Inhoudsopgave

2

Variabelen, datatypes en expressies

Een C-programma bestaat uit variabelen en functies. We gebruiken variabelen om data te bewerken, zoals berekenen van nieuwe waarden of testen of twee variabelen aan elkaar gelijk zijn. De compiler moet notie hebben welke variabelen in een programma gebruikt worden en hoe groot de variabelen zijn. De meeste moderen computersystemen werken met 64-bits eenheiden en dat betekent dat een variabele een bepaald bereik heeft; niet elk getal kunnen we in een variabele opslaan. We gebruiken functies om bewerkingen op de variabelen te beschrijven. We hebben al één functie gezien: `main`. Functies worden in detail besproken in hoofdstuk 4.

2.1 Variabelen

Een *variabale* is een object waaraan een waarde kan worden toegekend. Een variabele moet aan het C-programma kenbaar gemaakt worden. We noemen dat de *declaratie* van de variabele. Bij de declaratie wordt het *type* van de variabele opgegeven. Zo geeft de declaratie

```
int a;
```

aan dat in het programma de variabele `a` wordt gebruikt van het type `int`. Een `int` is een geheeltallig getal; het Engelse woord hiervoor is *integer*. Getallen met een komma, zoals 3,14159, kunnen we niet in een `int` opslaan. We kunnen nu in het programma de waarde van de variabele vastleggen met een *toekenning*. In het Engels is dat een *assignment*. Als we aan `a` de waarde 2 willen toekennen dan schrijven we

```
a = 2;
```

De C-compiler zorgt ervoor dat in het geheugen van de computer ruimte wordt gereserveerd om de variabele op te slaan. Als we in een programma de variabele `a` gebruiken dan weet de compiler op welk geheugenadres hij moet zoeken. We mogen de variabele onbeperkt aantal keer aanpassen. Zo kunnen we schrijven:

```
int a;
...
a = 2;
...
a = -3;
...
```

Listing 2.1: *Meerdere toekenningen aan een variabele.*

De naam van de variabelen mag 31 karakters lang zijn. Er is geen noodzaak (en zelfs *bad practice*) om namen uit één karakters te laten bestaan. Hoofdletters en kleine letters zijn *niet* gelijk aan elkaar dus `count` is niet hetzelfde als `COUNT`. Dit wordt kast-ongevoelig genoemd. Overigens accepteren C-compilers langere namen dan 31 karakters.

Er zijn enkele regels aan de namen van variabelen:

- Moet beginnen met een letter;
- Mag alleen letters, cijfers en de underscore bevatten;
- Mag geen keyword zijn;
- Mag geen bekende functienaam zijn;[1]
- Mag niet eerder gedeclareerd zijn;

Enkele goede voorbeelden:

```
Count  val23  loop_counter  ary2to4
```

Enkele foute voobeelden:

```
_loop  50cent  int  printf
```

De meeste C-compilers accepteren de underscore als eerste karakter. Dit wordt echter afgeraden omdat veel *systeemroutines* de underscore als eerste karakter gebruiken en dat kan problemen geven in de latere stadia van de compilatie.

Bij de declaratie van variabelen mogen ook gelijk waarden worden toegekend, op voorspraak dat de waarden constant zijn. Een voorbeeld is hieronder te zien:

```
int a = 2;
int b = 3+5;
```

2.2 Datatypes

Een variabele is van een bepaald *datatype*. We hebben er al één gezien, de `int`. Een datatype heeft een bepaalde grootte. Zo bestaat een `int` *meestal* uit 32 bits. Er zijn echter ook systemen en C-compilers waar een `int` uit 16 bits bestaat. Dat een `int` uit 32 bits

[1] Dat mag wel, maar dan kunnen we de functie niet in een programma gebruiken. We kunnen dus schrijven `int printf;` maar dan kan de `printf`-functie niet aangeroepen worden.

bestaat, betekent dat niet elk willekeurig getal aan de `int` kunnen worden toegekend. Er is een minimale en maximale waarde. Voor de `int` is de minimale waarde −2147483648 en de maximale waarde +2147483647.

Nu is de grootte van een `int` niet altijd noodzakelijk of toereikend in een bepaalde situatie. We kunnen dan kiezen voor een ander type. C kent een aantal geheeltallige datatypes van diverse grootten. Deze zijn weergegeven in tabel 2.1.

Tabel 2.1: *De dataypes die beschikbaar zijn in C.*

type	bits	bereik
`char`	8	−128 — +127
`short int`	16	−32768 — +32767
`int`	16	−32768 — +32767
	32	−2147483648 — +2147483647
`long int`	32	−2147483648 — +2147483647
`long long int`	64	−9223372036854775808 — +9223372036854775807

Een `char` wordt gebruikt om een karakter op te slaan en is 8 bits groot. De interpretatie van het opgeslagen karakter is afhankelijk van de computer waarom het programma draait. Merk op dat een `char` zowel negatieve als positieve waarden kan bevatten. De meest gebruikte codering is de ASCII-code. Om het karakter 'A' op te slaan schrijven we:

```
char karak;
karak = 'A';    /* assign character A */
```

Nu is de C-compiler erg coulant in het toekennen van waarden. We mogen voor de toekenning ook een getal gebruiken. We kunnen het bovenstaande ook schrijven als:

```
char karak;
karak = 65;    /* assign character A */
```

Uiteraard moet de waarde passen en we merken terloops op dat negatieve waarden geen echte karakters voorstellen. Een `short int` is meestal 16 bits. Een `int` is 16 of 32 bits groot afhankelijk van de gebruikte C-compiler en de onderliggende hardware. Een `long int` is meestal 32 bits. De C-standaard legt vast dat

grootte `short int` $\leq$ grootte `int` $\leq$ grootte `long int`

De keywords `short` en `long` worden *qualifiers* genoemd. Bij gebruik hiervan mag `int` weggelaten worden:

```
short sh;    /* short int */
long lo;     /* long int */
```

Als we aan een integer alleen maar positieve waarden of 0 toekennen dan kunnen we de qualifier `unsigned` bij declaratie opgeven. Het (positieve) bereik is dan ongeveer twee keer zo groot als bij de integer:

```
unsigned short int ush;    /* range 0 to 65535 */
```

Naast geheeltallige datatypes kent C ook een drietal *floating point* datatypes. Het Nederlandse woord hiervoor is *drijvende komma*. Hier onstaat al gelijk de eerste verwarring: in C wordt de komma vervangen door de punt. We schijven dus 3.14 en niet 3,14.

Een `float` is een datatype met een grootte van 32 bits. Een `double` is een datatype met een grootte van 64 bits. We zouden verwachten dat de `long double` dan een datatype van 128 bits is, maar dat is niet altijd waar. De `long double` is soms ook 80 bits, bijvoorbeeld in Intel-processoren.

Tabel 2.2: *De floating point dataypes die beschikbaar zijn in C.*

type	bits	kleinste getal	grootste getal
`float`	32	$\approx 1{,}18 \times 10^{-38}$	$\approx 3{,}4 \times 10^{38}$
`double`	64	$\approx 2.2 \times 10^{-308}$	$\approx 1.8 \times 10^{308}$
`long double`	80/128	[1])	[1])

[1]) Afhankelijk van de implementatie 80 of 128 bits.

Voorbeelden van floating point variabelen:

```
float f;
double d;
```

Met betrekking tot de nauwkeurigheid kunnen we nog het volgende vermelden. Een `float` heeft een nauwkeurigheid van ongeveer 6 decimale cijfers. Het heeft dus geen zin om meer cijfers toe te voegen, de extra cijfers worden genegeerd. Een `double` heeft een nauwkeurigheid van 16 decimale cijfers. Niet alle getallen kunnen exact in een floating point-variabele worden opgeslagen. Zo is 1/7 niet exact te representeren. Bij veelvuldig rekenen met floating point-variabelen treedt er verlies op in de representatie. Als we bijvoorbeeld 1/3 met 3 vermenigvuldigen, kan het zijn dat het resultaat 0,99999 is en niet 1,0.

Al deze datatypes worden *enkelvoudige datatypes* genoemd. Dat betekent dat de compiler ze ziet als één eenheid. Het is niet mogelijk om enkelvoudige datatypes te splitsen over meerdere andere datatypes.

Naast de genoemde datatypes bestaat er nog een datatype: de *pointer*. Een pointer is een variabele die het *adres* bevat van een (andere) variabele. Pointers worden in detail besproken in hoofdstuk 6.

2.3 Constanten

Een geheeltallige constante zoals `123` is van het type `int`. Door er een `L` achter te zetten wordt de constante een `long`. Dus `123L` is een `long`. Als een constante te groot is voor een `int` wordt het automatisch gezien als een `long`. Een unsigned constante wordt aangegeven met een `U` aan het einde. Dat mag in combinatie met de `L`. Een floating point-constante bestaat uit cijfers, optioneel met een punt. Dus `1.23` is een floating point-constante. Om 10-machten aan te geven wordt de *E*-notatie gebruikt. Zo staat `125.0E-3` voor 0,125. Een floating point-constante is automatisch van het type `double`, tenzij er een `F` aan het einde staat, dan is de constante van het type `float`.

Geheeltallige constanten kunnen op drie manieren worden ingevoerd: als decimaal getal, als octaal getal of als hexaxdecimaal getal. Een constante die begint met `0` wordt gezien als octaal getal. Een octaal getal bestaat alleen uit de cijfers 1 t/m 7. Een constante die begin met de *prefix* `0x` of `0X` wordt gezien als een hexadecimaal getal. Een constante die begint met de cijfers `1` t/m `9` wordt gezien als een decimaal getal.

```
int a = 0377;     /* octal 377 is decimal 255 */
int b = 0xa9;     /* hexadecimal A9 is decimal 169 */
int c = 127;      /* decimal 127 */
```

De C-standaard kent geen manier om binaire getallen in te voeren. Veel compilers ondersteunen dit toch door de prefix `0b` te gebruiken. Zo is de constante `0b10101001` gelijk aan 169.

Een *karakterconstante* is een geheel getal, gescheven als een één karakter tussen apostrofes. Zo is `'A'` een karakerconstante. De interne representatie is een geheel getal dat overeen komt met de karakterset van de computer. Over het algemeen wordt de ASCII-code gebruikt en komt `'A'` overeen met de waarde 65.

Niet alle karakters kunnen zo worden ingevoerd. Een voorbeeld is het karakter `'` zelf. Om dit karakter in te voeren gebruiken we een *escape sequence*. Een escape sequence bestaat uit een *backslash* (`\`) en een karakter. We kunnen de apostrofe dus voorstellen met `'\''`.

De C-standaard kent een hele verzameling van dit soort escape sequences. Eén daarvan hebben we al meerdere malen gezien: de newline. Deze wordt aangegeven met `'\n'`. Een aantal is vermeld in tabel 2.3.

Tabel 2.3: *Enkele escape sequences in C.*

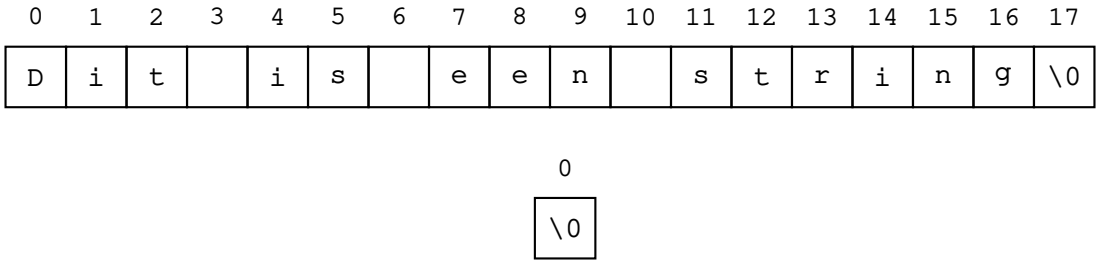
`\n`	newline	`\\`	backslash
`\r`	carriage return	`\'`	single quote
`\a`	bell	`\"`	double quote
`\b`	backspace	`\0`	null byte
`\f`	formfeed	`\t`	horizontal tab

Andere karakters kunnen worden gevormd door de backslash, gevolgd door de letter `x` en één of twee hexadecimale cijfers. We kunnen de letter `A` dus ook schrijven als `'\x41'`.

Een *string constante*, of kortweg string, is een rij van karakters die wordt begonnen en afgesloten met aanhalingstekens. De rij mag ook leeg zijn. Dan spreken we dan van een lege string. Twee voorbeelden:

```
"Dit is een string"
""  /* een lege string */
```

Let erop dat een string *geen* enkelvoudig datatype is zoals `int` en `double`. Een string is een rij karakters die in het geheugen liggen. Technisch gezien is een string een *array*. We kunnen dus niet de waarde van een string bepalen zoals dat wel mogelijk is van een `int` of een `char`. Aan het einde van een string wordt automatisch een nul-byte geplaatst. Er is dus altijd één karakter meer nodig dan het aantal karakters in een string. In figuur 2.1 zijn de twee strings afgebeeld.



Figuur 2.1: *Uitbeelding van twee strings.*

C kent geen ingebouwde operaties op strings, zoals inkorten, aan elkaar plakken, kopieren of lengte bepalen. Dit moet allemaal door de programmeur zelf ontworpen worden. Gelukkig kent de standaard C-bibliotheek een groot aantal functies voor het bewerken van strings. We komen hierop terug in hoofdstuk 5.

Bij de declaratie van variabelen mogen ook gelijk waarden worden toegekend, op voorspraak dat de waarden constant zijn. De constante waarde mag ook berekend worden. Een aantal voorbeelden is hieronder te zien:

```
int a = 2;
char z = 'z';
short a = -10+5;
long l = 123L;
float tau = 2.0F*3.14159F;
double e = 2.718281828;
double googol = 1E100;
unsigned char = 128+127;
```

Let erop dat de constante wordt geconverteerd naar het type van de variabele. Sommige compilers geven een waarschuwing als zo'n conversie plaatvindt.

```
int a = 25.6;   /* converted to 25 */
float b = 7;    /* converted to 7.0 */
```

Bij de declaratie van een variabele mag de qualifier `const` worden opgegeven dat inhoudt dat aan de variabele eenmalig een waarde wordt toegekend. De `const`-variabele kan daarna niet meer veranderen. Dat is bijzonder handig als we een constante waarde moeten gebruiken in een programma. Zo kunnen we na de declaratie

```
const int aantal = 10;
```

We kunnen `aantal` nu gebruiken als vervanging voor het getal 10. Als later blijkt dat het aantal moet worden aangepast, dan hoeven we alleen maar de variabele `aantal` aan te passen.

Uiteraard moet de constante passen. Als we bijvoorbeeld

```
char ch = 65536;
```

uitvoeren zal de compiler een waarschuwing geven dat de constante niet past in een `char`.

2.4 Expressies

Een *expressie* is elk stukje programma dat een uitkomst oplevert. Zo is `2+2` een expressie die de waarde `4` oplevert. We kunnen de uitkomst van een expressie toekennen aan een variabele:

```
a = 2 + 2;
```

Een expressie mag variabelen bevatten:

```
a = b - c;
```

Zelfs de regel

```
a;
```

is een expressie die `4` oplevert als de waarde van `a` `4` is. Ook het vergelijken van twee variabelen is een expressie en mag toegekend worden aan een variabele:

```
a = b == c;     /* note the == */
```

Als `b` gelijk is aan `c` dan wordt `a` gelijk aan 1. Als `b` ongelijk is aan `c` dan wordt `a` gelijk aan 0. We zullen het vergelijken van variabelen en constanten nader bekijken in hoofdstuk 3.

Een expressie mag willekeurig complex zijn, maar let op de voorrangsregels: vermenigvuldigen en delen gaat voor op optellen en aftrekken. Haakjes worden gebruikt de prioriteiten te veranderen:

```
a = (2+b)*5+c;
```

2.4.1 Rekenkundige operatoren

De *binaire* rekenkundige operatoren zijn `*`, `/`, `%`, `+` en `-`. Binair wil in dit verband zeggen dat de operatoren op twee variabelen (of constanten of expressies) werken. De expressie

```
a % b
```

berekent de rest van de deling van `a` gedeeld door `b`. Dit wordt de *modulus operator* genoemd. Daarnaast kunnen + en - ook als *unaire operator* gebruikt worden. Een voorbeeld hiervan is

```
int a = -5;
```

Let goed op de prioriteiten van de operatoren. De unaire operatoren gaan voor op vermenigvuldigen (`*`), delen (`/`) en modulus (`%`). De binaire optelling (`+`) en aftrekking (`-`) hebben en lagere prioriteit dan `*`, `/` en `%`. Haakjes kunnen de volgorde veranderen. Merk op dat de *associativiteit* van links naar rechts is. Dus de expressie

```
a / b / c
```

is equivalent aan

```
(a / b) / c
```

maar *niet* aan

```
a / (b / c)
```

Verder moet worden opgelet bij deling van geheeltallige getallen. Een deling rondt naar beneden af, dus de *fractie*, het deel van een getal na de komma, wordt weggelaten. Dat betekent dat

```
1 / 3 * 3
```

de waarde 0 oplevert. Eerst wordt 1/3 berekend en de geheeltallige uitkomst hiervan is 0. Daarna wordt de uitkomst met 3 vermenigvuldigd. De uitkomst is nog steeds 0.

2.4.2 Relationele operatoren

De zes *relationele operatoren* zijn

```
==   !=   >   >=   <   >=
```

Hierin zijn == gelijk aan en != ongelijk aan. Verder is > groter dan, >= is groter dan of gelijk aan, < is kleiner dan en <= is kleiner dan of gelijk aan. Deze hebben een lagere prioriteit dan de alle rekenkundige operatoren, dus `i < len-1` wordt gelezen als `i < (len-1)`. Overigens hebben == en != een lagere priotiteit dan de overige vier. Een relationele operator levert de waarde 1 op als de vergelijking waar is en anders levert de relationele operator 0 op.

De relationele operatoren hebben vooral nut bij beslissingen. Zie listing 2.2.

```
1 int main(void) {
2
3     int a = 2, b = 3;
4
5     if (a<b) {
6         printf("a␣is␣kleiner␣dan␣b\n");
7     }
8     return 0;
9 }
```

Listing 2.2: *Voorbeeld van een beslissing.*

2.4.3 Logische operatoren

De && - en || -operatoren zijn logische operatoren die expressies met elkaar verbinden. Ze hebben een relatie met de relationele operatoren. Zo wordt variable `isdig` in de expressie

```
isdig = ch>='0' && ch<='9';
```

gelijk aan 1 als `ch` een cijferkarakter is, anders wordt `isdig` 0. In de expressie

```
isbit = ch=='0' || ch=='1';
```

wordt `isbit` gelijk aan 1 als `ch` een '0' of een '1' is, anders wordt `isbit` 0. Het berekenen van een dergelijke expressie wordt gestopt op het moment dat de uitkomst al duidelijk is. Dus de berekening van

```
i==5 && j>3
```

wordt gestopt als `i` ongelijk aan 5 is. De uitkomst staat dan namelijk al vast. Dit wordt *shortcut evaluation* genoemd.

De unaire *negatieoperator* `!` zet een expressie die 0 oplevert om in een 1 en een expressie die *niet-nul* oplevert wordt omgezet in een 0. Een typisch gebruik van `!` is te zien in listing 2.3.

```
1  int main{void} {
2
3      int valid = a<5 || a>10;
4
5      if (!valid) {   /* if valid is 0 */
6          ...
7      }
8
9      return 0;
10 }
```

Listing 2.3: *Voorbeeld van de negatieoperator.*

Met al deze operatoren kunnen we willekeurig complexe expressies realiseren. Om bijvoorbeeld te bepalen of een jaartal een schikkeljaar (Engels: leap year) is, gebruiken we de volgende expressie:

```
int leap = (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
```

We zullen het even uitleggen. Een schikkeljaar is een jaar met als extra dag 29 februari. Zo'n jaartal is deelbaar door 4 en niet deelbaar door 100, of als het jaartal deelbaar is door 400, dan is het wel een schikkeljaar. Dus 1984 is een schikkeljaar, 2100 is geen schikkeljaar maar 2000 weer wel. Deze expressie wordt vaak gebruikt bij het bepalen van een geldige datum.

2.4.4 Bitsgewijze operatoren

C biedt zes zogenoemde *bitsgewijze operatoren*. Ze kunnen alleen maar gebruikt worden bij geheeltallige variabele, constanten of expressies. Deze zijn:

`&`	AND
`|`	OR
`^`	EXOR
`<<`	naar links schuiven
`>>`	naar rechts schuiven
`~`	one's complement (alle bits geïnverteerd)

De *waarheidstabellen* van de AND-, OR- en EXOR-functies zijn gegeven in tabel 2.4. De AND-functie wordt gebruikt om een of meer bits uit een variabele te selecteren of op 0 te zetten. De OR-functie wordt gebruikt om bits in een variabele op 1 te zetten en de EXOR-functie wordt gebruikt om bits in een variabele te inverteren.

De schuifoperator << schuift de bits in een variabele een aantal plekken naar links. De vrijgekomen bits worden opgevuld met nullen. De bits die aan de linkerkant "eruit vallen"

Tabel 2.4: *De drie bitsgewijze operatoren.*

(a) *AND-functie.*

0	0	0
0	1	0
1	0	0
1	1	1

(b) *OR-functie.*

0	0	0
0	1	1
1	0	1
1	1	1

(c) *EXOR-functie.*

0	0	0
0	1	1
1	0	1
1	1	0

verdwijnen. De expressie

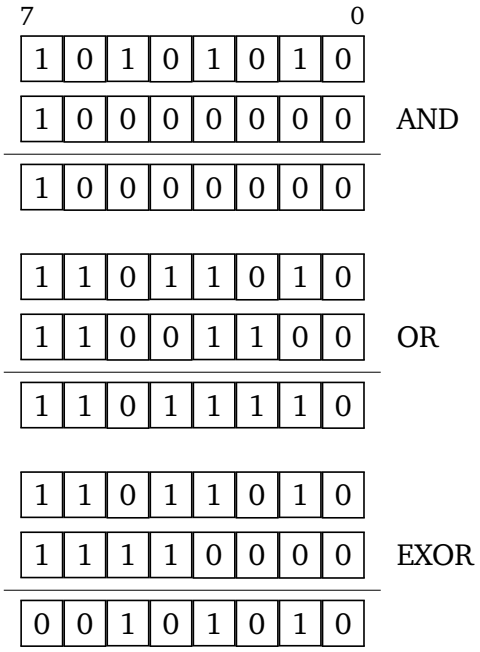
```
1 << 4
```

geeft als resultaat 16. De schuifoperator >> schuift de bits in een variabele een aantal plekken naar rechts. De vrijgekoment bits worden *bij unsigned variabelen* aangevuld met nullen. Bij *signed variabelen* worden ze aangevuld met de *tekenbit*. De bits die aan de rechterkant eruit vallen gaan verloren. De unaire operator ~ inverteert alle bits in een variabele. Een bit die 1 is wordt een 0 en een bit die 0 is wordt een 1. In veel boeken wordt dit *one's complement* genoemd.

De operatoren kunnen door elkaar gebruikt worden. Een voorbeeld is:

```
a = a & ~0xff;
```

Deze expressie zorgt ervoor dat de 8 minst significante bits allemaal 0 worden. In figuur 2.2 zijn drie voorbeelden te zien van bitmanupilatie met AND, OR en EXOR. Hiervoor zijn de gegevens uit tabel 2.4 gebruikt.



Figuur 2.2: *Drie voorbeelden van AND, OR en EXOR.*

De bitsgewijze operatoren zijn bijzonder handig bij het gebruik van microcontrollers. We geven ter illustratie een voorbeeld voor een ATmega-microcontroller. We willen testen of een

schakelaar is ingedrukt. We lezen de stand van de schakelaars in via de (speciale) variabele `PINA`. Als de schakelaar op bit 7 (hoogste bit van `PINA`) is ingedrukt, dan inverteren we de stand van de led die is gekoppeld aan bit 0 van de (speciale) variabele `PORTB`.

```
#include <avr/io.h>             /* PINA, PORTB, ... */

int main(void) {

    if (PINA & 0x80) {           /* if key is pressed ... */
        PORTB = PORTB ^ 0x01;    /* ... invert lower led */
    }

}
```

Listing 2.4: *bla*

2.5 Datatypeconversie

Het omzetten of converteren van het ene datatype naar het andere datatype kan op twee manieren worden gerealiseerd: automatisch door de C-compiler of expliciet door een *type cast* in het programma. We kunnen redelijkerwijs verwachten dat een "kleiner" type wordt omgezet naar een "groter" type. We noemen dat *promotie*. De regels voor automatische conversie zijn complex; we geven hier slecht een korte opsomming:

- Als een van de twee operanden een `long double` is, converteer de andere naar `long double`;
- Als een van de twee operanden een `double` is, converteer de andere naar `double`;
- Als een van de twee operanden een `float` is, converteer de andere naar `float`;
- Converteer `char` en `short` naar `int`;
- Als een van de twee operanden een `long` is, converteer de andere naar `long`.

De regels voor unsigned variabelen zijn nog lastiger als ze gecombineerd worden met signed integers. Het beste is om deze *mixed types* te vermijden.

Een conversie kan ook expliciet worden opgegeven. We noemen dit een *type cast*. We geven dit aan door het type tussen haken te zetten:

```
i = (int) ch;     /* promote character to integer */
```

In dit geval geeft dat geen problemen want een karakter past in een integer. Maar de cast:

```
ch = (char) i;     /* demote integer to character */
```

kan voor problemen zorgen als de waarde `i` te groot is om in een karakter te plaatsen. Er gaan dan bits verloren. Een mooi voorbeeld van een type case is het omrekenen van een temperatuur in graden Celsius naar graden Fahrenheit. Stel we hebben twee `floats` voor de temperaturen. Dan kan de conversie berekend worden met:

```
f = (float)9/5 * c + 32;
```

We hebben hier de integer 9 gecast naar een `float` want anders gaat bij de deling 9/5 informatie verloren. De constante 32 hoeft niet gecast te worden omdat eerder al de getallen en variabelen naar een `float` gecast zijn. Het is overigens *good practice* om `32.0` te schrijven.

2.6 Overflow

Overflow is de situatie als een expressie een waarde oplevert die te groot of te klein is en niet in de variabele kan worden opgeslagen. De C-compiler genereert geen code om hierop te testen. Dat zou wel kunnen, maar dat zou de executietijd van programma's nadelig beïnvloeden. De ontwerper van het programma moet hier zelf op toezien dat een overflow-conditie niet kan voorkomen. De reden dat overflow voorkomt is dat datatypes (en dus variabelen) uit een eindig aantal bits bestaan. Zo bestaat een `unsigned char` uit 8 bits is het grootste getal dat kan worden opgeslagen 255 (binair 11111111_2) bedraagt. Tellen we daar 1 bij op dat is het resultaat 256 (binair 100000000_2) maar dat kan niet in de variabele worden opgeslagen. Het resultaat is dat de overvloedige bits gewoonweg worden geschrapt.

Overflow bij floating point-getallen werkt anders. Dat heeft te maken met de gebruikte specificatie van de getallen. Naast representeerbare getallen kent de floating point-specificatie nog drie speciale getallen: $+\infty$ (oneindig positief), $-\infty$ (oneindig negatief) en `NaN` (Not A Number). Ze geeft de deling 1/0 als resultaat $+\infty$[2]. De deling 0/0 resulteert in NaN.

2.7 Vaste-lenge datatypes

Dat de grootte van integers aan elkaar gerelateerd zijn, kan in programma's voor problemen leiden. We weten nu niet of een `int` nou 16 of 32 bits is. Dit heeft geleid tot een aantal nieuwe datatypes die in de C99-standaard zijn vastgelegd. De grootte van deze types zijn exact. Zo is een `int8_t` een integer van precies 8 bits. Naast de integer met teken zijn er ook enkele types zonder teken. Zo is een `uint16_t` en integer van precies 16 bits met een minimale waarde van 0 en een maximale waarde van 65535.

Voor "normaal" gebruik zijn deze datatypes niet echt nodig. Maar bij het schrijven van programma's op microcontrollers is het vaak de enige goede methode om zeker ervan te zijn dat een bepaalde grootte van een variabele gegarandeerd is. Dit is met name belangrijk bij het gebruik van zogenoemde *I/O-adressen*. Dat zijn speciale adressen in het geheugen van de microcontroller waarmee communicatie met de buitenwereld mogelijk is, zoals het inlezen van de stand van een schakelaar of het laten branden van een led.

In de tabellen 2.5 en 2.6 zijn de nieuwe datatypes voor signed en unsigned te zien. Voordat we ze kunnen gebruiken moeten we eerst het *headerbestand* `stdint.h` laden. Dit is te zien in listing 2.5.

[2] Volgens de wiskunde is het resultaat van een deling door 0 onbepaald. De specificatie definieert echter de deling als oneiding.

Tabel 2.5: *De vaste-lengte dataypes die beschikbaar zijn in C (signed).*

type	bits	kleinste getal	grootste getal
`int8_t`	8	−128	+127
`int16_t`	16	−32768	+32767
`int32_t`	32	−2147483648	+2147483647
`int64_t`	64	−9223372036854775808	+9223372036854775807

Tabel 2.6: *De vaste-lengte dataypes die beschikbaar zijn in C (unsigned).*

type	bits	grootste getal
`uint8_t`	8	255
`uint16_t`	16	65535
`uint32_t`	32	4294967295
`uint64_t`	64	18446744073709551615

```
1  /* Load new datatypes */
2  #include <stdint.h>
3
4  int main(void) {
5
6      uint8_t byte = 0;
7      ...
8      while (byte<128) {
9          ...
10         byte = byte + 1;
11     }
12 }
```

Listing 2.5: *Voorbeeld van het gebruik van vaste-lengte datatypes.*

2.8 Overige operatoren

C kent veel operatoren. In deze paragraaf beschrijven we ze even kort. In de volgende hoofdstukken worden ze verder uitgelegd.

2.8.1 Grootte van een variabele

De grootte van een variabele of datatype in *bytes* kan berekend worden met de operator `sizeof`. Dit kan alleen tijdens compileren van het C-programma gebeuren. Tijdens het uitvoeren van het programma zijn alle grootten berekend. We zullen `sizeof` bespreken in hoofdstuk 5. Deze operator heeft een hoge prioriteit.

2.8.2 Verhogen of verlagen met 1

Een toekenning als

```
i = i + 1;
```

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

    if (sizeof i == sizeof long) {
        printf("De grootte van i is gelijk aan een long\n");
    }

    return 0;
}
```

Listing 2.6: *sizeof.*

kan geschreven worden als

```
i++;
```

of als

```
++i;
```

Het verschil is dat bij `i++` eerst de waarde van `i` wordt gebruikt en daarna met 1 wordt opgehoogd (dit wordt *postfix* genoemd), bij `++i` wordt de waarde van `i` eerst opgehoogd en daarn gebruikt (dit wordt *prefix* genoemd). Stel dat `i` is 5 dan zorgt

```
k = i++;
```

dat `k` gelijk is aan 5 en `i` gelijk is aan 6. Bij

```
k = ++i;
```

zijn `k` en `i` gelijk aan 6. We zullen deze operatoren tegenkomen in hoofdstuk 5. Deze operatoren hebben een hoge prioriteit.

2.8.3 Toekenningsoperatoren

Een toekenning als

```
i = i * 3;
```

mag geschreven worden als

```
i *= 3;
```

Let daarbij op dat de expressie aan de rechterkant van de toekenningsoperator als één eenheid wordt gezien. Dus

```
i *= y + 1;
```

wordt uitgewerkt als

```
i = i * (y + 1);
```

Zo'n beetje alle gangbare operatoren kunnen worden gebruikt. Een lijst is te vinden in tabel tab:vartoekenningsoperatoren. Deze operatoren hebben een lage prioteit.

Tabel 2.7: *Toekenningsoperatoren.*

Operatie		Evaluatie
`a *= b`	lees als	`a = (a) * (b)`
`a /= b`	lees als	`a = (a) / (b)`
`a %= b`	lees als	`a = (a) % (b)`
`a += b`	lees als	`a = (a) + (b)`
`a -= b`	lees als	`a = (a) - (b)`
`a <<= b`	lees als	`a = (a) << (b)`
`a >>= b`	lees als	`a = (a) >> (b)`
`a &= b`	lees als	`a = (a) & (b)`
`a |= b`	lees als	`(a) = a | (b)`
`a ^= b`	lees als	`(a) = a ^ (b)`

2.8.4 Nog enkele operatoren

De volgende opertoren worden de verder op in het boek besproken:

`[]` `->` `.` `*` (*dereference*) `&` (*address*) `?:` `,`

2.9 Voorrangsregels van operatoren

Bij het uitwerken van expressies worden voorrangsregels gehanteerd. C kent een groot aantal operatoren. We hebben de *meest voorkomende* operatoren op volgorde van prioriteit opgesomd in tabel 2.8. Merk op dat de toekenningsoperator = ook in de lijst voorkomt. Een in C-programma's gebruikelijke constructie is het inlezen van een karakter én gelijk testen of het een newline-karakter betreft:

```
while ((ch = getchar()) != '\n') { ... }
```

Hier gebeuren eigenlijk drie dingen. Eerst wordt de *functie* `getchar` aangeroepen om een karakter van het toetsenbord te lezen. Daarna wordt dit karakter toegekend aan variabele `ch`. De haakjes zorgen voor de juiste prioriteit. De uitkomst van de toekenning is de geheeltallige waarde van de toekenning en dat is het ingelezen karakter. Daarna wordt getest of het ingelezen karakter gelijk is aan het newline-karakter.

Tabel 2.8: *Voorrangsregels van de meeste operatoren.*

Operatie	**Associativiteit**
`()`	links naar rechts
`! ~ + - ++ --` (*type cast*) `sizeof`	rechts naar links
`* / %`	links naar rechts
`+ -`	links naar rechts
`<< >>`	links naar rechts
`< <= > >=`	links naar rechts
`== !=`	links naar rechts
`&`	links naar rechts
`^`	links naar rechts
`|`	links naar rechts
`&&`	links naar rechts
`||`	links naar rechts
`?:`	rechts naar links
`= += -= *= /= %= &= ^= |= <<= >>=`	rechts naar links
`,`	links naar rechts

3

Flow control

4

Functies

5

Array's

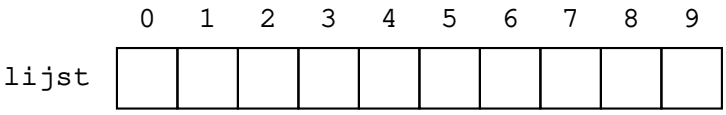
Een *array* is een manier om bij elkaar behorende gegevens te groeperen en te bewerken.

Een array bestaat uit een aantal *elementen*. De nummering van de elementen begint bij 0.

De declaratie van een array begint met het opgeven van het datatype zoals `int` en `double`, gevolgd door de naam van de array en het aantal elementen. In onderstaande listing wordt de array `lijst` met tien elementen van het type `int` gedeclareerd.

```
1 int lijst[10];
```

Listing 5.1: *Declaratie van een array met tien elementen*



Figuur 5.1: *Uitbeelding van een array met tien elementen.*

6

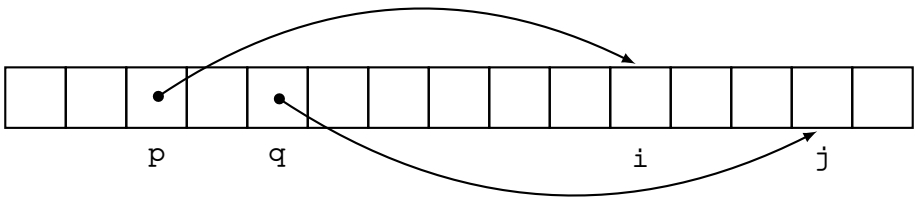
Pointers

Een pointer-variabele, of kortweg *pointer*, is een variabele waarvan de inhoud het adres is van een andere variabele. We zeggen dan ook wel dat de pointer *wijst* (Engels: “points to”) naar de andere variabele. Als een pointer naar een variabele wijst, is het mogelijk om via de pointer bij de variabele te komen.

Pointers zijn een krachtig middel om efficiënt gegevens te beheren en parameters over te dragen aan functies. Soms zijn pointers zelfs de enige manier voor het bewerken van data. Het is dan ook niet verwonderlijk dat in veel C-programma’s pointers gebruikt worden.

Pointers worden samen met het `goto`-statement in verband gebracht met het schrijven van ondoorzichtige programma’s. En zeker, onzorgvuldig gebruik van pointers komt de leesbaarheid en aantonen van correctheid van programma’s niet ten goede. Maar met discipline kunnen programma’s zeer efficient geschreven worden.

Een handige manier om pointers weer te geven, is door het geheugen van een computer voor te stellen als een rij vakjes. In figuur 6.1 is dat te zien. Elk vakje stelt een geheugenplaats voor[1]. In de figuur zijn de variabelen `i` en `j` te zien. De twee pointers `p` en `q` wijzen respectievelijk naar variabele `i` en `j`. Dit is weergegeven met de twee pijlen. De inhoud van pointer `p` is dus het adres van variabele `i` en de inhoud van pointer `q` is het adres van variabele `j`.



Figuur 6.1: *Uitbeelding van twee pointers naar variabelen in het geheugen wijzen.*

[1] We gaan hier gemakshalve vanuit dat elke variabele precies één geheugenplaats in beslag neemt. In de praktijk bestaan variabelen en pointers meestal uit meer dan één geheugenplaats.

Een pointer kan wijzen naar een enkelvoudige variabele, een (element van een) array, een structure of (het begin van) een functie. Een pointer kan niet wijzen naar een constante, een uitdrukking en een `register` variabele.

Het is ook mogelijk om een pointer naar het datatype `void` te laten "wijzen". Deze pointers worden generieke pointers genoemd. We zullen dit bespreken in paragraaf 6.3.

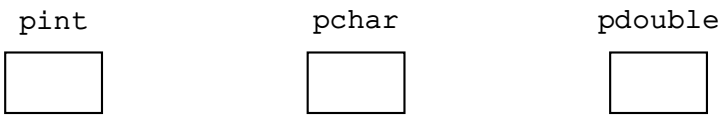
6.1 Pointers naar enkelvoudige datatypes

In listing 6.1 is de declaratie van enkele pointers van enkelvoudige datatypes te zien. Bij de declaratie moet het type variabele waarnaar de pointer wijst worden opgegeven. De asterisk (`*`) geeft aan dat het de declaratie van een pointer betreft.

```
1 int *pint;          /* pint is a pointer to an int */
2 char *pchar;        /* pchar is a pointer to a character */
3 double *pdouble;    /* pdouble is a pointer to a double */
```

Listing 6.1: *Enkele declaraties van pointers.*

We kunnen pointers uitbeelden door middel van vakjes, zoals te zien is in figuur 6.2. Elk vakje stelt een pointer voor. In beginsel hebben de pointers geen correcte inhoud. Er wordt dan wel gesproken dat de pointer nergens naar toe wijst, maar dat is feitelijk onjuist. Een pointer heeft altijd een adres als inhoud, maar het kan zijn dat de pointer niet naar een bekende variabele wijst.



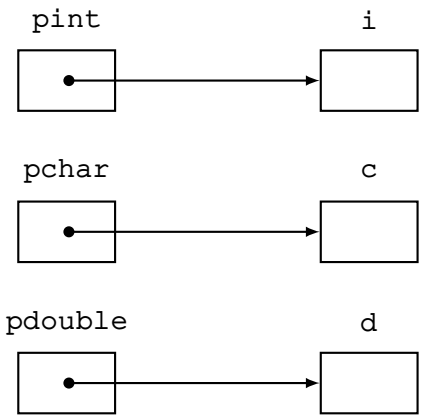
Figuur 6.2: *Voorstelling van drie pointers in het geheugen.*

Aan een pointer is het adres van een variabele van hetzelfde type toe te kennen. Hiervoor gebruiken we de *adres-operator* `&` (ampersand). In listing 6.2 is een aantal toekenningen van adressen te zien.

```
1  int i;           /* the variables */
2  char c;
3  double d;
4
5  int *pint;       /* the pointers */
6  char *pchar;
7  double *pdouble;
8
9  pint = &i;       /* pint points to variable i */
10 pchar = &c;      /* pchar points to variable c */
11 pdouble = &d;    /* pdouble points to variable d */
```

Listing 6.2: *Enkele toekenningen van adressen aan pointers.*

Nu de pointers geïnitialieerd zijn, kunnen we een voorstelling maken van de relatie tussen de pointers en de variabelen. Dit is te zien in figuur 6.3. Pointer `pint` wijst naar variabele `i`, pointer `pchar` wijst naar variabele `c` en pointer `pdouble` wijst naar variabele `d`.



Figuur 6.3: *Voorstelling van drie pointers die naar variabelen wijzen.*

Via de pointers kunnen we de variabelen gebruiken. Stel dat we variabele `i` met één willen verhogen. Dat kunnen we doen door gebruik te maken van de *indirectie* of *dereference* operator `*`. Deze operator heeft voorrang op de optelling.

```
1 *pint = *pint + 1;          /* increment i with one */
2 *pchar = *pchar + 1;        /* next character in ASCII table */
3 *pdouble = *pdouble + 1.0;  /* add 1.0 to double*/
```

Listing 6.3: *Gebruik van een pointer bij een toekenning.*

We mogen de dereference operator overal gebruiken waar een variabele gebruikt mag worden, bijvoorbeeld bij een optelling of in een test.

```
1  int i = 2, *pint;
2
3  ...
4
5  pint = &i;
6
7  ...
8
9  i = *pint + 1;    /* add 1 to variable i */
10
11 ...
12
13 if (*pint > 5) {
14     printf("Variabele␣is␣%d\n", *pint);
15 }
```

Listing 6.4: *Gebruik van een pointer bij het afdrukken van een variabele.*

Overigens kan tijdens declaratie ook gelijk de initialsatie van een pointer plaatsvinden. Deze declaratie kan verwarrend zijn. In onderstaande listing wordt de pointer `pint` gedeclareerd en geïnitialiseerd met het adres van variabele `i`. Het betreft hier dus *geen* dereference.

```
1 int i = 2;
2 int *pint = &i;     /* declare and initialize pint */
```

Listing 6.5: *Declaratie en initialisatie van een pointer.*

6.2 De NULL-pointer

In principe wijst een pointer naar een variabele (of beter: naar een geheugenadres). Om aan te geven dat een pointer niet naar een variabele wijst, kunnen we de *NULL-pointer* gebruiken. Let erop dat de NULL-pointer niet hetzelfde is als een niet-geïnitialiseerde pointer. Een NULL-pointer is een pointer waarin alle bits 0 zijn[2]. Een niet-geïnitialiseerde pointer heeft een willekeurige waarde[3]. In C is een preprocessor-macro genaamd `NULL` te gebruiken om een pointer als NULL-pointer te initialiseren. De C-standaard schrijft voor voor dat `NULL` wordt gedefinieerd in `locale.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, `time.h`, en `wchar.h`. Slechts een van deze header-bestanden is noodzakelijk om `NULL` te definiëren.

```
1 #include <stdio.h>
2
3 int *p = NULL;   /* p is initialized as NULL-pointer */
```

Listing 6.6: *Declaratie en initialisatie van een NULL-pointer.*

NULL-pointers kunnen *niet* gebruikt worden bij dereferentie. Dat veroorzaakt over het algemeen dat de executie van een programma wordt afgebroken.

```
1 #include <stdio.h>
2
3 int *p = NULL;   /* p is initialized as NULL-pointer */
4
5 *p = *p + 1;     /* Oops, dereference of NULL-pointer! */
```

Listing 6.7: *Dereferentie van een NULL-pointer.*

In een vergelijking zullen twee NULL-pointers altijd `true` opleveren.

[2] In het algemeent is de inhoud van een NULL-pointer het getal 0, maar dat is niet in alle gevallen zo. De C-standaard definieert de NULL-pointer als een pointer die niet naar een bekende variabele wijst.

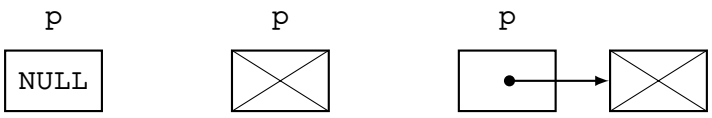
[3] In geval een pointer als globale variabele wordt gedeclareerd, zorgt de compiler ervoor dat de inhoud op 0 gezet wordt.

```c
int *p = NULL, *q = NULL;
...
if (p == q) {
    /* true */
}
```

Listing 6.8: *Vergelijken van twee NULL-pointers.*

De NULL-pointer wordt door diverse standaard functies gebruikt om aan te geven dat er een fout is geconstateerd. Zo geeft de functie `fopen` de waarde `NULL` terug als het niet gelukt is om een bestand te openen. De functie `malloc` geeft de waarde `NULL` terug als het niet gelukt is om een stuk geheugen te alloceren.

Bij het voorstellen van NULL-pointers zijn diverse mogelijkheden die gebruikt worden. Bij de linker voorstelling wordt het woord `NULL` in een vakje gezet, bij de middelste voorstelling wordt een kruis in het vakje gezet en bij de rechter voorstelling wordt een pijl getrokken naar een vakje met een kruis erin.



Figuur 6.4: *Drie voorstellingen van NULL-pointers.*

6.3 Pointer naar `void`

De void-pointer, ook wel *generieke pointer* genoemd, is een speciaal type pointer die naar elk type variabele kan wijzen. Een void-pointer wordt net als een gewone pointer gedeclareerd middels het keyword `void`. Toekenning aan een void-pointer gebeurt met de adres-operator `&`.

```c
int i;
char c;
double d;

void *p;  /* void-pointer */

p = &i;   /* valid */
p = &c;   /* valid */
p = &d;   /* valid */
```

Listing 6.9: *Declaratie en initialisatie van een void-pointer.*

Omdat het type van een void-pointer niet bekend is, kan een void-pointer niet zonder meer in een dereferentie gebruikt worden. De void-pointer moet expliciet gecast worden naar het correcte type. In de onderstaande listing gebruiken we pointer `p` om naar een `int` te wijzen. De type cast `(int *)` zorgt ervoor dat pointer `p` naar een `int` wijst. Door gebruik te maken van de dereference operator `*` kunnen we bij de inhoud van variabele `i`. De constructie `*(int *)` is dus een expliciete dereference naar een integer.

```
int i = 2;
void *p = &i;                    /* void-pointer */

...
*(int *) p = *(int *) p + 1;     /* explicit type cast */

...
printf("De waarde is %d\n", *(int *) p);
```

Listing 6.10: *Declaratie, initialisatie en deference van een void-pointer.*

6.4 Afdrukken van pointers

Het afdrukken van de waarde van een pointer kan met de `printf`-functie en de format specifier `%p`. Merk op dat de pointer van het type `void` moet zijn, maar veel compilers accepteren pointers naar een datatype.

```
#include <stdio.h>

int main() {

    int i = 2, *p = &i;

    printf("Pointer: %p\n", (void *) p);

    return 0;
}
```

Listing 6.11: *Afdrukken van een pointer.*

Noot: Bij 32-bits compilers is de grootte van een pointer 32 bits (4 bytes). Zo'n pointer kan maximaal 4 GB adresseren. Bij 64-bits compilers is de grootte van een pointer 64 bits (8 bytes). Zo'n pointer kan maximaal 16 EB (exa-bytes) adresseren.

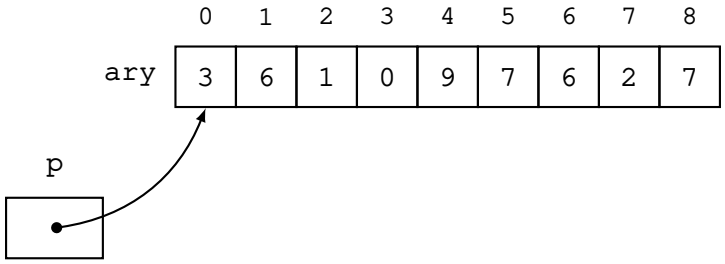
6.5 Pointers naar array's

We kunnen een pointer ook laten wijzen naar het eerste element van een array. Dit is te zien in listing 6.12. De array bestaat uit negen elementen van het type `int`. De pointer `p` laten we wijzen naar het eerste element van de array. We gebruiken hiervoor de adres-operator `&` en elementnummer 0.

```
int ary[] = {3,6,1,0,9,7,6,2,7};

int *p = &ary[0]; /* p points to first element for array */
```

Listing 6.12: *Een pointer naar het eerste element van een array.*

De uitbeelding hiervan is te zien in figuur 6.5.



Figuur 6.5: *Uitbeelding van een pointer naar het eerste element van een array.*

Omdat deze toekenning zeer vaak in een C-programma voorkomt, is er een verkorte notatie mogelijk. We kunnen in plaats van `&ary[0]` ook de naam van de array gebruiken: *de naam van een array is een synoniem voor een adres van het eerste element van de array*. Zie listing 6.13.

```
1 int ary[] = {3,6,1,0,9,7,6,2,7};
2
3 int *p = ary; /* p points to first element of array */
```

Listing 6.13: *Een pointer naar het eerste element van een array.*

Omdat de naam van een array een synoniem is, mag het dus niet gebruikt worden aan de linkerkant van een toekenning.

```
1 int *p, ary[] = {3,6,1,0,9,7,6,2,7};
2
3 p = ary;      /* correct use of pointer and array name */
4 ary[2] = *p; /* correct use of pointer and array element */
5
6 ary = p;      /* ERROR: array name cannot be used in this context */
```

Listing 6.14: *Een pointer naar het eerste element van een array.*

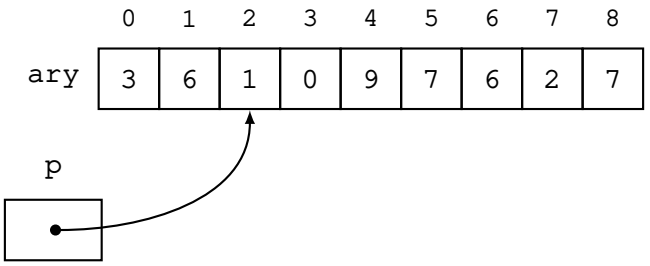
Het is ook mogelijk om een pointer naar een ander element van een array te laten wijzen. De compiler test niet of de toekenning binnen de array-grenzen ligt.

```
1 int ary[] = {3,6,1,0,9,7,6,2,7};
2
3 int *p = &ary[2]; /* p points to third element of array */
```

Listing 6.15: *Een pointer naar het derde element van een array.*

Een uitbeelding is te zien in figuur 6.6. In de figuur wijst `p` naar het derde element van `ary`. We kunnen nu de inhoud van dit element opvragen door `ary[2]` en door `*p`. Het is zelfs mogelijk om `p` als de naam van een array te beschouwen. Zie paragraaf 6.8.

We kunnen de lengte van `ary` bereken met de `sizeof`-operator. Dat kan alleen via `ary` omdat de compiler de lengte kan uitrekenen. Het kan *niet* via pointer `p` want dat is een pointer naar een `int`; pointer `p` “weet” niet dat er naar een array gewezen wordt.



Figuur 6.6: *Uitbeelding van een pointer naar het derde element van een array.*

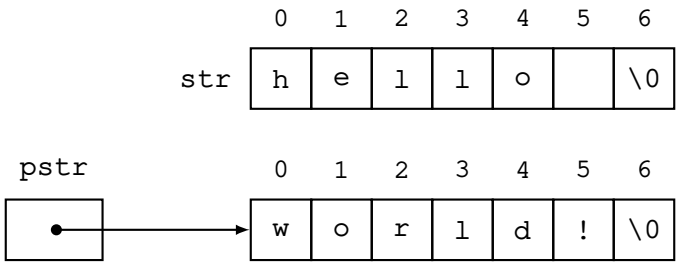
6.6 Strings

Een string in C is niets anders dan een array van karakters, afgesloten met een nul-karakter (`'\0'`). Er is dus altijd één geheugenplaats meer nodig dan het aantal karakters in de string. Een nul-karakter is niet hetzelfde als een NULL-pointer. Een nul-karakter is een byte met de inhoud 0 (alle bits zijn 0), een NULL-pointer is een pointer met de inhoud 0. in listing 6.16 zijn twee declaraties met strings te zien, een echte array met en string als inhoud en een pointer naar een string in het geheugen.

```
1 char str[] = "Hello␣";
2 char *pstr = "world!";
```

Listing 6.16: *Declaratie en initialisatie van twee C-strings.*

Merk op dat `str` niet aangepast mag worden, want dit is de naam van een array. Pointer `pstr` mag wel aangepast worden want `pstr` is een pointer naar het eerste element van de array. Een voorstelling van beide strings is te zien in figuur 6.7.



Figuur 6.7: *Uitbeelding van twee C-strings.*

Ook hier merken we op dat we de lengte van `str` kunnen berekenen met de `sizeof`-operator. De compiler heeft genoeg informatie beschikbaar. We kunnen de lengte van de tweede string *niet* door de compiler laten uitrekenen, want `pstr` in een pointer naar een `char`. Pointer `pstr` "weet" dus niet dat er naar een string gewezen wordt.

Toch is het mogelijk om tijdens het draaien van een programma de lengte van de string te vinden. We kunnen namelijk uitgaan van het feit dat een string in 'C' wordt afgesloten met een nul-karakter. Dit wordt uitgelegd in de volgende paragraaf.

6.7 Rekenen met pointers

Pointers kunnen rekenkundig worden aangepast dat vooral nuttig is bij het gebruik van array's. In het onderstaande programma wijst pointer `p` in eerste instantie naar het begin van de array `ary` (dus `ary[0]`). Daarna wordt `p` twee maal met 1 verhoogd en daarna met 3 verhoogd. Bij rekenkundige operaties op pointers wordt rekening gehouden met de grootte van de datatypes. De grootte van een `int` is in de regel vier bytes. Door de pointer met 1 te verhogen wordt dus naar de volgende `int` gewezen.

```
int ary[] = {3,6,1,0,9,7,6,2,7};
int *p = ary;  /* p pointe to ary[0] */

p = p + 1;     /* p points to ary[1] */
...
p = p + 1;     /* p points to ary[2] */
...
p = p + 3;     /* p points to ary[5] */
```

Listing 6.17: *Rekenen met pointers.*

Een mooi voorbeeld van het rekenen met pointers is het bepalen van de lengte van een C-string. In listing 6.18 wordt pointer `str` gedeclareerd en wijst naar het begin van de string. Pointer `begin` wijst ook naar het begin van de string. Daarna verhogen we pointer `str` totdat het einde van de string is bereikt. Daarna drukken we het verschil van de twee pointers af.

```
#include <stdio.h>

int main() {

    char *str = "Hallo wereld!";
    char *begin = str;

    while (*str != '\0') {   /* while not end of string ... */
        str = str + 1;       /* point to the next character */
    }

    printf("Lengte is %d\n", str-begin);
}
```

Listing 6.18: *Berekenen van de lengte van een string met behulp van pointers.*

Let erop dat de twee pointers naar elementen in dezelfde array moeten wijzen (of één na het laatste element). Alleen dan levert de aftrekking `str-begin` een gedefinieerd resultaat. De aftrekking is van het type `ptrdiff_t` (dat meestal gelijk is aan een `int`) en levert het verschil in elementen. Het onderstaande programmafragment geeft als uitvoer de waarde 3.

Vergelijken van twee pointers kan ook. Zo kunnen pointers op gelijkheid worden vergeleken, maar ongelijkheid kan ook. We zouden de `printf`-regel van listing 6.18 kunnen vervangen door de onderstaande programmafragment. Uiteraard moeten de twee pointers naar

```
int ary[] = {3,6,1,0,9,7,6,2,7};
int *p = &ary[2];
int *q = &ary[5];

printf("Verschil is %d\n", q-p);
```

Listing 6.19: *Het berekenen van het verschil van twee pointers.*

hetzelfde datatype wijzen en heeft de vergelijking alleen zin als de pointers naar elementen binnen dezelfde array wijzen.

```
    if (str>begin) {
        printf("Lengte is %d\n", str-begin);
    } else {
        printf("De string is leeg\n");
    }
```

Listing 6.20: *Vergelijken van twee pointers.*

6.8 Relatie tussen pointers en array's

De relatie tussen pointers en array's zijn zo sterk in 'C' verankerd, dat we er een aparte paragraaf aan wijden. In listing 6.21 zijn de declaratie en initialisatie van een array en een pointer te zien. De pointer `p` wijst na initialisatie naar het eerste element van de array.

```
int ary[] = {3,6,1,0,9,7,6,2,7};
int *p = ary;
```

Listing 6.21: *Declaratie en initialisatie van een array en een pointer.*

Om toegang te krijgen tot eerste element uit de array kunnen natuurlijk `ary[0]` gebruiken. Maar we kunnen via `p` ook bij het eerste element komen. Hiervoor gebruiken we `*p`. We mogen echten `p` ook lezen als de naam van een array. Om het het eerste element te komen, mogen we dus ook `p[0]` gebruiken. Om alle elementen van de array bij elkaar op te tellen, kunnen we dus schrijven:

```
int ary[] = {3,6,1,0,9,7,6,2,7};
int *p = ary;

int sum = p[0]+p[1]+p[2]+p[3]+p[4]+p[5]+p[6]+p[7]+p[8];
```

Listing 6.22: *Bepalen sum van elementen in een array.*

Aan de andere kant mogen we de naam van de array ook lezen als een pointer naar het eerste element. We kunnen de naam gebruiken in een dereference. Dat betekent dat `*ary`

identiek is aan `ary[0]` en dat `*(ary+2)` identiek is aan `ary[2]`. We hebben echter wel haken nodig bij `*(ary+2)` omdat de dereference-operator voorgaat op de optelling. Om de som van de array te bepalen mogen we dus schrijven:

```
int ary[] = {3,6,1,0,9,7,6,2,7};
int *p = ary;

int sum = *ary + *(ary+1) + *(ary+2) + *(ary+3) + ... ;
```

Listing 6.23: *Bepalen sum van elementen in een array.*

Het is mogelijk om een pointer naar een willekeurig element van de array te laten wijzen door de naam van de array als pointer te beschouwen. Als we `p` willen laten wijzen naar het derde element gebruiken we gewoon de toekenning `p = ary+2`. Na deze toekenning kunnen het derde element afdrukken door `p` als pointer of als array te beschouwen. Zie listing 6.24.

```
int ary[] = {3,6,1,0,9,7,6,2,7};

int *p = ary+2;                    /* p points to third element */

printf("Contents is %d\n", *p);    /* prints third element */
printf("Contents is %d\n", p[0]);  /* prints third element */
```

Listing 6.24: *Pointer die naar een element in een array wijst.*

Let erop dat we in het bovenstaande programmafragment `p[0]` hebben gebruikt. De pointer wijst naar het derde element dus `p[0]` betekent dat de inhoud van het derde element wordt afgedrukt.

Als we zeker weten dat we een correct element gebruiken, mogen we ook negatieve waarden voor het elementnummer gebruiken. In listing 6.25 wordt het adres van het derde element uit de array toegekend aan pointer `p`. We mogen dan `p[-1]` gebruiken omdat dit het tweede element uit de array betreft. We kunnen echter niet `ary[-1]` gebruiken want dit leidt tot het gebruik van een element buiten de array. Let erop dat C-compilers over het algemeen niet testen of een adressering binnen de arraygrenzen ligt.

```
int ary[] = {3,6,1,0,9,7,6,2,7};

int *p = ary+2;    /* p points to third element */

int a = p[-1];     /* legal: points to second element */
int b = ary[-1];   /* ILLEGAL: points outside array */
```

Listing 6.25: *Pointer die naar een element in een array wijst.*

6.9 Pointers als functie-argumenten

Net als “gewone” variabelen, kunnen ook pointers als argumenten bij het aanroepen van een functie gebruikt worden. Let erop dat een kopie van de pointers worden meegegeven. Via die kopie kunnen we bij de variabelen komen waar de pointers naartoe wijzen. We kunnen dus niet de pointers zelf aanpassen.

In listing 6.26 is te zien hoe we een functie `swap` definieren die de inhouden van twee variabelen verwisseld. Bij het aanroepen van de functie geven we de addressen mee van de te verwisselen variabelen. In de functie gebruiken we pointers om de inhouden te verwisselen.

```
1  void swap(int *pa, int *pb) {
2      int temp;
3
4      temp = *pa;
5      *pa = *pb;
6      *pb = temp;
7  }
```

Listing 6.26: *Het verwisselen van twee variabelen met behulp van pointers.*

Bij het aanroepen van de functie geven we de adressen van de variabelen mee. Dit is te zien in listing 6.27. Deze manier van argumentenoverdracht wordt *Call by Reference* genoemd.

```
1  void swap(int *pa, int *pb);   /* prototype of function swap */
2
3  int main(void) {
4
5      int a=2, b=3;              /* declare variables */
6
7      swap(&a, &b);              /* swap variables */
8
9      return 0;
10 }
```

Listing 6.27: *Aanroep van de functie.*

Een typisch voorbeeld is een functie die een string kopieert naar een andere string. De functie krijgt twee pointers naar strings als argumenten mee. Bij het aanroepen van de functie worden de namen van de twee strings als argumenten meegegeven. De naam van een string is immers een pointer naar het eerste karakter van de string. Het programma is te zien in listing 6.28.

Merk op dat de geheugenruimte voor de kopie groot genoeg moet zijn om de kopie op te slaan en dat de twee string elkaar in het geheugen niet mogen overlappen. De standaard C-bibliotheek heeft een functie `strcpy` die een efficiente implementatie is van het kopieren van strings.

Noot: een array kan alleen maar via een pointer als argument aan een functie worden doorgegeven. Dit is veel efficiënter dan de hele array mee te geven. In listing 6.28 wordt

```c
void string_copy(char *to, char *from) {

   if (from == NULL) {        /* sanity check */
       return;
   }

    while (*from != '\0') {   /* while not end of string ... */
        *to = *from;          /* copy character */
        to = to + 1;          /* point to the next character */
        from = from + 1;
    }
    *to = '\0';               /* terminate string */
}

int main() {

    char stra[] = "Hello world!";
    char strb[100];

    string_copy(strb, stra);  /* copy stra to strb */

    return 0;

}
```

Listing 6.28: *Functie voor het kopieren van een string.*

dus *niet* de hele array meegegeven, maar alleen de pointers naar de eerste elementen. We mogen daarom de pointers `to` en `from` ook als namen van array's beschouwen. Zie listing 6.29. Merk op dat we dus de lengte van de meegegeven array *niet* kunnen uitrekenen met de `sizeof`-operator. Er wordt immers een pointer meegegeven. Dat we toch het einde van een string kunnen bepalen, komt doordat een string wordt afgesloten met een nul-byte.

```c
void string_copy(char to[], char from[]) {

    int i=0;

    while (from[i] != '\0') {    /* while not end of string ... */
        to[i] = from[i];         /* copy character */
        i = i + 1;               /* point to the next character */
    }
    to[i] = '\0';                /* terminate string */
}
```

Listing 6.29: *Functie voor het kopieren van een string.*

Bij het overdragen van een array aan een functie moet expliciet de grootte worden opgegeven. Het is niet mogelijk om *in de functie* de grootte van de array uit te rekenen, er wordt immers een pointer meegegeven. Er is dus een extra parameter nodig waarmee we de grootte opgeven. Bij het aanroepen van de functie rekenen we de grootte uit en geven dit mee. Dit is te zien in listing 6.30. Let erop dat in `main` wél de grootte van de array kan

worden uitgerekend, want daar wordt de array gedeclareerd. We gebruiken twee keer de `sizeof`-operator, want `sizeof` geeft de grootte van een object in bytes. We moeten de grootte van de array in bytes delen door de grootte van één element in bytes.5

```
#include <stdio.h>
#include <stdlib.h>

void printarray(int ary[], int len) {

    int i;

    for (i=0; i<len; i++) {
        printf("%d ", ary[i]);
    }
}

int main(void) {

    int list[] = {1,2,3,4,5,6,7,8,9};
    int length = sizeof(list)/sizeof(list[0]);

    printarray(list, length);

    return 0;
}
```

Listing 6.30: *Meegeven van de grootte van een array.*

CALL BY REFERENCE...

In de programmeerwereld is het gebruikelijk om ondercheid te maken tussen twee manieren van argumentoverdracht: *Call by Value* en *Call by Reference*. Bij Call by Value wordt een kopie van de waarde van een variabele aan de functie meegegeven. Alleen de kopie kan veranderd worden door de functie. De variabele waarvan de kopie is gemaakt kan dus niet op deze manier veranderd worden. Bij Call by Reference wordt het adres van de variabele aan de functie meegegeven. Via dit adres is het dus wel mogelijk om de originele variabele te veranderen.

Sommige programmeurs beweren dat Call by Reference eigenlijk niet bestaat. En daar is wat voor te zeggen. Er wordt immers een waarde aan de functie meegegeven en dat is het adres van een variabele. Dit adres kan in de functie veranderd worden maar het originele adres waar de variabele in het geheugen staat wordt niet aangepast. Toch maken programmeurs onderscheid tussen deze twee manieren van argumentenoverdracht.

6.10 Pointer als return-waarde

Een pointer kan ook als return-waarde dienen. In het onderstaande voorbeeld wordt in een string gezocht naar een bepaalde karakter in een string. Als het karakter gevonden is, wordt een pointer naar het karakter teruggegeven. Als het karakter niet wordt gevonden wordt `NULL` teruggegeven. Na het uitvoeren van de functie moet hier op getest worden. Tevens wordt in het begin getest of de pointer naar de string wel een geldige waarde heeft. Geldig wil zeggen dat de pointer niet `NULL` is. Het is *good practice* om altijd te testen of een pointer `NULL` is.

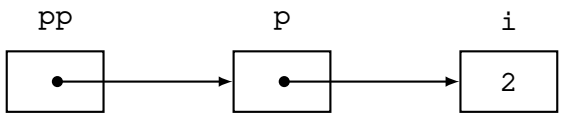
```
1  char *find_token(char *str, char ch) {
2
3      if (str == NULL) {        /* sanity check */
4          return NULL;
5      }
6
7      while (*str != '\0') {    /* while not end of string ... */
8          if (*str == ch) {     /* if character found ... */
9              return str;       /* return pointer */
10         }
11         str++;
12     }
13
14     return NULL;              /* character not found */
15 }
```

Listing 6.31: *Pointer als return-waarde.*

Let goed op de definitie van de functie `find_token`. Voor de functienaam is de dereference-operator geplaatst. Dit betekent dat de functie een pointer naar een karakter teruggeeft. Deze vorm wordt vaak verward met pointers naar functies. Zie paragraaf 6.15.

6.11 Pointers naar pointers

Een pointer kan ook gebruikt worden om naar een andere pointer te wijzen. In figuur 6.8 is te zien dat pointer `p` wijst naar variabele `i`. Met behulp van de dereferentie `*p` kunnen we bij de inhoud van variabele `i` komen. De pointer `pp` wijst naar `p`. We hebben nu een *dubbele dereferentie* nodig om via pointer `pp` bij de inhoud van variabele `i` te komen.



Figuur 6.8: *Uitbeelding van een pointer naar een pointer naar een `int`.*

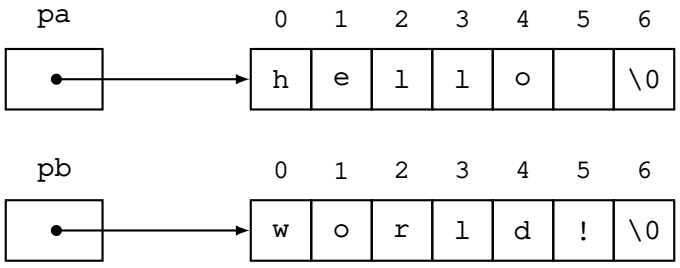
Voor de dubbele dereferentie gebruiken we twee keer de dereference operator `*`. Om toegang te krijgen tot de inhoud van variabele `i` via pointer `pp` gebruiken we dus `**pp`. Dit is te zien in de onderstaande listing.

Met behulp van pointers naar pointers kunnen de inhouden van twee pointers verwisselen. In figuur 6.9 is te zien dat de pointers `pa` en `pb` wijzen naar twee strings in het geheugen.

```c
int i = 2;          /* the integer */
int *p = &i;        /* p points to i */
int **pp = &p;      /* pp points to p */

printf("De waarde is %d\n", **pp);
```

Listing 6.32: *Voorbeeld van een pointer naar een pointer.*

Als we nu de strings willen “verwisselen”, hoeven we alleen maar de pointers er naartoe te verwisselen.



Figuur 6.9: *Uitbeelding van pointers naar strings.*

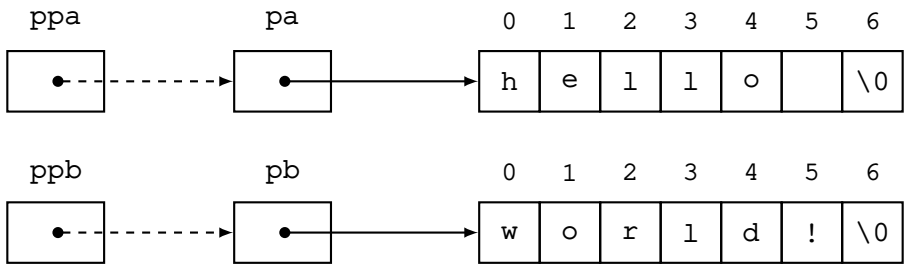
We kunnen dat doen met het onderstaande programmafragment. We declaren drie pointers en laten `pa` en `pb` wijzen naar strings. We gebruiken de pointer `temp` om de verwisseling tot stand te brengen.

```c
char *pa = "hello ";
char *pb = "world!";
char *temp;

temp = pa;            /* swap pa and pb */
pa = pb;
pb = temp;
```

Listing 6.33: *Verwisselen van twee pointers.*

Maar stel dat we zulke verwisselingen vaker in een programma moeten uitvoeren. Dan is het handig om een functie te gebruiken die dat voor ons doet. We geven aan de functie de adressen van de pointers mee zodat de functie ze kan verwisselen. Hoe dit eruit ziet, is te zien in figuur 6.10. De pointers `pa` en `pb` wijzen naar de strings. In de functie zijn twee pointers gedefinieerd die wijzen naar pointers naar strings. Dus `ppa` wijst naar pointer `pa` en `ppb` wijst naar pointer `pb`. We kunnen nu in de functie de inhouden van `pa` en `pb` verwisselen.

In listing 6.34 is de functie te zien voor het verwisselen van twee pointers. De functie heeft twee parameters `ppa` en `ppb` die een pointer zijn naar een pointer naar een string. In de functie declareren we een pointer `temp` die een pointer is naar string (of eigenlijk: een karakter). Met behulp van de dereference `*ppa` kopieren we de inhoud van pointer `pa` naar `temp`. Daarna kopieren we `pb` naar `pa` en als laatste kopieren we `temp` naar `pb`.



Figuur 6.10: *Uitbeelding van pointers naar pointers naar strings.*

```c
#include <stdio.h>
#include <stdlib.h>

void swapstr(char **ppa, char **ppb) {

    char *temp;

    temp = *ppa;       /* copy pa into temp */
    *ppa = *ppb;       /* copy pb into pa */
    *ppb = temp;       /* copy temp into pb */
}

int main()
{
    char *pa = "hello ";
    char *pb = "world!";

    printf("%s%s\n", pa, pb);
    swapstr(&pa, &pb);
    printf("%s%s\n", pa, pb);
    return 0;
}
```

Listing 6.34: *Functie voor het verwisselen van twee pointers.*

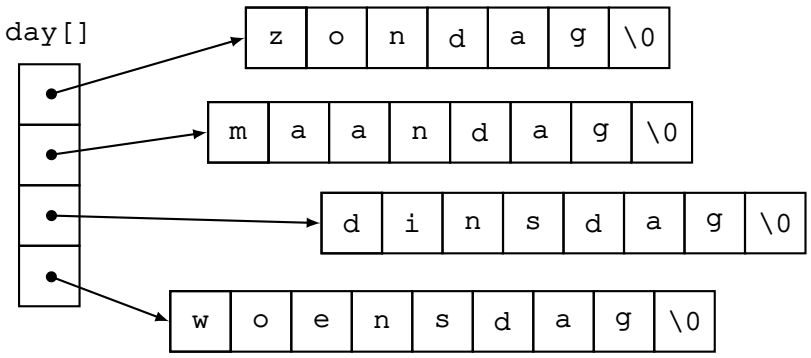
De uitvoer van dit programma is te zien in de onderstaande figuur.

Command Prompt

```
hello world!
world!hello
```

6.12 Array van pointers

Uiteraard kunnen we ook een array van pointers maken. We demonstreren dat aan de hand van een array van pointers naar karakters. Omdat het pointers zijn, kan een pointer ook wijzen naar het begin van een array van karakters, oftewel strings. Dit is te zien in figuur 6.11. Merk op dat de vier pointers naar karakters wijzen. Dat daar toevallig vier strings aan gekoppeld zijn, is vanuit het perspectief van de pointers niet belangrijk.



Figuur 6.11: *Voorstelling van een array van poiinters naar strings.*

We declareren een array van vier pointers naar karakters. Daarna laten we de pointers naar strings wijzen. Zie listing 6.35.

```
1 char *day[4];          /* array of four pointers to char */
2
3 day[0] = "zondag";     /* points to the 'z' */
4 day[1] = "maandag";    /* points to the 'm' */
5 day[2] = "dinsdag";    /* points to the 'd' */
6 day[3] = "woensdag";   /* points to the 'w' */
```

Listing 6.35: *Een array van pointers.*

Omdat dit soort toekenningen veel voorkomen, mogen de strings ook bij declaratie aan de pointers worden toegekend. Dit is te zien in listing 6.36.

```
1 char *day[4] = {"zondag", "maandag", "dinsdag", "woensdag"};
```

Listing 6.36: *Een array van pointers naar strings met initialisatie.*

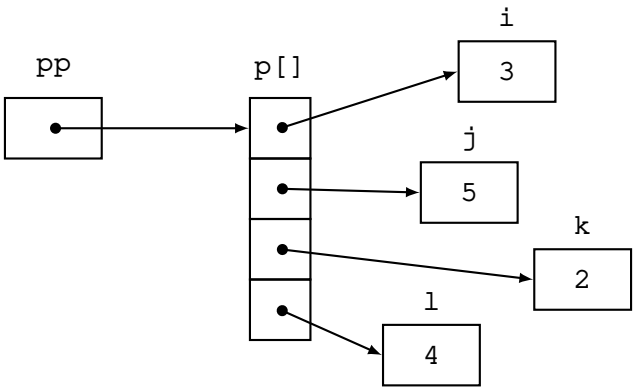
Opmerking: de C++-standaard verbiedt het gebruik van dit soort declaraties en initialisaties. Veel compilers geven echter een waarschuwing en gaan gewoon verder. Omdat de strings in dit soort constructies meestal niet veranderen, kan het keyword `const` voor de declaratie gezet worden, waarmee wordt aangegeven dat de strings niet veranderen. C++-compliers accepteren deze constructie.

6.13 Pointers naar een array van pointers

We bekijken nu een wat complexer voorbeeld van het gebruik van pointers. We declareren een array `p` van vier pointers naar integers. We vullen de array met de adressen van de integers `i`, `j`, `k` en `l`. Daarna declareren we een pointer `pp` die wijst naar (het eerste element van) de array. Een voorstelling van de variabelen is te zien in figuur 6.12.

De array `p` wordt gedeclareerd als:

```
int *p[4];
```



Figuur 6.12: *Voorstelling van een pointer naar een array van pointers naar integers.*

De rechte haken hebben een hogere prioriteit dan de dereference-operator. We lezen de declaratie dus als: p is een array van vier elementen en elk element is een een pointer die wijst naar een integer. De pointer pp wordt gedeclareerd als:

```
int **pp;
```

Let goed op wat hier staat: pp is een pointer naar een pointer naar een integer. Vanuit pointer pp is niet af te leiden dat pp wijst naar een array, alleen maar dat er twee dereferenties nodig zijn om bij een integer te komen. We moeten dat in het C-programma zelf scherp in de gaten houden.

We gebruiken de pointers zoals te zien is in listing 6.37. in regel 6 worden de vier integers gedeclareerd. In regel 7 declareren we array p en initialiseren de array met de adressen van de integers. We geven geen array-grootte op want de C-compiler kan dat zelf uitrekenen.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5
6      int i=3, j=5, k=2, l=4;
7      int *p[] = {&i, &j, &k, &l};
8      int **pp = p;
9
10     printf("&p:␣%p,␣&pp:␣%p,␣&p[0]:␣%p,␣&p[1]:␣%p,␣&p[2]:␣%p,␣&p[3]:
           ␣%p\n\n", &p, &pp, &p[0], &p[1], &p[2], &p[3]);
11
12     printf("&i:␣%p,␣p[0]:␣%p\n", &i, p[0]);
13     printf("&j:␣%p,␣p[1]:␣%p\n", &j, p[1]);
14     printf("&k:␣%p,␣p[2]:␣%p\n", &k, p[2]);
15     printf("&l:␣%p,␣p[3]:␣%p\n", &l, p[3]);
16
17     printf("\ni:␣%d,␣*p[0]:␣%d,␣**pp:␣%d\n", i, *p[0], **pp);
18
19     return 0;
20 }
```

Listing 6.37: *Voorbeeld van het gebruik van pointers.*

In regel 8 declareren we de pointer `pp` (let op het gebruik van de dubbele dereferentie-operator) en initialiseren `pp` met het adres van `p`. In regel 10 drukken alle adressen van de pointers af. In de regels 12 t/m 15 drukken we de adressen van de integers en de inhouden van de array-elementen af. Om variabele `i` af te drukken hebben we drie mogelijkheiden: `i` (de variabele), `*p[0]` (waar `p[0]` heen wijst) en `**pp`. Die wijst dus via dubbele dereferentie ook naar `i`.

Een mogelijke uitvoer is te zien in de onderstaande figuur. We zeggen hierbij mogelijk omdat het draaien van het programma op een computer andere waarden (adressen) kan opleveren. In de onderstaande figuur is te zien dat pointer met acht hexadecimale cijfers worden afgedrukt. Dat betekent dat de pointers met 32 bits worden opgeslagen. We hebben gebruik gemaakt van een 32-bits C-compiler.

Command Prompt

```
&p: 0061FDF0, &pp: 0061FDE8, &p[0]: 0061FDF0, &p[1]: 0061FDF8,
&p[2]: 0061FE00, &p[3]: 0061FE08

&i: 0061FE1C, p[0]: 0061FE1C
&j: 0061FE18, p[1]: 0061FE18
&k: 0061FE14, p[2]: 0061FE14
&l: 0061FE10, p[3]: 0061FE10

i: 3, *p[0]: 3, **pp: 3
```

6.14 Argumenten meegeven aan een C-programma

In besturingssystemen die C ondersteunen zoals Windows, Linux en Mac OS-X, is het mogelijk om een C-programma *command line argumenten* mee te geven. Bij het starten van een programma kunnen we (optioneel) gegevens invoeren en overdragen aan een gecompileerd C-programma. Als voorbeeld starten we het programma `myprog.exe` met de argumenten `argument1` en `argument2`. Het programma drukt eenvoudigweg alle argumenten op het beeldscherm af. Te zien is dat ook de programmanaam als argument wordt meegegeven.

Command Prompt

```
C:\Users\C> myprog.exe argument1 argument2

Aantal argumenten: 3

Argument 0: myprog.exe
Argument 1: argument1
Argument 2: argument2

C:\Users\C>
```

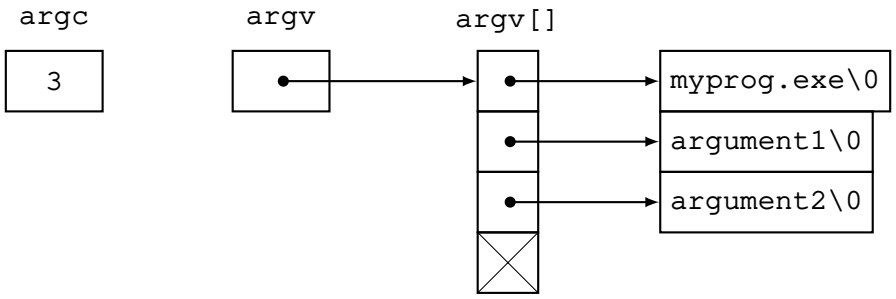
Elk C-programma krijgt per definitie de twee parameters `argc` en `argv` mee, die aan `main` worden meegegeven. Dit is te zien in listing 6.38.

```c
int main(int argc, char *argv[]) {

    /* rest of the code */

    return 0;
}
```

Listing 6.38: *Declaratie van de command line parameters.*

De integer `argc` (**arg**ument **c**ount) geeft aan hoeveel parameters aan het C-programma zijn meegegeven. De pointer `argv` (**arg**ument **v**ector) is een pointer naar een lijst van pointers naar strings, gedeclareerd als `*argv[]`. Elke string bevat een argument. Per definitie wijst `argv[0]` naar een string waarin de programmanaam vermeld staat. Dat houdt in dat `argc` dus minstens 1 is. Er zijn dan geen optionele argumenten meegegeven.

In het voorbeeldprogramma is `argc` dus 3 en zijn `argv[0]`, `argv[1]` en `argv[2]` pointers naar respectievelijk `myprog.exe`, `argument1` en `argument2`. In figuur 6.13 is een uitbeelding van de variabelen `argc` en `argv` te zien. De strings worden, zoals gebruikelijk in C, afgesloten met een nul-karakter. De C-standaard schrijft voor dat de lijst van pointers naar strings wordt afgesloten met een NULL-pointer.



Figuur 6.13: *Voorstelling van de variabelen `argc` en `argv`.*

Het programma `myprog.exe` is te zien in listing 6.39. Het programma drukt eerst de variabele `argc` af. Met behulp van een `for`-lus worden de argumenten één voor één afgedrukt. Merk op dat `argv[i]` een pointer is naar het i^e argument. We kunnen `argv[i]` dus direct gebruiken voor het afdrukken van de bijbehorende string.

Merk op dat `argv` een echte pointers is en niet de naam van een array. We mogen `argv` dus aanpassen. Dit is te zien in listing 6.40.

6.15 Pointers naar functies

Functies zijn stukken programma die ergens in het geheugen liggen opgeslagen. De naam van een functie is het adres van de eerste instructie van de functie. Het is dus mogelijk om de naam van een functie te gebruiken als een pointer. Dit worden *functiepointers* genoemd.

In listing 6.41 is te zien hoe een functiepointer wordt gedeclareerd. In de eerste regel wordt een functie gedefinieerd, een zogenoemde prototype, die een `int` als parameter meekrijgt

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int i;

    printf("\nAantal␣argumenten:␣%d\n\n", argc);
    for (i = 0; i<argc; i++) {
        printf("Argument␣%d:␣%s\n", i, argv[i]);
    }
    return 0;
}
```

Listing 6.39: *Het programma myprog.exe.*

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

    printf("\nArguments:␣");
    while (argc>0) {
        printf("%s␣", *argv);
        argv = argv + 1;
        argc = argc - 1;
    }
    return 0;
}
```

Listing 6.40: *Afdrukken van argumenten.*

en een `int` teruggeeft. In de tweede regel wordt `pf` gedeclareerd als een pointer naar een functie die een `int` meekrijgt en een `int` teruggeeft.

```c
int func(int a);        /* func is a function returning an int */
int (*pf)(int a);       /* pf is a pointer to a function
                                        returning an int */

pf = func;              /* assign pf */
```

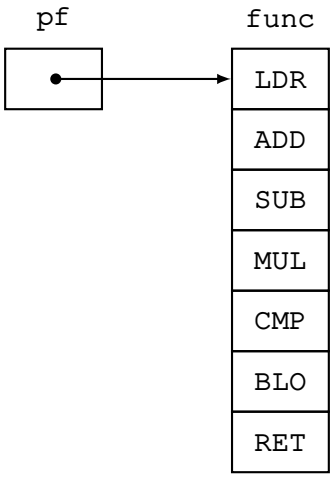
Listing 6.41: *Een functie en een pointer naar een functie.*

Let op het gebruik van de haakjes. Die zijn nodig om prioriteiten vast te leggen. Zonder de haken staat er `int *pf(int a)` en dan is `pf` een functie die een `int` als parameters meekrijgt en een pointer naar een `int` teruggeeft. Zie listing 6.42.

```
1 int (*pf)(int a);    /* pf: pointer to a function returning an int */
2 int *pf(int a);      /* pf: function returning a pointer to an int */
```

Listing 6.42: *Een functie en een pointer naar een functie.*

In figuur 6.14 is te zien hoe `pf` wijst naar de functie `func`. We kunnen nu `pf` gebruiken om de functie aan te roepen. In de figuur is een aantal instructies gezet.



Figuur 6.14: *Voorstelling van een pointer naar een functie.*

Het gebruik van functie-pointers komt niet zo vaak voor in C-programma's. Het is meer in gebruik bij het schrijven van besturingssystemen. We willen toch even twee functies de revue laten passeren waarbij functie-pointers gebruikt worden.

Quicksort

Quicksort is een sorteeralgoritme ontworpen door C.A.R. Hoare in 1962. Het is een van de efficientste sorteeralgoritmes voor algemeen gebruik. De exacte werking zullen we niet bespreken, er zijn genoeg boeken die dit beschrijven. De standaard C-bibliotheek bevat een implementatie onder de naam `qsort`. Het prototype van `qsort` is:

```
void qsort(void *base, size_t nitems, size_t size,
                       int (*compar)(const void *, const void*));
```

Hierin is `base` een pointer naar het eerste element van de array, `nitems` het aantal elementen in de array en `size` de grootte (in bytes) van één element. De functie `compar` behoeft wat speciale aandacht. Dit is een functie (door de ontwerper zelf te schrijven) die twee pointers naar twee elementen in de array meekrijgt. De pointer zijn van het type `void *` want we weten niet wat het datatypes van de elementen van de array zijn. De functie geeft een getal kleiner dan 0 als het eerste argument kleiner is dan het tweede element, 0 als de twee elementen gelijk zijn en een getal groter dan 0 als het eerste element groter is dan het tweede element.

In listing 6.43 is een programma te zien dat een array van integers sorteert. De functie `cmpint` vergelijkt twee integers uit de array. Let op de constructie om bij de integers te

```c
#include <stdio.h>
#include <stdlib.h>

int values[] = { 88, 56, 100, 2, 25 };

int cmpint (const void * a, const void * b) {
    /* explicit type case to pointers to integers */
    return ( *(int*)a - *(int*)b );
}

int main () {
    int n;

    printf("De array voor sorteren: \n");
    for( n = 0 ; n < 5; n++ ) {
        printf("%d ", values[n]);
    }

    qsort(values, 5, sizeof(int), cmpint);

    printf("\nDe array na sorteren: \n");
    for( n = 0 ; n < 5; n++ ) {
        printf("%d ", values[n]);
    }

    return(0);
}
```

Listing 6.43: *Sorteren van een array met quicksort.*

komen. De pointers zijn van het type `void *` dus er is een expliciete type case nodig naar een pointer naar een integer. Dat wordt gerealiseert door `(int *)`. Daarna wordt de pointer gebruikt om bij de integer te komen. Om parameter `a` te verkrijgen is dus `*(int *) a` nodig. Op deze wijze wordt ook parameter `b` gevonden en de functie geeft eenvoudigweg het verschil tussen de twee parameters terug.

6.16 Pointers naar vaste adressen

C is erg coulant bij het toekennen van adressen aan pointers. Zo is het mogelijk om een pointer naar een vast adres te laten wijzen. De toekenning en initialisatie

```c
int *p = (int *)0x40fe;
```

zorgt ervoor dat `p` een pointer is naar een integer op adres $40FE_{16}$ (hexadecimale notatie). Er is een expliciete type cast nodig om de integer `0x40fe` om te zetten naar een adres. Nu zullen dit soort toekenningen niet voorkomen op systemen waar een operating system op draait. Het gebruik van de pointer zal hoogst waarschijnlijk een crash van het programma veroorzaken. Maar op kleine computersystemen zonder operating system, de zogenoemde *bare metal*-systemen, is het vaak de enige manier om informatie naar binnen en naar buiten te krijgen. We geven hieronder een voorbeeld van het gebruik van dit soort pointers op een ATmega-microtroller van Atmel.

```
1 /*    Moet ik nog invullen */
```

Listing 6.44: *Het gebruik van pointers op een ATmega-microcontroller.*

6.17 Subtiele verschillen en complexe declaraties

We zullen in de praktijk nauwelijks complexere situaties tegenkomen dan dat we tot nu toe zijn tegengekomen. Toch willen we een bloemlezing geven van enkele bekende en onbekende, complexe declaraties.

`int *p`	`p`: pointer to int
`int **pp`	`pp`: pointer to pointer to int
`int ***ppp`	`ppp`: pointer to pointer to pointer to int
`int **pp[3]`	`pp`: array[3] of pointer to pointer to int
`char **argv`	`argv`: pointer to pointer to char
`char *argv[]`	`argv`: array[] of pointer to char
`int *list[5]`	`list`: array[5] of pointer to int
`int (*list)[5]`	`list`: pointer to array[5] of int
`int *(*list)[5]`	`list`: pointer to array[5] of pointer to int
`int *pf()`	`pf`: function returning a pointer to int
`int (*pf)()`	`pf`: pointer to function returning an int
`int *(*pf)()`	`pf`: pointer to function returning a pointer to int
`char (*(*x())[])()`	`x`: function returning pointer to array[] of pointer to function returning char.
`char (*(*x[3])())[5]`	`x`: array[3] of pointer to function returning pointer to array[5] of char.

Index