

# Arduino Programming

## Theorie Sessie 4



Pointers in C/C++  
Pointers en Functies  
Pointers en Arrays  
Pointer bewerkingen  
Pointer bewerkingen en Arrays



# Pointers in C/C++

## Wat is een pointer?

Een pointer is een speciale variable die geen waardes maar een geheugen adres bevat.

Dit is typisch het startadres van waar de inhoud van een andere variabele zich in het geheugen bevindt

# Pointers in C/C++

## Wat is een pointer?

Pointers wijzen naar andere variabelen en deze variabelen hebben uiteraard een data type

Pointers zélf hebben niet altijd hetzelfde type, er bestaan namelijk meerdere types pointers

en dit ondanks dat alle pointers dezelfde soort data stockeren namelijk één geheugenadres.



# Pointers in C/C++

**Wat is een pointer?**

*pointers moeten  
hetzelfde type hebben dan  
de variabelen waar ze naar verwijzen*



# Pointers in C/C++

## Wat is een pointer?

Pointers worden gedeclareerd door het plaatsen van een asterix ( \* ) voor het data type waar ze naar verwijzen:

```
int *voorbeeldpointer;
```

```
int *  voorbeeldpointer;
```

```
int*  voorbeeldpointer;
```





# Pointers in C/C++

## Wat is een pointer?

Pointers worden gedeclareerd door het plaatsen van een asterix ( \* ) voor het data type waar ze naar verwijzen:

**int \*voorbeeldpointer;**      → *meest voorkomende notatie*

int \* voorbeeldpointer;

int\* voorbeeldpointer;



# Pointers in C/C++

## Wat is een pointer?

Enkele voorbeelden:

***char \*c;***  
*char\** c;

is een pointer met naam “c” die wijst  
naar het startadres  
van een variabele van het type char

Een char omvat één byte in het geheugen  
De pointer wijst dus naar deze byte.



# Pointers in C/C++

## Wat is een pointer?

*int \*t;*

is een pointer met naam “t” die wijst  
naar het startadres  
van een variabele van het type int

Een int omvat twee bytes in het geheugen,  
de pointer wijst hier naar het éérste van de twee



# Pointers in C/C++

## Wat is een pointer?

*float \*s;*

is een pointer met naam “s” die wijst  
naar het startadres  
van een variabele van het type float

Een float omvat vier bytes in het geheugen,  
de pointer wijst hier naar het éérste van de vier



# Pointers in C/C++

## Hoe werken pointers ?

Als een pointer het adres bevat  
van waar een andere variabele zich bevindt

dan is de vraag:

hoe wordt de link gemaakt  
tussen zo'n pointer  
en een willekeurige variabele ?



# Pointers in C/C++

## Hoe werken pointers ?

Dit is het gemakkelijkst te verduidelijken  
aan de hand van een praktisch voorbeeld:

Stel de variabele “c” van het type “char”  
bevat als waarde de letter “a”.

De declaratie hiervan is:

```
char c = “a”;
```



# Pointers in C/C++

## Hoe werken pointers ?

	<u>Adres</u>	<u>Data</u>
	0x1152	.....
char c = "a";	0x1153	a
	0x1154	.....
	0x1155	.....
	0x1156	.....

# Pointers in C/C++

## Hoe werken pointers ?

we hebben vervolgens een pointer “p”  
die verwijst naar het type char.

De declaratie hiervan is:

```
char *p;
```





# Pointers in C/C++

## Hoe werken pointers ?

`char *p;`

<u>Adres</u>	<u>Data</u>
0x1382	

`char c = "a";`

<u>Adres</u>	<u>Data</u>
0x1152	.....
0x1153	a
0x1154	.....
0x1155	.....
0x1156	.....



# Pointers in C/C++

## Hoe werken pointers ?

dan kunnen we zorgen dat de pointer “p” wijst naar het startadres van de char variabele “c” op de volgende manier:

**p = &c;**

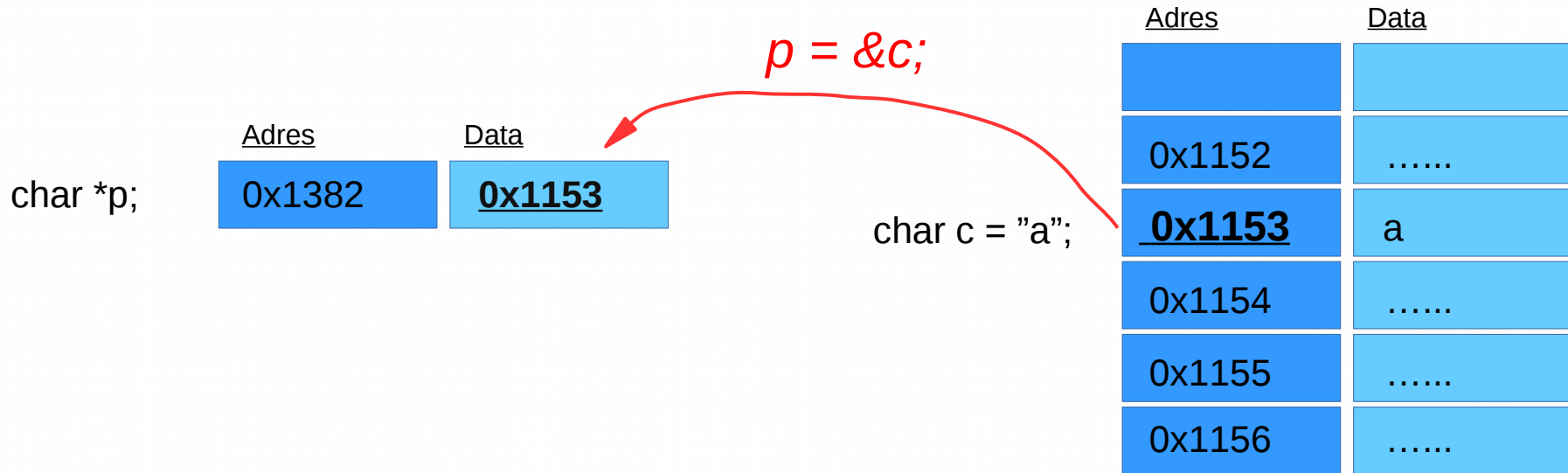
de ampersant moet men hierbij lezen als:

*& = “het geheugen-adres van”*



# Pointers in C/C++

## Hoe werken pointers ?



# Pointers in C/C++

## Hoe werken pointers ?

Correctie:

char \*p;

Adres	Data
0x1382	<u>0x11</u>
0x1383	<u>0x53</u>

$p = \&c;$

char c = "a";

Adres	Data
0x1152	.....
<u>0x1153</u>	a
0x1154	.....
0x1155	.....
0x1156	.....



één byte per geheugencel

8-bits microcontroller



# Pointers in C/C++

## Pointers als waarde-referentie

Pointers kunnen gebruikt worden als referentie naar de inhoud van een andere variabele (via het omwegje van het adres)

een pointer als waarde-referentie duid men aan door een asterix voor de pointer te plaatsen

bvb:

**\*p**



# Pointers in C/C++

## Pointers als waarde-referentie

```
char c='a'; → c staat op adres 0x1153  
char *p;  
p = &c;
```

*p*

*\*p*





# Pointers in C/C++

## Pointers als waarde-referentie

```
char c='a'; → c staat op adres 0x1153  
char *p;  
p = &c;
```

***p → is nu 0x1153***

***\*p***



# Pointers in C/C++

## Pointers als waarde-referentie

```
char c='a'; → c staat op adres 0x1153  
char *p;  
p = &c;
```

*p → is nu 0x1153*

*\*p → is nu 'a'*



# Pointers in C/C++

## Pointers als waarde-referentie

```
char c='a'; → c staat op adres 0x1153  
char *p;  
p = &c;
```

```
... Serial.println(p) → "0x1153"
```

```
... Serial.println(*p) → "c"
```



# Pointers in C/C++

## Pointers als waarde-referentie

```
char c='a'; → c staat op adres 0x1153  
char *p;  
p = &c;
```

*dit geeft een foutmelding omdat "Serial.println"  
last heeft met het printen van pointers*

```
... Serial.println(p) → "0x1153"
```

```
... Serial.println(*p) → "c"
```

# Pointers in C/C++

## Pointers als waarde-referentie

*Willen we het adres van een pointer  
weergeven via `Serial.println`*

*dan moeten we dit doen via een klein omwegje  
door éérst het type te veranderen naar een `int`*

`Serial.println(int(p))` → `"0x1153"`



# Pointers in C/C++

## Pointers als waarde-referentie

char \*p;

Adres	Data
0x1382	<u>0x11</u>
0x1383	<u>0x53</u>

$p = \&c;$

char c = "a";

Adres	Data
0x1152	.....
<u>0x1153</u>	a
0x1154	.....
0x1155	.....
0x1156	.....

*p*

*\*p*





# Pointers in C/C++

## Pointers als waarde-referentie

Zowel 'p' als '\*p' noemen we pointers

Het sterretje duid énkél aan  
over welke verwijzing het gaat:

- naar het geheugen adres
- of naar de *inhoud* van dat adres



# Pointers en Functies

## Nut van pointers

...wat kunnen we nu met zo'n pointer doen?

Een van de belangrijke toepassingen van pointers is het doorgeven van variabelen aan functies.

Stel de volgende functie:

```
int verdubbel( int getal ) {  
    getal = getal * 2;  
    return getal;  
}
```



# Pointers en Functies

## Nut van pointers

Dan kan men niet zomaar deze functie op de volgende manier aanroepen:

```
int gt = 7;  
verdubbel(gt);
```

het mág wel, dit geeft géén foutmelding...

... maar na het aanroepen is  
gt nog steeds 7 en niet 14 !



# Pointers en Functies

## Nut van pointers

Bij het aanroepen van de functie wordt er immers een nieuwe variabele gemaakt

*( int getal )*

de bewerking gebeurt op deze nieuwe variabele  
deze wordt de retourwaarde van de functie

*return getal;*



# Pointers en Functies

## Nut van pointers

Willen we dat de waarde van de variabele waarmee we de functie aanroepen aangepast wordt dan moeten we dit doen als:

*gt = verdubbel(gt);*

ook hier echter wordt nog steeds een kopie gemaakt van gt in het geheugen bij het aanroepen van de functie



# Pointers en Functies

## Nut van pointers

Deze kopie is echter zinloos.  
We doen er niets mee.

Het was niet onze bedoeling dat we nog  
de originele waarde van 'gt' zouden behouden

Ons doel was gewoon  
de waarde van 'gt' verdubbelen.





# Pointers en Functies

## Nut van pointers

Stel dat het nu niet om zo'n functie zou gaan die één parameter neemt, maar 1000 parameters...

dan zou er van elke parameter in de functie een lokale kopie gemaakt worden die vervolgens geretourneerd moet worden.

Stel nu dat het gaat om float's...



# Pointers in C/C++

## Nut van pointers

één float neemt 4 bytes in het geheugen  
dus dat wil dit zeggen dat deze operatie

$$\underline{1000 \times 4 \text{ byte's} = 4 \text{ kilobyte}}$$

aan onnodige extra variabelen zal creëren

maar variabelen komen in de Arduino  
terecht in het SRAM geheugen.

en deze SRAM is slechts 2 kilobytes groot ...

# Pointers en Functies

## Nut van pointers

Onnuttige kopies van variabelen kunnen snel dus een probleem worden in de kleine geheugens van een microcontroller !

Hiernaast moet de CPU van de chip ook nog 'ns héél hard werken om al deze kopies te maken en ook nog eens om ze terug te wissen !

Dit kan in de uitvoering van een programma dus enorme vertragingen opleveren.



# Pointers en Functies

**Nut van pointers**

de oplossing ... ?

**Pointers !!**



# Pointers en Functies

## Nut van pointers

We nemen opnieuw onze voorbeeld functie doch met enkele aanpassingen:

```
void verdubbel( int *pntr ) {  
    *pntr = *pntr * 2;  
}
```

de functie neemt nu een pointer als argument  
m.a.w. het adres van een andere variabele

en de bewerking is nu op de **inhoud** van dit adres

# Pointers en Functies

## Nut van pointers

```
void verdubbel( int *pntr ) {  
    *pntr = *pntr * 2;  
}
```

```
void setup() {  
    int gt = 7;  
    verdubbel ( &gt );  
}
```





# Pointers en Functies

## Nut van pointers

```
void verdubbel( int *pntr ) {  
    *pntr = *pntr * 2;  
}
```

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....

# Pointers en Functies

## Nut van pointers

verdubbel (&gt) = 0x1153



```
void verdubbel( int *pntr ) {  
    *pntr = *pntr * 2;  
}
```

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....

# Pointers en Functies

## Nut van pointers

```
void verdubbel( 0x1153 ) {  
    *pntr = *pntr * 2;  
}
```

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....

# Pointers en Functies

## Nut van pointers

```
void verdubbel( 0x1153 ) {  
    *pntr = *pntr * 2;  
}
```



*pntr*

*\*pntr*

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....

# Pointers en Functies

## Nut van pointers

```
void verdubbel( 0x1153 ) {  
    *pntr = *pntr * 2;  
}
```

***pntr = 0x1153***

***\*pntr***

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....



# Pointers en Functies

## Nut van pointers

```
void verdubbel( 0x1153 ) {  
    *pntr = *pntr * 2;  
}
```

***pntr = 0x1153***

***\*pntr***

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....





# Pointers en Functies

## Nut van pointers

```
void verdubbel( 0x1153 ) {  
    *pntr = *pntr * 2;  
}
```

***pntr = 0x1153***

***\*pntr = 7***

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....



# Pointers en Functies

## Nut van pointers

```
void verdubbel( 0x1153 ) {  
    *pntr = *pntr * 2;  
}
```



$*pntr * 2 \rightarrow 7 * 2 = 14$

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....

# Pointers en Functies

## Nut van pointers

```
void verdubbel( 0x1153 ) {  
    *pntr = *pntr * 2;  
}
```



*\*pntr* \* 2 → 7 \* 2 = 14

*\*pntr* = 14

int gt = 7;

Adres	Data
0x1152	.....
0x1153	7
0x1154	.....
0x1155	.....
0x1156	.....

# Pointers en Functies

## Nut van pointers

```
void verdubbel( 0x1153 ) {  
    *pntr = *pntr * 2;  
}
```



**\*pntr** \* 2 → 7 \* 2 = 14

**\*pntr** = 14

int gt = 7;

Adres	Data
0x1152	.....
0x1153	14
0x1154	.....
0x1155	.....
0x1156	.....

# Pointers en Functies

## Nut van pointers

```
void verdubbel( int *pntr ) {  
    *pntr = *pntr * 2;  
}  
  
void setup() {  
    int gt = 7;  
    verdubbel ( &gt );  
    Serial.println(gt);    → 14  
}
```

er is géén kopie van gt gemaakt

# Pointers en Arrays





# Pointers en Arrays

## Arrays in het geheugen

Stel we hebben volgende array:

*byte b[4] = { 0,1,2,3 };*

Hoe ziet deze array er dan uit  
in het geheugen van de microcontroller ?



# Pointers en Arrays

## Arrays in het geheugen

*byte b[4] = { 0,1,2,3 };*

	<u>Adres</u>	<u>Data</u>
	0x101F	
<i>b[0] →</i>	0x1020	0
<i>b[1] →</i>	0x1021	1
<i>...</i>	0x1022	2
<i>b[3] →</i>	0x1022	3
	0x1023	

# Pointers en Arrays

## Arrays in het geheugen

We kunnen hier dan  
elk element apart aanspreken:

*byte elementje = b[2];*

... maar zo'n array in zijn geheel aanspreken  
in C/C++ is niet zoals in andere talen



# Pointers en Arrays

## Arrays in het geheugen

indien we nu het volgende proberen:

```
byte b[4] = { 0,1,2,3 };  
byte kopie[4];
```

```
kopie = b;
```

dan krijgen we een error !!

*“invalid array assignment”*



# Pointers en Arrays

## Arrays in het geheugen

in tegenstelling tot andere programmeer talen  
“bevat” de naam van de array niet “alle elementen”

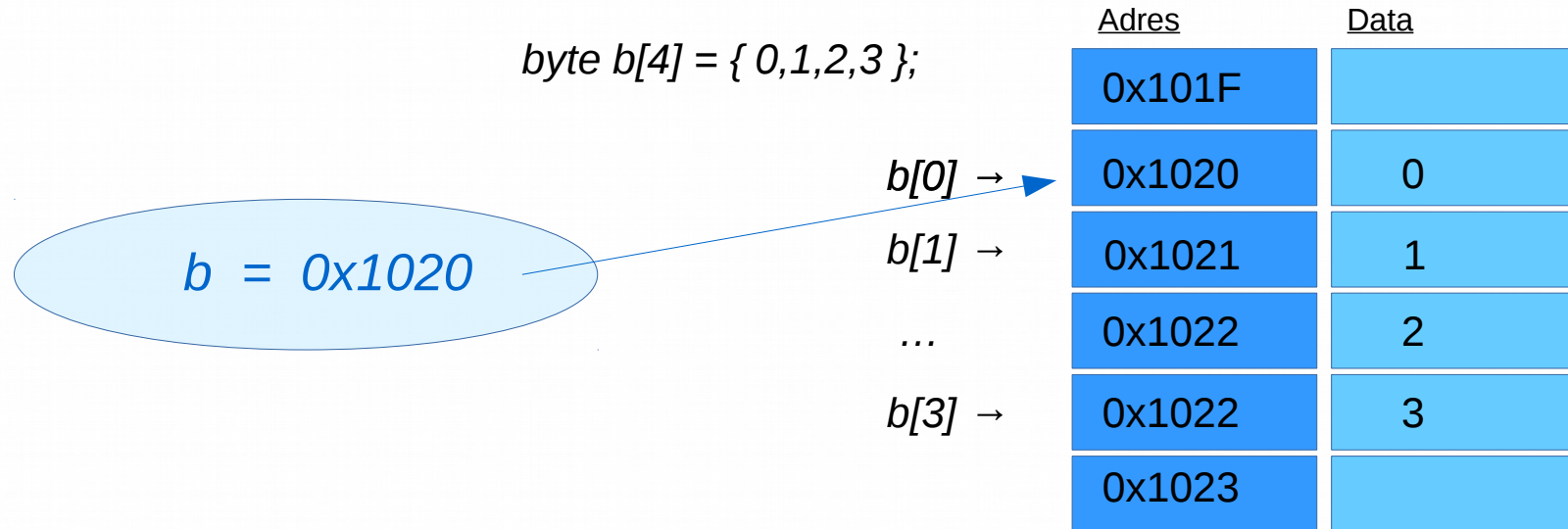
**in C/C++ en dus ook Arduino Code**  
**is de naam van een array een pointer**

de naam van een array bevat hierbij het adres  
van het éérste element van deze array



# Pointers en Arrays

## Arrays in het geheugen





# Pointers en Arrays

## Arrays in het geheugen

Variabele:

```
int a = 7;  
int *p;
```

```
p = &a;
```

```
Serial.println( *p) → 7
```

Array:

```
int a[ ] = { 1,2,3 };  
int *p;
```

```
p = a;
```

```
Serial.println ( *p) = 1
```



# Pointer bewerkingen



# Pointer bewerkingen

## **pointer++**

veronderstel even dat drie variabelen  
opeenvolgend in het geheugen staan

```
char a = "x";  
char b = "y";  
char c = "z";
```

en dat we een pointer hebben  
die naar het adres van de éérste variabele  
van deze drie variabelen verwijst

```
char *p = &a;
```



# Pointer bewerkingen

**pointer++**

char \*p;

<u>Adres</u>	<u>Data</u>
0x1382	<u>0x1153</u>

char a = "x";

char b = "y";

char c = "z";

<u>Adres</u>	<u>Data</u>
0x1152	.....
0x1153	x
0x1154	y
0x1155	z
0x1156	.....



# Pointer bewerkingen

## **pointer++**

dan is de inhoud van de pointer p:

```
Serial.println( int(p) ) → 0x1153;
```

als we nu de inhoud van deze pointer aanpassen,  
dan wijst de pointer ook in het geheugen  
naar een andere locatie

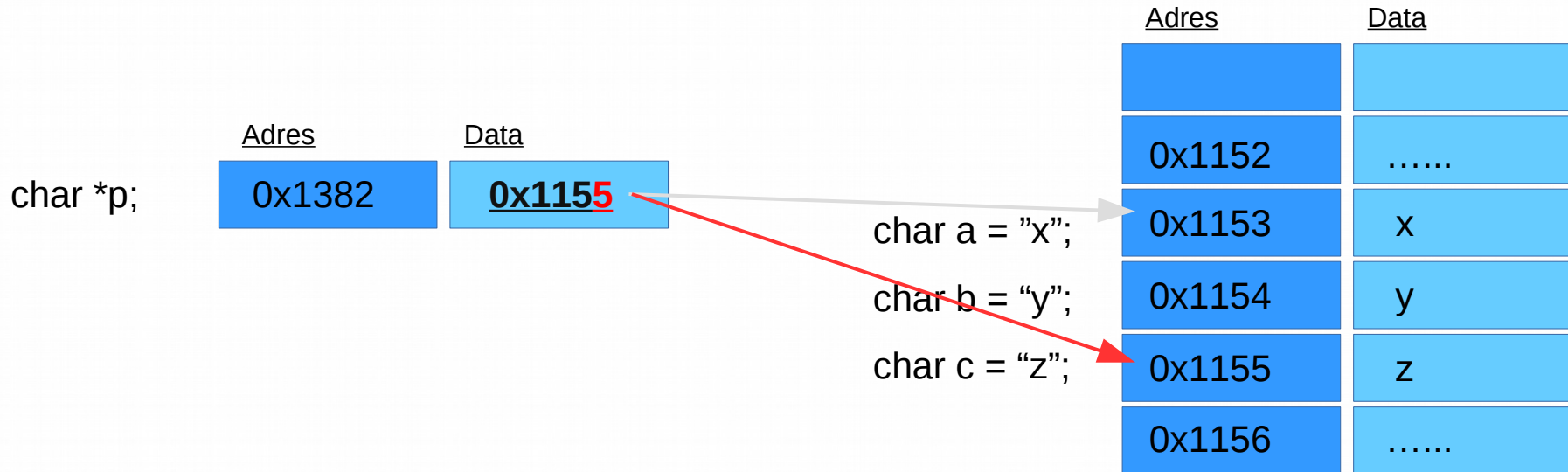
```
p = 0x1155;
```

```
Serial.println ( int(p) ) → 0x1155;
```



# Pointer bewerkingen

**pointer++**





# Pointer bewerkingen

## **pointer++**

Als we dan kijken naar de inhoud waar de pointer naar gaat refereren dan krijgen we dit:

```
p = &a;
```

```
Serial.println( int(p) ) → 0x1153
```

```
Serial.println( *p ) → "x"
```

```
p = 0x1155;
```

```
Serial.println( int(p) ) → 0x1155
```

```
Serial.println( *p ) → "z"
```

# Pointer bewerkingen

## **pointer++**

het veranderen van het adres van een pointer kan eveneens door het toepassen van wiskundige bewerkingen en/of bestaande operatoren:

***$p = p + 2;$***

***$p = p - 7;$***

***$p++;$***

***$p-- ;$***

# Pointer bewerkingen

## **pointer++**

de bestaande operatoren of bewerkingen werken evenwel niet hoe men in éérste instantie zou verwachten.

Bij dergelijke operatoren is het van belang wat het pointer type is

We nemen even de incrementor operator om de exacte werking te illustreren:



# Pointer bewerkingen

## pointer++

wanneer we een pointer met een incrementer ophogen dan zal het adres in de pointer niet met één verhoogd worden

maar met het aantal bytes dat een variable van het type waar de pointer naar wijst groot is

Enkele voorbeelden...



# Pointer bewerkingen

## **pointer++**

een 'char' is één byte groot:

```
char a;  
char *p = &a;  
Serial.println(int(p)); → 0x1153;  
p++;  
Serial.println(int(p)); → 0x1154;
```

*p++ zal de pointer hier met  
één geheugenplaats ophogen*



# Pointer bewerkingen

## **pointer++**

een 'int' is twee bytes groot:

```
int i;
```

```
int *p = &i;
```

```
Serial.println(int(p)); → 0x1160;
```

```
p++;
```

```
Serial.println(int(p)); → 0x1162;
```

*p++ zal hier de pointer met  
twee geheugenplaatsen ophogen*





# Pointer bewerkingen

## **pointer++**

een 'float' is vier bytes groot:

```
float f;
```

```
float *p = &f;
```

```
Serial.println(int(p)); → 0x1184;
```

```
p++;
```

```
Serial.println(int(p)); → 0x1188;
```

*p++ zal de pointer hier met  
vier geheugenplaatsen ophogen*



# Pointer bewerkingen

## **pointer++**

het voordeel hiervan is dat wanneer  
meerdere variabelen van hetzelfde type  
na elkaar in het geheugen staan...  
(denk o.a. aan arrays...)

...bewerkingen/operators op een pointer  
die pointer direct zullen doen verwijzen  
naar het startadres van een andere variabele

# Pointer bewerkingen

**pointer++**

int \*p;

Adres	Data
0x1382	<u>0x1160</u>

int i = 32000;

int j = 43231;

Adres	Data
0x115F	
0x1160	32000
0x1161	
0x1162	43231
0x1163	



# Pointer bewerkingen

**pointer++**

int \*p;

Adres	Data
0x1382	<u>0x1160</u>

***p++;***

int i = 32000;

int j = 43231;

Adres	Data
0x115F	
0x1160	32000
0x1161	
0x1162	43231
0x1163	



# Pointer bewerkingen

**OPGEPAST !!!!!**

$$p++ \neq *p++$$

$p++$  → verhoogt het adres in de pointer

$*p++$  → verhoogt de inhoud van  
wat er op het adres staat  
waar de pointer naar verwijst

# Pointer bewerkingen en Arrays





# Pointer bewerkingen en Arrays

## Elementen loop

gezien de naam van een array een pointer is  
naar het eerste element van de array

en alle elementen van een array elkaar  
logisch opvolgen in het geheugen

laat dit ons toe om bewerkingen op pointers  
te gebruiken om de verschillende elementen  
van die array te bereiken

# Pointer bewerkingen en Arrays

## Elementen loop

We illustreren dit even met een loop doorheen de elementen van een array:

*byte m[ 5 ] = { 0, 1, 2, 3, 4 };*

“m” is dus een pointer naar het adres van het eerste element:

*m → 0x3300;*

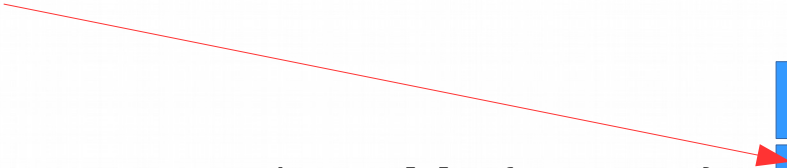


# Pointer bewerkingen en Arrays

## Elementen loop

m = 0x3300

byte m[5] = { 0,1,2,3,4 };



Adres	Data
0x3300	0
0x3301	1
0x3302	2
0x3303	3
0x3304	4

# Pointer bewerkingen en Arrays

## Elementen loop

`m = 0x3300`

`*m = 0`

**`byte *p = m;`**

`byte m[5] = { 0,1,2,3,4 };`

Adres	Data
0x3300	0
0x3301	1
0x3302	2
0x3303	3
0x3304	4



# Pointer bewerkingen en Arrays

## Elementen loop

`m = 0x3300`

`*m = 0`

**`byte *p = m;`**

`p = 0x3300`

`*p = 0`

`byte m[5] = { 0,1,2,3,4 };`

Adres	Data
0x3300	0
0x3301	1
0x3302	2
0x3303	3
0x3304	4



# Pointer bewerkingen en Arrays

## Elementen loop

```
byte *p = m;
```

```
p = 0x3300
```

```
*p = 0
```

```
byte m[5] = { 0,1,2,3,4 };
```

```
p++;
```

```
p = ?
```

```
*p = ?
```

Adres

Data

0x3300	0
0x3301	1
0x3302	2
0x3303	3
0x3304	4





# Pointer bewerkingen en Arrays

## Elementen loop

```
byte *p = m;
```

```
p = 0x3300
```

```
*p = 0
```

```
byte m[5] = { 0,1,2,3,4 };
```

```
p++;
```

```
p = 0x3301
```

```
*p = 1
```

Adres	Data
0x3300	0
0x3301	1
0x3302	2
0x3303	3
0x3304	4



# Pointer bewerkingen en Arrays

## Elementen loop

*p++;*

*p = 0x3301*

*\*p = 1*

byte m[5] = { 0,1,2,3,4 };

*p++;*

Adres	Data
0x3300	0
0x3301	1
0x3302	2
0x3303	3
0x3304	4



# Pointer bewerkingen en Arrays

## Elementen loop

*p*++;

*p* = 0x3301

\**p* = 1

byte m[5] = { 0,1,2,3,4 };

*p*++;

*p* = 0x3302

\**p* = 2

Adres	Data
0x3300	0
0x3301	1
0x3302	2
0x3303	3
0x3304	4



# Pointer bewerkingen en Arrays

## Elementen loop

in code wordt dit dan bvb:

```
byte m[5] = { 0,1,2,3,4 };  
byte *p = m;
```

```
for ( int n=0; n<5; n++) {  
    Serial.println ( *p );  
    p++;  
}
```



# Pointer bewerkingen en Arrays

## Elementen loop

een terechte opmerking is dat deze code langer is dan het equivalent met de individuele elementen:

```
byte m[5] = { 0,1,2,3,4 };
```

```
byte *p = m;
```

```
for ( int i=0; i<5; i++) {  
    Serial.println ( *p );  
    p++;  
}
```

```
byte m[5] = { 0,1,2,3,4 };
```

```
for ( int i=0; i<5; i++) {  
    Serial.println ( m[ i ] );  
}
```



# Pointer bewerkingen en Arrays

## Elementen loop

kijken we naar het doorgeven van een array  
aan een functie dan zien we meteen het voordeel:

```
void drukaf ( byte *p ) {  
    for ( int i=0; i<5; i++) {  
        Serial.println ( *p );  
        p++;  
    }  
}
```

```
byte m[5] = { 0,1,2,3,4 };
```

```
drukaf(m);
```





# Pointer bewerkingen en Arrays

## Elementen loop

kijken we naar het doorgeven van een array  
aan een functie dan zien we meteen het voordeel:

```
void drukaf ( byte *p ) {  
    for ( int i=0; i<5; i++) {  
        Serial.println ( *p );  
        p++;  
    }  
}
```

```
byte m[5] = { 0,1,2,3,4 };
```

```
drukaf(m);
```



# Pointer bewerkingen en Arrays

## Elementen loop

kijken we naar het doorgeven van een array  
aan een functie dan zien we meteen het voordeel:

```
void drukaf ( byte *p ) {  
    for ( int i=0; i<5; i++) {  
        Serial.println ( *p );  
        p++;  
    }  
}
```

*zonder pointers zouden  
we hier elk element van  
de array afzonderlijk  
moeten doorgeven*

```
byte m[5] = { 0,1,2,3,4 };
```

```
drukaf(m);
```

*en van elk element  
zou een kopie  
gemaakt worden*

