranderen in “setup()” dan heeft dit geen invloed op “LokaleVariabele” in de functie “loop()”!
Ze heten hetzelfde, maar ze hebben ieder hun eigen geheugen locatie. Je snapt al dat dit to verwarring kan leiden!

```
1  // LokaleVariabele kan hier NIET gebruikt worden
2
3  void setup() {
4    // put your setup code here, to run once:
5    int LokaleVariabele;
6    // LokaleVariabele kan hier WEL gebruikt worden, maar dit is niet dezelfde als in loop()
7  }
8
9  void loop() {
10   // put your main code here, to run repeatedly:
11   int LokaleVariabele;
12   // LokaleVariabele kan hier WEL gebruikt worden, maar dit is niet dezelfde als in setup()
13 }
```

Laten we eens gaan kijken naar een werkend programma waarbij we er een (schijnbaar) rommeltje van gaan maken.

We zien hier ook meteen een nieuwe constructie welke we bij “const” ook hebben gezien: het definiëren van een variabele en het meteen een waarde geven, allemaal in 1 instructie: int A = 0;. Deze regel (regel 1) maakt de GLOBALE variabele “A”.

In zowel “setup()” als “loop()” maken we weer een variabele “A” en geven we het een waarde. Deze variabelen bestaan dus alleen maar in hun eigen blok en zijn LOKALE variabelen.

We kunnen de functie “delay(x)” gebruiken om een programma te pauzeren voor “x” milli seconden (1000 = 1 seconde)

Draai dit programma maar eens op de Arduino.

```
1  int A = 0;
2
3  void setup() {
4    int A;
5    A = 1;
6    Serial.begin(9600);
7
8    // print the values to the serial monitor
9    Serial.print("Setup: A = ");
10   Serial.println(A);
11 }
12
13 void loop() {
14   int A;
15   A = 2;
16
17   // print the values to the serial monitor
18   Serial.print("Loop: A = ");
19   Serial.println(A);
20
21   delay(100000);
22 }
```

De output (uitvoer) zal zijn:

Setup: A = 1
Loop: A = 2

Om het effect van LOKAAL en GLOBAAL te zien, gaan we nu even in “setup()” twee regels uitschakelen met behulp can commentaar-tekens (//).
Start het programma weer op de Arduino:

```
1  int A = 0;
2
3  void setup() {
4   // int A;
5   // A = 1;
6   Serial.begin(9600);
7
8   // print the values to the serial monitor
9   Serial.print("Setup: A = ");
10  Serial.println(A);
11 }
12
13 void loop() {
14  int A;
15  A = 2;
16
17  // print the values to the serial monitor
18  Serial.print("Loop: A = ");
19  Serial.println(A);
20
21  delay(100000);
22 }
```

De output wordt nu:

Setup: A = 0
Loop: A = 2

De globale variabele is nu in gebruik in de “setup()” functie. Maar, in de “loop()” functie wordt deze globale variabele niet gebruikt omdat er een lokale variabele is met dezelfde naam – en deze lokale variabele wordt dus in “loop()” gebruikt.

Laten we eens alleen met een globale variabele werken en diens waarde in de verschillende delen veranderen:

```
1  int A = 0;
2
3  void setup() {
4    Serial.begin(9600);
5
6    Serial.print("Setup: Eerste A = ");
7    Serial.println(A);
8
9    A = 10;
10   // print the values to the serial monitor
11   Serial.print("Setup: Tweede A = ");
12   Serial.println(A);
13 }
14
15 void loop() {
16   Serial.begin(9600);
17
18   Serial.print("Loop: Eerst A = ");
19   Serial.println(A);
20
21   A = 20;
22
23   // print the values to the serial monitor
24   Serial.print("Loop: Tweede A = ");
25   Serial.println(A);
26
27   delay(100000);
28 }
```

De variabele “A” is nu een GLOBALE variabele – Dus het kan overal gezien en gebruikt worden.

De output zou er zo uit moeten zien:

Setup: Eerste A = 0
Setup: Tweede A = 10
Loop: Eerste A = 10
Loop: Tweede A = 20

Dus we definiëren de globale variabele “A” en geven deze de waarde nul (0) – het staat immers buiten alle blokken.
In “setup()”, geeft de “Serial.print” weer wat de waarde van “A” is en dat het gelezen en gebruikt kan worden.
Hier veranderen we de waarde van “A” naar 10, en geven deze ook weer.
In “loop()”, laten we ook weer zien dat we “A” kunnen lezen en dat het nu de nieuwe waarde heeft

die we in “setup()” hebben opgegeven.
En vervolgens veranderen we deze weer.

Nummer Typen en Weergave

Er zijn verschillende nummer system om nummer te representeren of weer te geven. Kijk maar eens naar deze Wiki Lijst van Nummer Systemen. Voor ons programmeurs zijn de volgende drie eigenlijk de meets gebruikte nummersystemen: Decimaal, Hexadecimaal en Binaire nummers (in die volgorde). Vaak zullen we echter gewoon decimale nummer gebruiken. Maar wat zijn dan die nummersystemen?

Decimale Nummers

Het decimale systeem is het systeem wat we dagelijks gebruiken. Het nummersysteem waarbij we 10 verschillende symbolen gebruiken: 0, 1, 2, 3, 4, 5, 6, 7, 8, en 9. Omdat dit 10 symbolen heeft, noemen we dit het “Decimale” systeem (Deci betekend 10).

Ik ben er wel zeker van dat je weet hoe het tellen in het decimale systeem werkt, we doen het immers elke dag. Maar om de andere twee systemen beter te begrijpen, gaan we toch even door op hoe het “tellen” werkt voor een decimaal systeem …

Als we beginnen met tellen dan hebben we 0, 1, 2, 3, 4, 5, 6, 7, 8, en 9. voor het volgende nummer (dus 9+1) starten we weer met nul en verhogen voorgaand nummer met 1. Welk nummer voor “9” hoor ik je al vragen?

Stel we zetten een aantal nullen voor een nummer, de waarde van het nummer veranderd hierdoor niks en de extra nullen hebben geen toegevoegde waarde.Laten we eens beginnen met 3 nullen voor het nummer.

We gaan weer tellen: 0001, 0002, … , 0008, 0009, 0010, 0011, 0012, … , 0018, 0019, 0020, 0021, … 0099, 0100, 0101, … etc.

Elke keer als een nummer voorbij de “9” wil gaan, dan starten we dat nummer met 0 (nul) en verhogen het voorgaande nummer met 1.

Snappie?

Dus als we het nummer 0999 hebben, en deze met 1 willen verhogen, dan zal deze 9 terug naar nul gezet worden (0990), vervolgens verhogen we het nummer ervoor met 1 (0990), maar daardoor wordt dit nummer ook weer groter dan 9 en wordt dus ook weer teruggezet op nul (0900), hierdoor moet het nummer daarvoor ook weer met 1 verhoogd worden (0900), maar daardoor moet dat nummer ook weer op nul worden gezet (0000) en het nummer ervoor weer met 1 worden verhoogd waardoor we eindigen met 1000.

Ben je er nog? Even onthouden, want we hebben dit later weer nodig.

We weten natuurlijk al dat we een decimaal nummer met Serial.print(nummer); kunnen weergeven. We kunnen het ook schrijven alsSerial.print(nummer, DEC);, waarbij DEC aangeeft dat we het nummer als een decimaal nummer willen zien. Deze methode (DEC) werkt overigens het beste met gehelen nummers, floating point nummers (nummers met cijfers achter de komma) kunnen onverwachtse resultaten geven.

Binaire Nummers

Het binaire nummersysteem kent eigenlijk maar twee nummers: 0 en 1. Daarom heet het een “binair” system, “bi” wil zeggen “twee”.

In tegenstelling tot wate je zou verwachten, vindt het binaire systeem zijn oorsprong niet in de computerwereld – het bestaat al eeuwen voor we wisten wat een computer zou kunnen zijn. Het is helaas in het Engels, maar deze Wiki pagina over de geschiedenis is zeer zeker leuk om eens te lezen!

Dat even terzijde, het binaire system is natuurlijk wel bij uitstek geschikt voor computers omdat het maar twee waarden kent: true of false, ja of nee, of: aan of uit.

Het tellen in het binaire systeem werkt overigens hetzelfde als bij het decimale systeem, alleen hebben we dan wel veel minder nummers om mee te werken.

Dus, inclusief de denkbeeldige nullen voor het nummer, laten we eens gaan tellen: 0000, 0001 en de nummers zijn op. Dus zetten we het nummer weer terug naar nul, en verhogen het voorgaande nummer weer met 1. Net zoals we dat zagen bij het decimale systeem, dus als we doorgaan met tellen: 0010, 0011, en weer zijn de nummer op, dus herhalen we weer nummer naar nul zetten en voorgaand nummer met 1 verhogen, enz. 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111.

Nu we weten hoe we tellen, zouden we een conversie tabel kunnen maken zodat we binaire nummers om kunnen zetten naar decimale nummers:

Binaire Nummers	
Binair nummer	Decimaal nummer
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12

Binaire Nummers	
Binair nummer	Decimaal nummer
1101	13
1110	14
1111	15

Er is echter een eenvoudigere methode dan steeds een hele tabel uit de kast halen (en dan heb ik het niet over calculator op jouw computer, want dat gaat natuurlijk nig sneller).

Als je een beetje bekend bent met rekenen “tot de macht”, dan kan het volgende handig zijn voor het berekenen van de decimale waarde van een binair getal. Als je namelijk een table moet maken voor b.v. 16 bits binaire nummers, b.v. 1101 1000 0000 0101, dan kon dat weleens een lange tabel gaan worden.Er is een manier om dat “sneller” te doen (OK, de calculator op jouw PC doet het VEEL sneller).

Elke positie in een binair nummer heeft een waarde, en als we positie van rechts naar links tellen dan is dat:
“2 tot de machtpositie” vermenigvuldigd met 0 of 1, net wat de bit waarde in die positie is.

We gebruiken “2” omdat het binaire nummersysteem maar 2 waarden kent: 0 en 1.

Dus als we naar een 4 bit nummer (dit noemt men een nibble!) gaan kijken, bijvoorbeeld: 1010.

Binaire conversie			
Positie 3	Positie 2	Positie 1	Positie 0
1	0	1	0
23 = 2×2×2 = 8	22 = 2×2 = 4	21 = 2	20 = 1
1×8	0x4	1×2	0x1
= 8	= 0	= 2	= 0

Het resultaat is de som van deze nummers, of te wel: 1010 binair = 8+0+2+0 = 10 decimaal.

De meeste programmeurs vergeten de details van deze truc gemakkelijk, zoals gezegd: de rekenmachine van onze computer kan dit sneller, en de meeste programmeertalen hebben functies ingebouwd die dit ook snel kunnen doen voor ons.

Iets om te onthouden: een byte is 8 bits en als alle bits de waarde 1 hebben, dan krijgen we 1111 1111, wat meteen de maximale waarde van een byte is: 255.

Je kunt binaire nummers overigens gewoon in jouw programma code gebruiken, door er “0b” (nul-b) voor te typen. Bijvoorbeeld: 0b1010

Het weergeven van binaire nummers werkt ook gewoon, als voorbeeld voor het nummer 1010: Serial.println(0b1010);

Dit print echter de decimale waarde van het binaire nummer, wat dus hetzlefde werkt als Serial.println(0b1010, DEC);

Om een nummer nu als binair te printen, gebruiken we: Serial.println(0b1010, BIN);
Dit werkt voor alle nummersystemen, dus als we een decimaal nummer meegeven dan levert Serial.println(10, BIN); de output “1010”.

Hexadecimale Nummers

OK, we hebben dus Binair en Decimaal gezien, maar wat zijn dan Hexadecimale Nummers?

Overigens: de meeste mensen gebruiken de term “hex” om aan te geven dat ze het over hexadecimale nummers hebben.

In het Hexadecimale systeem hebben we 16 nummers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, en F. (Hex = 6 en Deci = 10)

Het tellen werkt hetzelfde als we hebben gezien bij decimale nummers en binaire nummers, al hebben we nu dus meer “nummers” beschikbaar om mee te werken.

Als voorbeeld, weer met de extra nullen ervoor: 0001, 0002, … , 0009, 000A, … 000F, 0010, 0011,… 001F, 0020, 0021,… etc.

Conversie naar een decimaal nummer lijkt wat lastiger, maar werkt vergelijkbaar als bij conversie van binaire nummers.

Stel we hebben het hexadecimale nummer 2FA. Weer van rechts naar links tellen voor de posities, dan moeten we de waarde van het nummer vermenigvuldigen met 16 tot de machtpositie nummer. We gebruiken hier “16” omdat het hexadecimale systeem 16 nummers kent. De som van de uitkomste is weer de decimale waarde.

In een voorbeeld:

Hexadecimale conversie		
Positie 2	Positie 1	Positie 3
2 × 162	F × 161	A x 160
= 2 × 256	= 15 (F) × 16	= 10 × 1
= 512	= 240	= 10

De, 2FA Hexadecimaal is hetzelfde als 512+240+10 = 762 decimaal.

Ook hexadecimale nummers kunnen we in onze code gebruiken door er “0x” (nul-x) voor te typen, dus bijvoorbeeld: 0x2FA

We kunnen ze dus ook printen, zoals we eerder hebben gezien, met bijvoorbeeld Serial.println(0x2FA); maar dit print dus een nummer in decimaal formaat. Wil je nu liever de hexadecimale waarde als output zien, dna gebruiken we Serial.println(0x2FA, HEX);. En dit werkt dus ook weer met decimale nummer en binaire nummers.

Nummers Weergeven

Hieronder een stukje demo code how we met decimale, binaire en hexadecimale nummers kunnen werken.

```
1  void setup() {
2    Serial.begin(9600);
3
4    int decNummer = 10;
5    int hexNummer = 0x2FA;
6    int binNummer = 0b10001010;
7
8    Serial.println("Alle Nummers als decimaal:");
9    Serial.println(decNummer);
10   Serial.println(decNummer, DEC);
11   Serial.println(hexNummer);
12   Serial.println(hexNummer, DEC);
13   Serial.println(binNummer);
14   Serial.println(binNummer, DEC);
15
16   Serial.println("Alle Nummers als Hexadecimaal:");
17   Serial.println(decNummer, HEX);
18   Serial.println(hexNummer, HEX);
19   Serial.println(binNummer, HEX);
20
21   Serial.println("Alle Nummers als Binair:");
22   Serial.println(decNummer, BIN);
23   Serial.println(hexNummer, BIN);
24   Serial.println(binNummer, BIN);
25 }
26
27 void loop() {
28   // leave empty for now
29 }
```

De output van dit voorbeeld (merk op dat “, DEC” toevoegen dus niet nodig is):

Alle Nummers als decimaal:
10
10
762
762
138
138

Alle Nummers als Hexadecimaal:
A
2FA
8A
Alle Nummers als Binair:
1010
1011111010
10001010

3.6. Operators (bewerkers)

Nu dat we wat gezien hebben van Data Types, Variabelen, Scope (bereik) en hoe we ze kunnen weergeven (print), tijd om eens te gaan kijken, naar wat men noemt “Operators“. “Operator” is het Engelse woord voor “Bewerker” – beetje beroerde vertaling, maar het geeft toch wel weer wat ze doen.

Een Operator (bewerker) is een symbool dat iets doet met data.

Dus een Operator “doet iets” met data en is meestal een symbool. Denk daarbij bijvoorbeeld aan het “+” symbool, welke we gebruiken om “optellen” aan te geven in berekeningen. De “+” operator bewerkt dus de data door het op te tellen.

Voor Arduino Programmeren, zouden we 6 basis “groepen” van operators kunnen maken (zie ook de Arduino Reference pagina’s, welke helaas in het Engels zijn).
In dit eerste stukje kijken we naar operators voor berekeningen, maar daarna gaan we ook kijken naar andere operators, bijvoorbeeld gebruikt voor het vergelijken van data (informatie).

Rekenkundige Operators

Zoals we al eerder zagen; Operators (bewerkers) kunnen gebruikt worden voor berekeningen, en de meeste van deze symbolen ken je eigenlijk al wel. Sommige van deze symbolen zijn echter net even anders op de computer.

Als voorbeeld, op school heb je geleerd dat een deling er zo uit ziet: 12 : 4 = 3 (in sommige landen gebruikt men het “÷” teken)
Op de computer, en dus ook in de programmeertaal C, schrijft men dit echter als volgt: 12 / 4 = 3

We zien dit ook bij vermenigvuldigen. Op school schrijf je: 3 × 4 = 12 (in wiskunde kan dit ook een “.” zijn)
Op de computer is dit echter: 3 * 4 = 12

Rekenkundige Operators

Symbool	Doel
=	Toewijzen
+	Optellen
–	Aftrekken
*	Vermenigvuldigen

Rekenkundige Operators	
Symbool	Doel
/	Delen
%	Modulo

De onbekende jongen in het rijtje kan de “modulo” operator zijn.
Modulo is wat we op school “deelrest” noemen bij een staartdeling. Als voorbeeld: Stel we delen 5 door 2 met een staartdeling, dan is het antwoord 2 (immers: 2×2=4) en de deelrest is dan 1 omdat 5 – 4 = 1, en 1 kunnen we niet door 2 delen.
Dus,… de “modulo” van 5 gedeeld door 2 is: 5 % 2 = 1.

Nog even een paar voorbeelden:
Stel we delen 10 door 5, dan is het antwoord 2 en de deelrest 0, dus 10 % 5 = 0.
Zo zien we ook dat 33 % 7 = 5, dus 33 delen door 7, resulteert in 5, omdat de deling 4 levert (4×7=28) en de deelrest 5 (33-28=5) is.

Die andere rare jongen is het “is gelijk” teken (=), of te wel een “toewijzing” waarbij bijvoorbeeld een waarde wordt gekopieerd naar b.v. een variabele.

Dit zijn de basis rekenkundige operators. Maar vergeet niet dat een programma in principe een verzameling instructies is waarin we met informatie (data) werken. Dus niet alleen maar rekenen maar ook vergelijken. Niet belangrijk om te onthouden; maar in principe is vergelijk gebaseerd op rekenen.

Vergelijking Operators

Als we bezig zijn met Arduino Programmeren, dan zul je snel merken dat het belangrijk is dat we beslissingen kunnen maken in een programma. Bijvoorbeeld: het licht aanzetten als het donker is. Hiervoor moeten we zaken gaan vergelijken en dat is waar deze groep operators van toepassing gaan zijn.

Je hoeft de volgende operators niet meteen te onthouden, naarmate je meer werkt met beslissingen in een programma, ga je ze vanzelf onthouden.

Bedenk ook dat een vergelijking altijd een antwoord geeft die waar (true) of onwaar (false) is!

Vergelijking levert altijd een waar (true”) of onwaar (false) antwoord – dit is dus een boolean antwoord!

Dus als voorbeeld: 5 > 2 levert waar (true) en 5 < 2 levert onwaar (“false”).
Ehm maar wat betekenen die tekentjes nou? Even in de volgende tabel kijken:

Comparison Operatos	
Symbool	Doel
==	is gelijk aan
!=	is niet gelijk aan

Comparison Operatos	
Symbool	Doel
<	is kleiner dan
>	is groter dan
<=	is kleiner dan of gelijk aan
>=	is groter dan of gelijk aan

Het onthouden van de betekenis van deze symbolen is niet altijd even gemakkelijk en daarom hebben we hiervoor een ezelsbruggetje.

Als we voor het “<” teken een plaatje zetten, net als hieronder afgebeeld, dan wordt het een “K“. De “K” van “Kleiner dan”.
Makkelijk he? Het andere teken wordt daarmee automatisch het tegenovergestelde natuurlijk; groter dan.

Ezelsbruggetje – Kleiner dan

Hieronder dan een klein programma om dit in werking te zien. Vergeet niet: een vergelijking resulteert in een boolean antwoord, en een boolean wordt als nummer opgeslagen dus “true” (1) en “false” (0). Als we een boolean printen (weergeven) dan zien we dus het nummer en niet “true” of “false”.

```
1  boolean A;
2
3  void setup() {
4    Serial.begin(9600);
5
6    A = 5 < 2;
7    Serial.print("A = ");
8    Serial.println(A);
9  }
10
11 void loop() {
12 }
```

Dit voorbeeld doet het volgende:

We definiëren de variabele “A” als een “boolean” (dus een variabele die de waarde “true” of “false” heeft).
We gebruiken de vergelijking operator “<” – in andere woorden, we kijken vergelijken 5 met 2 en kijken dus of 5 kleiner is dan 2. Het voor de hand liggende antwoord is dus natuurlijk ONWAAR (false). Omdat booleans echter als nummer worden opgeslagen, resulteert dit dus in de weergave van het getal “0” (nul).

Meer details over vergelijken in het volgende hoofdstuk …

Boolean Operators

Ook al kan het gebruik van vergelijking operators wat verwarrend zijn, boolean operators zijn misschien nog verwarrender – maar; in het volgende hoofdstuk zal duidelijk worden wat we hier allemaal mee kunnen doen.

Boolean operators zijn operators die we in de gewone taal uitspreken als “en”, “of” en “niet”.

Boolean Operator	
Symbool	Doel
&&	AND (en)
||	OR (of)
!	NOT (niet)

Ik zal proberen deze voorbeelden uit te leggen, maar het geeft niets als je het allemaal nog niet 100% kunt bevatten.

AND (&&) of te wel: “EN”

De AND operator (engels voor “en”) kijkt of twee waarden waar zijn.

Stel we hebben 2 boolean waarden, “ZietErUitAlsEenEend” en “KlinktAlsEenEend“.

We gaan deze twee waarden gebruiken om te bepalen of we te maken hebben met een “Eend“, en daarvoor moeten beiden WAAR (true) zijn.
Dus we hebben te maken met een Eend als ZietErUitAlsEenEend AND (EN) KlinktAlsEenEend beiden waar (true) zijn.

Eend = ZietErUitAlsEenEend && KlinktAlsEenEend.

Als een van deze waarden niet waar (false) is, of als zelfs beiden niet waar zijn, dan hebben we dus niet (false) met een Eend te maken.

Hieronder een simpele tabel met voorbeelden. AND levert een WAAR als beide waarden WAAR zijn.

And (en) Resultaten

Waarde1	Waarde2	Resultaat van Waarde1 AND (en) Waarder2
true	true	= true
true	false	= false
false	true	= false
false	false	= false

OR (||) of te wel: “OF”

De OR operator (engels voor “of”) kijkt of minstens 1 van de 2 waarden waar is.

Laten we als voorbeeld de volgende boolean variabelen bekijken: “DonkerBuiten” en “Na21Uur” (na 9 uur ’s avonds).
Om te kijken of de buiten verlichting aan moet (variabele: “LichtenAan“) moet minstens een van deze twee variabelen waar (true) zijn.

Dus LichtenAan als het DonkerBuiten OR (OF) Na21Uur.

LichtenAan = DonkerBuiten || Na21Uur.

Weer een tabelletje met de mogelijke combinaties. OR levert dus een WAAR (true) als minimaal 1 van de 2 waarden WAAR is.

Or (of) Resultaten

Waarde1	Waarde2	Resultaat van Waarde2 OR (of) Waarde2
true	true	= true
true	false	= true
false	true	= true

Or (of) Resultaten		
Waarde1	Waarde2	Resultaat van Waarde2 OR (of) Waarde2
false	false	= false

NOT (!) of te wel: “NIET”

De NOT (Engels voor “niet”) draait een boolean om,
Dus een “true”→”false” or “false”→”true” …

De boolean “not” operator draait een boolean om naar diens tegengestelde waarde. Zie het als een licht schakelaar waarbij je het licht aan en uit kunt zetten. Het schakelen van aan naar uit is wat “not” doet. Het geeft het tegenovergestelde antwoord.

Als voorbeeld, als “LichtenAan” waar is (true), dan is NOT (niet) “LichtenAan” dus onwaar (false). Maar als “LichtenAan” niet waar (false) is, dan is NOT “LichtenAan” dus waar (true).

Dus: NOT true = false, en NOT false = true.

!true = false
!false = true

We hebben hier geen tabel voor nodig, maar toch even voor het compleet zijn:

NOT (niet) resultaten	
Waarde1	Resultaat van NOT Waarde1
true	false
false	true

Wat kleine experimenten

We hebben nu een aantal operators (bewerkers) doorlopen, laten we eens wat gaan spelen met wat we geleerd hebben.

```
1  void setup() {
2    Serial.begin(9600);
3
4    Serial.print("Het antwoord is: ");
5    Serial.println(true && false);
6  }
7
8  void loop() {
9  }
```

In dit voorbeeld, gebruiken we de “AND” (&&) boolean operator.

Even niet vergeten dat booleans dus als nummer opgeslagen worden he? Dus true=1 en false=0 wanneer deze weergegeven worden.

In regel 5 kunnen we ook een aantal andere operator voorbeelden zetten:

```
Serial.println(true && true);   // AND; true and true  = true  = 1
Serial.println(true || true);   // OR ; true or true   = true  = 1
Serial.println(false || false); // OR ; false or false = false = 0
Serial.println(false || true);  // OR ; false or true  = true  = 1
Serial.println(!true);          // NOT; not true       = false = 0
```

Samengestelde (Compound) Operators

Compound Operators, ook wel “Samengestelde” operators, worden vaak gebruikt om minder te hoeven typen als we een programma schrijven. Dit wordt vaak gebruikt, maar persoonlijk ben ik er niet zo’n fan van omdat het de duidelijkheid of leesbaarheid van jouw programma nadelig beïnvloed. Omdat ze toch zo vaak gebruikt worden, zul je ze wel moeten leren kennen – helaas.

Meestal combineert een Compound Operator twee of meer operators of stappen in 1 stap. Dit maakt leesbaarheid dus wat minder. Merk ook opdat ik hieronder niet alle compound operators vermeldt.

Een Compound Operator (samengestelde operator) combineert een aantal operators of stappen naar een enkele stap …

Compound (Samengestelde) Operators

Symbol	Purpose	Example

++	waarde verhogen	A++; // verhoog A met 1 en geef de oude waarde van A terug
		++A; // verhoog A met 1 en geef de nieuwe waarde van A terug

—	waarde verlagen	A–; // verlaag A met 1 en geef de oude waarde van A terug
		–A; // verlaag A met 1 en geef de nieuwe waarde van A terug

+=	samengetrokken optelling	A += B; // hetzelfde als A = A + B;
-=	samengetrokken aftrekken	A -= B; // hetzelfde als A = A – B;
*=	samengetrokken vermenigvuldiging	A *= B; // hetzelfde als A = A * B;

Compound (Samengestelde) Operators

Symbol	Purpose	Example
/=	samengetrokken deling	A /= B; // hetzelfde als A = A / B;
%=	samengetrokken modulo	A %= B; // hetzelfde als A = A % B;

Zoals je hier uit de tabel al kunt opmaken: soms is zo'n compound operator moeilijk te volgen of eenvoudig te verwarren. Omdat we er ook niets mee winnen, voor wat betreft ons eind programma betreft, raad ik beginners zeker aan om deze niet of zo min mogelijk te gebruiken.

Een eenvoudig voorbeeld:

```
1  int A = 0;
2
3  void setup() {
4    Serial.begin(9600);
5
6    Serial.print("Setup: A = ");
7    Serial.println(A);
8  }
9
10 void loop() {
11   Serial.begin(9600);
12
13   A += 1; // the same as: A = A + 1;
14
15   // print the values to the serial monitor
16   Serial.print("Tellen in de loop():  A = ");
17   Serial.println(A);
18
19   delay(1000);
20 }
```

Your output will look like this:

```
Setup: A = 0
Tellen in de loop(): A = 1
Tellen in de loop(): A = 2
Tellen in de loop(): A = 3
Tellen in de loop(): A = 4
Tellen in de loop(): A = 5
Tellen in de loop(): A = 6
Tellen in de loop(): A = 7
...
```

Als we nu regel 13 zouden veranderen naar het beter leesbare A = A + 1; , en dit op de Arduino starten, dan zullen we 2 dingen zien.
Als eerste zul je zien dat de compiler precies hetzelfde resultaat levert wat omvang betreft voor beiden voorbeelden (dusA+=1; enA=A+1;) …
Je zult nu ook zien dat A = A + 1; beter leesbaar is.

Dus nogmaals, zeker voor beginners: beperk het gebruik van deze samengestelde (compound) operators.

Pointer Access Operators en Bitwise Operators

Deze twee groepen slaan we hier gewoon over omdat deze toch aanzienlijk meer ervaring eisen. Als je wat meer programmeer ervaring opgedaan hebt, dan zul je vanzelf merken wanneer je ze nodig hebt en het kan zelfs zijn dat je ze nooit nodig zult hebben. 

OK, we hebben weer een hoop informatie voor onze kiezen gekregen en ik adviseer iedereen dan ook om er eens lekker mee te gaan experimenteren wat behulpzaam kan worden in het volgende hoofdstuk van Arduino Programmeren voor Beginners.

Volgende hoofdstuk: Arduino Programmeren voor Beginners – Deel 4: Beslissingen