# Deel 1, de basics

## Intro

Deze cursus is bedoeld voor mensen die met een Arduino werken of hebben gewerkt en soms een beetje in de knoop komen met de grootheid of complexiteit van hun programma's of waarbij ze vervelende bugs hebben meegemaakt.

Het werkend krijgen van een Arduino en het iets te laten doen wat jij wilt en het goed programmeren van de Arduino zijn twee heel verschillende dingen. Collega RudyB heeft een goede zaak verricht betreft het eerste en ik wil graag een woordje doen over het tweede.

Het doel van dit eerste deel van de cursus is de lezer bepaalde kennis mee te geven van de programmeertaal waarmee hij beter zijn programma kan ordenen, zijn programma kan onderhouden en daarmee bugs kan voorkomen.

In dit eerste deel leer je de ins en outs van de volgende dingen:

- variabelen types en hun scopes
- geavanceerde datatypes.
- statements en logic operators.
- functies
- stijl

In het tweede hoofdstuk leer ik je de basics van een "state-machine", een code structuur waarmee je complexere handelingen kan verrichten.

Ook om je programma op te delen met functies en je code te verdelen over meerdere bestandjes opdat je bestanden kleiner en overzichtelijker kunnen worden komen aan bod in het derde deel.

En in deel 3 leer ik je ook hoe je je computer code voor je kan laten maken en hele project folders opzet. Daarvoor verschaf ik de tooling die je daarvoor nodig heb.

Ik probeer deze 'cursus' in een Jip en Janneke taal uit te leggen waar ik het ook uitgelegd in had willen hebben op school. Ik leg de focus vooral meer op C dan op C++ desondanks dat Arduino in C++ is geschreven. C++ is namelijk een verschrikkelijk ingewikkelde taal met een erg complexe syntax. Op de 'classes' na, wordt er geen over-ingewikkelde C++ elementen behandeld. En met alleen C kan je prima een Arduino programmeren.

# Inhoudsopgave

## Hoofstuk 1, bare mininum

Een basis programma zoals in Arduino bestaat uit minimaal deze twee componenten. De void setup() en de void loop(). In het example lijstje van Arduino staat de 'BareMininum' sketch. Deze sketch is precies dat.

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Zonder een van deze functies kan de arduino IDE niet compileren. De void setup wordt slechts eenmalig uitgevoerd. We gebruiken de setup daarom om bepaalde zaken te initialiseren. De pinModes zetten of de seriele communicatie (Serial.begin()) zijn hier goede voorbeelden van.De void loop() wordt eindeloos herhaalt.

Deze gebruikte syntax is Arduino eigen en is niet conform de 'realiteit'. Onderhuids bestaat dit stukje code in Arduino.

```
int main(void)
{
    init();

    initVariant();

    setup();

    for (;;) {
        loop();
        if (serialEventRun) serialEventRun();
    }

    return 0;
}
```

Init() initialiseert bepaalde zaken van de Arduino. Deze functie zorgt er ook voor dat millis() en micros() werken. Wat initVariant() doet, weet ik niet precies maar het zal iets soortgelijks zijn. De serial event run is een dingetje om de Serial Event te laten plaatsvinden als je er gebruik van maakt. Wat wij zelf programmeren, zijn de bodies van setup() en loop(). int main(void) is standaard C en C++ syntax. Elke C en C++ programma heeft een main functie.

# Hoofstuk 2, Variabelen

Variabelen zijn niks meer dan namen die wij toekennen aan een stuk geheugen wat onze programma's kunnen manipuleren. Er zijn verschillende type variabelen tot ons beschikbaar waarbij de grootste bestaat uit 8 bytes. Een variabele is meestal signed of unsigned. Een signed variabele kan negatief zijn waar een unsigned variabele dat niet kan.

Voordat ik verder ga, moet ik eerst twee termen uitleggen, declareren en initialiseren. Met declareren bedoelen we het aanmaken van een variabele en met initialiseren stoppen we er een waarde in. Declareren en initialiseren kan tegelijk, maar het initialiseren kan ook later en het is ook niet noodzakelijk. Het is wel belangrijk dat je er wel eerst een waarde in een variabele stop alvorens je hem probeert uit te lezen.

De syntax voor het aanmaken en initialiseren hoeft waarschijnlijk niet uitgelegd te worden, desalnietemin geef ik voorbeelden van de verschillende mogelijkheden.

```
uint16_t xPos = 1, yPos = 2, zPos = 3;
int32_t val1 = 100, val2;
float pi = 3.141568;
int16_t result, sum, division;
```

## §2.1 bool

De eerst variabele type is de bool. Een bool kan true of false zijn. De bool consumeert wel een hele byte aan geheugen terwijl er maar een bit van benutten. Ik raad ook niet om deze te gebruiken omdat er betere alternatieven voor zijn. De syntax:

```
bool var;
```

## §2.2 char

De tweede variabele type is de char en deze is 1 byte of 8 bit groot en kan maximaal 256 verschillende waardes bevatten. De char staat voor character en is bedoeld om ascii characters in op te slaan. Je kan echter ook een getal in een char opslaan. De char bestaat ook in signed en unsigned variant. Een signed char is bedoeld voor waardes van -128 tot +127 en een unsigned char is bedoeld voor waardes tussen de 0 en 255.

In Arduino kan je ook gebruik maken van de 'byte' een 'byte' is door Arduino gedefinieerd als een unsigned char. Tevens heb je ook nog de int8_t en de uint8_t maar dit is essentieel hetzelfde als een signed char en een unsigned char respectievelijk.

```
char x; // character x wordt gedeclareerd
signed char b = 5; // b wordt geclareerd en geinitialiseerd
unsigned char
byte    // hetzelfde als unsigned char
int8_t  // hetzelfde als signed char
uint8_t // hetzelfde als unsigned char
```

## §2.3 int

Een 2 byte of 16 bit variabele is de integer of int. In Arduino is een int 2 bytes maar op andere platformen kan dit verschillen. Een integer is standaard signed en kan dus negatief worden. Een unsigned int kan niet negatief worden en kan maximaal 65535 worden. Ook deze types kan je schrijven als int16_t en uint16_t.

```
int
signed int // hetzelfde als int
unsigned int
int16_t // het zelfde als een signed int
uint16_t// hetzelfde als een unsigned int
```

Omdat integers per platform kunnen verschillen in grootte doe je er altijd goed aan om de intX_t en uintX_t notaties te gebruiken. Dan is ten alle tijden duidelijk over wat voor variabele we het hebben.

## §2.4 long en long long

De laatste type voor de integers is de ‘long’ en deze is 4 byte of 32 bit groot. De long is net als de int standaard signed. De long kan ook net als de int en de char zowel signed als unsiged zijn. En ook voor de long mag je int32_t en uint32_t gebruiken.

```
long
signed long // hetzelfde als long
long int    // hetzelfde als long
unsigned long
int32_t // het zelfde als een signed long
uint32_t// hetzelfde als een unsigned long
long long
signed long long
unsigned long long
int64_t // het zelfde als een signed long long of long long
uint64_t// hetzelfde als een unsigned long long
```

We mogen ook twee maal long achter elkaar tikken dat geeft ons een variabele van 64 bits. Deze kan zowel signed als unsigned zijn. Er is een kleine kans dat je deze variabele ooit nodig zal hebben. Het bereik van een normale unsigned long is al 4.29 miljard. Een long long kan tot

## §2.5 float & double

Je hebt buiten de integers om ook nog de float en de double. Dit zijn variabele type bedoeld voor decimale getallen. De float is 32 bit groot en de double is 64 bit groot. Dan kunnen we ook nog de long double gebruiken en die is wel 128 bit groot. Ik maak zelf nooit gebruik van deze variabele typen. En ik raad ook niet aan om ze te gebruiken. Het rekenen met deze getallen gebruikt meer performance dan het rekenen met integers. Het is ook niet mogelijk om elke bit-wise operator te gebruiken op deze variabele types.

```
float
double
long double
```

Ze kunnen ook problemen geven met vergelijkingen.

```
float x = 1.1;
double y = 1.1;

if(x == y) {
```

Dit soort vergelijkingen kan tot problemen leiden. x is niet precies hetzelfde als y. Het zijn verschillende datatypen waarbij de 1.1 anders wordt opgeslagen. De bits zijn daarom niet 1 op 1 het zelfde en daarom is x niet gelijk aan y ondanks het feit dat ze allebei de waarde 1.1 hebben.

Het is soms net wat je aan het programmeren heb. Als je bijvoorbeeld een temperatuur sensor heb en je wilt de temperatuur in 100$^{ste}$ decimalen opslaan, dan kan je er voor kiezen om een float of een double te gebruiken. Ook voor rekenkundige functies zoals sin(), cos() en tan() zijn deze variabele type nodig.

# Hoofdstuk 3, Constantes

Behalve variabelen zijn er ook contantes om mee te werken. Ze bestaan in drie varianten.

## §3.1 De macro

Als eerste is er de macro.

```
#define ledPin 13
```

De #define macro is niks meer dan een simpele tekst vervanger. De macro wordt niet uitgevoerd door de compiler maar de preprocessor. Dat houdt in dat als eerst alle vermeldigen van 'ledPin' worden vervangen door 13. Dus:

```
digitalWrite(ledPin, LOW);
// wordt omgezet in:
digitalWrite(13, LOW);
```

Macros worden nog besproken verder in hoofstuk 3 document. Je kan er namelijk veel meer mee dan constantes mee maken. Er is ook een sterk verdeelde mening over macro's die met een hoop broodjes aap komen.

## §3.2 De const int

In C++ en dus in Arduino mogen we ook de const int gebruiken.

```
const int constante = 7;
```

De oudere C compilers mochten ook de const int gebruiken, alleen die compilers erkende ze ook echt als variabelen. Je mocht ze weliswaar niet wijzigen, maar je mocht ze ook niet gebruiken als case labels in een switch case.

## §3.3 De enum

Enum staat voor enumerator hetgeen 'teller' betekent. Een enum gebruik je vaak bij een switch-case en state machines. Doordat je in het programma zelf alleen maar de namen gebruikt, maken de daadwerkelijk waardes niet meer uit. Een enum begint met tellen vanaf 0, maar je kan ook zelf waardes toekennen.

```
enum states {
    startPosition = 4,
    middlePosition, // 5
    endPosition,    // 6
};
```

Aanschouw dit voorbeeld. De switch case wordt later uitgelegd.

```
state = startPosition;
switch (state) {
case startPosition:
    state = middlePosition;
    break;

case middlePosition:
    state = endPosition;
    break;

case endPosition:
    state = startPosition;
    break;
}
```

state is hier een variabele. In state worden telkens andere waardes gestopt. Omdat ik hier discreet ben met het gebruik van de enum namen, maakt het eigenlijk niet uit welke waardes er in staat. Het maakt hier dus niet uit of 'startPosition' nu 4 of 26 is.

Wat wel opgemerkt moet worden is dat een enum een klein beetje geheugen van de Arduino consumeert. Daar hebben de const int en de #define geen last van. Overigens kan je met een #defines en const int exact hetzelfde doen als met een enum. Ik had ook kunnen doen:

```
enum states {
    startPosition = 4
};
const int middlePosition = 5;
#define endPosition 6
```

# Hoofdstuk 4, Functies

Functies zijn de meest belangrijke bouwblokken van meeste programmeertalen. Ze helpen om geheugen te besparen en om programma's meer overzichtelijk te houden.

## §4.1 wat is een functie?

Functies heb je ongetwijfeld al veel gezien zonder dat je precies wist wat het nu was. Als je RudyB's cursus gevolgd heb dan heb je onder andere de volgende functies ontmoet:

- digitalWrite()
- digitalRead()
- millis()
- analogWrite()

Je kan functies herkennen aan de ronde haken (). Een functie is een blokje code die ergens buiten de void loop() staan. In void loop() kan je deze functies aanroepen of 'callen'. De basis syntax ziet er zo uit.

```
void loop()
{
    flashLed();
}

void flashLed() {
    digitalWrite(ledPin, HIGH);
    delay(1000); // delays willen we nooit meer zien
    digitalWrite(ledPin, HIGH);
    delay(1000);
}
```

## §4.2 functie prototype

Er is ‘technish’ gezien een ding  incorrect. Hoewel dit compileert, is de syntax eigenlijk niet goed. In principe is de volgende regel waar:

*Als je een functie callt, moet de functie zelf boven de call liggen of anders moet er een functie prototype gemaakt worden.*

Natuurlijk een cryptische zin als je hem voor het eerst leest. In principe als je dus een functie aan roept. Dan moet de code van de functie zelf boven de plaats liggen van waar je hem aan roept. In dit geval moet je doen:

```
void flashLed() {
    digitalWrite(ledPin, HIGH);
    delay(1000); // delays willen we nooit meer zien
    digitalWrite(ledPin, HIGH);
    delay(1000);
}

void loop()
{
    flashLed();
}
```

Waarom compileert het nu wel goed? Dit is iets wat de Arduino IDE voor ons doet. Als je op de ‘compileren’ knop klikt, gebeuren er een heleboel dingen. Een van die dingen is dat de IDE de prototypes aanmaakt van alle functies in het .INO bestand. Dus nu denk je: En wat is een prototype precies?

Een functie prototype is een soort declaratie van een functie. Wij vertellen aan de compiler met een prototype, dat elders in het programma deze functie bestaat. De compiler werkt van boven naar beneden. Als de compiler ziet dat er een functie wordt ge`call`ed die hij nog niet heeft gezien, geeft de compiler een error. Door de prototype te tikken, snapt de compiler dat er op het moment dat een functie wordt ge`call`ed deze functie ook bestaat. Met de prototype zeggen we eigenlijk tegen de compiler: “deze functie bestaat, ik beloof het”. De syntax met wat commentaar:

```
void flasLed(); // Ik ben een functie prototype

void loop()
{
    flashLed(); // Ik roep deze functie aan
}

void flashLed() { // Ik ben de functie
    // deze 4 regels zijn de functie body.
    digitalWrite(ledPin, HIGH);
    delay(1000); // delays willen we nooit meer zien
    digitalWrite(ledPin, HIGH);
    delay(1000);
}
```

## §4.3 argumenten

Functies hebben nog twee belangrijke eigenschappen. Je kan ze argumenten meegeven waarmee ze kunnen werken of ze kunnen argumenten 'returnen' of ze kunnen het allebei. 'Void' betekent hier dat deze functie niets returnt.

Nu gaan we de functie flashLed() uitbreiden met twee argumenten; de pin nummer en de delay tijd.

```
void flasLed(byte, unsigned int);

void loop()
{
    flashLed(ledPin, 500);
    flashLed(andereLedPin, 100);

}

void flashLed(byte pin, unsigned int delayTime) {

    // deze 4 regels zijn de functie body.
    digitalWrite(pin, HIGH);
    delay(delayTime); // delays willen we nooit meer zien
    digitalWrite(pin, HIGH);
    delay(delayTime);
}
```

We laten nu eerst ledPin een flits geven van een halve seconde, en daarna laten we een andereLedPin een flits van 100ms geven. Tussen de functie haken () staan de argumenten en hun types. Pin is van het type byte en is 8 bit. Deze kan maximaal de waarde 255 bevatten. Omdat geen arduino zoveel pinnen heeft, volstaat een byte. De delay tijd daarentegen is een unsigned int en is daarmee 16bit groot en kan dus maximaal 65535 worden.

Nu kunnen we met deze functie elke output pin gedurende een variabele tijd aanzetten. Stel nu dat je denkt: "deze functie hoef ik niet meer aan te passen" dan zou je de functie in een ander bestand plaatsen. Hoe dit werkt, wordt nog uitgelegd.

Let er ook op dat je de prototype aanpast met de datatypes. In dit geval moet je 'byte' en 'unsigned int' tussen de functie haken van de prototype zetten. Deze datatypes moeten ten alle tijden matchen met die van de functie zelf.

## §4.4 return

Nu laten we flashLed() de delay tijd ophogen met 5 en de nieuwe waarde returnen we. Om de ge'return'de waarde op te vangen en te kunnen gebruiken, hebben we een variabele nodig van het type unsigned int. De code komt er dan zo uit te zien (met setup weggelaten):

```
unsigned int flasLed(byte, int); // proto
unsigned int flasLed(byte, int); // proto
unsigned int time = 100; // variabele

void loop()
{
    time = flashLed(andereLedPin, time);
}
unsigned int flashLed(byte pin, unsigned int delayTime) {
    // deze 4 regels zijn de functie body.
    digitalWrite(pin, HIGH);
    delay(delayTime); // delays willen we nooit meer zien
    digitalWrite(pin, HIGH);
    delay(delayTime);

    return delayTime + 5;
}
```

De werking is als volgt: De tijd die we gebruiken voor de flits duratie is opgeslagen in de variabele 'time'. We beginnen met een waarde van 100.

flashLed() neemt deze 'time' als argument. De waarde van 'time' wordt nu in 'delayTime' gestopt. De functie knippert vervolgens de output 1x en als laatste wordt de meegegeven tijd + 5 ge`return`ed.

De opgehoogde waarde wordt dan in 'time' gestopt in de void loop. En time is nu 5 hoger dan dat hij was. De led zal nu alsmaar langzamer en langzamer gaan knipperen. Totdat na 65 seconde deze een keer overflowt en de led weer snel gaat knipperen.

Zo kunnen functies dus waardes retourneren op de plaats waar ze worden aangeroepen. De functies millis() en micros() retourneren waardes. En digitalRead(pin) retourneert een waarde en neemt een waarde als argument.

Dit is vaak handig voor rekenkundige functies:

```
long MUL(long arg1, long arg2) {
    return arg1 * arg2;
}
long DIV(long arg1, long arg2) {
    return arg1 / arg2;
}
long MOD(long arg1, long arg2) {
    return arg1 % arg2;
}
long ADD(long arg1, long arg2) {
    return arg1 + arg2;
}
long SUB(long arg1, long arg2) {
    return arg1 - arg2;
}

long result;
void loop()
{
    result = MUL(5, 6);
    Serial.println(result); // print 30

    result = SUB(15, 56);
    Serial.println(result); // print -41
    // etc
}
```

Als een functie niets retourneert dan is de functie van het type void. Void betekent ‘leegte’. Ook als je geen argumenten in een functie wilt hebben, mag je void tikken tussen de ronde haken() van de functie en zijn prototype.

Als een functie een waarde moet retourneren, dan moet hij dit ook echt doen. Dit is verplicht. Als een functie niets retourneert dan kan je eerder uit de functie springen door middel van ook een return statement, maar zonder een waarde er achter.

## §4.5 functies callen functies

Je kan ook functies call'en vanuit andere functies. Dit mag niet onbeperkt. Ik weet niet wat het limiet hiervan is, maar het kan voorkomen dat je programma 'undefined behaviour' gaat vertonen als je teveel nested functies aanroept.

```
void foo();
void loop() {
    foo(); // prints "foobar" op monitor
}
void bar() {
// omdat bar boven de call staat, is er geen prototype nodig
    Serial.println("bar");
}
void foo() {
    Serial.print("foo");
    bar();
}
```

## §4.6 conditional statements callen functies

Wat we ook met functie kunnen doen, is ze aanroepen vanuit een condional statement (if en while). Dit doe je in Arduino meestal met de millis() functie.

```
if(millis() - millisPrev >= interval) {
    millisPrev = millis();
    // body
}
```

De functie millis() retourneert de tijd in ms, trekt de vorige sample hier van af en vergelijkt het resultaat met het ingestelde interval.

Om een led te toggelen, kan je dit doen:

```
digitalWrite(ledPin, !digitalRead(ledPin));
```

Als tweede argument wordt de waarde gebruikt die wordt geretourneerd door digitalRead();

## §4.7 passing by reference

Functies kennen een nadeel en dat is dat ze slechts een argument kunnen returnen. Om dit aan te pakken, kunnen we gebruik maken van pointers. Pointers zijn voor beginnelingen erg moeilijk te behappen dus als je dit nog niet snap, kan je het negeren.

Ik ga nu een functie tonen die gebruik maakt van het *passing-by-reference principe*. De functie moet twee opgegeven waardes aanpassen en returnen. Dit doen wij door de adressen (de geheugenlocaties) van de variabelen mee te geven als argumenten. De functie gebruikt twee pointers die elk wijzen naar een van die adressen. Het adres van een variabele geef je aan door de ampersand & voor een variabele te tikken. &var1 betekent 'adres van var1'.

```
int var1 = 5, var2 = 10;

void foobar(int *ptr1, int *ptr2) {
    *ptr1 += 9; // *ptr1 wijst naar var1
    *ptr2 += 15;// *ptr2 wijst naar var2
}

void loop()
{
    Serial.println(var1); // print 5
    Serial.println(var2); // print 10

    foobar(&var1, &var2); // & betekent adres van…

    Serial.println(var1); // print 14
    Serial.println(var2); // print 25
}
```

Wanneer we foobar aanroepen, geven we mee de adressen van 'var1' en 'var2'. '*ptr1' is een pointer die nu naar var1 wijst en '*ptr2' is een pointer die naar var2 wijst.

Je kan deze pointers zien als lokale variabelen die hetzelfde geheugen gebruiken als het geheugen van die variabelen die we meegeven. Door de waardes van *ptr1 en *ptr2 aan te passen, zijn we eigenlijk de waardes van var1 en var2 aan het aanpassen. Dit principe is handig als je meer dan 1 datatype wilt returnen. Bovendien werkt dit principe ook met complexere datatypes zoals arrays en structs. Maar dit is meer iets voor de gevorderde programmeur, dus maak je geen zorgen als je hier nog niks van begrijpt.

## §4.8 Interrupt Service Routine

Een interrupt service routine is een functie die buiten de scope van het hele programma staat. Een ISR wordt aangeroepen door bijvoorbeeld externe invloeden zoals een flank verandering op een bepaald input pin. Als je ISR's met de rest van het programma wilt laten communiceren, dan moet dat via globale variabelen. Je mag namelijk nooit functie calls doen vanuit een ISR, als je dit probeert dan kan je 'undefined behaviour' (ongedefenieerd gedrag) tegen komen. Globale variabelen die worden gebruikt in een ISR dienen te worden voorzien van het 'volatile' keyword in de declaratie.

```
volatile int x;
```

Meestal maakt het niet uit. Dit is een voorzorgsmaatregel die we nemen tegen het optimalisatie proces van de compiler. Als de compiler tegenkomt,

```
if(0) {
```

dan wordt de body achter deze if verwijderd omdat de compiler snapt dat deze nooit uitgevoerd kan worden. Als de compiler tegenkomt

```
if(x) {
```

Dan kan de body verwijderd worden als x nooit een andere waarde krijgt in de scope van het programma. Omdat de interrupts buiten deze scope bestaan, worden acties op globale variabele door ISRs niet erkend.

Ik zal dit nu uitleggen met een voorbeeld. Als je hebt

```
int x;

void foobar() {
    if(x == 10) {
        // body
    }
}

void ISR()
{
    x++;
}
```

Er is de globale x gedeclareerd, maar x is niet geinitialiseerd. X wordt alleen opgehoogd in een ISR en kan zodoende wel of niet de waarde 10 bereiken. Omdat de compiler interrupts in dit opzicht niet snapt, denkt de compiler dat x nooit en nimmer 10 kan worden. Daarom zal de compiler de if-statement in zijn geheel verwijderen. Als we voor int x nu volatile tikken dan voorkomen we dat de compiler de if-statement verwijderd. Als we in foobar zelf x zouden manipuleren…

```
void foobar() {
    if(x == 10) {
        // body
    }

    x++;
}
```

..dan zou de compiler nooit de if-statement weghalen en dan zou volatile niet nodig zijn. Om toch dit zeldzame probleem nooit tegen te komen, is het raadzaam om ten alle tijden globale variabelen die in interrupts voorkomen te voorzien van het keyword 'volatile'.

# Hoofdstuk 5, Geavanceerdere data types

## §5.1 arrays

Je kan met deze variabelen types ook arrays maken. Arrays zijn een soort container die meerdere elementen van een bepaalde variabele type omvat. De syntax is als volgt:

```
long someArray[] = {5, 70000, 6}; // gebruikt 3 x 4 = 12 bytes
for(int i = 0; i < 3; i++ ) {
    Serial.println(someArray[i]);
}
// prints 5
// 70000
// 6
```

Arrays kan je herkennen aan de rechte blokhaken []. Arrays kunnen erg handig zijn in combinatie met for-loops. Je kan 1 element uit het array benaderen door tussen de blokhaken een index nummer te zetten. Een array is in C/C++ geïndexeerd met 0. Dat wil zeggen dat het eerste element eigenlijk het 0$^{e}$ elelement is. In het voorbeeld gebruik ik de variabele 'i' in de for-loop als index. In dit voorbeeld wordt het array ook geïnitialiseerd door middel van de accolades. De compiler weet zo dat het array 3 elementen bevat van het type long en moet zodoende 12 bytes aan geheugen reserveren.

Een array hoeft niet meteen geïnitialiseerd te worden. Dit kan ook later. Het is in dit geval bij het declareren wel nodig dat je de grootte van het array aangeeft tussen de blokhaken.

```
long someArray[3];
void setup() {
    someArray[0] = 5;
    someArray[1] = 70000;
    someArray[2] = 6;
}
```

Doet hetzelfde als het vorige voorbeeld.

Arrays kunnen ook 2 of meer dimensionaal voorkomen. Dit kan je zien aan meer dan 1 setje blokhaken[][]. Het volgende voorbeeld is een stukje code die ik gebruik om een 4 x 4 keypad uit te lezen.

```
char keys[][] = {
    {'1','2','3','A'} ,
    {'4','5','6','B'} ,
    {'7','8','9','C'} ,
    {'*','0','#','D'}
};

char key = keys[row][column];
```

Je mag optioneel tussen de blokhaken de groottes van 4 neerzetten. Als je dit doet, kan je alle binnenste accolades weghalen. Maar persoonlijk vind ik dat moeilijk doen.

Je kan arrays ook als argument meegeven aan functies. Dit werkt onderhuids iets anders dan bij normale variabelen. Als je dit doet, ben je eigenlijk stiekem het adres van het array aan het meegeven. Je maakt automatisch gebruik van het eerder uitgelegde principe ‘passing by reference’.

Als je vervolgens in je functie de lokale array aan het aanpassen bent, dan ben je eigenlijk je originele array aan het aanpassen. Om deze reden is het ook altijd zinloos om een array te retourneren.

```
void myFunction(int localArray[]) {
```

Je kan dan deze functie aanroepen met een ander array.

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

void myFunction(globalArray]) {
```

Deze regel:

```
void myFunction(int localArray[])
```

is voor het programma hetzelfde als:

```
void myFunction(int *localArray)
```

## §5.2 structs

### §5.2.1 declareren van structs

Een andere manier om data bij elkaar te bewaren is door middel van een structure of struct. Een struct kan je zien als een groepje van variabele en functies die bij elkaar horen. Het primaire doel van een struct is om je variabele te orderenen. Met een struct kan je ook object georienteerd programmeren. Een voorbeeld ziet er zo uit.

```
struct structName {
    int a;
    int b;
};
```

Structs en de syntax van structs zijn enigszins een beetje ingewikkeld. Om te beginnen, moet je weten dat wij zojuist een nieuwe data type gemaakt hebben.

```
struct structName
```

Dit is de data type, je kan dit zien als een vervanging voor bijvoorbeeld ‘int’. Je gebruikt deze ook op dezelfde wijze gebruiken als ‘int’. Met deze data type kunnen we nieuwe variabelen maken. Alleen noemen we deze variabele nu objecten. Elk object van het type ‘struct structName’ heeft namelijk twee variabelen (int a en int b). We gaan nu twee nieuw objecten aanmaken genaamd foo en bar.

```
// data type     | objecten
struct structName foo, bar;
```

We hebben nu twee objecten gemaakt. Beide objecten hebben elk nu de variabelen int a en int b. Deze kunnen we benaderen met:

```
foo.a = 5;
if(bar.b == 4) {
```

De syntax is dus object.variabele

We mogen en kunnen ook objecten aanmaken direct achter het maken van de struct. We mogen doen:

```
struct {
    int a;
    int b;
} foo, bar;
```

Ter vervanging van:

```
struct structName foo, bar;
```

Als we het nu zo doen, mogen we ‘structName’ weghalen. We spreken dan van een naamloze struct. Je kan niet met een naamloze struct achteraf nog nieuwe objecten mee te maken. In dit voorbeeld zijn we dus beperkt tot foo en bar. Als we meer objecten willen aanmaken, dan moeten we deze objecten of direct achter foo en bar zetten, of we gebruiken wel een struct naam zodat we met struct structName nog nieuwe objecten willen kunnen maken.

Deze code is een combinatie van beide manieren en dit werkt hetzelfde:

```
struct structName {
    int a;
    int b;
} foo;
struct structName bar;
```

Foo wordt direct gemaakt achter de struct en bar wordt achteraf gedeclareerd.

Het voordeel van het gebruik van een naam voor de struct is dat je ook lokale structs kan aanmaken in functies. Je kan dan ook structs meegeven aan een functie als argument of je kan een functie een struct laten retourneren.

### §5.2.2 Initialiseren van structs

Je kan structs op twee manieren initialiseren. Je kan ze direct initialiseren of je kan ze in functies initialiseren. In functies kan initialiseren er zo uit zien

```
void initStructs() {
    foo.a = 1;
    foo.b = 2;
    bar.a = 4;
    bar.b = 5;
}
```

Het direct initialiseren doen we bijna hetzelfde als we doen met arrays. We gebruiken accolades {} en dit kunnen we op twee manieren doen. Met de eerste manier, is de volgorde van belang. Het maakt bij het initialiseren niet uit of we de objecten direct achter een struct aanmaken of achteraf. Daarom zal ik voor de duidelijkheid beide voorbeelden laten zien.

```
struct structName {
    int a;
    int b;
} foo = {3, 4}, bar = {5, 6};

struct structName foobar = {7, 8}, hello = {9, 10};
```

Als ik bijvoorbeeld alleen a wil initialiseren zou ik ook alle $2^e$ getallen weg kunnen laten, maar andersom kan dat niet. Als ik tien variabelen heb, en ik wil slechts alleen de laatste initialiseren dan moet ik of alles initialiseren, of ik moet de volgorde van de variabele in de structs omdraaien of we vergeten deze methode en we gaan manier twee gebruiken.

Er is namelijk een leuk truukje wat we mogen doen met structs initialiseren. Op deze methode kunnen we initialiseren slechts dat wat we willen en is de volgorde onbelangrijk.

```
struct structName {
    int a;
    int b;
} foo = {.a = 3, .b = 4}, bar = {.b = 5, .a = 6};

struct structName foobar = {.a = 7}, hello = {.b = 9};
```

Als we dus een punt . tikken gevolgd door de naam van de variable dan kunnen we die variabele een functie meegeven.

### §5.2.3 structs met functies

Behalve variabelen kan een struct ook functies bevatten. Strikt gezien kan een struct niet echt functies bevatten maar wel pointers naar de functies. De syntax in een struct voor een functie pointer ziet er ingewikkeld uit, maar dit is het ook het enige ingewikkelde er aan.

Bekijk het volgende voorbeeld.

```
void func1() {
    Serial.print("hello ");
}
void func2() {
    Serial.println("world");
}
struct structName {
    int a;
    int b;
    void (*Function)();
}
struct structName foo = {3, 4, func1};
struct structNamebar = {.Function = func2};
```

Ik heb hier twee functies gemaakt en twee objecten. Deze struct heeft nu ook een functie pointer die 'Function' heet. Deze pointer kunnen we naar elke functie laten wijzen. Dit doe ik ook met de initialisatie van de objecten. Je mag natuurlijk ook doen:

```
void printStuff() {
    foo.Function = func1; // functie pointer wijst naar func1
    bar.Function = func2; // functie pointer wijst naar func2

    foo.Function(); // callt eigenlijk func1
    bar.Function(); // callt eigenlijk func2
} // er staat nu 'hello world' op het scherm
```

Zo zie je ook meteen hoe je de functies kan aanroepen.

### §5.2.3 struct arrays en struct argumenten

Omdat structs voor de compiler niet anders is dan een normale variabele type, mag je met structs dus alles doen als dat wat je met een int zou doen. Je mag een array van struct objecten maken, je mag een struct object meegeven als argument aan een functie en een functie kan een argument rerourneren.

Een array van struct objecten:

```
struct name {
    int a;
    int b;
} object[2];

object[0].a = 6;
object[1].b = 7;
```

Het volgende programma heeft een array met 2 struct objecten en de functie foobar neemt een struct als argument, verhoogt a en b met 5 en retourneert de struct.

```
struct name {
    int a;
    int b;
} object[2] = {{1,2},{3,4}};

struct name foobar(struct name localStruct) {
    localStruct.a += 5;
    localStruct.b += 5;

    return localStruct;
}
void setup() {
    Serial.begin(115200);
}

void loop() {
    object[0] = foobar(object[0]);
    object[1] = foobar(object[1]);

    Serial.println(object[0].a); Serial.println(object[0].b);
    Serial.println(object[1].a); Serial.println(object[1].b);
}  // prints 6789
```

De foobar functie ziet er naar mijn mening vreemd uit. Het blijft ook een vreemd gezicht dat er twee woorden gebruikt worden voor slechts een data type.

Ik laat nu hetzelfde voorbeeld zien, maar dan anders gecodeerd. Ik ga gebruik maken van het ‘passing by reference’ principe. Het probleem met de bovenstaande functie is dat het voor de Arduino relatief veel werk kost om een struct als argument mee te geven en vervolgens een struct te retourneren. Onderhuids gebeurt dit hele proces voor elke variabele van de struct. Met slechts twee variabelen valt dit mee, maar wat als het er 30 zijn? Dan heeft de arduino wel even nodig.

```
struct name {
    int a;
    int b;
} object[2] = {{1,2},{3,4}};

void foobar(struct name *localStructPtr) {
    localStructPtr->a += 5;
    localStructPtr->b += 5;
}

void loop() {

    foobar(&object[0]);
    foobar(&object[1]);

    Serial.println(object[0].a); Serial.println(object[0].b);
    Serial.println(object[1].a); Serial.println(object[1].b);
} // prints 6789
```

Het verschil is de functie foobat. De 'struct name localStruct' is nu veranderd in 'name *localStructPtr'. localStructPtr is nu een pointer van het type 'name'. Wanneer we foobar aanroepen, geven we weer de adressen mee van de objecten. Dit is minder werk dan het meegeven van alle variabelen van deze objecten. De localStructPtr wijst nu bij de eerste call van foobar naar object[0] en de tweede keer wijst de pointer naar object[1].

In foobar staat ook iets nieuws. Omdat localStructPtr nu een pointer is, moeten we in plaats van de . de -> operator gebruiken. Hij doet exact hetzelfde als de . We kunnen aan de -> ook zien dat het een pointer is.

Dit:

```
localStructPtr->a += 5;
```

is namelijk hetzelfde als:

```
(*localStructPtr).a += 5;
```

Beide syntax is geldig.

## §5.3 unions

Als we variabelen hebben die we nooit tegelijk nodig hebben en waarbij we de opgeslagen waardes niet hoeven vast te houden, kunnen we gebruik maken van een union. Een union kan je zien als een struct, alleen de variabelen delen hetzelfde geheugen. De syntax is als volgt:

```
union {
  byte b;
  char c;
} myUnion ;
```

b en c gebruiken nu hetzelfde geheugen. Je kan ze benaderen op dezelfde manier als bij een struct.

```
myUnion.b = 48;
Serial.println(myUnion.c); // print '0'
```

Decimaal 48 is '0'. Als b een andere waarde krijgt, krijgt c dat ook.

Een union opzich is niet bijzonder interessant. Ik gebruik ze nagenoeg nooit. Ik ben wel een mooi voorbeeld tegengekomen. Iemand op het Arduino forum had ooit het probleem dat hij een float variabele over de seriele bus wilde versturen naar een andere arduino. Op een float of double mag je geen bit wise operators gebruiken. Je mag de bits van een float niet shiften en dat was precies wat hij wel moest doen. De oplossing was het gebruik van een union en dat zag er zo uit:

```
union {
  long l;
  float f;
} myUnion;
```

De waarde die hij wilde versturen was myUnion.f. Om toch de bits te kunnen shiften, gebruikte hij een long variabele van deze union. De long bits mogen wel geshift worden. Het versturen van de 'float' zag er zo uit:

```
Serial.write(myUnion.l >> 24);
Serial.write(myUnion.l >> 16);
Serial.write(myUnion.l >>  8);
Serial.write(myUnion.l);
```

## §5.4 bitfields

bitfields zijn een toevoeging aan de struct. Je kan namelijk in een struct zelf variabelen aanmaken met een instelbaar aantal bits. Dit doe je door achter de variabele declaratie een dubbele punt : te zetten met gewenste aantal bits wat je er voor wilt reserveren. Ik laat een struct zien, die een byte op de Arduino in beslag neemt, waarvan we elke bit apart kunnen benaderen.

```
struct {
    uint8_t bit0 : 1;
    uint8_t bit1 : 1;
    uint8_t bit2 : 1;
    uint8_t bit3 : 1;
    uint8_t nibble : 4;
} status;

status.bit1 = 0;
status.bit3 = 1;
status.nibble = 9;
```

Let op dat bit0 t/m bit 3 elke 1 bit zijn. Je kan er daarom een 0 of een 1 in stoppen. Je kan er ook andere waardes in stoppen, maar dan ga je vreemd gedrag krijgen.

De nibble heeft 4 bits en kan hiermee maximaal de waarde 15 hebben. Bij bitfields maakt het uit welke variabele type er voor tikt. Voor het geheugengebruik maakt dit niets uit. Maar voor de resultaten wel. Als ik namelijk ‘int’ of een andere ‘signed’ type gebruikt zou hebben, dan zouden de bits niet 1 maar -1 tonen bij een Serial.printlb(). Waarom leg ik nu niet uit, ik raad je wel aan om voor bitfields unsigned variabele te gebruiken. Bitfields zijn handig voor vlaggen te maken of om bijvoorbeeld met nibbles (halve byte) of tri-state variabelen (gebruikt in motorola protocol) te werken.

We kunnen bit fields ook nog combineren met een union. Beschouw deze code:

```
union {
    struct {
        unsigned int nibble1 : 4;
        unsigned int nibble2 : 4;
    };
    struct {
        unsigned int bit0 : 1;
        unsigned int bit1 : 1;
        unsigned int bit2 : 1;
        unsigned int bit3 : 1;
        unsigned int bit4 : 1;
        unsigned int bit5 : 1;
        unsigned int bit6 : 1;
        unsigned int bit7 : 1;
    };
} foo;
```

We hebben een union genaamd ‘foo’. Foo heeft twee structs. Beide structs gebruiken elk 8 bits. Omdat ze in een union zitten, delen ze geheugen voor deze 8 bits. Struct 1 heeft 2 nibbles (1 nibble is een halve byte) en struct 2 heeft 8 losse bits.

Omdat deze structs in een union zitten, hoeven ze geen eigen naam te hebben. De union zelf mag een naam hebben, maar ik heb in dit voorbeelds slechts een union object (foo) gemaakt.

Waarom willen we dit nu doen? We kunnen de nibbles van foo namelijk gebruiken om 4 bits van foo tegelijk aan te sturen. Stel nu dat ik bit0 en bit3 wil clear’en en bit 1 en 2 wil setten zonder de overige 4 bits aan te raken. Precies dat kunnen we doen met deze ene regel.

```
foo.nibble1 = 0b0110;
```

Op dezelfde wijze kan ik met foo.nibble2 de andere 4 bits manipuleren. Als ik bijvoorbeeld foo.nibble2 gelijk maakt aan 0xF; dan hebben bits 4 t/m 7 allemaal de waarde 1.

## §5.5 typedef

typedef is een keywoard in C en je kan deze gebruiken om zelf een nieuwe variabele type te maken. Een voorbeeld hiervan is de 'byte' type variabele van Arduino. In C noch C++ bestaat er een 'byte' type variabele. In Arduino is de byte namelijk ge'typedef't als een unsigned char.

```
typedef unsigned char byte;
```

Dit lijkt sterk op:

```
#define byte unsigned char
```

Dit laatste werkt, maar dit gaat geheid problemen opleveren in je programma. Waarom, dat wordt uitgelegd in deel 2.

Met de typedef kan je dus zelf variabele type 'verzinnen'. Dit is met name handig voor gebruik met structs. Ik gebruik nu een typedef om een nieuwe type te maken, genaamd 'Trein'.

```
typedef unsigned char byte;

typedef struct Treinen {
    byte adres;
    byte snelheid;
    byte F1_F8;
    byte F9_F16;
    // etc
} Trein;
```

Dit is een struct om een tein objecten mee te maken. Laat ik nu drie treinen aanmaken:

```
Trein Re460;
Trein RE4_4;
Trein AE6_6;
```

Ik kan de drie treinen ook nog meteen initialiseren. Onderhuids wordt 'Trein' vervangen door 'struct Treinen'. Het voordeel is dat we nu meer overal struct moeten tikken.

```
Trein Re460 = {3,0,0,0};
Trein RE4_4 = {4};
Trein AE6_6 = {.adres = 5};
```

In dit voorbeeld heb ik bij de Re460 alle waardes geinitialiseerd en bij de andere twee treinen heb ik alleen de adressen geinitialiseerd.

Sommige 'goede' programmeurs zijn tegen het gebruik van dergelijke typedefs. Waarom is mij ook een raadsel, ze kunnen ontzettend handig zijn. Typedefs zijn met name handig voor structs. Zonder een typedef zou je bij alle functies waar je struct gebruikt het woord 'struct' gevolgd door nog twee worden moeten tikken.

## §5.6 Classes

De classe is een datatype die uit C++ komt. De classe is onderhuids hetzelfde als een struct alleen de syntax is anders. En er zijn kleine subtiele verschillen.

Net als de struct is een classe slechts een container of een beschrijving van iets. Met een classe kunnen we ook Objecten aanmaken. Bijna elke half-knappe Arduino library maakt gebruik van deze classes.

Ik gebruik voor de syntax voorbeelden van de Arduino website. Op deze website https://www.arduino.cc/en/Hacking/LibraryTutorial wordt uitgelegd in detail hoe je zelf classes kan gebruiken om zelf een library te maken. Arduino doet dit ook meteen in andere bestanden dan het .ino bestand. Zo leer je meteen ook iets over het splitsen van code in meerdere bestanden. Classen en hun functies (methodes) hoeven niet in andere bestanden te staan. Je mag in je .ino bestand ook classes aanmaken. Ik laat nu het Arduino voorbeeld zien met een extraatje. Dit mag in je .ino zetten.

```
class Morse // de classe
{
  public:
    Morse(int pin); // constructor
    void dot();     // public 'method'
    void dash();    // public 'method'
    int publicVar;  // public 'variabele'
  private:
    int _pin;            // private variabele
    void someFunction(); // private 'method'
};
```

Het verschil tussen ‘public’ en ‘private’ is dat public variabelen en functies buiten de classe gebruikt mogen worden. De ‘private’ variabele en functies mogen alleen gebruikt worden door andere functies van de classe. In dit voorbeeld mogen _pin en someFunction() alleen maar gebruikt worden in void dot(), void dash() en de constructor ‘Morse’.

Ik wil je ook vertellen dat we een functie van een classe methodes noemen en variabelen van classes noemen we attributen.

Als je alleen al dit in je programma zet, kan het al compileren. Je gebruikt op dit moment nog niet eens geheugen hiervoor. Op dit moment hebben we namelijk alleen nog maar een nieuwe data type gedefinieerd.

De syntax van de methodes ziet er iets anders uit dan wat je tot nu toe gezien heb.

```
Morse::Morse(int pin) { // constructor
  _pin = pin;
  publicVar = 10;
  someFunction();
}

void Morse::dot() { // 'method' van de classe
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
  someFunction();
}

void Morse::dash() { // 'method' van de classe
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
  someFunction();
}

void Morse::someFunction() {
    Serial.println("someFunction");
}
```

Als eerste wordt hier gebruik gemaakt van de ‘nameSpace’ operator :: . Het is in dit geval duidelijk te zien dat deze functies nauw zijn verbonden aan de classe ‘Morse’. Voor het programma is dit ook waar. Alleen objecten die met de Morse classe gemaakt zijn, mogen deze functies gebruiken. We zijn nu vrij om nog meer classes te maken die dezelfde functie namen gebruiken.

Wat je ook ziet is dat is dat de functies, dot en dash en de constructor de private functie ‘someFunction’ aanroepen. En alleen de constructor zet hier de waarde 10 in publicVar (ook een private variabele).

Op dit punt hebben we technische gezien nog steeds geen geheugen geconsumeerd van de Arduino. Als je compileert, zie je echter dat de IDE toch beweert dat je programma groter is geworden. Dat komt door de gebruikte Arduino functies, delay en digitalWrite. Het gebruiken van deze Arduino functies leidt er toe dat het compilatie proces voor ons deze functies gaat opzoeken en mee compileert. Als je arduino functies vervangt door standaard C/C++ operaties, zal je programma geen extra geheugen gebruiken.

Hoewel we nu ook methodes hebben gemaakt voor onze klasse, hebben we nog steeds geen objecten aangemaakt.

We zien op dit punt wel al verschillen tussen een classe en een struct. Een struct heeft geen private varibelen en private methodes. Het is de regel dat je een classe ‘moet’ gebruiken wanneer je ten minste een private variabele heb. Een classe heeft tevens ook al functie bodies. Bij een struct moeten we een functie pointer aan een andere ‘standaard’ functie koppelen. Bij een classe wordt dat voor ons gedaan.

Nu gaan we de classe in gebruik nemen en er objecten mee aanmaken. Dit doen we met de constructor.

```
Morse::Morse(int pin) { // constructor
  _pin = pin;
  publicVar = 10;
  someFunction();
}
```

De constructor wijkt af van andere functies (of methodes) doordat er geen type voor staat. Er staat geen void of int voor de functie. De constructor heeft ook dezelfde naam als de classe. Dit is standaard en ook vereist.

Met de constructor kunnen we zoveel Objecten maken als we willen.

```
Morse object1(5), object2(4);
```

Deze objecten werken hetzelfde als de struct objecten. We kunnen nu de alleen de 'public' functies en variabelen gebruiken met:

```
object1.dot();
object2.dash();
```

Door de private functies en variabelen kan je gemakkelijker iets complexere constructies maken dan je standaard kan met structs.

Met classe objecten mag en kan je hetzelfde doen met structs omtrent arrays en pointers. 'passing by reference' werkt op dezelfde wijze met -> ook het maken van arrays ziet er hetzelfde uit.

Normaliter zet je de classe zelf in een header file en alle functies zet je in een aparte source file. Dit heeft als doel om je code op te splitsen om zo meer leesbaarheid te maken. Tevens kan je dan je nieuwe files met de classes en functies ook gebruiken voor eventuele andere projecten als dat wenselijk is.

## §5.7 variabele scope

Variabelen hebben een zogenaamde ‘scope’. Dit kan je zien als het gebied van een variabele waarin hij bestaat. Een simpel voorbeeld om het  principe te illustreren:

```
byte d = 'd';

void foobar() {
    char c = 'b';

    c++;
    d++;

    Serial.println(c); //  ‘c’
    Serial.println(d);//  'd', 'e' etc
}
```

Er zijn twee variabelen; ‘d’ en ‘c’. ‘c’ is een lokale variabele die alleen bestaat in de functie foobar(). ‘d’ is gedeclareerd buiten functies om en bestaat overal in het bestand. Dit is wat we een globale variabele noemen. ‘c’ onthoudt niet zijn waarde buiten foobar. Als foobar 10x is uitgevoerd, dan er is er 10x ‘c’ geprint. Omdat de variabele ‘d’ ook buiten foobar bestaat, blijft deze zijn waarde behouden. Als foobar 3x wordt uitgevoerd dan wordt er ‘d’, ‘e’ en ‘f’ geprint.

Het gebruik van globale variabele is over het algemeen niet wenselijk, doch soms noodzakelijk. Stel dat nu iemand anders met deze code gaat spelen. Deze persoon ontdekt al gauw dat foobar() twee dingen print, wat hij dan nog niet weet is dat foobar() de waarde van ‘d’ aanpast. Als hij foobar niet opzoekt, kan hij dus foutieve code schrijven waarbij ‘d’ onbedoeld wordt opgehoogd.

De namen die worden gebruikt voor lokale variabele mogen hergebruikt worden. Zowel functie ‘foo()’ als functie ‘bar()’ mogen dezelfde namen gebruiken voor hun lokale variabele.

Een lokale variabele mag tevens ook dezelfde naam gebruiken als een bestaande globale variabele. Dit kan tot verwarrende resultaten leiden.

```
byte x = 5;

void foobar() {
    byte x  = 6;
    Serial.println(x);
}
```

Wordt er nu 5 of 6 geprint? Er wordt 6 geprint. Maar wat nu als we wel in foobar de waarde van de globale x willen veranderen? Omdat Arduino is geschreven in C++ mogen we de :: resolution operator gebruiken. Dat ziet er zo uit.

```
byte x = 5;

void foobar() {
    byte x  = 6;
    Serial.println(::x); // 5
    Serial.println(x);   // 6
}
```

Als we dit in C zouden willen dan moeten we ‘extern’ gaan gebruiken.

```
byte x = 5;

void foobar() {
    byte x  = 6;
    Serial.println(x);   // 6

    {
        extern byte x;
        Serial.println(x); // 5
    }
}
```

We mogen ten alle tijden extra accolades gebruiken voor precies dit soort doeleiden. Wat extern doet, is ‘extenden’ van de scope van een variabele of functie. Extern is een bijzonder ingewikkeld begrip in C. In de toekomst komt er meer informatie over het gebruik.

Ik hoop dat je immers begrepen heb dat het simpelweg een erg slecht idee is om voor globale als lokale variabele dezelfde namen te gebruiken. Just don’t.

# Hoofdstuk 6, Conditonele statements

C/C++ kent verscheidene statements. Deze bepalen welke taken er worden uitgevoerd en waarom. Het waarom gedeelte noemen we de ‘statement’ en de uit te voeren code noemen we de body.

## §6.1, If en Else

Ik begin met de meest simpele de ‘if-statement’ en de bijbehorende ‘else’. Ik leg tevens ook iets uit over de algemene syntax. De syntax van de if-statement is als volgt:

```
if(/* statement */) {
    // body
}
```

Lees: Als ‘statement’ waar is, dan wordt ‘body’ uitgevoerd. In de statement kunnen een hele boel dingen staan. De statement is waar als de statement ongelijk is aan ‘0’. In het volgende hoofdstuk leg ik uit wat je allemaal in de statement kan zetten. Voor nu leg ik alleen uit wat het doet.

Aanvullend is er de ‘else’:

```
if(/* statement */) {
    // body 1
}
else {
    // body 2
}
```

Lees: Als statement waar is, voer body 1 uit. En anders voer body 2 uit. Moge het duidelijk zijn dat een ‘else’ altijd een ‘if’ nodig heeft.  Dit kunnen we verder uitbreiden met de ‘else - if’. Dit is slechts een combinatie van de 2.

```
if(/* statement 1 */) {
    // body 1
}
else if(/* statement 2 */) {
    // body 2
}
```

Lees: Als statement 1 waar is, voer body 1 uit of als statement 2 waar is voer body 2 uit. Voer niets uit als beide statements niet waar zijn.

Deze constructie kan ongelimiteerd worden uitgebreid met meer else-if statements en hij kan optioneel eindigen met een enkele ‘else’ onderop.

```
if(x == 1) {
    // body als x gelijk is aan 1
}
else if(x == 2) {
    // body als x gelijk is aan 2
}
else if(x == 3) {
    // body als x gelijk is aan 3
}
else { // als x iets anders is dan 1,2 of 3
    // body
}
```

Dit is een complete 'boom' van if else-statements. In een body mag je dus van alles neer zetten. Je kan hier ook andere if statements in zetten als je wilt.

```
if(x == 1) {
    if(y == 2) {
        // body
    }
    else if(y == 1) {
        // body
    }
}
```

Er is nog een belangrijk detail. Als je maar 1 regel code wilt uitvoeren achter een if-statement dan zijn de accolades niet verplicht. Je mag dus het volgende typen.

```
if(/* statement */) foo();
else                bar();

// of
if(/* statement */)
    foo();
else
    bar();
```

Hoewel dit 'technisch' gezien correct is, raad ik ten strengste aan om ten alle tijden accolades te gebruiken. Stel nu dat je de volgende toevoeging tikt:

```
if(/* statement */)
    foo();
else
    bar();
    foobar();
```

foobar() is hier geen deel van de else body. Alleen bar() is dat. Foobar() wordt nu onconditioneel uitgevoerd.

Ik wil dus dat iedereen die dit leest, tikt:

```
if(/* statement */) { foo(); }
else                { bar(); foobar(); }

// of
if(/* statement */) {
    foo();
}
else {
    bar();
    foobar();
}
```

Wat je hier ook ziet is dat achter de eerste ‘else’ twee code regels staan en dat ze ‘creatief’ zijn uitgelijnd.

‘Goede’ programmeurs hebben de neiging om hier over te gaan zeiken. Ik kan je echter vertellen uit mijn ervaring dat meer dan een code regel op een lijn en creatief uitlijnen echt wonderen kan doen voor de leesbaarheid van je code.

In dit eerste voorbeeld kan je namelijk nog steeds zien wanneer foo() wordt uitgevoerd en wanneer bar() een foobar() worden uitgevoerd. Het verschil is dat het eerste voorbeeld 2 lijnen in beslag neemt en het tweede ‘correcte’ voorbeeld zeven lijnen in beslag neemt.

Het is namelijk geaccepteerd om een ‘else’ achter de sluitings accolade } van de voorgaande ‘if’ te zetten. Dit ziet er als volgt uit en ik vind het verschrikkelijk omdat het breekt met de rest van de syntax in C.

```
if(/* statement */) {
    // body
} else {
    // body
}


if(/* statement */) {
    // body
} else if(/* statement */) {
    // body
}
```

Dit voorbeeld heeft echter wel mijn voorkeur:

```
     if(x == 1) { /* body */}
else if(x == 2) { /* body */}
else if(x == 3) { /* body */}
else            { /* body */}
```

Ik zal je nu een voorbeeld laten zien die door sommige ‘goede’ programmeurs wel geaccepteerd wordt doch compleet niet logisch is.

```
if(y == 5) {
  if(x == 1)
    foo();
} else
  bar();
```

Als je goed kijkt, zie je dat de ‘else’ bij de eerste ‘if’ hoort. Maar op het eerste oogopslag zou je denken dat de ‘else’ bij de tweede ‘if’ hoort omdat ze dezelfde inspringsafstand of ‘indendatie’ hebben. Ik moest hiervoor ook de grootte van de indendatie  veranderen naar 2. Als je dit dus ‘correct’ wilt ‘neerkalken’:

```
if(y == 5) {
    if(x == 1) {
        foo();
    }
}
else {
    bar();
}
```

Dit neemt naar mijn mening enige onduidelijkheid weg. Duidelijk te zien is naar bij welke if de else hoort. C en C++ zijn in dit opzicht eigenlijk geen perfecte programmeer talen. Er is een syntax bedacht waar tegenstrijdigheden in zit. Na een openings accolade { moet een inspringing of indentatie komen. Als je dus onder een if maar een code regel heb staan dan heeft deze een inspringing… ook als er geen accolades om heen staan.

```
if(/* statement */)
    foo();
```

Geen openings accolade { maar wel een inspringing. Dat breekt met de regels in mijn optiek. En dezelfde onlogica doet zich voor als je een else achter een sluiting accolade } zet. Ook dan is de inspringing raar.

Het komt wel eens voor dat een if-statement zo groot en ingewikkeld wordt dat hij niet in een regel past en dat je dat niet eens meer wilt. Je mag een statement in meer dan 1 regel plaatsen. Er zijn meer dan een manier om dit orderen en de volgende heeft mijn voorkeur

```
if( (x > xMin && x < xMax)
||  (y > yMin && y < yMax) ) {
    // body
}
```

Je moet ook erg goed op de haakjes letten. Deze zou met minder haken goed gaan omdat && voorrang heeft op ||. Als je alle && om zou zetten in || en de || in een && en dan ook nog de haken zou weg halen dan zou je onverwachtste resultaten krijgen.

Ik kies er zelf voor om de || in dit geval op gelijke horizontale afstand te zetten als 'if' om daarmee duidelijkheid te creeren dat we hier te maken hebben met een multi-regel if-statement. Andere mensen kiezen hiervoor:

```
if( (x > xMin && x < xMax) ||
    (y > yMin && y < yMax) ) {
    // body
}
```

Ik vind manier één beter omdat de && en || dan op dezelfde plaats komen te staan op gelijke afstand met de if.

Nu is deze if-statement nog simpel. Ik heb complexere gezien die tot twaalf regels in beslag namen. Deze was zelfs zo complex dat de programmeur nog een extra wit regel had geplaatst in de if-statement zelf. Je snapt dus dat je zo'n ingewikkelde if-statement niet op een regel wilt hebben. Dat is technisch wel mogelijk. Hij past dan alleen niet meer op je scherm. Bovendien heb je dan helemaal geen overzicht meer.

Ik heb een leuk voorbeeldje van het internet geplukt van een beginneling waar een fout in staat. Zijn eerste poging was:

```
if (sex == "m") && ((age >= 18) && (age <= 35)) && (military = "yes") || (pushups >= 50))
```

Hier alleen al kloppen de haken niet. Het is hier ook helemaal niet duidelijk wat de programmeur wilt. Toen kwam hij bij zijn laatste poging:

```
if (((sex == "m") && ((age >= 18) && (age <= 35)) && ((military = "yes") || (pushups >= 50))) {
```

De fout hier is dat er een enkele = is gebruikt in plaats van een dubbele. Hij heeft zelfs nog teveel haken gebruikt ook. Dat kan echter geen kwaad. Liever teveel haken dan te weinig. Je ziet hier dus chaotische code van een beginneling die er niet uit komt. Het is ook nog steeds niet precies duidelijk aan welke condities precies voldaan moet worden en wat belangrijk is. Als we nu deze if-statement anders neerkalken, krijgen we:

```
if( (sex == "m")
&&( (age >= 18) && (age <= 35) )
&&( (military == "yes") || (pushups >= 50) )  ) {
    // body
```

Je kan hieruit makkelijker opmaken, wanneer het geheel nu waar is. De persoon moet een man zijn EN hij moet tussen de 18 en 35 jaar oud zijn EN hij moet of al soldaat zijn of meer dan 50 pushups kunnen doen.

## §6.2 Switch-case

De switch-case statement is nauwverbonden aan een if-else keten, maar het is sneller in uitvoering. Beiden hebben hun voor en nadelen. Wat beter is voor jou programma, moet je zelf bepalen. Ik leg slechts de werking en de syntax uit. De syntax kan als volgt zijn:

```
switch(x) {
case 1:
    // body

case 2:
    // body
    break;

case 3:
    // body
    break;

default:
    // body
    break;
}
```

Het werkt als volgt: De switch-statement bekijkt de waarde van x en voert afhankelijk van deze waarde 1 van de cases uit. Als er geen case is met een corresponderende waarde dan wordt de body van de default case uitgevoerd. De default case is echter optioneel en je kan hem weglaten. Dit is de equivalent van de laatste else in een langere if-else keten.

Misschien zijn er twee dingen die je zijn opgevallen. De eerste case heeft geen break statement en er is geen inspringing na de openings accolade {.

De break statement beëindigt een case. Met break kan je ook uit functie bodies en while-loops springen maar dit volgt later.

Als case 2 is uitgevoerd, zorgt de break er voor dat case 3 of elke andere case niet meer wordt uitgevoerd. Wat gebeurt er nu als x gelijk is aan 1? In dat geval wordt de body van case 1 wel uitgevoerd, maar de body van case 2 wordt daarna ook uitgevoerd. Wanneer de code de break tegenkomt, wordt er weer uit de switch-case gesprongen. Als je een switch-case hebt zonder deze breaks dan spreken we van een ‘fall-through switch-case’. De laatste case (of default label) moet wel een break hebben.

Het tweede opvallende, is dat er na een openings accolade { geen inspringing volgt. Dit heeft een speciale reden. Ik leg eerst iets uit over de ‘case’. Een case is een label. Deze labels zijn de enige labels die we gebruiken in C/C++. Een label kan je zien als een overblijfsel van een lagere programmeer taal genaamd ‘Assembly’. In Assembly staan labels altijd zonder inspringen.

Dat gezegd hebbende. Het breekt aan een kant wel met de rest van de syntax over dat er een inspringing moet komen na een {. Je mag een switch case ook zo opzetten:

```
switch(x) {
    case 1:
    // body

    case 2:
    // body
    break;

    case 3:
    // body
    break;

    default:
    // body
    break;
}
```

Ik vind beide manieren prima. Beide manier worden ook door anderen geaccepteerd.

Dat wat achter case staat, moet een constante zijn. Dit mag een character zijn; ‘A’ of ‘5’. Het mag een nummer zijn; 0,1,2. Wat ik bijna altijd aanraad om te doen is om geen nummers te gebruiken maar om een #define, const int of een enum te gebruiken. Soms is een nummer wel beter, als je bijvoorbeeld een X of een Y coördinaat  in een switch-case voert, dan is een nummer logischer.

```
enum { // 'enumerates' the constants
    chickenCanFly = 1,
    foobar,
}
const int cows_can_fly = 5;
#define PIGS_CAN_FLY 4

switch(x) {
case chickenCanFly: // 1
    // body
    break;

case foobar: // 2
    // body
    break;

case PIGS_CAN_FLY: // 4
    // body
    break;

case cows_can_fly: // 5
    // body
    break;

case '0': // decimaal 48
    // body
    break;
}
```

Dit is allemaal geldige syntax. Het verschil tussen enum, #define en const int komt nog later aan bod.

Een nadeel van een switch-case zijn tevens deze constantes. Wat je er bijvoorbeeld niet mee kan is om een bepaalde getallen gebieden te kijken. Er is geen officiële switch-case substitutie voor:

```
     if(x >= 0  && x < 10) { // als 0 <= x < 10
    // body
}
else if(x >= 10 && x < 20) { // als 10 <= x < 20
    // body
}
else if(x >= 20 && x < 30) { // als 20 <= x < 30
    // body
} // etc
```

Onofficieel is er voor dit specifieke scenario wel iets wat je kan doen.

```
byte y = x / 10;

switch(y) {
case 0: // 0 - 9
    // body
    break;

case 1: // 10 - 19
    // body
    break;

case 2: // 20 - 29
    // body
    break;
```

Omdat de gebieden uit 10 getallen bestaan, delen we x door 10 en stoppen het resultaat in y die we weer gebruiken voor de switch-case. Dit is een programmeer truukje die ik graag wil delen.

Of je een rij if-statements, if-else statements of een switch-case moet gebruiken is afhankelijk van wat je wilt. Als er meer dan 1 case uitgevoerd moet kunnen worden, is een keten van if-statements de manier. Als je in onregelmatige getallen gebieden moet gaan kijken of stringen teksts moet vergelijken dan zijn if-else ketens het beste. En soms is een fall-through switch-case erg handig.

Een veel voorkomende pitfall is het vergeten van het tikken van een break statement, let hier goed op. Dit wilt ervaren programmeurs ook wel eens overkomen.

Als een switch-case in een functie ligt, dan mag elke case ook beeindigd worden met een return statement. Dit is handig als we een look-up table moeten maken. Stel nu dat we een ascii character die een hexadecimaal getal voorstelt moeten omzetten naar een decimale waarde. De waardes ‘A’ – ‘F’ liggen niet in de buurt van ‘0’ – ‘9’. We kunnen wel berekenen wat de waarde moet zijn, maar qua uitvoer is het sneller om een look-up table te maken. Dat kunnen we als volgt doen:

```
int8_t hex2dec(char c) {
    switch(c) {
    case '0': return 0;
    case '1': return 1;
    case '2': return 2;
    case '3': return 3;
    case '4': return 4;
    case '5': return 5;
    case '6': return 6;
    case '7': return 7;
    case '8': return 8;
    case '9': return 9;
    case 'A': return 10;
    case 'B': return 11;
    case 'C': return 12;
    case 'D': return 13;
    case 'E': return 14;
    case 'F': return 15;
    default:  return -1;
    }
}
```

Als c een van de ascii waardes bevat zal de functie de corresponderende decimale waarde rerourneren. Als c een ‘ongeldige waarde’ (een waarde waarvoor geen case is) bevat, dan retourneert de functie -1 met default label.

## §6.3, While

De while-loop is een van de twee loop functie die we kennen in C/C++. De syntax is heel simpel.

```
while(/* statement */) {
    // body
}
```

Als de statement waar is, wordt de body hier uitgevoerd. Net als bij de 'if'. Het verschil is dat na het uitvoeren van de body, de statement opnieuw wordt bekeken. Als die dan nog steeds waar is, wordt de body opnieuw uitgevoerd. Je moet hier dus ook alert zijn op wat je doet. Deze while:

```
byte x = 0;
while(x != 7) {
    x += 2;
}
```

Zal nooit stoppen omdat x nooit 7 zal zijn, maar deze while:

```
byte x = 0;
while(x != 6) {
    x += 2;
}
```

Zal wel stoppen omdat x wel 6 wordt.

Er bestaat ook nog de do-while loop.

```
do {
    // body
}
while(/* statement */);
```

De body wordt onconditioneel 1x uitgevoerd, daarna wordt de while bekeken of de body nogmaals uitgevoerd moet worden.

Ervaring heeft me geleerd dat deze while lussen nagenoeg overbodig zijn in een goed programma. Ik gebruik ze bijna nooit. Je kan dus een fout maken en per ongeluk een eindeloze while-loop maken. En processen van je programma, mogen eigenlijk geen andere processen van je programma blokkeren. Bovendien met een goede structuur heb je simpelweg geen while loops nodig. En een do-while die heb ik nog nooit nodig gehad of gebruikt.

## §6.4, De for-loop

De for-loop is een loop die een x aantal keer uitgevoerd wordt. De syntax is als volgt:

```
byte i;
for(i = 0; i < 5; i++) {
    // body
}
```

In C++ en Arduino mag je ook nog doen:

```
for(byte i = 0; i < 5; i++) {
    // body
}
```

In de statement gebeuren drie dingen. Een variabele, in ons geval 'i', wordt geïnitialiseerd. Hij krijgt een waarde, 0 in dit geval. Dan wordt bekeken door: i < 5 of de body wel of niet moet worden uitgevoerd. En als laatste wordt i opgehoogd met 1. Deze loop zal dus precies 5x worden uitgevoerd.

Je kan van alles en nog wat tikken. De for-loop kan ook terugtellen, stapjes van 2 maken of naar links of naar rechts shiften. Hierbij bestaat dezelfde pitfall als bij de while loop. Je kan op meer dan een eindeloze for-loop maken.

```
for(byte i = 0; i < 1000; i += 2) {
    // body
}
```

Wordt eindeloos herhaalt omdat de byte i niet hoger dan 255 kan worden.

Dit zijn alle conditionele statements in C/C++. Het volgende hoofdstuk gaat de operators dan weten we ook wat we precies in de statements kunnen zetten.

## §6.5 Stijl

### §6.1.1 accolades

Zoals je gezien heb in paragraaf §6.1 en §6.2, is er meer dan een manier om switch-case labels en accolades te plaatsen (of niet te plaatsen). Er is meer dan een stijl mogelijk en meer dan een stijl wordt door anderen geaccepteerd. Van de volgende voorbeelden worden alleen manieren 1 en 2 geaccepteerd.

```
if(/* conditie */) // 1
{
    foo();
}

if(/* conditie */) { // 2
    bar();
}

if(/*conditie 1*/) { // 4
    foo();
    if(/*conditie 2*/) {
        bar();
        foobar(); } }
// scheelt heel veel wit regels
```

Manier drie heb ik zelf bedacht. Ik heb er al menig mensen boos mee gemaakt, maar ik blijf hem gebruiken. In een andere taal genaamd ‘python’ zijn er helemaal geen accolades. Python gebruikt slechts inspringingen.

```
if /* conditie */ :
    foo()
    bar()
```

Ik heb dit als erg prettig ervaren tot nu toe. Zodoende ging ik het zelfde doen voor C. Ik zette alle sluitings accolades achter de laatste regel. En er zit een heel simpele logica in. Na elke openings { accolade volgt een inspringing op de volgende regel en na elke sluit } accolade is er een inspringing minder. Dit werkt ook prima als er meer dan een sluit accolade achter elkaar staat. Ik heb ooit de volgende functie ontworpen. Ik zal hem op alle drie manieren tonen, de slechte, de algemeen geaccepteerde en de goede.

```
void readInputs()
{
    byte slave, pin, element, /*IO,*/ state;
    for(slave=0; slave<nMcp; slave++)
    {                                       // for all MCP23017 devices
        static unsigned int inputPrev[8] = {0,0,0,0,0,0,0,0}, input = 0;
        input = mcp[slave].getInput(portB) | (mcp[slave].getInput(portA) << 8);
        for(pin=0;pin<16;pin++)
        {
            if((input & (1 << pin)) != (inputPrev[slave] & (1 << pin)))
            {
                inputPrev[slave] = input;
                if(input & (1<<pin))
                {
                    state = 0;
                }
                else
                {
                    state = 1;
                }
                IO = pin + slave * 16;
                for(element=0; element<elementAmmount; element++)
                {
                    unsigned int eeAddress = IO * 8 ;
                    EEPROM.get(eeAddress, Array);
                    if(type != 255)
                    {
                        if(!debug) {
                            sendState(state);
                        }
                        if(type == decouplerObject)
                        {
                            setOutput(outputIO, state);
                        }
                        if(hasLedIO == YES)
                        {
                            setOutput(ledIO, state);
                        }
                        return;
                    }
                }
            }
        }
    }
}
```

```
void readInputs() {
    byte slave, pin, element, /*IO,*/ state;
    for(slave=0; slave<nMcp; slave++){
        static unsigned int inputPrev[8] = {0,0,0,0,0,0,0,0}, input = 0;
        input = mcp[slave].getInput(portB) | (mcp[slave].getInput(portA) << 8);
        for(pin=0;pin<16;pin++) {
            if((input & (1 << pin)) != (inputPrev[slave] & (1 << pin))) {
                inputPrev[slave] = input;
                if(input & (1<<pin)) {
                    state = 0;
                }
                else {
                    state = 1;
                }
                IO = pin + slave * 16;
                for(element=0; element<elementAmmount; element++){
                    unsigned int eeAddress = IO * 8 ;
                    EEPROM.get(eeAddress, Array);
                    if(type != 255) {
                        if(!debug) {
                            sendState(state);
                        }
                        if(type == decouplerObject) {
                            setOutput(outputIO, state);
                        }
                        if(hasLedIO == YES) {
                            setOutput(ledIO, state);
                        }
                        return;
                    }
                }
            }
        }
    }
}

void readInputs() {
    byte slave, pin, element, /*IO,*/ state;
    for(slave=0; slave<nMcp; slave++){                                      // for all MCP23017 de
        static unsigned int inputPrev[8] = {0,0,0,0,0,0,0,0}, input = 0;
        input = mcp[slave].getInput(portB) | (mcp[slave].getInput(portA) << 8); // read both I/O ports
        for(pin=0;pin<16;pin++) {                                           // for all 16 I/O pins
            if((input & (1 << pin)) != (inputPrev[slave] & (1 << pin))) {   // if an input (
                inputPrev[slave] = input;
                if(input & (1<<pin))    state = 0;                          // store the state of
                else                    state = 1;
                IO = pin + slave * 16;                                      // calculate which IO
                for(element=0; element<elementAmmount; element++){          // for every rail
                    unsigned int eeAddress = IO * 8 ;
                    EEPROM.get(eeAddress, Array);                           // fetch ID from EEPRO
                    if(type != 255) {
                        if(!debug)                  sendState(state);
                        if(type == decouplerObject) setOutput(outputIO, state);
                        if(hasLedIO == YES)         setOutput(ledIO, state);
                        return; } } } } }              // if type = 255, the device is not defined
```

Nu is het net waar je gelukkig van wordt. Ik geloof niet dat zo'n hele trap van sluit accolades ook maar iets toevoegd aan de leesbaarheid. Ik geloof ook niet dat zo veel witregels echt nodig zijn. Ik weet wel dat methode drie achtien regels in beslag neemt en methode één wel 44. Ik heb methode drie al een langere tijd in gebruik, en ik vind hem echt erg fijn. Het oogt misschien druk, maar als je er eenmaal mee werkt, valt het reuze mee. Je kan het zien als het lezen van een boek. Manier één is dan net als het toevoegen van een wit regel na elke zin. Waarschijnlijk bevoordert de leesbaarheid, je maakt er alleen niemand gelukkig mee als je op die wijze een boek schrijft.

Nu komt het puntje bij paaltje. Er zijn slechts drie dingen die ik je wil meegeven over accolade gebruik.

1. Het maakt niet uit welke stijl je kiest, zolang je hem maar consistent gebruikt.
2. Oordeel niet te snel over iets waar je nog nooit mee gewerkt heb.
3. Denk voor je zelf. Dat wat men ooit gekozen heeft als standaard, hoeft niet persé ook de 'beste' methode te zijn.

### §6.5.2 camelCase en underscores

Omdat we variabelen en functies lang niet altijd in een woord uit kunnen drukken, is er de noodzaak om meerdere woorden achter elkaar te tikken. Meestal maakt men gebruik van camelCase. Hierbij begint het eerste woord met een kleine letter en elk aansluitend woord begint met een hoofdletter. Dit maakt het lezen van code aanzienlijk prettiger. Sommige mensen gebruiken in plaats van camelCase under_scores om leesbaarheid te krijgen. Dit mag ook en het is net wat je fijner vindt. Omdat macro's vaak met HOOFDLETTERS worden geschreven, is het wel een standaard om voor macro's wel underscores te gebruiken.

De 'class' datatype wordt meestal begonnen met een Hoofdletter. Als je naar Arduino code kijkt, zie je het veel voorkomen. Serial.begin, Wire.begin etc. Je kan om deze reden met typedef ook zelf gemaakte data type namen met een Hoofdletter maken. Dan kan je zien dat je met een complexere datatype te maken heb.

In de volgende delen van de cursus, maak ik gebruik van software timers. Elk van deze timer eindigt bij mij met de hoofdletter T bijvoorbeeld "blinkT". Dit is slechts een syntax ding van mij. Zo kan ik zien welke variabele een timer is.

Het is ook hier belangrijk dat je consistent blijft in de stijl. Als je stijl overal hetzelfde is, kan je makkelijker onthouden of zelfs raden hoe iets heet. Ik kwam in assembly source code ooit een functie tegen die de IO moest initialiseren. De functie heette: "initi_o". Dit is dus hoe je het niet moet doen. Niemand zal dit ooit kunnen raden, logsichere benamingen zouden zijn; init_io of initIO.

Ik raad ook af om veelvuldig afkortingen te gebruiken. Als je dat doet dan ben je meteen ook de enige die ze snapt. Ook als je afkortingen in jouw oren nog zo logisch zijn, voor een anders zijn en blijven het afkortingen. Sommigen zijn wel zo simpel zoals 'ptr' voor 'pointer', die snappen mensen wel.

Verzin dus goede duidelijke namen die beschrijven waarvoor de variabele of functie gebruikt wordt. Wees hierin consistent.

# Hoofdstuk 7, Operators

Operators gebruiken we onder anderen in de conditionele statements (logic operators). Ook gebruiken we ze om mee te rekenen. Net zoals dat vermenigvuldigen voorgaat op optellen en aftrekken, is er een dergelijke rangorde onder de operators. Als eerste behandelen we de relationele operators die worden gebruikt in de conditionele statements.

== betekent gelijk aan

```
if(x == y)
```

In deze if statement wordt x met y vergeleken. De body wordt alleen uitgevoerd als x gelijk is aan y. Een veel voorkomende fout onder beginnelingen is dat er een enkele = wordt getikt

```
if(x = y)
```

In deze if gebeurt iets compleet anders. x neemt hier de waarde over van y. Deze operatie retourneert een logische '1' en daarmee wordt de body van de if ook uitgevoerd. Gebruik voor dergelijke vergelijkingen altijd twee ==.

!= analoog aan de ==. Werkt net omgekeerd en betekent 'is ongelijk aan'.

```
If(x != y)
```

De body wordt nu alleen uitgevoerd als x ongelijk is aan y

<, >, <= en >= kleiner dan, groter dan, kleiner of gelijk aan en groter of gelijk aan.

```
if(x < y)
```

Als x kleiner is dan y wordt de body uitgevoerd.

In het geval we combinaties willen maken van meerdere statements kunnen de OR ||, de AND && en de NOT ! gebruiken. Dit zijn de 3 logical operators

```
if(x < y && z >= 7)
```

Als x kleiner is dan y en z is groter en gelijker dan 7 dan wordt de body uitgevoerd.

```
if(x < 3 || y >= 1)
```

Als x kleinder is dan 3 of y is groter of gelijk aan 1 dan wordt de body uitgevoerd.

```
if(x)
```

In principe wordt elke byte waarde die niet gelijk is aan 0 als 'waar' beschouwd door de if-statement. Ook als x een negatieve signed waarde is. Dus als er ook maar 1 bitje geset is, dan is if waar.

```
if(!x)
```

De ! operator is de logical NOT. In dit voorbeeld is de if alleen waar als x gelijk is aan 0.

Dan hebben we nog de enkele | (OR) en de enkele & (AND). Dit zijn bit-wise operators omdat deze operators bits aan kunnen passen. De & wordt gebruikt om bytes met elkaar te AND'en. Stel dat we van een variabele alleen zijn geintereseerd in de 6e bitDat kan er als volgt uit zien:

```
if(x & 0b01000000)
```

Als bit 6 van x hoog is dan is het resultaat van x & 0b01000000 waar en dan wordt de body uitgevoerd. Alle andere bits van x doen er niet toe.

```
if(!(x & 0b00100000))
```

Zo is deze statement alleen waar als de 5e bit van x niet is geset.

De enkele | in een if-statement:

```
#define AND_MASK1 0b00001000
#define AND_MASK2 0b00100000

if(x & (AND_MASK1 | AND_MASK2))
```

Ik heb hier 2 constantes gemaakt en ik gebruik ze als AND maskers. In deze if statement worden eerste beide AND maskers met elkaar ge'OR'ed en het resultaat wordt ge'AND'ed met x. De if statement is nu alleen waar als 3e en 5e bit van x allebei hoog zijn. Het is hier extreem belangrijk dat er om de OR instructie ronde haken staan. De & staat namelijk hoger in de operator rangorde dan |.

De | en & operators kunnen ook gebruikt worden om variabelen aan te passen.

```
x = x | 9; // set bit 0 en bit 3
// of
x |= 9;
```

Analoog hieraan kunnen we het zelfde doen met de & om bits te clearen in plaats van de setten

```
x = x & ~9; // clears bit 0 en bit 3
// of
x &= ~9;
```

Het ~ is ook een NOT operator. Alleen dit is de bitwise NOT en niet de logical NOT. Dit betekent de inverse van de variabele. 9 is binair gelijk aan 0b00001001 zodoende is ~9 gelijk aan het omgekeerde 0b11110110. Door x te AND'en met deze waarde worden alleen de 0e en 3e bit gecleared.

De ^ is de XOR of exclusive OR operator. De exclusive OR operator kan worden gebruikt om verschillen tussen bits te bekijken en om bits te toggelen. Beschouw deze if:

```
if(x ^ y)
```

Deze if is alleen waar als er verschil zit tussen x en y. Hij doet in dit geval hetzelfde als !=.

```
if(x ^ (y & 0b00001111))
```

In dit voorbeeld is de if waar als tenminste 1 van de eerste 4 bits x waar zijn, of als de achterste 4 bits van x verschillen met de achterste 4 bits van y. Je ziet dus dat de XOR ^ tot ingewikkelde contructies kan leiden.

```
x = x ^ 0b00001111; // of
x ^= 0b00001111;
```

Deze lijn klapt de achterste 4 bits van x om, ongeacht wat ze zijn.

De overige bitwise operators zijn de << en >>. Dit zijn de shift operators en ze worden gebruikt om alle bits van een variabele op te schuiven naar links of naar rechts.

```
byte y = 0b00001111, x = 0;
x = y << 4;
Serial.println(x, BIN); // prints 11110000
```

Of:

```
byte x = 0;
x = x | (1 << 4); // sets the 4th bit of x
Serial.println(x,BIN); // prints 10000
```

De laatste operators die ik wil behandelen zijn de rekenkundige operators. Deze zijn vrij simpel. Je hebt: vermenigvuldigen *, delen /, modulo %, optellen + en aftrekken -. Ik leg alleen het delen en de module uit omdat de rest me wel duidelijk lijkt.

Als je in C een integer getal deelt door andere integer getal dan is het resultaat ook een integer getal. Deze wordt naar beneden afgerond. Zo is 3 / 5 gelijk aan 0 en 5 / 3 is gelijk aan 1. De module % geeft het restant terug. Als je 2 deelt door 3 dan is het resulaat 0 maar het restant is 2.

```
10 / 3 = 3
10 % 3 = 1
```

3 past 3x in 10. Daarom is het resultaat 3. Als je 3 x 3 aftrekt van 10, dan houd je er 1 over en dat is het restant en dat is waar de modulo operator goed voor is.

Stel nu als je wilt controlleren of een getal even of oneven is. Dat kan op 2 manieren. Je kan de $0^e$ bit controlleren. Als deze waar is, dan is het getal oneven.

```
if(x & 0b1) {
    // getal is oneven, want de 0e bit is waar
}
else {
    // getal is even want de 0e bit is niet waar
```

Of je gebruikt de modulo operator:

```
if(x % 2) {
    // getal is even want het restant is niet 0
else {
    // getal is oneven want het restant is 0
```

0 % 2 = 0
1 % 2 = 1
2 % 2 = 0
3 % 2 = 1
etc etc.

Alle bit-wise operators en rekenkundige operators kunnen gebruikt worden met de assignment operator =. Zo worden al deze operators allemaal asignment operators genoemd. De standaard assignment operator in al zijn glorie:

```
x = 5;
```

Is duidelijk wat het doet.

Voor alle bitwise en rekenkundige operators mag je dit.

```
x = x + 5;
y = y & 0b00001111;
```

omzetten in:

```
x += 5;
y &= 0b00001111;
```

Dit mag je dus doen met:

```
<<=, >>=, +=, -=, ^=, |=, &=, *=, %= en /=
```

Er zijn nog twee onbehandelde operators. De “ternary conitional” ‘ ?: ’. Deze is 100% overbodig, je kan hem altijd vervangen door een if-else statement en hij is altijd moeilijk te lezen. Geen regel C code met de ? is ooit leesbaar geweest. De syntax is als volgt:

```
max = (a > b) ? a : b;
```

Als dat wat links van de ? waar is, dan is max gelijk aan ‘a’ en anders is max gelijk aan ‘b’.

Arduino maakt wel gebruik van de ternary operator in combinatie met macro’s. De functies min(), max(), abs() en constrain() zijn hier voorbeelden van. Deze macro’s zijn vermomd als functies. Ze staan immers niet in hoofdletters. Je kan ze dus ook niet gebruiken met een prefix of postfix, x++, --y. Dit staat overigens wel vermeld op hun website dat je dit niet moet doen.

De laatste operators zijn de postfix var++/var—en de prefix ++var/--var. Dit zijn ook simpele rekenkundige operators. Ze verhogen of verlagen een variabele met 1. Wat ze speciaal maakt, is dat je ze in een functie argument mag zetten. Je mag doen:

```
void bar(byte _var1, byte _var2) {
    Serial.printlln(_var1);
    Serial.printlln(_var2);
}

void foo() {
    byte x = 5;
    byte y = 6;

    bar(++x, y--); // print 6 en 6
}
```

Als ++ of – voor een variabele staan (prefix) dan worden eerste de waardes opgehoogd of verlaagd en dan als argument aan de functie meegegeven. Als de ++ of – achter de variabele staan dan wordt eerst de waarde meegegeven en daarna opgehoogd/verlaagd. Hoewel ze handig kunnen zijn, zijn ze ook overbodig. Ze kunnen soms ook onverwachtse resultaten geven in combinatie met macro's. Ik raad zelf aan om ze te vermijden, ze te vervangen door += 1 of -= 1 en ze al zeker niet in functie argumenten te zetten. Voorkomen is in dit geval beter dan een mogelijke bug ervaren.

# Hoofdstuk 8, Rekenen met de Arduino.

Rekenen in C en met name met een microcontroller daar kan je een heel vak aan toewijden op het HBO. Het vervelenste is dat berekeningen per platform kunnen verschillen.

Ik wil de twee grootste pitfalls behandelen.

1. Gebruik altijd haakjes als je niet zeker weet of je berekening werkt. De operator rangorde is te vinden op internet, maar als je het even niet meer weet, zet haakjes om je deel berekeningen.
2. Let op de overflows. Dit is de moeilijkste, want je ziet hem niet altijd aankomen. Als je twee bytes met elkaar vermenigvuldig dan heb je een grote kans dat het resultaat niet meer past in een byte. Bekijk de volgende code

```
uint16_t result;

uint8_t val1 = 100;
uint8_t val2 = 200;

result = val1 * val2;
```

Het resultaat past in principe in 'result' die een 16 bit grootte heeft. Deze instructie verloopt ook goed.

```
uint16_t result;
uint8_t val1 = 100;
uint8_t val2 = 200;

result = val1 * val2 / 2;
```

Van deze instructie zou je ook verwachten dat de uitkomst goed is, de waarheid is echter iets anders. Het probleem is dat er ongewenst en tevens onverwachts een automatische type-case plaats vindt naar een 8 bit variabele, voordat het resultaat in 'result' wordt gezet. Zodoende vinden er overflows plaats en krijgt je een vreemd onverwachts antwoord. Dit probleem los je op door een typecase naar een 16 bit waarde te gebruiken. Dat ziet er zo uit:

```
uint16_t result;
uint8_t val1 = 100;
uint8_t val2 = 200;

result = val1 * val2 / (uint16_t)2;
```

Het resultaat van de som wordt nu wel goed opgeslagen. Wanneer er wel of niet ongewenste typecasts plaatsvinden, is soms lastig te voorspellen. Ik heb namelijk ook een compiler gehad die het tweede voorbeeld wel goed afhandelde.

Om dit te voorkomen is het raadzaam om altijd een typecast te gebruiken naar een long. Dan forceer je altijd je eigen typecast waardoor er geen ongewenst gedrag kan voorkomen en waarschijnlijk past dat waarmee je rekent altijd wel in een long. Je mag natuurlijk ook een long long of een int64_t gebruiken als je echt zeker van wilt zijn. De rekenfuncties van Arduino werken om deze redenen ook met longs.

Je kan natuurlijk ook functies gebruiken om mee te rekenen. Als je per functie slechts twee waardes gebruikt om mee te berekenen dan is er een kleinere kans dat er iets mis gaat.

Stel dat we deze berekening willen doen (200 + 300) x (400 + 100). Dan kan je proberen:

```
unsigned int x;

x = (200 + 300) * (400 + 100); // 250000??
```

Zonder te testen durf ik niet eens met zekerheid te zeggen dat deze regel doet wat wij verwachten. Als ik er (uint32_t) voorzet dat zal hij het wel doen. Maar laten we nu eens functies gebruiken.

```
x = MUL(ADD(200,300), ADD(400,100)); // 250000!!

long MUL(long x, long y) {
    return x * y;
}
long ADD(long x, long y) {
    return x + y;
}
```

Voor deze berekening durf ik mijn hand wel voor in het vuur te steken. De functies gebruiken voor alles long en ze doen slechts 1 berekening.

Het is belangrijk te onthouden dat een Arduino altijd maar 1 ding tegelijk kan uitvoeren. Ook als je een lange regel code tikt, de arduino doet slechts een berekening per keer. Het is daarom raadzaam om elke wiskunige actie tussen haken () te zetten en ten alle tijden een type-case mee te geven. Zo voorkom je meeste bugs. En test altijd je berekeningen of ze de gewenste resultaten opleveren.

Arduino heeft ook regenkundige functies ingebakken voor de wat hogere wiskunde (sinus, worteltrekken etc). Op deze website staat een compleet overzicht van de beschikbare functies.

https://www.arduino.cc/en/Math/H

# Hoofstuk 9, Arduino functionaliteit en randaparatuur

De mensen van Arduino hebben een uitgebreide set functies gemaakt voor ons die we kunnen gebruiken voor onze progrmma's. En vele elektronica fabrikanten hebben allerlei handige apparaten gemaakt die we aan een Arduino kunnen koppelen. We hebben klok modules, temperatuur sensor en lcd schermen om er een paar te noemen. Het merendeel van deze apparaten kunnen we met serieel, I2C of SPI aansturen.

## §2.1 De seriele communicatie

De Seriele library van Arduino komt met zeer goede functie. Ik leg eerst de basis werking uit en daarna leg ik de belangrijke functies uit. Ik geef ook een paar voorbeelden van truukjes om ingelezen data te verwerken.

De seriele commincatie is meer dan alleen software. Het in- en uitklokken van bytes wordt gedaan door een hardware component genaamd UART. Deze UART module is een onderdeel van de microcontroller zelf. De UART kan gelijktijdig bytes uitklokken en inklokken. Het in- of uitklokken gebeurt wel met een byte tegelijk. De frequentie waar dit klokken gebeurt, heet de baudrate en wordt uitgedrukt in bits per seconde of BPS. Een byte bestaat hier uit 8 databits, een stop bit en een start bit.

Bij de seriele communicatie wordt gebruikt gemaakt van twee 64 byte buffers om zowel de te verzenden als de te ontvangen bytes in op te slaan. Als de Arduino een byte ontvangt, dan wordt het seriele ISR aangeroepen en deze ontvangen byte wordt in de Rx buffer geladen. Met Serial.read() kunnen wij deze buffer uitlezen. Omgekeerd gebeurt er hetzelfde met de Tx buffer. Wanneer er een byte verstuurd is, wordt ook het seriele ISR uitgevoerd. Het ISR kijkt in de Tx buffer of er iets te versturen is, als er iets te versturen is dan haalt het ISR de byte uit de Tx buffer en klokt hem uit. De verstuur buffer of Tx buffer kunnen we vullen met Serial.print(), Serial.println() of Serial.write().

Als we met de verzend functies de buffer sneller vullen dan het ISR kan legen dan zou de Tx buffer overflowen. Arduino heeft daarvoor een beveiliging ingebouwd. Als de Tx buffer vol is dan blokkeren de print en de write functie het programma, totdat er genoeg ruimte is om de data in de Tx buffer te laden. Dit voorkomt dataverlies, maar het blokkeert wel je programma. Omgekeerd bij het ontvangen kan de Rx buffer wel overflowen. Daarom is het belangrijk dat we er op letten dat we de Rx buffer altijd zo snel mogelijk weer uitlezen om dat te voorkomen.

De seriele communicatie kan opgezet worden met de functie Serial.begin(baudrate). Deze functie moet eenmalig aanworden geroepen. Je kan de functie vaker aanroepen om bijvoorbeeld verschillende baudrates te proberen. We kunnen optioneel een extra argument meegeven aan deze functie om het aantal databits en de pariteitbits te gebruiken.

We kunnen behalve met acht bits ook met vijf, zes of zeven bits werken. We kunnen ook een tweede stopbit toevoegen. Pariteitbit is een extra bit die we kunnen gebruiken voor error detectie. De pariteitbit kunnen we op even of oneven zetten. Deze pariteitbit werkt als volgt als we hem op oneven zetten: Deze wordt wel of niet geset door de zender. De bit wordt gezet als het aantal databits in de byte oneven is en de bit wordt dus niet gezet als het aantal data bits even is. Als de pariteitbit op even is geze, is het natuurlijk net andersom.

De ontvanger leest deze bit uit om te controlleren of dit klopt met de databits. Deze vorm van error detectie werkt goed als er en bit verandert. Als er twee bits tegelijk corruperen tijdens data overdracht dan werkt dit niet. De praktijk heeft echter aangetoond dat deze methode goed genoeg

werkt. Voor onze hobby projecten maakt het geen drol of we hem wel of niet gebruiken. Het is wel belangrijk dat beide apparaten exact hetzelfde zijn geconfigureerd.

De syntax van deze configuratie ziet er als volgt uit.

```
Serial.begin(baudrate, SERIAL_DPS);
```

Hierbij is de D, het aantal databits (5,6,7,8). De P is pariteitbit en kan zijn: N voor neutraal, E voor even of O voor odd (oneven). De S is het aantal stop bits. Deze is 1 of 2. Dus het kan er zo uit zien:

```
Serial.begin(9600, SERIAL_6O2); // 9600BPS, 6 databits, odd pariteit bit, 2 stop bits
Serial.begin(38400, SERIAL_8N1);// 38400BPS, 8 databits, geen pariteit bit, 1 stop bit
```

Normaliter maken we geen gebruik van pariteitsbits en gebruiken we een stop bit met acht data bits. Dat is standaard.
Het vesturen van data kunnen we doen met een drietal functies.

```
Serial.print();
Serial.println();
Serial.write();
```

De print en de println functies zijn nagenoeg identiek. Alleen de println voegt nog een newline \n en een carriage return \r achter het bericht. Dit zorgt er voor dat regels tekst onder elkaar komen te staan.

De print functies accepteren bijna elke datatype en deze formatteert ze naar iets wat wij kunnen lezen. De write functie accepteert alleen integers en characters en stuurt de waarde van die variabele op. De write functie neemt ook maar een byte als argument. Als je probeert om een 16 bit integer mee te geven, dan worden alleen de achterste 8 bits, de least significant bits, gestuurd.
Om het verschil tussen write en print te illustreren:

```
Serial.print(1); // stuurt '1' in ascii format. Dus 49 deciamaal
Serial.write(1); // stuurt 1 decimaal
```

De print en println komen voor alle verscheidene datatypes met extra argumenten om deze te formateren. Van de Arduino website heb ik deze voorbeelden ontleend:

```
Serial.print(78, BIN);    //prints "1001110"
Serial.print(78, OCT);    //prints "116"
Serial.print(78, DEC);    //prints "78"
Serial.print(78, HEX);    //prints "4E"
Serial.print(1.23456, 0); //prints "1"
Serial.print(1.23456, 2); //prints "1.23"
Serial.print(1.23456, 4); //prints "1.2346"
```

De print functies gebruiken onderhuids weer de write functie om de bytes in de buffer te scheppen.

Het inlezen van data uit de buffer doen we met Serial.read(). En met Serial.available() kijken we of we er data is binnengekomen. De Serial.available() functie retourneert het aantal bytes in de Rx buffer die we kunnen uitlezen. Het inlezen en verwerken van seriele data kan soms nog best ingewikkeld worden. Mogelijke problemen zijn data transmissies met variabele data grootte of data is geformateerd in ASCII in plaats van decimale getallen (De zend arduino doet print(1) ipv write(1)). De seriele monitor van Arduino formateert getallen ook om naar de ASCII standaard.

Stel nu dat we via de monitor een waarde willen opgeven tussen de en 0 en 999. Als we 255 versturen, versturen we eigenlijk ‘2’, ‘5’ en ‘5’. We moeten dan drie bytes inlezen. En als we 0 sturen of eigenlijk ‘0’ moeten we maar een byte inlezen. ‘0’ is 48 decimaal. Dus van de ontvangen byte moeten we nog 48 of ‘0’ aftrekken om de correcte decimale waarde er uit te halen.

De code om zo een getal in te lezen, kan er zo uit zien:

```
int val;
void setup() {
    Serial.begin(115200);
}
void loop() {
    if(Serial.available()) {
        delay(3); // quick en dirty manier
        byte byteAmount = Serial.available(); // aantal ontvangen bytes
        val = 0;

        switch(byteAmount) {
        case 3: val += (Serial.read() - '0') * 100;
        case 2: val += (Serial.read() - '0') * 10;
        case 1: val += (Serial.read() - '0'); // * 1 is niet nodig
            break;
        }
        Serial.print("value = "); Serial.println(val);
    }
}
```

De werking is vrij simpel. De if-statement icm serial.available kijkt of er al een of meerdere bytes zijn ontvangen. Als dat het geval is, gebruik ik een dirty delay om te wachten op eventuele vervolg bytes. Dit kan je ook doen met millis(), maar voor het gemak doe ik even een delay gebruiken. Na de 3 millisecondes, gebruik ik opnieuw Serial.available om te kijken of er nog bytes zijn bijgekomen. Deze waarde sla ik op in ‘byteAmount’. Ik reset de waarde val en ik voer byteAmount aan een switch-case. Afhankelijk of er nu een, twee of drie bytes zijn ontvangen, wordt er een case uitgevoerd. Als we drie bytes hebben ontvangen, weten we dat het ontvangen getal groter is dan 100. Dan springen we naar case 3. In deze case wordt de ontvangen waarde opgehaald met Serial.read() daarvan trekken we ‘0’ of 48 van af, we vermenigvuldigen het resultaat met 100 en tellen dit op bij ‘val’.

Omdat er geen break statement staat, vallen we door naar case 2. Nu doen we hetzelfde truukje met de tientallen. We trekken weer ‘0’ van de ontvangen waarde af, vermenigvuldigen het resultaat met 10 en tellen dit op bij ‘val’. En als laatste doen we dit riedeltje nogmaals in case 1 waarna we de waarde op het scherm printen voor een visuele feedback. Dit programma kan je direct in de Arduino IDE zetten en uploaden naar een board om te testen.

Je kan er ook voor kiezen om te wachten totdat je een X aantal bytes heb ontvangen voordat je ze uit de buffer gaat scheppen.

```
if(Serial.available() >= 5) { // als we 5 bytes hebben ontvangen
    array[0] = Serial.read();
    array[1] = Serial.read();
    array[2] = Serial.read();
    array[3] = Serial.read();
    array[4] = Serial.read();
}
```

Je kan ook aan het begin van je data pakket opgeven hoeveel bytes je gaat sturen als je meerdere arduino's met elkaar wilt laten communiceren. Om te verzenden doe je dan:

```
Serial.write(5);
Serial.print("hello");
```

En om te in te lezen, hebben we iets beters nodig. Ik wil nu een arduino een tekst laten inlezen en tonen op een lcd scherm. Dat zou ik met deze code doen:

```
void readSerialBus() {
    static bool isReceiving = false;
    static byte messageSize = 0;

    if(Serial.available() { // als er tenminste 1 byte is te lezen

        if(isReceiving == false) {
            messageSize = Serial.read();
            isReceiving = true;
        }
        else {
            if(Serial.available() >= messageSize) {
                lcd.setCur(1,1);               // set cursor positie op lcd scherm
                for(byte i = 0; i < messageSize; i++) {
                    array[i] = Serial.read();   // store bytes in array
                    lcd.print(array[i]);      // echo byte op lcd scherm
                }
                isReceiving = false; // voor het volgende bericht
            }
        }
    }
}
```

Bij de eerste byte die we inlezen, is de boolean 'isReceiving' nog false. Dan lezen we de byte waarde in en gebruiken deze voor 'messageSize'. We zetten 'isReceiving' op true om de rest van het bericht in te lezen.

Als in de else body, serial.Available() dezelfde waarde krijgt als messageSize dan wordt de rest van de functie uitgevoerd. De cursor op het lcd scherm wordt gezet. En de bytes worden door middel van een for-loop ingelezen en 1 voor 1 naar het lcd scherm gestuurd. Wanneer alle bytes ontvangen zijn, wordt 'isReceiving' weer op false gezet voor het volgende bericht.

Het is meestal wenselijk om voor jezelf een bepaald protocol op te zetten tussen twee apparaten. In 1 byte passen 256 verschilldende combinaties voor instructies. Dan kunnen we deze ook nog onderverdelen in sub-instructies of vervolg bytes inbouwen. Kijk bijvoorbeeld maar naar de DCC++ centrale, ook daarin staat een heel opgezet serieel protocol.

Ik vind het zelf meestal handig om een enum met commando's te combineren met een switch-case. Ik laat een voorbeeld zien van een dcc centrale. Ik begin met drie instructies; setSpeed, setHeadLight en setFunction. Elke instructie wordt gevolg door een adres en afhankelijk van welke instructie er verstuurd is, volgen er een of meerdere vervolg bytes met informatie voor de trein.

Ik vind het zelf fijn om de cases van een switch-case klein te houden om zo een beetje overzichtelijk te houden. Deze constructie is ook erg modulair, het is makkelijk om nieuwe instructies toe te voegen.

```
enum serialCommands {
    setSpeed = '0',
    setHeadlight,
    setFunction,
};

void readSerialBus() {
    static bool isReceiving = false;
    static byte command;

    if(Serial.available()) {
        byte serialByte = Serial.read();

        if(!isReceiving) {
            isReceiving = true;
            command = serialByte;
        }
        else {
            switch(command){
                default: Serial.println("unknown command");      isReceiving = false; break;
                case setSpeed:     if(setSpeedF(serialByte))     isReceiving = false; break;
                case setHeadlight: if(setHeadlightF(serialByte)) isReceiving = false; break;
                case setFunction:  if(setFunctionF(serialByte))  isReceiving = false; break;
            }
        }
    }
}
```

Elke byte wordt meteen afgehandeld, er wordt niet eerst gewacht totdat er een x aantal bytes zijn ontvangen. De eerste byte wordt opgeslagen in 'command' en de isReceiving flag wordt weer op true geset. Wanneer er een tweede byte komt, dan wordt de switch-case uitgevoerd en afhankelijk van welk commando is ontvangen wordt een van de drie cases uitgevoerd.

Elke case roept een functie aan en paast de serialByte als argument mee. In deze functies vindt de verdere afhandeling plaats van de vervolg bytes. De functies worden aangeroepen vanuit een if-statement. Deze functies retourneren namelijk 'true' wanneer alle bytes zijn ontvangen. Wanneer dit gebeurt, wordt de isReceiving flag weer op false geset en kan de volgende instructie worden ontvangen.

Dan nu deze vervolg functies. Deze functies hebben dezelfde naam als een van de serialCommands + de hoofdletter F. De F van Functie is toegevoegd om onderscheid te maken tussen de constantes van de enum en de functies. Zo hebben ze dezelfde naam en toch ook weer niet.

De functie om een trein functie te zetten kan er zo uitzien:

```
static bool setFunctionF(byte serialByte) {
    static byte followByte = 0;

    switch(followByte++) {
        default followByte = 1;
        case 0: address = serialByte; return false;
        case 1: function = serialByte return false;
        case 2: state = serialByte;   return true;
    }
}
```

De ‘followByte’ variabele wordt gebruikt om er voor te zorgen dat de juiste case wordt uitgevoerd. Hij wordt ook elke keer opgehoogd. Bij de eerste keer dat deze functie vanuit readSerialBus wordt aangeroepen, is followByte 0. Daarom wordt case 0 als eerst uitgevoerd. In deze case wordt de waarde van de serialByte opgeslagen in een globale variabele genaamd ‘address’. Door de ++ operator wordt de volgende byte afgehandeld in case 1 en de laatste byte wordt afgehandeld in case 2. In case 2 wordt tevens ook true geretourneerd. De functie readSerialBus() weet dan dat alle bytes binnen zijn en set de isReceiving flag weer op 0 voor de volgende instructie.

Ik gebruik voor de followByte een apart truukje. Nadat case 2 is uitgevoerd, heeft followByte de waarde 3. Er is geen case 3 maar er is wel een default label. Als er nu een nieuwe instructie komt om een trein functie te zetten dan wordt de default label uitgevoerd. De followByte wordt dan op 1 gezet en door de afwezigheid van de break valt de code door naar case 1 waar het adres weer wordt ingelezen. Doordat we achter de default label followByte weer 1 gemaakt hebben, zal de volgende byte weer in case 1 afgehandeld worden.

Als we nu functie 8 laag willen maken van loc 36 dan moeten we naar deze controller de volgende byte reeks sturen: ‘2’ (instructie ‘setFunction’), 36 (address), 8 (f8), 0(uit).

Deze structuur stelt ons in staat om op een geordende wijze een varierend aantal bytes af te handelen. Deze structuur is ook makkelijk uitbreidbaar. Als je dus ooit zelf een DCC centrale wilt programmeren dan is dit een handige manier om je communicatie op te zetten met bijvoorbeeld een handregelaar.

Zoals je nu misschien wel snapt, zijn er nog meer wegen die naar Rome leiden. Mijn voorbeelden zijn niet de enige voorbeelden. Je kan er gebruik van maken maar je kan ook proberen zelf iets te verzinnen.

## §2.2 I2C

I2C spreken we uit als ‘ie kwadraat cee’. I2C staat eigenlijk voor IIC hetgeen ‘inter integrated circuit’ betekent. I2C is een bi-drectionele communicatie bus waarmee we met slechts twee IO meer dan een stukje rand appartuur kunnen bedienen. I2C is bedoeld om op een printplaat zelf gebruikt te worden en niet voor langere afstanden. Als je randapparatuur wilt bekabelen, let er dan op dat je draden niet te lang worden. Hou ongeveer een lengte aan van 50cm. De bi-directionale data lijn heet de SDA en de klok lijn heet de SCL. De master bedient de kloklijn en zowel master als slave kunnen de datalijn bedienen.

Voor I2C zijn er menig apparaten ontwikkeld; temperatuursensors, real time clock modules, IO extenders, pwm drivers, DACs en ADCs om er een paar te noemen. Arduino heeft voor het gebruik van de I2C bus de ‘Wire’ library geschreven. Om de library te gebruiken moet je

```
#include <Wire.h>
```

Tikken. De bus wordt geinitialiseerd met

```
Wire.begin();
```

Hoe je een apparaat moet aansturen, verschilt per apparaat. Doorgaans stuur je minimaal twee bytes. De eerste byte wordt veelal gebruikt om een bepaald intern register te selecteren. De waarde van de tweede byte wordt dan in dit register weggeschreven. De counter die de registers bijhoudt, incrementeert na een schrijfactie. Dat wil zeggen dat eventuele vervolg bytes in opvolgende registers worden  weggeschreven.

Een I2C data-overdracht bestaat doorgaans uit een startbit gevolgd door een adres, een of meer vervolg bytes en een stop bit. In code ziet dit er zo uit:

```
Wire.beginTransmission(0x28);
Wire.write(0x0E);    // register
Wire.write(someVal); // waarde
Wire.endTransmission();
```

I2C heeft een ingebouwde error detectie methode. Een slave bevestigt elke ontvangen byte met een acknowledge bit of ACK. Het tegenovergestelde noemen we een NACK bit.

Wire.endTransmission() stuurt de stop bit en retourneert de status van een transmissie. 0 betekent dat er geen problemen waren, 1 betekent dat de verstuur buffer is overflow’ed, 2 betekent dat er een NACK is ontvangen na de adres byte, 3 betekent dat er een NACK is ontvangen na een data byte en 4 is een overige error.

I2C kan ook worden gebruikt om registers uit te lezen. Dit doe je door eerst via boven genoemde methode een register te selecteren om vervolgens met de functie Wire.requestFrom(adres, aantalBytes) bytes op te vragen van een slave apparaat.

```
Wire.beginTransmission(0x28);
Wire.write(0x0E);    // register
Wire.endTransmission();

Wire.requestFrom(0x28, 2); // vraag om 2 bytes

byte x = Wire.read(); // haal byte 1 uit buffer
byte y = Wire.read(); // haal byte 2 uit buffer
```

De request functie vraagt de twee bytes op en slaat ze op in de ontvangst buffer. Met Wire.read() kunnen de bytes uit de buffer gelepeld worden. Dit werkt hetzelfde als bij de seriele communicatie. Je kan ook gebruik maken van Wire.available() maar dit heb je doorgaans niet nodig.

Libraries die I2C apparaten kunnen aansturen, gebruiken deze functies onderhuids. Als je dus een dergelijke library gebruik, hoef je niet zelf deze functies aan te roepen.

Behalve als een I2C master kan een Arduino ook als een I2C slave fungeren. Een I2C master heeft standaard adres 0, als je een arduino als slave instelt, is het nodig om een adres tussen de ronde haken van Wire.begin() te plaatsen.

Een I2C slave gebruikt zogenaamde events om I2C data af te handelen. In de void setup kan je twee verschillende events configureren. Het eerste event is de 'onReceive()' event. De syntax ziet er zo uit:

```
Wire.onReceive(receiveEvent);
```

Deze onReceive functie koppelt het event aan een functie naar keuze, in dit geval is dat dus de functie:

```
void receiveEvent(int byteAantal)
```

In deze functie kan je met Wire.read() de ontvangen bytes inlezen.

Het kan ook voorkomen dat de master Arduino bytes wilt opvragen van een Arduino slave. Dit doen we op dezelfde methode alleen gebruiken we de functie:

```
Wire.onRequest(requestEvent)
```

Nu is de functie requestEvent gekoppeld aan het event. Als deze event plaatsvindt dan wilt de master antwoord krijgen van de slave. We moeten de master dit antwoord verschaffen. Dit kunnen we doen door Wire.write() te gebruiken. Het aantal bytes wat we sturen moet kloppen met het aantal wat de master verwacht. We kunnen in dit scenario helaas niet ruiken hoeveel bytes de master verwacht. Het idee is namelijk dat je zowel master als slave programmeert, je moet dan ook zelf bijhouden en weten wat je doet.

## §2.3 SPI

SPI staat voor Serial Peripheral Interface (seriele randapparatuur interface) en is net als I2C een communicatie bus om met randappartuur te werken. SPI maakt geen gebruik van adressen, maar van extra IO lijnen om met een slaves te communiceren. De SPI bestaat in plaats van twee uit driebus lijnen; MOSI _> master out slave in, MISO -> master in slave out en de klok lijn -> SCK. Verder is er een slave select lijn aanwezig voor elke aanwezige slave. Op de MOSI lijn kan de master data uitklokken naar de slaves. Via de MISO lijn kunnen de slaves data sturen op het klok signaal van de master.

Om de SPI bus te gebruiken moet je de library includen:

```
#include <SPI.h>
```

De SPI bus wordt geinitialiseerd met:

```
SPI.begin();
```

Om bytes heen en weer te sturen naar of van een slave, moet eerste de slave select lijn laag worden gemaakt met een digitalWrite instructie. Daarna kan je met de functie:

```
SPI.transfer();
```

Zowel data versturen als inlezen. Als je tussen de ronde haken een byte als argument zet, dan wordt deze byte uitgeklokt. Als je geen argument meegeeft, retourneert deze functie een waarde van de slave. Aanvullende bestaat er ook de functie:

```
SPI.transfer16();
```

Voor 16 bit waardes. Deze werkt hetzelfde als de eerste.

Door de slave select lijnen zou de master dezelfde byte naar meer dan een slave tegelijk kunnen uitklokken.

SPI heeft nog meer aanvullende functies voor optionele bus intellingen. SPI is namelijk vele malen sneller dan I2C en heeft ook meer functionaliteiten. Er kunnen meerde bussnelheden worden ingesteld en er zijn meerdere modi mogelijk. Ik ben hier zelf niet echt bekend mee en refereer daarom verder naar de Arduino website over SPI: https://www.arduino.cc/en/Reference/SPI. De basis werking is in ieder geval behandeld.

Voor zowel SPI als I2C zijn er in de Arduino IDE zat voorbeelden te vinden in de voorbeelden map.

## §2.4 rand apparatuur.

In deze paragraaf wil ik slechts een lijst neerzetten met randapparatuur om in een indicatie te geven van je wat je allemaal wel niet kan doen met een Arduino.

- Bluetooth module voor draadloze communicatie met je telefoon of een andere arduino
- Ethernet of wifi modules voor netwerk en internet verbindingen
- Temperatuur sensors
- Vloeistof temperatuur sensors
- Luchtvochtigheid sensors
- Bodemvochtigheid sensors
- Magneet sensors (hall sensors)
- RFID of NFC modules
- Gyroscopen en accelerometers
- Lcd schermen
- Grafische displays
- Laser sensors
- Infarood sensors en zenders (tv afstandsbediening)
- Alcohol sensors
- Gas sensors (CO2)
- Druk sensors
- Luchtdruk sensors
- Gewicht sensor
- Kleuren sensor
- Camera modules
- Mp3 en andere geluidsmodules
- Ultrasone afstandssensor
- Lichtintensiteit sensor
- Waterstroom sensor
- Stroommeet sensor
- Radar sensor
- Hartslag sensor
- Peltier element (voor koeling)
- Spraakherkenning module
- Beweging melders
- IO extenders (SPI, I2C)
- Bewegingssensor
- Led strips
- Relays voor 230V apparaten te schakelen
- Motoren (servo, DC, stappenmotors, brushless)
- Ledmatrix
- Solenoide
- 2.4GHz ontvanger

De mogelijkheden zijn eindeloos.