

C

Een inleiding in de programmeertaal C

Jesse op den Brouw

Eerste druk

©2020 Jesse op den Brouw, Den Haag

Versie: 0.1

Datum: 27 april 2020

DE HAAGSE
HOGESCHOOL

De auteur kan niet aansprakelijk worden gesteld voor enige schade, in welke vorm dan ook, die voortvloeit uit informatie in dit boek. Evenmin kan de auteur aansprakelijk worden gesteld voor enige schade die voortvloeit uit het gebruik, het onvermogen tot gebruik of de resultaten van het gebruik van informatie in dit boek.

Dit boek mag overal voor gebruikt worden, commercieel en niet-commercieel, mits de wijzigingen worden gedeeld en naam van de originele auteur wordt gemeld. De broncode van dit boek is beschikbaar op https://github.com/jesseopdenbrouw/book_c.



C van Jesse op den Brouw is in licentie gegeven volgens een [Creative Commons Naamsvermelding-NietCommercieel-GelijkDelen 3.0 Nederland-licentie](#).

Suggesties en/of opmerkingen over dit boek kunnen worden gestuurd naar:
J.E.J.opdenBrouw@hhs.nl.

Voorwoord

Een van de beste boeken over C is *The C Programming Language* van Brian Kernighan en Dennis Ritchie [1]. Het boek is kort en bondig maar laat toch ruimte om de tekst te ondersteunen met voorbeelden. Helaas zijn veel van de voorbeelden lastig te volgen voor de beginnende programmeur. Zoals de schrijvers zelf opmerken:

The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. [...] C is not a big language, and it is not well served by a big book.

Dit boek is bedoeld als inleiding op de programmeertaal C en is niet toereikend om alle facetten van de taal te leren. Dat is ook niet de bedoeling van dit boek. We laten alleen maar zien wat gebruikelijk is bij het ontwikkelen van C-programma's.

Veel boeken over C beschrijven het ontwikkelen van programma's op het Unix operating system en afgeleide varianten zoals Linux, FreeBSD en Mac OS-X. Dat is echter niet de werkwijze van veel programmeurs. Veel software wordt ontwikkeld met een Integrated Development Environment (IDE). Bekende IDE's zijn Visual Studio, Code:Blocks en Xcode. Het is natuurlijk niet mogelijk om alle mogelijkheden van zulke IDE's te beschrijven (of af te beelden).

Dit boek is opgemaakt in L^AT_EX [2] (L^AT_EX-engine = pdfL^AT_EX 1.40.21). L^AT_EX leent zich uitstekend voor het opmaken van lopende tekst, tabellen, figuren, programmacode, vergelijkingen en referenties. De gebruikte L^AT_EX-distributie is TexLive uit 2020 [3]. Als editor is TexStudio [4] gebruikt. Tekst is gezet in Charter [5], een van de standaard fonts in L^AT_EX. De keuze hiervoor is dat het een prettig te lezen lettertype is, een *slanted* letterserie heeft en een bijbehorende wiskundige tekenset heeft. Code is opgemaakt in Nimbus Mono [6] met behulp van de *listings*-package [7]. Voor het tekenen van array's, pointers en flowcharts is *TikZ/PGF* gebruikt [8]. Alle figuren zijn door de auteur zelf ontwikkeld, behalve de logo's van Creative Commons en De Haagse Hogeschool.

Natuurlijk zullen docenten ook nu opmerken dat dit boek niet aan al hun verwachtingen voldoet. Dat zal altijd wel zo blijven. Daarom wordt de broncode van vrij beschikbaar. Eenieder die dit wil, kan de inhoud vrijelijk aanpassen, mits wijzigingen gedeeld worden. Hiermee wordt dit boek een levend document dat nooit af is. De broncode van dit boek is beschikbaar op https://github.com/jesseopdenbrouw/book_c.

Leeswijzer

Hoofdstuk 1 geeft in vogelvlucht enkele belangrijke concepten van C weer. Er wordt gesproken over wat een computer en een compiler is en hoe een uitvoerbaar bestand wordt gegenereerd. We beginnen met het geven van voorbeelden die de lezer in ontwikkelsoftware kan invoeren om zo de uitvoer te bekijken. We beschrijven kort hoe uitvoer naar beeldscherm en invoer van het toetsenbord wordt gerealiseerd. We bespreken het concept van variabelen datatypes. Verder wordt even stilgestaan bij beslissen en herhalen, array's en functie. In dit hoofdstuk worden geen array's, pointers en structures behandeld.

Hoofdstuk 2 gaat dieper in op variabelen, constanten en expressies. Alle beschikbare datatypes worden behandeld als mede enkele vaste-lengte datatypes. Er wordt uitgelegd wat een expressie is en hoe expressies kunnen worden gebruikt bij berekeningen en beslissingen. C kent een aantal eigenaardige taalconstructies zoals de increment en decrement operatoren, de conditionele expressie en de komma-operator. Een aantal operatoren wordt in dit hoofdstuk niet besproken zoals de adres- en dereferentie-operatoren, de element-operator en de member-operatoren. Die volgen in de verdere hoofdstukken.

Hoofdstuk 3 laat zien hoe de executie van een programma kan worden beïnvloed met beslissingen en herhalingen.

Hoofdstuk 4 is een hoofdstuk over het documenteren van een (deel van een) programma met behulp van flowcharts en ligt in het verlengde van hoofdstuk 3. De symbolen worden uitgelegd en getoond wordt hoe bepaalde taalconstructies op grafische wijzen kunnen worden beschreven.

Hoofdstuk 5 beschrijft het gebruik van functies om een programma op te delen in kleine eenheden die uitgevoerd kunnen worden. We laten zien hoe een functie gegevens meekrijgt en hoe gegevens worden teruggegeven. Verder laten we zien hoe functies zichzelf kunnen aanroepen: de recursieve functie. We hebben een paragraaf opgenomen over de *computational complexity* van een recursieve variant van de reeks van Fibonacci.

Hoofdstuk 6 gaat over array's.

Hoofdstuk 7 gaat over pointers. Er wordt uitgelegd wat een pointer is en wat de relatie met array's is. Verder wordt beschreven hoe pointers een rol spelen in het doorgeven van veel informatie aan functies. Pointerstructuren kunnen bijzonder complex zijn en we schuwen dan ook niet om pointers-naar-pointers te behandelen. Het gebruik van command line argumenten wordt uitgelegd zodat een uitvoerbaar programma kan worden gestart met optionele argumenten.

Hoofdstuk 8 gaat over structures en union die handig zijn bij het gebruik van complexe datastructuren. We leggen uit wat een structure is en hoe de gegevens binnen een structure kunnen worden benaderd. Structuren kunnen als argumenten en returnwaarde van functies dienen maar als de structures erg groot zijn is het handiger om pointers te gebruiken. We beschrijven kort de union als een probaat middel om de hoeveelheid gebruikte geheugen in te dammen (althoewel dat bij moderne PC's eigenlijk geen rol speelt). Bitfields worden niet besproken.

Studiewijzer

In onderstaande tabel wordt een overzicht gegeven van de stof die een student bij een eerste introductie onderwezen zou moeten krijgen. Uiteraard is iedereen vrij om zelf de onderwerpen te kiezen.

Tekst

Verantwoording inhoud

Dit boek voldoet alleen aan aandachtspunt 4,01 van de basis Basic of Knowledge and Skills (BoKS) Elektrotechniek. De BoKS is te vinden via [9].

Website

Op de website <http://ds.opdenbrouw.nl> zijn slides, practicumopdrachten en aanvullende informatie te vinden. De laatste versie van dit boek wordt hierop gepubliceerd. Er zijn ook voorbeeldprojecten voor Visual Studio en Code::Blocks te vinden.

Dankbetuigingen

Dit boek had niet tot stand kunnen komen zonder hulp van een aantal mensen. Ik wil collega Harry Broeders van Hogeschool Rotterdam bedanken voor zijn geweldige bijdrage. Niet alleen op technisch gebied, maar ook taalkundig en op de indeling van dit boek. Collega Ben Kuiper heeft een waardevolle bijdrage geleverd op technisch en taalkundig gebied.

De ASCII-tabel op pagina 128 is ontwikkeld door Victor Eijkhout. Deze tabel kan gevonden worden op <http://www.ctan.org/tex-archive/info/ascii-chart>.

Jesse op den Brouw, 2020.

Inhoudsopgave

| | |
|--|------------|
| Voorwoord | iii |
| 1 Un tour de C | 1 |
| 1.1 De computer | 2 |
| 1.2 De compiler | 3 |
| 1.3 Bibliotheken en functies | 4 |
| 1.4 Ontwikkelssystemen | 4 |
| 1.5 Een minimaal C-programma | 6 |
| 1.6 Afdrukken op het scherm | 7 |
| 1.7 Invoer van het toetsenbord | 8 |
| 1.8 Keywords | 10 |
| 1.9 Beslissingen | 10 |
| 1.10 Herhalingen | 12 |
| 1.11 Array's | 13 |
| 1.12 Functies | 14 |
| 1.13 En verder... | 15 |
| 2 Variabelen, datatypes en expressies | 17 |
| 2.1 Variabelen | 17 |
| 2.2 Datatypes | 18 |
| 2.3 Constanten | 20 |
| 2.4 Expressies | 23 |
| 2.4.1 Rekenkundige operatoren | 23 |
| 2.4.2 Relationale operatoren | 24 |
| 2.4.3 Logische operatoren | 24 |
| 2.4.4 Bitsgewijze operatoren | 25 |
| 2.5 Datatypeconversie | 27 |
| 2.6 Overflow | 28 |
| 2.7 Vaste-lenge datatypes | 28 |
| 2.8 Enumeraties | 29 |
| 2.9 Overige operatoren | 30 |
| 2.9.1 Grootte van een variabele | 30 |
| 2.9.2 Verhogen of verlagen met 1 | 30 |
| 2.9.3 Toekenningsoperatoren | 31 |
| 2.9.4 Conditionele expressie | 31 |
| 2.9.5 Komma-operator | 32 |

| | | |
|----------|---|-----------|
| 2.9.6 | Nog enkele operatoren | 32 |
| 2.10 | Voorrangsregels van operatoren | 32 |
| 3 | Beslissen en herhalen | 35 |
| 4 | Flowcharts | 37 |
| 4.1 | if-statement | 38 |
| 4.2 | if-else-statement | 39 |
| 4.3 | while-lus en do-while-lus | 39 |
| 4.4 | for-lus | 41 |
| 4.5 | switch-statement | 42 |
| 4.6 | Voorbeeld | 43 |
| 5 | Functies | 45 |
| 5.1 | Een eenvoudige functie | 46 |
| 5.2 | Functies met parameters | 47 |
| 5.3 | Functies met een returnwaarde | 51 |
| 5.4 | Zichtbaarheid en levensduur van lokale variabelen | 55 |
| 5.5 | Rekursieve functies | 57 |
| 5.6 | Complexiteit van recursieve functies | 60 |
| 5.7 | Pointers en functies als parameter | 63 |
| 5.8 | Functies die meer dan één waarde teruggeven | 63 |
| 6 | Array's | 65 |
| 6.1 | Declaratie en selectie | 65 |
| 6.2 | Initialisatie | 66 |
| 6.3 | Eendimensionale array | 67 |
| 6.4 | Aantal elementen van een array bepalen | 68 |
| 6.5 | Tweedimensionale array | 69 |
| 6.6 | Afdrukken van array | 70 |
| 6.7 | Array's en functies | 71 |
| 6.8 | Array's vergelijken | 73 |
| 6.9 | Strings | 74 |
| 6.9.1 | Stringfuncties | 75 |
| 7 | Pointers | 77 |
| 7.1 | Pointers naar enkelvoudige datatypes | 78 |
| 7.2 | De NULL-pointer | 80 |
| 7.3 | Pointer naar void | 81 |
| 7.4 | Afdrukken van pointers | 81 |
| 7.5 | Pointers naar array's | 82 |
| 7.6 | Strings | 84 |
| 7.7 | Rekenen met pointers | 85 |
| 7.8 | Relatie tussen pointers en array's | 86 |
| 7.9 | Pointers als functie-argumenten | 87 |
| 7.10 | Pointer als return-waarde | 90 |
| 7.11 | Pointers naar pointers | 91 |
| 7.12 | Array van pointers | 94 |

| | | |
|----------|--|------------|
| 7.13 | Pointers naar een array van pointers | 95 |
| 7.14 | Argumenten meegeven aan een C-programma | 96 |
| 7.15 | Pointers naar functies | 98 |
| 7.16 | Dynamische geheugenallocatie | 100 |
| 7.17 | Pointers naar vaste adressen | 103 |
| 7.18 | Subtiele verschillen en complexe declaraties | 103 |
| 8 | Structures | 105 |
| 8.1 | Definitie en declaratie | 105 |
| 8.2 | Toegang tot members | 106 |
| 8.3 | Functies met structures | 106 |
| 8.4 | Typedef | 107 |
| 8.5 | Pointers naar structures | 108 |
| 8.6 | Array van structures | 110 |
| 8.7 | Structures binnen structures | 110 |
| 8.8 | Teruggeven van meerdere variabelen | 112 |
| 8.9 | Unions | 114 |
| 9 | Invoer en uitvoer | 117 |
| 9.1 | Inlezen van een getal met scanf | 117 |
| 9.1.1 | Problemen met scanf | 117 |
| | Bibliografie | 125 |
| A | De ASCII-tabel | 127 |
| B | Tutorial Visual Studio | 129 |
| C | Vorrangsregels van operatoren | 135 |
| | Index | 137 |

1

Un tour de C

C is al een oude taal. De taal is rond 1970 ontworpen door Dennis Ritchie¹ en is dus al zo'n 50 jaar oud. Hij was bezig met het schrijven van een besturingssysteem (Engels: operating system)² en zocht naar een mogelijkheid om dit te schrijven op een hoger niveau dat gebruikelijk voor die tijd. In die tijd werden besturingssystemen geschreven in assembly, een taal die dicht de hardware van de computer ligt. Programma's schrijven in assembly is tijdrovend en gevoelig voor fouten van de programmeur.

C is een zogenoemde derde-generatie-taal (3GL) en zorgt ervoor dat programma's gestructureerd kunnen worden geschreven zonder dat de programmeur kennis hoeft te hebben van de hardware waarop het programma draait. Toch biedt de taal constructies om die hardware in te stellen, een voorwaarde voor het schrijven van een operating system. Daarom is de taal geliefd bij programmeurs van kleine computersystemen waarop geen operating system draait, de zogenoemde *bare metal systems*. Het is vaak ook de enige taal die gebruikt kan worden op dit soort systemen, naast assembly.

C is geschreven voor ervaren programmeurs die behoefte hebben voor het schrijven van compacte programma's dat te zien is in de vele, soms onduidelijke, taalconstructies. Het is eigenlijk niet geschikt voor beginnende programmeurs. Het is zeker mogelijk om programma's te schrijven die niet correct werken, maar met enige discipline kan ook de minder ervaren programmeur de taal prima gebruiken.

C is een algemeen bruikbare taal (*general purpose language*) en is dus niet voor een specifiek doeleind ontworpen (behalve voor het schrijven van operating systems). Zo kunnen met C wiskundige berekeningen worden uitgevoerd maar de programmeur moet zelf alles schrijven. Er zijn wel *functies* beschikbaar voor bijvoorbeeld sinus, cosinus en tangens. Er

¹ Dennis MacAlister Ritchie (1941 – 2011). Hij was de ontwerper van de programmeertaal C en was een van de ontwerpers van Unix. Bekende afgeleiden van Unix zijn Linux en FreeBSD.

² Een besturingssysteem is een programma dat draait op een computer en zorgt voor het beschikbaar stellen van *resources* aan de gebruiker (lees: programma's). Zo zorgt een besturingssysteem ervoor dat de bestanden op de harde schijf netjes worden opgeslagen en beschikbaar zijn. Daarnaast zorgt een besturingssysteem ervoor dat programma's netjes naast elkaar kunnen draaien. Drie bekende besturingssystemen zijn Windows, Linux en OS-X.

zijn echter talen die dit veel beter ondersteunen. Ook het verwerken van bestanden is mogelijk maar het manipuleren van *strings* (een rij karakters) moet door de programmeur zelf worden uitgewerkt. Ook hier zijn diverse functies beschikbaar die het programmeerwerk enigszins verlichten.

C is een *imperatieve* programmeertaal, een van de bekende programmeerparadigma's³. C moet concurreren met vele andere talen zoals C++, C#, Python en Java. C++ is ontwikkeld door Bjarne Stroustrup als “de betere C” en ondersteunt *objectgeoriënteerd* programmeren. C# is ontwikkeld door Microsoft en is de *de facto* programmeertaal op Windows-systemen. Python is ontwikkeld door het Centrum voor Wiskunde en Informatica van de Universiteit van Amsterdam.

1.1 De computer

Een *computer* is een elektronisch, digitaal systeem dat *data* verwerkt aan de hand van een lijst *instructies*. Het Engelse woord voor verwerken is “to process”. En daar komt de naam vandaan van een belangrijk onderdeel van de computer: de *processor*.

De instructies die de processor kan uitvoeren zijn zeer eenvoudig. Voorbeelden zijn “tel op” en “trek af” en “bepaal of het ene getal kleiner is dan het andere getal”. Een processor kan niet in één keer “doe dit tien keer en bereken de wortel van diverse getallen en druk af op het scherm” uitvoeren. Als we dat willen, dan moeten we dit opdelen in de eenvoudige instructies die de processor wel kan verwerken. Alle instructies bij elkaar wordt een *programma* genoemd en de verzamelnaam voor alle programma's wordt *software* genoemd.

Een computer heeft naast de processor *geheugen* om de data en instructies op te slaan. Er zijn twee soorten geheugens: ROM (Read Only Memory) is een geheugen waarvan de inhoud niet gewijzigd kan worden; RAM (Random Access Memory) is geheugen waarvan de inhoud wel gewijzigd kan worden. Over het algemeen wordt het programma in de ROM opgeslagen en wordt de data in de RAM opgeslagen.

Naast ROM en RAM heeft de computer nog invoer- en uitvoermogelijkheden, anders is er geen communicatie met de buitenwereld mogelijk. De verzamelnaam is *I/O* dat “input” en “output” betekent. Het is best lastig om de invoer en uitvoer te realiseren. Daarom wordt bij de gangbare computers een stuk software geladen (of het is al aanwezig in de ROM) om dat voor de gebruiker (en programmeur) te vereenvoudigen. Die software wordt een *besturingssysteem* genoemd, maar gangbaar is om de Engelse naam Operating System te gebruiken. Bekende besturingssystemen zijn Windows, Mac OS-X en Linux.

Niet alle gegevens zijn in het geheugen van de computer aanwezig. Een computer heeft (vaak) *secundair geheugen*. Voorbeelden van secundair geheugen zijn harddisk (harde schijf), USB-sticks en SD-cards. De informatie op deze geheugendragers blijft aanwezig ook al wordt de computer uitgezet en weer aangezet. Om de informatie beschikbaar te maken worden de gegevens in *bestanden* (Engels: file) gezet. De manier waarop dat wordt gerealiseerd wordt een *bestandssysteem* (Engels: file system) genoemd. Bekende bestandssystemen zijn NTFS op Windows en ext3 op Linux. Het besturingssysteem zorgt ervoor dat de bestanden op ordentelijke wijze kunnen worden benaderd.

³ Een programmeerparadigma is een manier van programmeren en een wijze waarop een programma wordt vormgegeven.

In de praktijk komen twee soorten computers tegen: de algemeen bruikbare computers waarop een besturingssysteem draait (PC, laptops) en dat een grote hoeveelheid aan verschillende programma's kan uitvoeren en de zogenoemde *embedded systems*. Een embedded system is meestal geschikt om één taak uit te voeren en is voorzien van een kleine processor met geheugen en I/O-faciliteiten. Al deze componenten zijn op één ic (Integrated Circuit of chip genoemd) geplaatst. We noemen zo'n processorsysteem een *microcontroller*. Voorbeelden van embedded systems zijn (de besturingen van) wasmachines, koelkasten, televisies, thermostaten, horloges en bloeddrukmeters. De markt voor embedded systems is trouwens vele malen groter dan de markt voor algemene bruikbare computers.

De microcontroller heeft meestal geen besturingssysteem. Dat noemen we *bare metal* systemen. Dat betekent dat de programmeur alles zelf moet ontwikkelen. Gelukkig leveren fabrikanten ontwikkelsystemen om programma's voor microcontrollers te produceren. Dat worden *compiler suites* of *toolchains* genoemd.

1.2 De compiler

C is een zogenoemde derde-generatie-taal. De broncode van een C-programma is voor een mens gewoon te lezen. Deze broncode kan niet direct door de processor worden uitgevoerd. De *C-compiler* vertaalt de broncode naar instructies die de processor wel kan uitvoeren. Dat vertalen wordt *compileren* genoemd. De instructies worden in bitpatronen in het geheugen opgeslagen. De bitpatronen noemen we *machinecode*. Elk type processor heeft zijn eigen verzameling bitpatronen, dat wordt de *Instruction Set Architecture* genoemd. De ISA van een Intel-processor is dus anders dan die van een ATmega-processor, maar we kunnen wel zeggen dat elke processor ongeveer dezelfde soort instructies bevat. De C-compiler moet weten voor welke processor een C-programma wordt vertaald.

Een C-programma kan niet worden uitgevoerd door de computer. We moeten eerst een programma compileren zodat instructies worden gegenereerd die de processor wel kan uitvoeren. Het vertaalde programma wordt een *uitvoerbaar bestand* of *executable* genoemd. Een uitvoerbaar bestand wordt net als een C-programma opgeslagen op de harddisk. Bij het ontwikkelen van programma's voor microcontrollers wordt gebruik gemaakt van PC of laptop met een besturingssysteem en een toolchain. We noemen zo'n compiler een *cross compiler*. De uitvoerbare instructies worden via de PC in het geheugen van de microcontroller *geprogrammeerd*.

In het licht van het compilatieproces noemen we nog twee begrippen die we vaak zullen gebruiken: *compile-time* en *runtime*. Met compile-time bedoelen we dat iets tijdens compileren van een C-programma bekend moet zijn. Een voorbeeld hiervan is het bepalen van de grootte van een variabele. Met runtime bedoelen we dat tijdens het uitvoeren van een programma iets wordt bepaald, bijvoorbeeld hoe vaak iets moet worden uitgerekend.

Hoewel een derde-generatie-taal suggereert dat we een C-programma door verschillende compilers kunnen laten vertalen en op verschillende computersystemen (lees: processoren) kunnen uitvoeren, moeten we helaas opmerken dat dat niet het geval is. Het is zeker mogelijk om een programma te schrijven dat door de Microsoft C-compiler en door de GNU C-compiler kan worden vertaald. Maar er zijn ook veel verschillen. Zo kan de grootte van gegevens door verschillende compilers op verschillende wijze worden geïnterpreteerd. Ook het bestandssysteem is op computers verschillend. Windows en Linux doen dat op hun eigen

wijze. Het is dus best lastig om een programma te schrijven dat op beide besturingssystemen zonder problemen draait. En wat te denken van een microcontroller die niet eens een bestandssysteem heeft. Daar heeft het bewerken van bestanden geen zinnige betekenis.

1.3 Bibliotheken en functies

Stel dat we een klein stukje C-programma hebben geschreven dat iets voor ons uitrekent, bijvoorbeeld het gemiddelde van een aantal getallen. We willen dit stukje programma vaker gebruiken in een groot C-programma. Dan plaatsen we de berekening in een *functie*. We kunnen nu in ons C-programma de functie meerdere malen *aanroepen*. De functie hoeft dus maar één keer geprogrammeerd te worden om meerdere keren gebruikt te worden.

Er zijn ook functies die al geschreven zijn, zoals het afdrukken van tekst op het scherm. We hoeven dus niet zelf een functie te schrijven die dat voor ons doet. Als deze functies zijn ondergebracht in *bibliotheken* (Engels: library). Als tijdens het compileren zo'n functie nodig is, dan zoekt de compiler in de bibliotheek of de functie beschikbaar is. De compiler “plakt” dan de functie bij het programma. Dat plakken wordt *linken* genoemd.

Bij de C-compiler wordt een zeer uitgebreide bibliotheek geleverd, de zogenoemde *standard library*. De standard library bevat functies voor het afdrukken van tekst en gegevens, het bewerken van bestanden en het manipuleren van *strings* (een stukje tekst opgeslagen in het geheugen). Daarnaast wordt ook de *mathematical library* meegeleverd waarin functies voor het berekenen van sinus, cosinus, logaritmen en e-machten zijn opgeslagen. Tijdens het compileren moet opgegeven worden dat moet worden gezocht in de bibliotheken.

1.4 Ontwikkelssystemen

De meeste boeken gaan uit van een *command line interface* op een Unix-derivaat. Met behulp van een *editor* wordt een programma ingevoerd en wordt op de commandoregel de C-compiler gestart. Daarna wordt het programma (ook via de commandoregel) gestart. Natuurlijk is het ook mogelijk om alles met behulp van de command line te doen. Deze werkwijze wordt veel gebruikt op Linux-systemen maar kan ook op Windows worden gebruikt. Een voorbeeld is te zien in figuur 1.1.

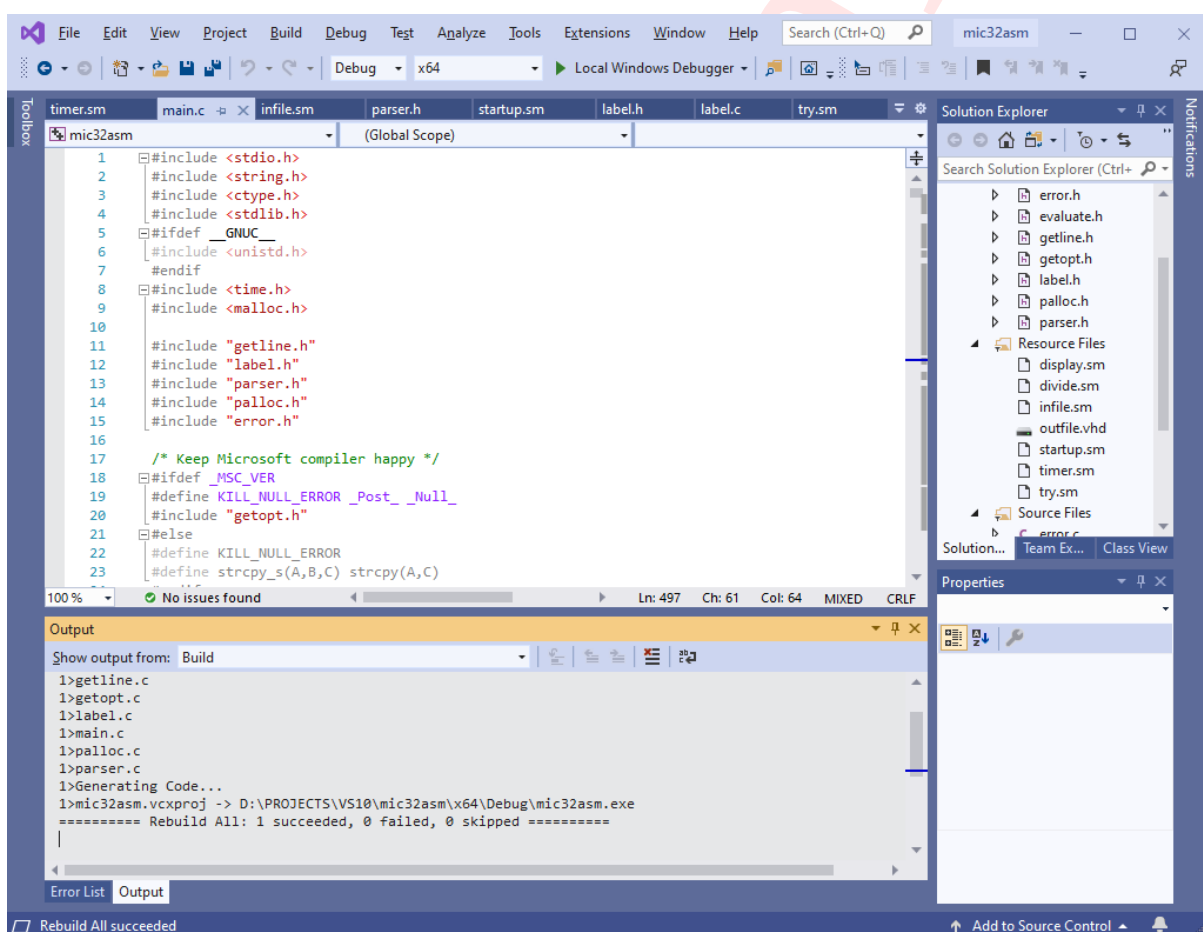
| | |
|------------------------------------|-------------------------------|
| C:\> Command Prompt | |
| C:\Users\C> notepad mooi.c | (start Notepad) |
| C:\Users\C> gcc -o mooi.exe mooi.c | (start C-compiler) |
| C:\Users\C> .\mooi.exe | (start uitvoerbaar programma) |
| C is een mooie taal | (de uitvoer op het scherm) |
| C:\Users\C> | |

Figuur 1.1: Een voorbeeld van een command line interface.

Dit is echter niet de werkwijze van veel programmeurs. Gelukkig zijn er goede ontwikkel-systemen (IDE: Integrated Development Environment) die het programmeerwerk verlichten. Bekende systemen zijn Microsoft Visual Studio, Code::Blocks en Apple's Xcode. Zulke

ontwikkelssystemen zorgen ervoor dat de programmeur gemakkelijk het programma kan invoeren, de compiler kan starten en het programma kan *debuggen*. Dat laatste is vaak nodig omdat blijkt dat (de uitvoer) een programma niet verloopt zoals de programmeur het voor ogen had. Met debuggen wordt het programma stap voor stap doorlopen en kan de programmeur (of is het debugger) de inhoud van *variabelen* bekijken. Ook kan de programmeur bepalen of de *statements* (opdrachten voor de computer) op de juiste volgorde worden uitgevoerd.

Een voorbeeld van Microsoft Visual Studio is te zien in figuur 1.2. Visual Studio ondersteunt het ontwikkelen van software voor Windows-computers met behulp van een groot aantal talen: C, C++, C#, Python. Het is zelfs mogelijk om software te ontwikkelen voor Linux-systemen. Een korte introductie wordt gegeven in bijlage B.



Figuur 1.2: Voorbeeld van Microsoft Visual Stdio.

We hebben nu een beeld van hoe op een computer een C-programma wordt vertaald naar instructies voor een computer. De computer kan het C-programma niet direct uitvoeren, het programma moet gecompileerd worden. Als we een wijzing in het C-programma willen doorvoeren, dan moeten we het C-programma uiteraard opnieuw compileren.

We zullen in de overige paragrafen in sneltreinvaart enkele concepten van C bespreken. In de volgende hoofdstukken wordt de taal verder uitgediept.

1.5 Een minimaal C-programma

We beginnen met het meest simpele C-programma dat mogelijk is. We willen hiermee uitlegen wat er gebeurt als dit programma gecompileerd en uitgevoerd wordt. Het programma is te zien in listing 1.1. Het programma doet in feite helemaal niets, althans niet dat de gebruiker van het programma kan waarnemen. Toch gebeuren er wel degelijk dingen “onder de motorkap”.

```
1 int main(void)  
2 {  
3     return 0;  
4 }
```

Listing 1.1: Een minimaal C-programma.

Dit programma kan niet direct door de computer worden uitgevoerd, het moet eerst worden gecompileerd. Na compilatie is een *uitvoerbaar bestand* beschikbaar dat wel door de computer kan worden uitgevoerd. De werking op een PC of laptop is als volgt. Als het uitvoerbare programma wordt gestart, dan zal het besturingssysteem het uitvoerbare programma in het geheugen van de computer laden. Dus ergens in het geheugen van de computer liggen de instructies die het programma vormen opgeslagen. Het besturingssysteem start het programma door de processor naar de eerste instructie van het programma te leiden. Het programma (dat is het uitvoerbare programma) zal nu instructies uitvoeren. Na afloop van het programma wordt de besturing weer teruggeven aan het besturingssysteem.

Het C-programma begint met de definitie van de *functie* `main`. Een functie is een aantal instructies samengepakt onder een gemeenschappelijke noemer. *Elk C-programma heeft een functie `main`*. Tussen de haken staat het keyword `void`, dat aangeeft dat het uitvoerbaar programma geen gegevens meekrijgt van het besturingssysteem⁴. Vóór `main` staat het keyword `int` dat aangeeft dat `main` een geheel getal teruggeeft aan het besturingssysteem.

Statements die in het C-programma gebruikt worden, zijn afgebakend met *acolades*. Het Engelse woord hier voor is *curly braces* of gewoon *braces*. De acolades geven het begin en einde aan van *block*. Een blok begint met een accolade-openen (`{`) en eindigt met een accolade-sluiten (`}`). Over de plaats van de acolades zijn er diverse meningen. In listing 1.1 is de accolade-openen geplaatst onder de definitie van `main`, maar het is ook gebruikelijk om de accolade-openen te schrijven achter de definitie van `main`.

Binnen `main` zien we één *statement*. Een statement is een opdracht in de C-taal. Het statement wordt gevormd door het keyword `return` gevolgd door het getal 0 en een punt-komma. Bij uitvoer van dit statement wordt de waarde 0 teruggegeven aan het besturingssysteem. Dat behoeft enige uitleg. Een (gecompileerd) programma wordt gestart door het besturingssysteem. Aan het einde wordt het programma afgesloten. Het besturingssysteem “ruimt” het programma op en zorgt ervoor dat het gebruikte geheugen weer vrijgegeven wordt voor volgende programma’s. We kunnen aan het besturingssysteem een getal teruggeven, in dit geval 0. Het is aan het besturingssysteem om hier wat mee te doen.

⁴ Dat kan wel, zie hoofdstuk 7.

Gebruikelijk is om 0 terug te geven als alles goed verlopen is. Een ander getal dan 0 geeft over het algemeen aan dat er iets fout gegaan is.

1.6 Afdrukken op het scherm

We kunnen het eerste programma interessanter maken door een regel tekst op het scherm af te drukken. Het programma in listing 1.2 drukt de regel *De som van 3 en 7 is 10* op het scherm af. We zullen het programma stap voor stap doorlopen.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 3;
6     int b = 7;
7
8     int som;
9
10    som = a+b;
11
12    printf("De_som_van_%d_en_%d_is_%d\n", a, b, som);
13
14    return 0;
15 }
```

Listing 1.2: Afdrukken van de som van twee getallen.

In regel 1 wordt een zogenoemd *header-bestand* geladen, in dit geval het bestand `stdio.h`. We leggen zo meteen uit waarom dat nodig is.

In regel 3 wordt kenbaar gemaakt dat het programma de functie `main` heeft. Een C-programma heeft de *altijd* de functie `main`. Het (gecompileerde) programma wordt hier gestart.

In regel 5 en 6 worden twee *variabelen* gedeclareerd, de variabelen `a` en `b`. Technisch gezien is een variabele een plek in het geheugen van de computer. Een variabele kan door het programma gebruikt worden om gegevens te bewerken. Declaratie wil zeggen dat een variabele kenbaar wordt gemaakt. Bij de declaratie wordt opgegeven wat het type is van de variabele. Dit gebeurt middels het keyword `int`, dat betekent dat `a` en `b` alleen geheeltallige getallen kan opslaan (Engels: integer). Aan de variabelen worden waarden toegekend. We noemen dit *initialisatie* van variabelen.

In regel 8 wordt de variabele `som` gedeclareerd zonder initialisatie. Dat betekent dat op dat moment de waarde (of inhoud) van de variabele *onbekend* is. Vervolgens wordt in regel 10 de som van `a` en `b` berekend middels de *optel-operator* `+` en wordt het resultaat *toegekend* aan variabele `som`. Vanaf regel 10 is de waarde van `som` dus 10. In regel 10 zien we zogenoemde *expressies*. Zo is `a + b` een expressie en is de toekenning `som = ...` ook een expressie. De term *expressie* komt veel voor in C.

In regel 12 wordt de functie `printf` aangeroepen. Deze functie is al geschreven is zit in de standard library. De functie krijgt vier *argumenten* mee, gegevens die in de functie verwerkt worden. Het eerste argument is een *string*. Een string is een stukje tekst, maar we spreken ook wel van een rij karakters. Daarna volgen de (waarden) van de variabelen `a`, `b` en `som`.

Het eerste argument van `printf` wordt een *format string* genoemd. Dat ligt niet in C-taal vast maar wordt algemeen gebruikt. De format string bevat karakters, zogenoemde *format specifications* en een *escape sequence*. Een format specification begint met een procentteken gevolgd door een letter. De functie `printf` gebruikt deze format specifications om de gegevens af te drukken. De eerste `%d` zorgt ervoor dat variabele `a` wordt afgedrukt als een decimaal geheel getal. Op overeenkomstige wijze worden ook `b` en `som` afgedrukt. Aan het einde van de format string is een *escape sequence* te zien, in dit geval `\n`. Dit zorgt ervoor dat een volgende afdruk wordt begonnen aan het begin van de volgende regel (Engels: *newline*).

In regel 14 wordt met het keyword `return` aangegeven dat het programma wordt afgesloten. Dat de return-waarde een geheel getal moet zijn, kunnen we zien aan de definitie van de functie `main`. We zien in regel 3 dat `main` een geheel getal teruggeeft (keyword `int`) en geen argumenten meekrijgt (keyword `void`).

Hoe weet de C-compiler nou hoe de functie `printf` moet worden aangeroepen? Dat wordt geregeld met een *function prototype*. We hoeven dat zelf niet op te geven want dat is al gedaan en is te vinden in het header-bestand `stdio.h`. De eerste regel geeft dus aan dat dit bestand geladen moet worden. De tekens `<en>` geven aan dat gezocht moet worden op een bepaalde plek in het bestandssysteem. We hoeven dat verder niet te weten.

1.7 Invoer van het toetsenbord

We kunnen het vorige programma interessanter maken door aan de gebruiker te vragen om twee gehele getallen in te voeren. Naast het afdrukken van tekst met de functie `printf` maken we nu ook gebruik van de functie `scanf` om gehele getallen in te lezen. Het programma is te zien in listing 1.3.

In regel 1 laden we weer het header-bestand `stdio.h`. Dit header-bestand is nodig om de functie-prototypes van `printf` en `scanf` te laden. In regel 4 maken we gebruik van een *pragma*. Dit is een aanwijzing voor de C-compiler om iets te doen of te laten. Waarom deze *pragma* nodig is, kunnen we lezen in het kader op pagina 9.

Het programma volgt verder de lijn van listing 1.2. In de functie `main` declareren we drie variabelen. Daarna drukken we in regel 10 een stukje tekst af dat aangeeft wat de gebruiker moet doen. In regel 14 wordt de functie `scanf` aangeroepen die ervoor zorgt dat een geheel getal wordt ingelezen van het toetsenbord en in variabele `a` wordt gezet.

De format specifier `%d` hadden we al eerder gezien, maar de constructie `&a` is nieuw. De ampersand (`&`) zorgt ervoor dat aan de functie `scanf` het *adres* van variabele `a` meegegeven wordt. Het adres van de variabele is de plek waar de variabele in het geheugen ligt. Op deze manier kan `scanf` de informatie op je juiste plek zetten. In regel 17 wordt hetzelfde gedaan voor variabele `b`. We drukken voor het inlezen nog even netjes af hoe de gebruiker moet handelen. In regel 15 rekenen we de som uit van variabelen `a` en `b` en kennen dat toe

```

1 #include <stdio.h>
2
3 /* Make Visual Studio happy */
4 #pragma warning(disable : 4996)
5
6 int main(void)
7 {
8     int a;
9     int b;
10
11     int som;
12
13     printf("Geef_een_getal:_");
14     scanf("%d", &a);
15
16     printf("Geef_nog_een_getal:_");
17     scanf("%d", &b);
18
19     som = a + b;
20
21     printf("De_som_van_%d_en_%d_is_%d\n", a, b, som);
22
23     return 0;
24 }

```

Listing 1.3: Invoer van de gebruiker opvragen.

aan variabele `som`. In regel 18 drukken we de drie variabelen af. Het programma wordt afgesloten in regel 20.

Als we het programma starten dan moeten we twee gehele getallen invoeren. Een mogelijke uitvoer van het programma is te zien in figuur 1.3. De invoer van de gebruiker is vet afgedrukt. Overigens zullen ontwikkelsystemen zoals Visual Studio aan het einde van het

To scanf OR NOT TO scanf...

De Microsoft C-compiler bestempelt `scanf` als “onveilig”. Een compilatie met `scanf` zal eindigen met een foutmelding. In plaats daarvan moet de functie `scanf_s` worden gebruikt. Helaas ondersteunen andere compilers deze functie niet. Dat zal resulteren in een programma dat niet door iedere compiler kan worden vertaald. Om het probleem te omzeilen hebben we gebruik gemaakt van een *pragma*. In regel 4 geven we aan dat de C-compiler fout 4996 moet negeren. Op andere compilers, bijvoorbeeld de GNU-C compiler, wordt deze regel overgeslagen (er volgt wel een waarschuwing). Overigens wordt op vele fora gewaarschuwd voor de onveiligheid van `scanf` en worden alternatieven gegeven. Wij gebruiken `scanf` hier wel *for the sake of simplicity*. Het is beter om `scanf` te vermijden.

programma wachten tot de gebruiker op een toets drukt, anders is door de snelheid van het uitvoeren van het programma niet te zien wat is afgedrukt.

```
C:\ Command Prompt

Geef een getal: 7
Geef nog een getal: 4
De som van 7 en 4 is 11
```

Figuur 1.3: Uitvoer van het programma in listing 1.3.

1.8 Keywords

In tabel 1.1 is een lijst te zien met gereserveerde woorden. Een aantal van deze woorden hebben we algezien zoals `int`, `void` en `return`. Deze woorden worden *keywords* en genoemd en geven de compiler aanwijzingen wat er moet gebeuren.

Tabel 1.1: Een lijst met keywords in de C-taal.

| | | | |
|-----------------------|---------------------|-----------------------|-----------------------|
| <code>auto</code> | <code>double</code> | <code>int</code> | <code>struct</code> |
| <code>break</code> | <code>else</code> | <code>long</code> | <code>switch</code> |
| <code>case</code> | <code>enum</code> | <code>register</code> | <code>typedef</code> |
| <code>char</code> | <code>extern</code> | <code>return</code> | <code>union</code> |
| <code>const</code> | <code>float</code> | <code>short</code> | <code>unsigned</code> |
| <code>continue</code> | <code>for</code> | <code>signed</code> | <code>void</code> |
| <code>default</code> | <code>goto</code> | <code>sizeof</code> | <code>volatile</code> |
| <code>do</code> | <code>if</code> | <code>static</code> | <code>while</code> |

Een aantal van deze keywords dient als *qualifier* voor andere keywords. Zo kunnen we een variabele declareren als

```
unsigned long int a;
```

Dit geeft de compiler aanwijzingen over de grootte (het aantal bits) van variabele `a`.

1.9 Beslissingen

Met behulp van de keywords `if` en `else` kunnen in het programma beslissingen nemen op basis van een *conditie*. Dit is te zien in listing 1.4. We declareren twee variabelen `a` en `b` en kennen gelijk de waarden toe. In regel 8 is te zien dat getest wordt of `a` kleiner is dan `b`. We noemen het kleiner-dan-teken een *relationele operator*. Als inderdaad blijkt dat `a` kleiner is dan `b`, dat betekent dat de conditie `a < b` waar is, worden de statements tussen de accolades in regel 9 en 11 uitgevoerd. Is de conditie niet waar, dan worden de regels overgeslagen. Overigens mogen in dit geval de accolades weggelaten worden omdat maar één statement wordt uitgevoerd. Het is echter aan te bevelen om ze toch te gebruiken, omdat misschien later er nog statements toegevoegd worden. Een bekend voorbeeld van het niet-gebruiken van accolades is Apple's *gotofail SSL security bug* [10].

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 7;
6     int b = 9;
7
8     if (a < b)
9     {
10         printf("a_is_kleiner_dan_b\n");
11     }
12
13     return 0;
14 }

```

Listing 1.4: Afdrukken van tekst op basis van een beslissing.

Een `if`-statement kan ook gevolgd worden door een `else`-keyword en een `else`-keyword kan ook weer gevolgd worden door een `if`-keyword. In het Engels wordt dit een *multy-way branch* (to branch = vertakken) genoemd. Zie listing 1.5.

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 7;
6     int b = 9;
7
8     if (a < b)
9     {
10         printf("a_is_kleiner_dan_b\n");
11     }
12     else if (a == b)
13     {
14         printf("a_is_gelijk_aan_b\n");
15     }
16     else
17     {
18         printf("a_is_groter_dan_b\n");
19     }
20     return 0;
21 }

```

Listing 1.5: Afdrukken van tekst op basis van een beslissing.

Technisch gezien hoort de `else` in regel 16 bij de `if` in regel 12. Let erop dat het vergelijken van `a` en `b` op gelijkheid in regel 12 een *dubbele is-gelijk-teken* (`==`) bevat.

1.10 Herhalingen

Stel dat we de kwadraten van 1 t/m 10 willen afdrukken. We kunnen ervoor kiezen om tien `print`-functies aan te roepen. Maar we merken direct op dat we in feite tien keer hetzelfde moeten doen. We kunnen het afdrukken van de tien kwadraten vormgeven met een *herhaling*. Dit is te zien in listing 1.6.

We beginnen het programma met de declaratie van een aantal variabelen. Vervolgens stellen we de ondergrens, bovengrens en stapgrootte in in de regels 8 t/m 10. We willen beginnen bij de ondergrens, stoppen bij de bovengrens en elke herhaling nieuwe waarden afdrukken. Daarvoor gebruiken we de variabele `getal`. Zo'n variabele wordt een *lusvariabele* genoemd.

In regel 12 zetten we de lusvariabele op de ondergrens en gaan de lus uitvoeren. De lus wordt gekenmerkt door het keyword `while`. Achter het keyword `while` staat, tussen de haken, de voorwaarde waarop de lus moet worden uitgevoerd. Zolang de conditie `getal <= bovengrens` waar is, worden de statements binnen de accolades uitgevoerd. We noemen dat een *iteratie*. Is de conditie niet waar dan wordt verder gegaan met het statement die volgt op het `while`-statement.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int ondergrens, bovengrens, stap;
6     int getal, kwadraat;
7
8     ondergrens = 1;
9     bovengrens = 10;
10    stap = 1;
11
12    getal = ondergrens;
13    while (getal <= bovengrens)
14    {
15        kwadraat = getal * getal;
16        printf("Het_kwadraad_van_%3d_is_%3d\n", getal, kwadraat);
17        getal = getal + stap;
18    }
19
20    return 0;
21 }
```

Listing 1.6: *caption*.

Binnen de accolades van het `while`-statement zien we drie statements. In regel 15 wordt het kwadraat berekend van de (huidige) waarde van de lusvariabele. Daarna worden de lusvariabele en het kwadraat afgedrukt. In regel 17 wordt de lusvariabele aangepast naar de nieuwe (volgende) waarde door de stapgrootte erbij op te tellen⁵.

⁵ Het is tegenwoordig in het Nederlands gebruikelijk om het Engelse woord *updaten* te gebruiken.

In geval van het kwadratenprogramma kunnen we ook een ander herhalingsstatement gebruiken: het `for`-statement. Het is een compacte schrijfwijze van het `while`-statement. We kunnen het berekenen van het kwadraat ook binnen de parameter van de functie plaatsen. Zo sparen we een variabele uit. Zie listing 1.7.

```
1 // ...
2 for (getal = ondergrens; getal <= bovengrens;
3     getal = getal + stap)
4 {
5     printf("Het_kwadraad_van_%3d_is_%3d\n",
6           getal, getal*getal);
7 }
8 //...
```

Listing 1.7: Gebruik van een *for*-statement.

1.11 Array's

Een *array* is een lijst variabelen onder een gemeenschappelijke noemer. Zo declareren we vijf floating point-getallen (getallen met een komma) met

```
double lijst[5];
```

en kunnen we gebruik maken van de variabelen

```
lijst[0] lijst[1] lijst[2] lijst[3] lijst[4]
```

Tussen de blokhaken `[]` staat het *elementnummer* en de variabelen worden *elementen* genoemd. In listing 1.8 is een programma te zien met een array.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double lijst[5] = {3.14159, 2.78182, 0.57721, 1.41421, 1.61803};
6     double som = 0.0;
7     int index;
8
9     for (index = 0; index < 5; index = index + 1) {
10         som = som + lijst[index];
11     }
12
13     printf("Gemiddelde:_%f\n", som / 5.0);
14
15     return 0;
16 }
```

Listing 1.8: Gemiddelde van vijf getallen.

De array mag bij declaratie gelijk geïnitieerd worden zoals te zien is in regel 5⁶. We berekenen van deze vijf elementen de gemiddelde waarde. We doen dat door eerst de som van de vijf element te berekenen en vervolgens te delen door 5. Met behulp van een `for`-statement wordt “langs de array gelopen”; bij elke iteratie selecteren we een element en tellen dat op bij de som van de tot dan toe gesommeerde elementen. De regel

```
som = som + lijst[index];
```

selecteert dus een element uit de array en telt dat op bij de som. In regel 13 drukken we het gemiddelde af door de som te delen door 5. We hoeven daarvoor niet een aparte variabele te declareren, we kunnen als argument gewoon `som / 5.0` gebruiken.

1.12 Functies

C biedt een handige manier om een groep statement, bijvoorbeeld berekeningen, onder te brengen in een *functie*. Als de functie eenmaal geschreven is hoeven we ons niet druk te maken over hoe de berekeningen worden gedaan, we hoeven alleen maar te weten hoe we de functie moeten gebruiken. We hebben al drie functies gezien: `printf`, `scanf` en `main`. `printf` en `scanf` zijn al beschikbaar via de standard library. We hoeven niet te weten hoe de functies werken, alleen maar hoe ze moeten worden aangeroepen.

Laten we eens de wortels bepalen van de getallen 0 t/m 10. Daartoe schrijven we een functie `sqrt_babylonian` die de wortel van een getal bepaalt volgens de Babylonische methode. Dit is een iteratieve methode om de wortel van een getal steeds nauwkeuriger te benaderen. De manier waarop dat gebeurt is te zien in vergelijking (1.1).

$$\begin{aligned} x_0 &= S && \text{beginsituatie} \\ x_{n+1} &= \frac{1}{2} \cdot \left(x_n + \frac{S}{x_n} \right) && \text{nieuwe situatie} \\ \sqrt{S} &= \lim_{n \rightarrow \infty} x_n && \text{uiteindelijke resultaat} \end{aligned} \tag{1.1}$$

We lezen dit als volgt. We beginnen met het getal S waarvan we de wortel willen berekenen ($x_0 = S$). De tweede, iteratieve stap, is het berekenen van de volgende benadering. We berekenen dat met de middelste vergelijking uit (1.1). De laatste stap geeft aan dat als we de wortel willen bepalen, de tweede stap oneindig vaak herhaald moet worden. Natuurlijk is dat niet realiseerbaar. We kunnen maar een eindig aantal stappen uitrekenen. Het blijkt dat voor de wortel slechts tien stappen nodig zijn om de wortel redelijk te benaderen.

De functie is te zien in listing 1.9. We maken hier gebruik van het datatype `double`, een floating point-datatype met een nauwkeurigheid van ongeveer 16 decimale cijfers. Zowel het argument als het returnwaarde is van dit datatype. We beginnen met de functiedefinitie

```
double square_root(double s)
```

Binnen de functie maken we gebruik van de twee variabelen `xiter` en `i`. `xiter` wordt gebruikt om de nieuwe waarde van de wortel te berekenen en `i` wordt gebruikt om het aantal iteraties bij te houden. We testen in regels 8 t/m 10 of het argument 0 is en geven dan direct de waarde 0 terug.

⁶ We hebben als getallen vijf wiskundige constanten genomen. De lezer wordt uitgedaagd uit te zoeken welke constanten dat zijn.


```

1 #include <stdio.h>
2
3 double square_root(double s)
4 {
5     double xiter = s;
6     int i;
7
8     if (s == 0.0)
9     {
10         return 0.0;
11     }
12
13     for (i = 0; i < 10; i = i + 1)
14     {
15         xiter = 0.5 * (xiter + s / xiter);
16     }
17
18     return xiter;
19 }
20
21 int main(void)
22 {
23     double i;
24
25     for (i = 0.0; i < 11.0; i = i + 1.0)
26     {
27         printf("De wortel van %6.3f is %6.6f\n", i, square_root(i));
28     }
29
30     return 0;
31 }

```

Listing 1.9: caption.

In het `for`-statement worden tien iteraties van de middelste formule uit (1.1) uitgevoerd. Bij elke doorgang wordt de waarde van de wortel steeds beter benaderd. Als het `for`-statement klaar is wordt deze waarde via `return` teruggegeven aan de aanroeper. In `main` worden met behulp van een `for`-statement de wortels van 0 t/m 10 berekend en afgedrukt.

Overigens bevat de *mathematical library* een implementatie van de wortel-functie. We hoeven dat niet zelf te schrijven.

1.13 En verder...

We hebben nu enkele concepten van C uitgelegd. Maar de taal kan meer. Wat we niet behandeld hebben zijn *pointers*, *structures* en *bestandsverwerking*. Deze concepten zullen in de volgende hoofdstukken worden uitgelegd.

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 moderne computersystemen werken met 64-bits eenheden 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 5.

2.1 Variabelen

Een *variabele* 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:

```

1 int a;
2 ...
3 a = 2;
4 ...
5 a = -3;
6 ...
7 a = 5;

```

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 voorbeelden:

```
_loop 50cent int printf
```

De meeste C-compilers accepteren de underscore als eerste karakter. Dit wordt echter afgeraden omdat veel *systemroutines* 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 voorpraak 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

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.

ook systemen en C-compilers waar een `int` uit 16 bits bestaat. Dat een `int` uit 32 bits 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 datatypes die beschikbaar zijn in C.

| type | bits | bereik |
|----------------------------|------|--|
| <code>char</code> | 8 | $-128 \text{ — } +127$ |
| <code>short int</code> | 16 | $-32768 \text{ — } +32767$ |
| <code>int</code> | 16 | $-32768 \text{ — } +32767$ |
| | 32 | $-2147483648 \text{ — } +2147483647$ |
| <code>long int</code> | 32 | $-2147483648 \text{ — } +2147483647$ |
| <code>long long int</code> | 64 | $-9223372036854775808 \text{ — } +9223372036854775807$ |

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 telkens 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:

$$\text{grootte short int} \leq \text{grootte int} \leq \text{grootte long int}$$

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 signed variant:

```
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 ontstaat al gelijk de eerste verwarring: in C wordt de komma vervangen door de punt, zoals gebruikelijk is in Engelstalige literatuur. We schrijven 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* datatypes die beschikbaar zijn in C.

| type | bits | kleinste getal | grootste getal |
|--------------------------|--------|--------------------------------|-------------------------------|
| <code>float</code> | 32 | $\approx 1,18 \times 10^{-38}$ | $\approx 3,4 \times 10^{38}$ |
| <code>double</code> | 64 | $\approx 2.2 \times 10^{-308}$ | $\approx 1.8 \times 10^{308}$ |
| <code>long double</code> | 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 ongeveer 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. De *floating-point* weergave maakt het mogelijk om een breed dynamisch bereik van waarden te bestrijken met een constant aantal significante bits.

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 7.

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 hexadecimaal 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 begint 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, geschreven als een één karakter tussen apostrofes. Zo is `'A'` een karakterconstante. 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`. De standaard ASCII-tabel is te vinden in bijlage A.

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.

| | | | |
|-----------------|-----------------|-----------------|----------------|
| <code>\n</code> | newline | <code>\\</code> | backslash |
| <code>\r</code> | carriage return | <code>\'</code> | single quote |
| <code>\a</code> | bell | <code>\"</code> | double quote |
| <code>\b</code> | backspace | <code>\0</code> | null byte |
| <code>\f</code> | formfeed | <code>\t</code> | horizontal tab |

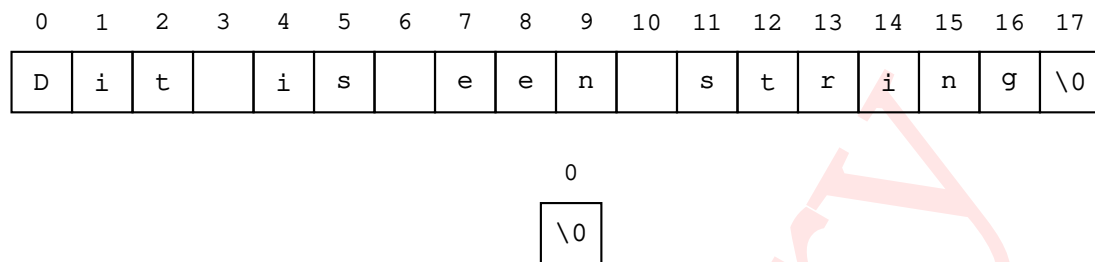
Andere karakters kunnen worden gevormd door de backslash, gevolgd door de letter `x` en één of twee hexadecimale cijfers. We kunnen de letter `A` dus ook schrijven als `'\x41'`.

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"
"" /* The empty 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-karakter 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, kopiëren 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 6.

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 plaatsvindt.

```
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;
```

de variabele `aantal` 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 gaan voor op optellen en aftrekken. Haakjes worden gebruikt de prioriteiten te veranderen:

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

2.4.1 Rekenkundige operatoren

De *binair* 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 *unair operator* gebruikt worden. Een voorbeeld hiervan is

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

Let goed op de prioriteiten van de operatoren. De unair operatoren gaan voor op vermenigvuldigen ($*$), delen ($/$) en modulus ($\%$). De binair optelling ($+$) en aftrekking ($-$) hebben een 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 prioriteit 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 variabele `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 schrikkeljaar (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 schrikkeljaar 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 schrikkeljaar. Dus 1984 is een schrikkeljaar, 2100 is geen schrikkeljaar 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:

| | |
|-----------------------|---|
| <code>&</code> | AND |
| <code> </code> | OR |
| <code>^</code> | EXOR |
| <code><<</code> | naar links schuiven |
| <code>>></code> | naar rechts schuiven |
| <code>~</code> | 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.

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 |

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” gaan verloren. De expressie

```
1 << 4
```

geeft als resultaat 16. De schuifoperator `>>` schuift de bits in een variabele een aantal plekken naar rechts. De vrijgekomen 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 en de overige bits ongemoeid laat. In figuur 2.2 zijn drie voorbeelden te zien van bitmanipulaties met AND, OR en EXOR. Hiervoor zijn de gegevens uit tabel 2.4 gebruikt.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|------|
| 7 | | | | | | | | 0 | |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | AND |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | | |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | | OR |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | | |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | | |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | | EXOR |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | | |

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`.

```
1 #include <avr/io.h>          /* PINA, PORTB, ... */
2
3 int main(void) {
4
5     if (PINA & 0x80) {        /* if key is pressed ... */
6         PORTB = PORTB ^ 0x01; /* ... invert lower led */
7     }
8
9 }
```

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, zeker 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 van `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 `float`s 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 en het grootste getal dat kan worden opgeslagen is 255 (binair 1111111₂). Tellen we daar 1 bij op dat is het resultaat 256 (binair 10000000₂) 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` nu 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` een 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 header-bestand `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 oneindig.

Tabel 2.5: De vaste-lengte datatypes 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 datatypes 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 Enumeraties

Er is in C een mogelijkheid om een eigen datatype te definiëren in combinatie met een lijst van constanten: een *enumeratie*. Een enumeratie is een lijst van geheeltallige constanten in de vorm van

```
enum vartype { TUNKNOWN; TINT; TFLOAT; TDOUBLE };
```

De eerste naam binnen de accolades krijgt de waarde 0, de volgende de waarde 1 enzovoorts. Maar het is ook mogelijk om expliciete waarden toe te kennen

```
enum vartype { TUNKNOWN; TINT=11; TFLOAT=13; TDOUBLE };
```

In deze enumeratie krijgt TUNKNOWN de waarde 0 (impliciet), TINT de waarde 11 (expliciet), TFLOAT de waarde 13 (expliciet) en TDOUBLE de waarde 14 (impliciet). Merk op dat er niet echt een nieuw datatype wordt gerealiseerd. De compiler zoekt uit of de enumeratie in een van de integer datatypes past.

2.9 Overige operatoren

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

2.9.1 Grootte van een variabele

De grootte van een variabele of datatype in *bytes* kan berekend worden met de operator `sizeof`. Dit kan alleen *tijdens het compileren* van het C-programma gebeuren. Tijdens het uitvoeren van het programma zijn alle grootten berekend. Er is één restrictie in het gebruik van `sizeof`: bij het gebruik met een datatype *moeten* haakjes gebruikt worden. Zie listing 2.6. Deze operator heeft een hoge prioriteit. We zullen `sizeof` nader bespreken in hoofdstuk 6.

```
1 int main(void) {  
2     int i;  
3  
4     if (sizeof i == sizeof (long)) {  
5         printf("De_grootte_van_i_is_gelijk_aan_een_long\n");  
6     }  
7  
8     return 0;  
9 }
```

Listing 2.6: Gebruik van `sizeof`.

2.9.2 Verhogen of verlagen met 1

Een toekenning als

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

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 daarna 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 6. Deze operatoren hebben een hoge prioriteit.

Tabel 2.7: Toekenningsoperatoren.

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

2.9.3 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 2.7. Deze operatoren hebben een lage prioriteit. Merk op dat bij de kolom "Evaluatie" de expressies `a` en `b` omringd zijn door haakjes. Dat geeft ook gelijk aan dat deze expressies eerst geëvalueerd worden voordat de operatie en toekenning plaatsvindt.

2.9.4 Conditionele expressie

De *conditionele expressie* is een operator die aan de hand van de waarde van een variabele (of constante of expressie) een van de twee opgegeven expressies uitvoert. Als we de maximale waarde van `a` en `b` willen bepalen dan kunnen we schrijven:

```
max = (a>b) ? a : b;    /* max = maximum(a, b) */
```

Een typisch voorbeeld is het afdrukken van de meervoudsvorm van een woord (of niet). Stel dat we aan het einde van een programma afdrukken hoeveel documenten verwerkt

zijn. Dan kunnen we met behulp van een conditionele expressie het woord document of documenten afdrukken. We doen dat zoals te zien is in de regels 11 en 12 in listing 2.7.

```
1 #include <stdio.h>
2
3 int main(void) {
4
5     int nrdocs = 0;
6
7     //...
8     /*          +----- nrdcos          */
9     /*          |          +---- " " or "en" */
10    /*          v          v                */
11    printf("In_totaal_%d_document%s_verwerkt\n", nrdocs,
12           (nrdocs == 1) ? " " : "en"));
13
14    return 0;
15 }
```

Listing 2.7: Het afdrukken van een meervoudsvorm.

De eerste parameter (%d) is het aantal documenten dat verwerkt is (nrdocs). De tweede parameter is een string (%s). De conditionele expressie

```
(nrdocs == 1) ? " " : "en";
```

geeft de (lege) string " " als nrdocs gelijk is aan 1 of de string "en" als nrdocs ongelijk aan 1 is. De haken rond de expressie nrdocs == 1 zijn eigenlijk niet nodig want de prioriteit van de conditionele expressie is erg laag, maar het komt de leesbaarheid wel ten goede.

2.9.5 Komma-operator

De komma-operator scheidt twee expressies die één statement worden gezien; We bespreken de komma-operator in hoofdstuk 3.

2.9.6 Nog enkele operatoren

De volgende operatoren worden de verderop in het boek besproken:

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

2.10 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 ongelijk 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 |

Uit deze tabel zijn de operatoren voor array's (`[]`), pointers (`*` en `&`) en structures (`->` en `.`) weggelaten. Deze operatoren worden behandeld in volgende hoofdstukken.

Veel van deze operatoren volgen uit de noodzaak (of is het drang) om programma's compact te schrijven. Doorgewinterde programmeurs kunnen op deze manier vlot, zij het onleesbare, code produceren. Een eerste gedachte is dat zulke constructies bijdragen aan compacte uitvoerbare bestanden. Maar de huidige generatie compilers is zeer goed in het herkennen van de taal en genereren al bijna-minimale uitvoerbare bestanden. Het is beter om eerst een leesbaar programma te schrijven dat correct werkt en daarna gaan nadenken of het slimmer kan. Donald Knuth schreef al eens [11]:

Premature optimization is the root of all evil.

De lezer is gewaarschuwd.

3

Beslissen en herhalen

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5
6     int i, ione, j;
7
8     for (i=0, ione=1, j=9; i<10; i++, ione++, j--) {
9         printf("%2dx9_=%1d%1d\n", ione, i, j);
10    }
11    return 0;
12 }
```

Listing 3.1: Afdrukken van de tafel van 9 met behulp van de komma-operator.

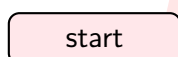
4

Flowcharts

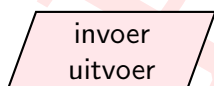
Een *flowchart* (Nederlands: stroomschema) is een grafische weergave van een (deel van een) computerprogramma. Flowcharts gebruiken rechthoeken, ovalen, ruiten en mogelijk andere vormen om het type stap te definiëren om samen met verbindingspijlen de stroom en volgorde te definiëren. Ze kunnen variëren van eenvoudige, met de hand getekende diagrammen tot uitgebreide computergegenereerde diagrammen die meerdere stappen en routes weergeven. Als we alle verschillende vormen van stroomdiagrammen beschouwen, zijn ze een van de meest voorkomende diagrammen op aarde, die door zowel technische als niet-technische mensen gebruikt worden. Ze zijn gerelateerd aan andere populaire diagrammen, zoals Data Flow Diagrams (DFD's) en Unified Modeling Language (UML) activiteitendiagrammen.

Het is belangrijk om een flowchart overzichtelijk te houden. Het heeft geen zin om op een A4'tje bijvoorbeeld 40 symbolen te tekenen. Afhankelijk van de grootte van het programma moeten we stukken code “comprimeren”. Zo zouden we de code voor het inlezen van tien elementen van een array als één statement in een flowchart kunnen tekenen met de tekst “lees array in”.

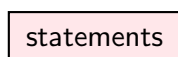
De gebruikte symbolen in een flowchart staan hieronder weergegeven:



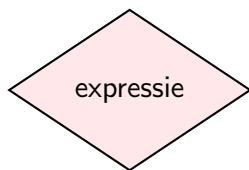
Dit symbool geeft het begin of het einde van de flowchart aan. Naast “start” en “stop” kunnen ook de termen “begin” en “einde” gebruikt worden.



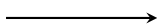
Dit symbool geeft invoer of uitvoer aan, bijvoorbeeld het inlezen van een getal of het afdrukken van tekst.



Dit symbool wordt gebruikt om statements anders dan invoer en uitvoer aan te duiden. Een statement kan bijvoorbeeld een berekening zijn, maar ook een functie-aanroep. Meerdere sequentiële statements mogen samengenomen worden in één symbool.



Dit symbool geeft een beslissing aan. De expressie kan alleen maar “waar” of “niet waar” opleveren. Een beslissing levert een vertakking op. De vertakkingen kunnen naar links of rechts zijn én naar beneden (dus niet links en rechts). Bij de uitgaande vertakkingen worden de woorden “yes” en “no” geschreven. Naast “yes” en “no” kunnen ook de termen “true” of “T” en “false” of “F” gebruikt worden.



Een pijl geeft een *flow* aan. De pijl begint bij een van de symbolen en eindigt bij een symbool of een andere pijl. Bij een symbool komt altijd maar één pijl aan.



Een connector kan gebruikt worden om een plek aan te geven waar twee pijlpunten samenkomen. Dit symbool wordt niet in dit boek gebruikt.

Voorbeelden van invoer en uitvoer zijn: “lees *i*” en “print *k*”. Voorbeelden van statements zijn (geen invoer en uitvoer): “ $j = j + 1$ ” en “ $k = 2 * j$ ”. Voorbeelden van expressies zijn: “ $j < 10$ ” en “ $a > 5$ en $a < 25$ ”.

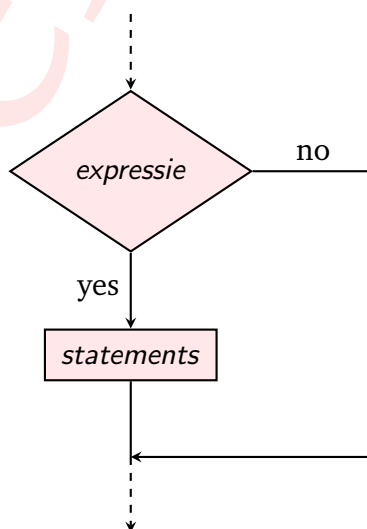
4.1 if-statement

Een if-statement is te modelleren door middel van een beslissing en een statement.

```
1 if (expressie) {
2     statements
3 }
```

Listing 4.1: *if*-statement in C

De flowchart is gegeven in figuur 4.1.



Figuur 4.1: Flowchart van een *if*-statement.

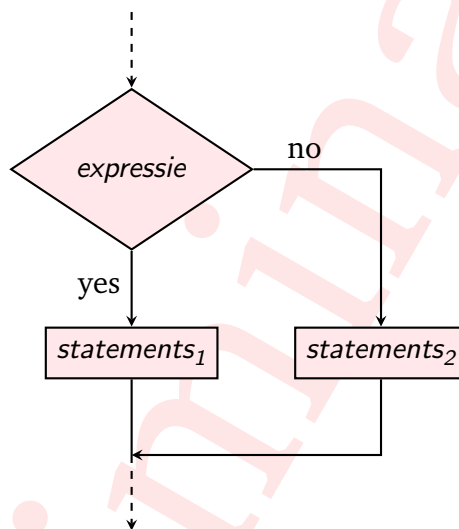
4.2 if-else-statement

Een if-else-statement van de vorm

```
1 if (expressie) {  
2     statements1  
3 else {  
4     statements2  
5 }
```

Listing 4.2: *if-else-statement in C.*

is in een flowchart te tekenen zoals te zien is in figuur 4.2.



Figuur 4.2: *Flowchart van een if-else-statement.*

4.3 while-lus en do-while-lus

Een while-lus heeft de gedaante:

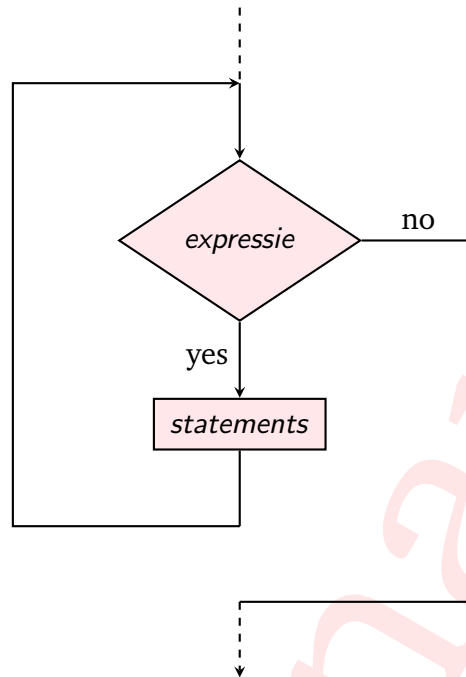
```
1 while (expressie) {  
2     statements  
3 }
```

Listing 4.3: *while-lus in C.*

wordt weergegeven als flowchart in figuur 4.3. De do-while-lus heeft in C de volgende gedaante:

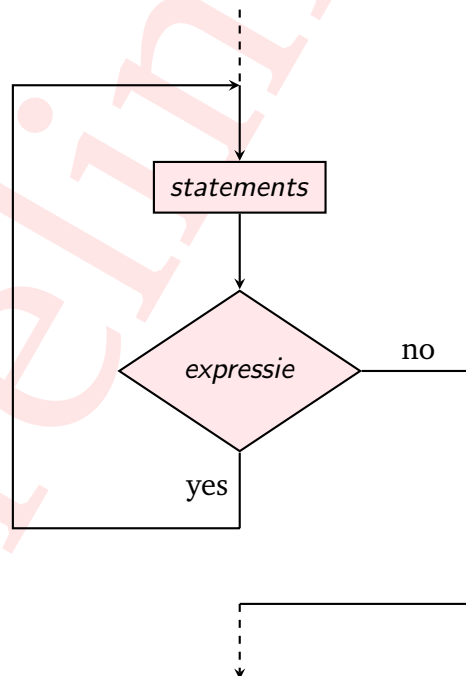
```
1 do {  
2     statements  
3 } while (expressie);
```

Listing 4.4: *Do-while-lus in C.*



Figuur 4.3: Flowchart van een *while*-lus.

De flowchart is te vinden in figuur 4.4. Let erop dat *statements* minstens één keer wordt uitgevoerd, ook al is *expressie* niet waar.



Figuur 4.4: Flowchart van een *do-while*-lus.

4.4 for-lus

Een for-lus van de gedaante:

```
1 for (statements1; expressie; statements2) {  
2     statements3  
3 }
```

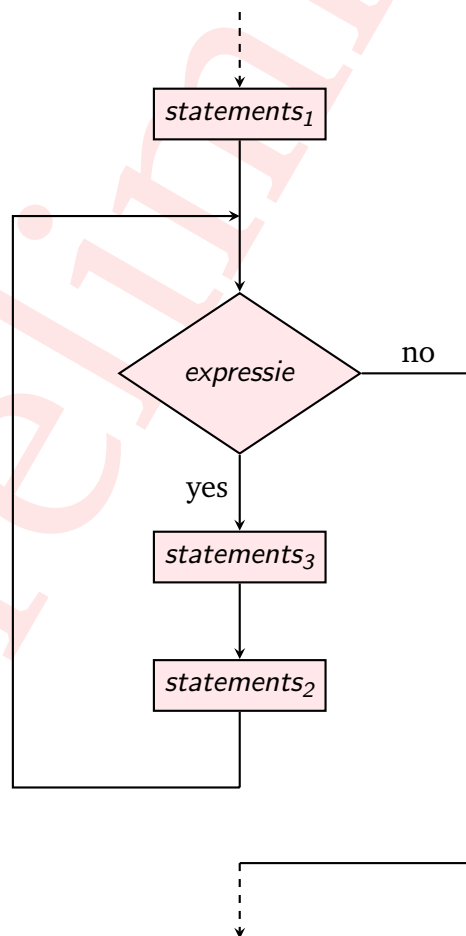
Listing 4.5: *for-lus* in C.

is *semantisch identiek* aan:

```
1 statements1  
2 while (expressie) {  
3     statements3  
4     statements2  
5 }
```

Listing 4.6: *for-lus* herschreven als *while-lus* in C.

Let erop dat $statements_2$ ná $statements_3$ komt. De flowchart is te vinden in figuur 4.5.



Figuur 4.5: Flowchart van een *for-lus*.

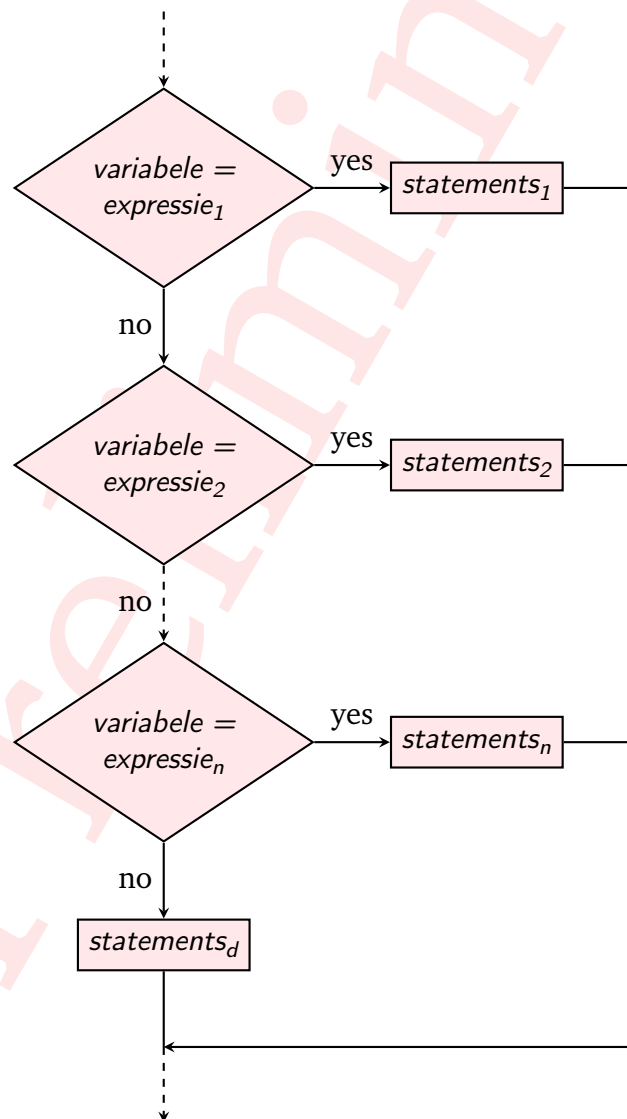
4.5 switch-statement

Een switch-statement is een meervoudige if-statement met steeds dezelfde variabele.

```
1 switch (variabele) {  
2     case expressie1: statements1  
3     case expressie2: statements2  
4     ...  
5     case expressien: statementsn  
6     default : statementsd  
7 }
```

Listing 4.7: switch-statement in C.

Merk op dat *expressie*₁, *expressie*₂, ..., *expressie*_n een constante moeten opleveren. De flowchart is te zien in figuur 4.6.



Figuur 4.6: Flowchart van een switch-statement.

4.6 Voorbeeld

We zullen een flowchart presenteren van een kort programma. Hieronder is een eenvoudig C-programma gegeven dat een variabele k inleest en vervolgens de som bepaalt van alle getallen tussen 1 en k (inclusief), dus:

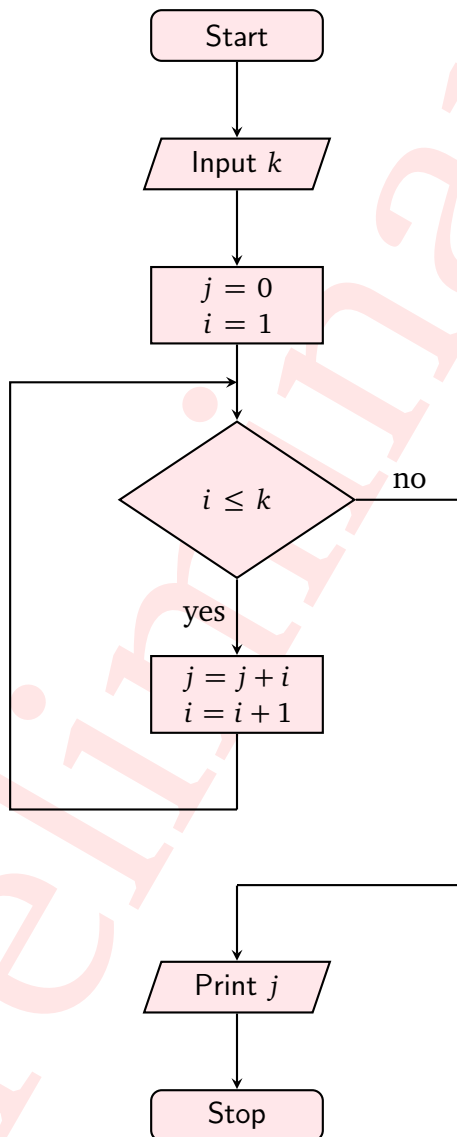
$$j = 1 + 2 + 3 + 4 + \dots + k \quad (4.1)$$

```
1 #include <stdio.h>
2
3 int main(void) {
4
5     int k, i, j;
6
7     scanf("%d", &k);
8
9     j=0;
10
11    for (i=1; i<=k; i=i+1) {
12        j=j+i;
13    }
14
15    printf("%d\n", j);
16
17    return 0;
18 }
```

Listing 4.8: Codevoorbeeld in C.

We hebben hier te maken met het begin en einde van het programma, het inlezen en afdrukken van variabelen en het doorlopen van een for-lus waarbij variabelen worden aangepast. Uiteindelijk komen uit bij het einde van het programma.

In figuur 4.7 is de flowchart te zien.



Figuur 4.7: Voorbeeld van een flowchart.

5

Functies

Als een programma langer wordt, dan wordt het al snel onoverzichtelijk. Sommige stukken code komen misschien meerdere keren in het programma voor. Bijvoorbeeld het inlezen van een positief getal. Dit kopiëren van code zorgt ervoor dat het programma slecht onderhoudbaar wordt. Als er namelijk een wijziging in deze code moet worden doorgevoerd dan moeten ook alle kopietjes worden aangepast. Ook zijn delen uit zo'n lang programma niet eenvoudig te gebruiken in een ander programma. De code is slecht herbruikbaar.

We kunnen deze problemen oplossen door het gebruik van *functies*. Een logisch bij elkaar behorend stuk code wordt dan ergens apart (in een functie) geplaatst. Deze functie kunnen we vervolgens naar believen aanroepen om de code van de functie uit te voeren.

Een functie kan eenvoudige van aard zijn en slechts een simpel stukje programma bevatten. Maar we kunnen functies ook complexer maken door ze argumenten mee te geven. Daarmee kunnen we de functie voor hetzelfde doeleind gebruiken, maar kan het stukje programma wat meer doen. We kunnen een functie ook informatie laten teruggeven. Op die manier is het mogelijk om een functie te schrijven die gegevens meekrijgt, daar iets mee laten doen en dan een resultaat teruggeeft. Een voorbeeld hiervan is een functie om de sinus van een hoek te laten berekenen.

Er zijn ook functies die *zichzelf* aanroepen. Dat worden recursieve functies genoemd. De functie roept zichzelf aan met (meestal) een gewijzigd argument. Op deze manier kunnen complexe programmeerproblemen van elegant worden opgelost maar we zullen zien dat het wel zijn weerslag heeft op het geheugen en de executietijd.

Binnen een functie kunnen we variabelen declareren. Deze variabelen zijn alleen binnen de functie beschikbaar. De variabele wordt aangemaakt als de functie begint en wordt verwijderd als de functie wordt geëindigd. De informatie die variabele bevat, is dan ook niet meer beschikbaar.

5.1 Een eenvoudige functie

Als sterk versimpeld voorbeeld beschouwen we het programma dat gegeven is in listing 5.1.

```
1 #include <stdio.h>
2
3 int main(void) {
4     // ...
5     printf("\n"); // sla 3 regels over
6     printf("\n");
7     printf("\n");
8     // ...
9     printf("\n"); // sla 3 regels over
10    printf("\n");
11    printf("\n");
12    // ...
13    return 0;
14 }
```

Listing 5.1: Een programma waar op twee plaatsen drie regels overgeslagen worden.

Op twee plaatsen in dit programma worden, in de uitvoer van het programma, drie regels overgeslagen. In plaats van deze code te dupliceren kunnen we deze code ook opnemen in een functie. Het is belangrijk om de functie een duidelijke naam te geven die aangeeft wat de functie doet. In dit geval is gekozen voor de naam `sla_3_regels_over`. Het programma waarbij gebruikt gemaakt wordt van een functie is gegeven in listing 5.2.

```
1 #include <stdio.h>
2
3 void sla_3_regels_over(void) {
4     printf("\n");
5     printf("\n");
6     printf("\n");
7 }
8
9 int main(void) {
10    // ...
11    sla_3_regels_over();
12    // ...
13    sla_3_regels_over();
14    // ...
15    return 0;
16 }
```

Listing 5.2: Gebruik van de functie `sla_3_regels_over`.

Bovenin het programma wordt de functie `sla_3_regels_over` gedefinieerd. Het keyword `void` betekent leeg. Het gebruik van `void` vóór de functienaam geeft aan dat deze

functie niets teruggeeft. Verderop in dit hoofdstuk zullen we zien hoe een functie indien gewenst wel iets kan teruggeven. Het gebruik van `void` ná de functienaam, tussen de haken, geeft aan dat aan deze functie niets meegegeven kan worden bij aanroep. Verderop in dit hoofdstuk zullen we zien hoe aan een functie, indien gewenst, wel iets kan worden meegegeven bij aanroep. Vervolgens wordt deze functie in `main` twee maal aangeroepen met de code `sla_3_regels_over()`. De haakjes `()` achter de functienaam geven aan dat de functie aangeroepen moet worden.

Als dit programma gestart wordt, begint de uitvoering zoals gebruikelijk bij `main`. Als de aanroep `sla_3_regels_over()` moet worden uitgevoerd, wordt naar de code van deze functie gesprongen. Aan het einde van de functie wordt teruggekeerd achter de plaats waar de functie is aangeroepen en wordt de uitvoering van het programma daar voortgezet. Bij het aanroepen van een functie “onthoudt” de processor dus waarvandaan de functie aangeroepen wordt, zodat het programma na afloop van de functie na deze aanroep kan worden vervolgd.

De functie moet “gezien” zijn voordat de functie aangeroepen kan worden. Vandaar dat de definitie van de functie `sla_3_regels_over()` boven `main` geplaatst is. Het is echter niet nodig om de volledige code van de functie voor `main` te definiëren. Het is voldoende om alleen de eerste regel van de functie voor `main` te definiëren. Dit wordt dan een *functiedeclaratie* of *functieprototype* genoemd. De volledige code van de functie moet natuurlijk nog wel worden gedefinieerd. Dit kan bijvoorbeeld na `main` maar kan ook in een apart bestand. We komen hier verderop in deze paragraaf op terug.

In listing 5.3 zien we hoe een functiedeclaratie kan worden gebruikt¹.

5.2 Functies met parameters

Als een nog steeds sterk versimpeld voorbeeld beschouwen we nu het programma dat gegeven is in listing 5.4

Op twee plaatsen in dit programma worden, in de uitvoer van het programma, regels overgeslagen. De eerste keer worden drie regels overgeslagen en de tweede keer worden vier regels overgeslagen. We kunnen nu een functie definiëren om drie regels over te slaan en nog een andere functie om vier regels over te slaan. Maar het zou natuurlijk veel handiger zijn als we een functie zouden kunnen definiëren waarmee we een variabeel aantal regels kunnen overslaan. Dit is mogelijk door een functie met een zogenoemde *parameter* te definiëren. Bij aanroep van de functie moet dan een zogenoemd *argument* worden meegegeven. De waarde van het argument wordt dan bij aanroep van de functie naar de parameter gekopieerd.

In listing 5.5 zien we hoe de functie `sla_regel_over` is gedefinieerd. De functie heeft een parameter van het type `int`. Bij de eerste aanroep van de functie wordt het argument 3 meegegeven. Deze waarde wordt bij aanroep *gekopieerd* naar de parameter `aantal`. Een parameter is in feite niets anders dan een variabele die bij het aanroepen van de functie

¹ Als we de functiedeclaratie vergeten, zal het programma wel compileren, al zal de compiler wel een waarschuwing geven. De compiler is in dit geval namelijk niet in staat om te controleren of de functie correct wordt aangeroepen. Het is dus sterk aan te raden om als een functie niet boven `main` gedefinieerd is een functiedeclaratie te gebruiken. Overigens geeft Visual Studio standaard een foutmelding.

```

1 #include <stdio.h>
2
3 void sla_3_regels_over(void);
4
5 int main(void) {
6     // ...
7     sla_3_regels_over();
8     // ...
9     sla_3_regels_over();
10    // ...
11    return 0;
12 }
13
14 void sla_3_regels_over(void) {
15     printf("\n");
16     printf("\n");
17     printf("\n");
18 }

```

Listing 5.3: Een programma waarin een functiedeclaratie gebruikt is..

```

1 #include <stdio.h>
2
3 int main(void) {
4     // ...
5     printf("\n"); // sla 3 regels over
6     printf("\n");
7     printf("\n");
8     // ...
9     printf("\n"); // sla 4 regels over
10    printf("\n");
11    printf("\n");
12    printf("\n");
13    // ...
14    return 0;
15 }

```

Listing 5.4: Een programma waar op twee plaatsen een aantal regels overgeslagen wordt.

geïnitieerd wordt met de waarde van het bij aanroep meegegeven argument. De code van de functie wordt vervolgens uitgevoerd. Doordat de parameter `aantal` is geïnitieerd met de waarde 3 wordt de code in het `for`-statement drie maal herhaald en daardoor worden in de uitvoer drie regels overgeslagen. De parameter `aantal` is alleen in de functie `sla_regels_over` te gebruiken. Buiten de functie is de parameter onbekend. Bij de tweede aanroep van de functie wordt het argument 4 meegegeven. Ook deze waarde wordt bij aanroep gekopieerd naar de parameter `aantal`. De code van de functie wordt vervolgens weer uitgevoerd. Doordat de parameter `aantal` nu is geïnitieerd met de

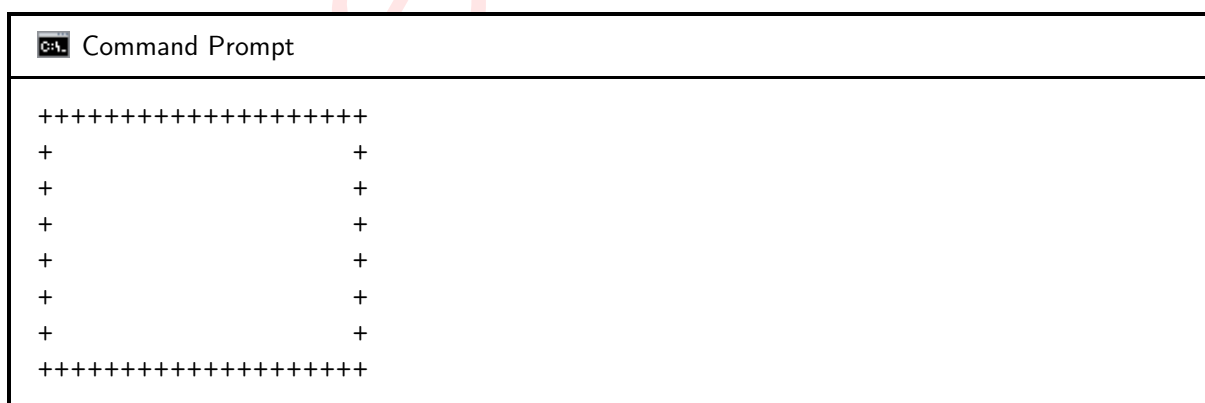
waarde 4 wordt de code in het `for`-statement vier maal herhaald en daardoor worden in de uitvoer vier regels overgeslagen.

```
1 #include <stdio.h>
2
3 void sla_regels_over(int aantal) {
4     for (int teller = 0; teller < aantal; teller++) {
5         printf("\n");
6     }
7 }
8
9 int main(void) {
10     // ...
11     sla_regels_over(3);
12     // ...
13     sla_regels_over(4);
14     // ...
15     return 0;
16 }
```

Listing 5.5: Een programma waar op twee plaatsen een aantal regels wordt overgeslagen.

Een functie kan meerdere parameters hebben. De verschillende parameters worden van elkaar gescheiden door een komma. Elke parameter heeft zijn eigen typeaanduiding. Bij aanroep moet het aantal argumenten overeenkomen met het aantal parameters, en de datatypes moeten overeenkomen óf ze worden geconverteerd. De waarde van het eerste argument wordt gekopieerd naar de eerste parameter en de waarde van het tweede argument wordt gekopieerd naar de tweede parameter enzovoort.

In listing 5.6 is een voorbeeld gegeven van een functie met twee parameters. De functie `print_rechthoek` drukt een rechthoek af met een bepaalde breedte en hoogte. De uitvoer van het in 5.6 gegeven programma is te zien in de figuur 5.1.



```
C:\> Command Prompt

+++++
+
+
+
+
+
+
+++++
+
+
+
+
+
+
+
+++++
```

Figuur 5.1: Uitvoer van het programma `print_rechthoek`

De functie is bedoeld voor het tekenen van een rechthoek met een breedte groter dan 2 en kleiner dan 80 en een hoogte van groter dan 2 en kleiner dan 40. De standaardfunctie

```

1 #include <stdio.h>
2 #include <assert.h>
3
4 void print_lijn(int lengte) {
5
6     assert (lengte > 2 && lengte < 80);
7     for (int teller = 0; teller < lengte; teller++) {
8         printf("+");
9     }
10    printf("\n");
11 }
12
13 void print_rechthoek(int breedte, int hoogte) {
14
15     assert (hoogte > 2 && hoogte < 40);
16     print_lijn(breedte);
17     for (int regel = 0; regel < hoogte - 2; regel++) {
18         printf("+");
19         for (int teller = 0; teller < breedte - 2; teller++) {
20             printf("_");
21         }
22         printf("+\n");
23     }
24     print_lijn(breedte);
25 }
26
27 int main(void) {
28
29     print_rechthoek(20, 8);
30     return 0;
31 }

```

Listing 5.6: Een voorbeeld van een functie met twee parameters.

`assert` wordt gebruikt om te controleren of de meegegeven argumenten aan deze voorwaarden voldoen. Als de expressie die in de `assert` wordt gedefinieerd `false` oplevert, dan wordt het programma afgebroken en wordt een foutmelding gegeven.

Merk op dat de functie `print_rechthoek` gebruik maakt van de functie `print_lijn` om de boven- en onderkant van de rechthoek te printen. De uitvoering van het programma start in `main`. Vervolgens wordt de functie `print_rechthoek` aangeroepen. De terugkeerlocatie wordt “onthouden” door de processor door het ergens in het geheugen op te slaan. Vervolgens wordt vanuit de functie `print_rechthoek` de functie `print_lijn` aangeroepen. Ook deze terugkeerlocatie wordt opgeslagen door de processor.

Na afloop van de functie `print_lijn` wordt het programma vervolgd op de terugkeerlocatie die als laatste is opgeslagen. Vervolgens wordt de `for`-instructie in `print_rechthoek` uitgevoerd en daarna wordt `print_lijn` nogmaals aangeroepen. Ook deze terugkeerlocatie wordt weer opgeslagen door de processor. Na afloop van de functie `print_lijn`

wordt het programma vervolgd op de terugkeerlocatie die zojuist is opgeslagen. Na afloop van de functie `print_rechthoek` wordt het programma vervolgd op de als eerste opgeslagen terugkeerlocatie. De volgorde waarin terugkeerlocaties worden opgeslagen en weer worden opgeroepen wordt *LIFO* (Last In First Out) genoemd. Hoe dat opslaan van die terugkeeradressen wordt gerealiseerd is te zien in het kader op pagina 51.

5.3 Functies met een returnwaarde

Tot nu toe hebben we functies geschreven die iets afdrukken met behulp van `printf`. Als we kijken naar de standaard C-functie `sin` dan zien we dat deze functie niets afdruckt maar de sinus van het argument teruggeeft. Een goed ontworpen functie moet in meerdere programma's gebruikt kunnen worden. Functies die het berekende resultaat meteen afdrukken zijn eigenlijk helemaal niet handig om te gebruiken in verschillende programma's. Stel eens voor dat de functie `sin` de sinus van het argument meteen zou afdrukken in plaats van het resultaat terug te geven. Deze functie zou dan slechts in een beperkt aantal gevallen te gebruiken zijn. De versie die het resultaat teruggeeft is in veel meer gevallen te gebruiken. Soms zal de waarde van de sinus afgedrukt moeten worden, door de returnwaarde van de `sin`-functie als argument door te geven aan de `printf`-functie:

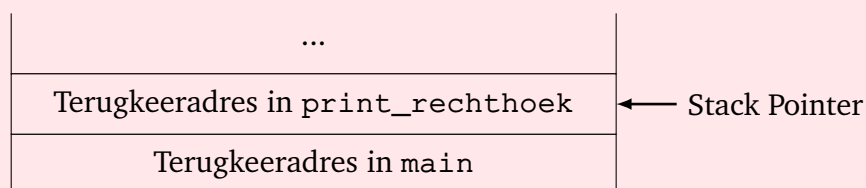
```
printf("%f"), sin(arg)); /* sin(arg) is printed */
```

Maar meestal zal de waarde van de sinus gebruikt worden in een berekening:

```
overstaande_rechthoekszijde = sin(hoek) * schuine_zijde;
```

STAPELEN MAAR...

De processor heeft een hardware-mechanisme om terugkeeradressen op te slaan, de zogenoemde *stack*. De stack is te vergelijken met een stapeltje A4-tjes. We kunnen er een vel opleggen of afhaken. De volgorde ligt vast; het laatste vel dat erop gelegd wordt, wordt ook weer als eerste eraf gehaald. De stack wordt gerealiseerd met een stukje RAM en de *stack pointer*. Dit is een speciaal aangewezen register in de processor. De bovenkant van de stack wordt aangewezen door de stack pointer. Elke keer als een functie wordt aangeroepen, wordt het terugkeeradres op de stack geplaatst en de stack pointer aangepast. Elke keer dat de functie geëindigd is, wordt het terugkeeradres van de stack gehaald en op dat adres gaat het programma daar verder. Uiteraard wordt de stack pointer ook dan aangepast. In figuur 5.2 is de stack getekend op het moment dat processor bezig is met het uitvoeren van de functie `print_lijn`.



Figuur 5.2: De stack op het moment dat de functie `print_lijn` wordt uitgevoerd.

De definitie van een C-functie die een waarde teruggeeft begint niet met `void` maar met het type van de waarde die wordt teruggegeven. Dit wordt het *returntype* van de functie genoemd. Vanuit de functie kan een waarde van het returntype worden teruggegeven met behulp van de `return`-statement. Er kan slechts één waarde worden teruggegeven via de `return`-statement.

In listing 5.7 is een voorbeeld gegeven van een functie die een waarde van het type `double` teruggeeft. Deze functie berekent het gemiddelde van drie als argumenten meegegeven gehele getallen.

```
1 #include <stdio.h>
2
3 double gemiddelde(int getal1, int getal2, int getal3) {
4
5     double resultaat = (getal1 + getal2 + getal3) / 3.0;
6     return resultaat;
7 }
8
9 int main(void) {
10
11     double gem = gemiddelde(27, 29, 33);
12     printf("%f\n", gem);
13     return 0;
14 }
```

Listing 5.7: Een eenvoudig voorbeeld van een functie met een returntype.

De definitie van de functie `gemiddelde` begint met de typeaanduiding `double` waarmee aangegeven wordt dat deze functie een waarde van het type `double` teruggeeft. In de functie wordt een `return`-statement gebruikt. De waarde van de expressie achter `return` wordt gekopieerd naar de plaats waar de functie wordt aangeroepen. De aanroep van de functie wordt als het ware vervangen door de returnwaarde. In `main` wordt de waarde die wordt teruggegeven door de functie `gemiddelde` opgeslagen in de variabele `gem` en vervolgens afgedrukt met behulp van `printf`.

Er is in het voorbeeld dat is weergegeven in listing 5.7 gebruik gemaakt van de variabelen `resultaat` en `gem`, maar beide variabelen zijn in feite niet nodig. In listing 5.8 is een compactere versie van dit voorbeeld weergegeven.

Een functie kan meerdere `return`-statements bevatten. Zodra een `return`-statement wordt uitgevoerd wordt de functie beëindigd. Als eenvoudig voorbeeld is in listing 5.9 een functie gegeven die de maximale waarde van twee, als argumenten meegegeven, gehele getallen teruggeeft. Merk op dat je deze functie ook kan gebruiken om de maximale waarde van 3 getallen te bepalen door de returnwaarde van de ene aanroep te gebruiken als argument van de volgende aanroep.

Dat werkt als volgt. In regel 15 worden drie gehele getallen ingelezen. Daarna wordt het maximum van deze drie getallen afgedrukt. Als argument voor `printf` wordt, naast de format string, een aanroep naar de functie `max` gerealiseerd. Dat argument is

```

1 #include <stdio.h>
2
3 double gemiddelde(int getal1, int getal2, int getal3) {
4
5     return (getal1 + getal2 + getal3) / 3.0;
6 }
7
8 int main(void) {
9
10    printf("%f\n", gemiddelde(27, 29, 33));
11    return 0;
12 }

```

Listing 5.8: Een compactere versie van het programma uit listing 5.7.

```

1 #include <stdio.h>
2
3 int max(int getal1, int getal2) {
4
5     if (getal1 > getal2) {
6         return getal1;
7     } else {
8         return getal2;
9     }
10 }
11
12 int main(void) {
13
14     int i1, i2, i3;
15     scanf("%d%d%d", &i1, &i2, &i3);
16     printf("De_maximale_waarde_is:_%d\n", max(i1, max(i2, i3)));
17     return 0;
18 }

```

Listing 5.9: Een programma dat drie gehele getallen inleest en de maximale waarde afdruckt.

```
max(i1, max(i2, i3))
```

en berekent eerst het maximum van `i2` en `i3` en de returnwaarde daarvan wordt samen met `i1` nog een keer in een aanroep naar `max` gebruikt. Het voordeel hiervan is dat er geen extra variabelen hoeven te worden gedeclareerd.

Omdat de functie meteen wordt beëindigd als `return` wordt uitgevoerd, is de `else` in de functie `max` overbodig. Zie listing 5.10 voor een compactere versie van deze functie.

Tot nu toe waren de functies die we als voorbeeld hebben bekeken nog niet zo heel goed bruikbaar in de praktijk. Daarom sluiten we deze paragraaf af met een functie die we wel degelijk in de praktijk zouden kunnen toepassen. In listing 5.11 is een functie `lees_geheel_getal`

```

1 int max(int getal1, int getal2) {
2
3     if (getal1 > getal2) {
4         return getal1;
5     }
6     return getal2;
7 }

```

Listing 5.10: Een compactere versie van de functie *max*.

gegeven die een geheel getal inleest, controleert of de ingevoerde waarde een getal is en of deze waarde tussen een als argument meegegeven minimale en maximale waarde ligt (inclusief). Zolang dit niet zo is, wordt een foutmelding gegeven en kan de gebruiker het opnieuw proberen,

```

1 #include <stdio.h>
2
3 /* Keep Visual Studio happy */
4 #pragma warning(disable : 4996)
5
6 int lees_geheel_getal(int min, int max) {
7     int getal, ret;
8
9     printf("Geef een geheel getal [%d..%d]:_", min, max);
10    ret = scanf("%d", &getal);
11
12    while (ret != 1 || getal < min || getal > max) {
13        char karakter;
14        do {
15            scanf("%c", &karakter);
16        } while (karakter != '\n');
17        printf("Onjuiste invoer. Probeer het opnieuw!\n");
18        printf("Geef een geheel getal [%d..%d]:_", min, max);
19        ret = scanf("%d", &getal);
20    }
21    return getal;
22 }
23
24 int main(void) {
25     printf("Het ingelezen toetscijfer is %d.\n", lees_geheel_getal
26         (1, 10));
27     return 0;
28 }

```

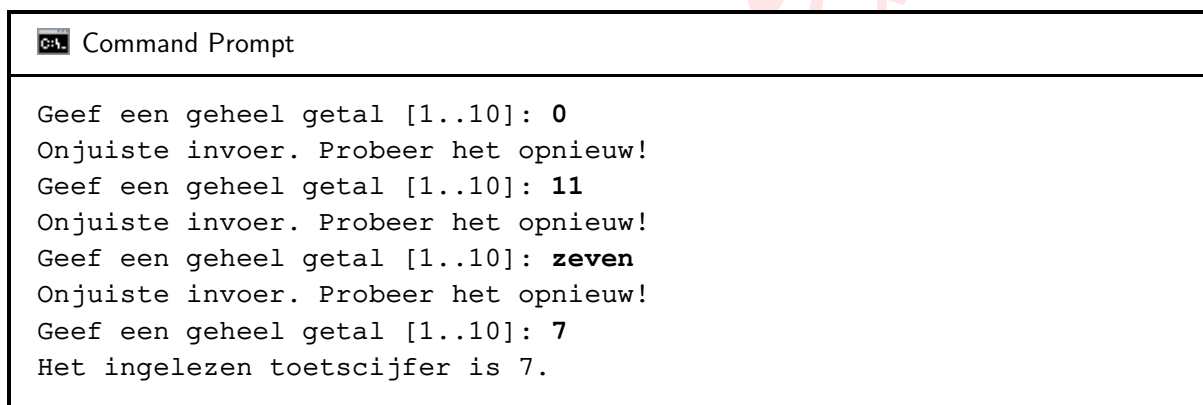
Listing 5.11: Een functie om een geheel getal in te lezen.

We hebben hier gebruik gemaakt van de returnwaarde van `scanf`. In regel 10 wordt `scanf` aangeroepen om een integer in te lezen en in variabele `getal` te zetten. De returnwaarde

van `scanf` wordt in variabele `ret` gezet. Als deze waarde 1 is, dan is het gelukt om een integer in te lezen. Zo niet, dan is het `scanf` niet gelukt om een integer in te lezen. In het `while`-statement in regel 12 kijken we of het inlezen *niet* gelukt is (`ret != 1`). Als dat waar is, dan wordt het `while`-statement uitgevoerd. Zo niet, dan kijken we nog of het getal buiten de grenzen ligt (`getal < min || getal > max`). Ook dan wordt het `while`-statement uitgevoerd.

In het `while`-statement wordt met behulp van een `do while`-statement een voor een karakters ingelezen totdat een newline gevonden is. Dan is de invoerbuffer van het toetsenbord leeg en kan opnieuw om een getal gevraagd worden. Zie meer over `scanf` en inlezen van het toetsenbord paragraaf 9.1.

De uitvoer van het in listing 5.11 gegeven programma is te zien in figuur 5.3. De door de gebruiker ingetypte invoer is vet weergegeven.



```
C:\> Command Prompt

Geef een geheel getal [1..10]: 0
Onjuiste invoer. Probeer het opnieuw!
Geef een geheel getal [1..10]: 11
Onjuiste invoer. Probeer het opnieuw!
Geef een geheel getal [1..10]: zeven
Onjuiste invoer. Probeer het opnieuw!
Geef een geheel getal [1..10]: 7
Het ingelezen toetscijfer is 7.
```

Figuur 5.3: Invoer van het programma

5.4 Zichtbaarheid en levensduur van lokale variabelen

Variabelen die binnen een functie gedeclareerd zijn, zijn alleen zichtbaar in de functie. We noemen dat lokale variabelen. Ze worden aangemaakt als de functie begint en worden verwijderd als de functie terugkeert naar de aanroeper. Dat betekent dat de levensduur loopt van het begin van de uitvoer van een functie tot de terugkeer naar de aanroeper. Lokale variabelen kunnen geïnitieerd worden. Elke keer als de functie aangeroepen wordt, krijgt een nieuwe versie van de variabele dan zijn waarde.

Parameters van de functie gedragen zich als lokale variabelen. Er wordt een kopie gemaakt van het bijbehorende argument en die kopie is zichtbaar in de functie. Parameters mogen daarom ook in een functie aangepast worden; de originele variabelen (het argument) wordt dus niet aangepast. Ook de levensduur van parameters is identiek aan die van lokale variabelen.

Omdat er een kopie van een variabele aan de parameters wordt meegegeven, kan de originele variabele niet worden aangepast. Een bekend voorbeeld is een functie die twee variabelen verwisseld. Dit komt onder andere voor in sorteerprogramma's. We definiëren een functie `wissel` met twee parameters `a` en `b`. De functie en de functieaanroep zijn te zien in listing 5.12. In de functie wordt een lokale variabele `hulpje` aangemaakt en met

behulp ervan worden `a` en `b` verwisseld. De functie `wissel` wordt met twee argumenten `x` en `y` aangeroepen. Vooraf en daarna worden `x` en `y` op het scherm afgedrukt.

```
1 #include <stdio.h>
2
3 void wissel(int a, int b)
4 {
5     int hulpje = a;
6     a = b;
7     b = hulpje;
8 }
9
10 int main(void)
11 {
12     int x = 7, y = 8;
13     printf("x=%d en y=%d\n", x, y);
14     wissel(x, y);
15     printf("x=%d en y=%d\n", x, y);
16     return 0;
17 }
```

Listing 5.12: Een functie om twee argumenten te verwisselen (foutief).

In figuur 5.4 is de uitvoer van het programma te zien. We merken op dat `x` en `y` niet verwisseld zijn. Dat kan ook niet, want er worden kopieën van `x` en `y` meegegeven. In functie `wissel` worden alleen de kopieën verwisseld, niet de originele variabelen.


Toch is het mogelijk om `x` en `y` te verwisselen. Hoe dat moet, zien we in hoofdstuk 7.

Lokale variabelen worden aan het begin van het uitvoeren van een functie aangemaakt en aan het einde weer verwijderd. Soms is het nodig om een lokale variabele te gebruiken die niet steeds opnieuw wordt aangemaakt en afgebroken wordt, maar behouden blijft *over aanroepen heen*. Dat kan door bij de declaratie van een lokale variabele de qualifier `static` te gebruiken. Zo'n lokale variabele wordt bij het starten van het programma geïnitieerd, expliciet door een constante expressie of impliciet met 0, en is beschikbaar zolang het programma draait.

CALL BY VALUE

De methode waarop een variabele wordt doorgegeven als parameters van een functie wordt *Call by Value* genoemd. Er wordt een kopie gemaakt van de variabele en die komt via de functieaanroep in een parameter terecht. De parameter kan worden aangepast, maar de originele variabele blijft zijn waarde behouden.

Er is ook nog een andere methode, *Call by Reference*. Deze methode wordt in hoofdstuk 7 besproken.

| |
|--|
|  Command Prompt |
| <pre>x = 7 en y = 8 x = 7 en y = 8</pre> |

Figuur 5.4: Uitvoer van de functie *wissel*.

In listing 5.13 is een functie te zien met een `static` lokale variabele. De variabele wordt bij het starten van het programma met 0 geïnitieerd (dit kunnen we ook achterwege laten). Elke keer dat we de functie aanroepen wordt de variabele met 1 verhoogd en wordt het resultaat teruggegeven aan de aanroeper.

```
1 #include <stdio.h>
2
3 int geefterug(void) {
4
5     static int count = 0;
6
7     count++;
8     return count;
9 }
10
11 int main(void) {
12
13     for (int i = 0; i < 10; i++) {
14         printf("Aanroep_%d\n", geefterug());
15     }
16
17     return 0;
18 }
```

Listing 5.13: Een functie met een `static` variabele.

5.5 Recursieve functies

Een functie die *zichzelf* aanroept wordt een *recursieve functie* genoemd. Een veel gebruikt voorbeeld van een toepassing van een recursieve functie is het berekenen van de faculteit van een natuurlijk getal n . De faculteit van n is gedefinieerd zoals gegeven is in (5.1).

$$n! = \prod_{i=1}^n i = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n \quad (5.1)$$

Dus bijvoorbeeld $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. Verder is afgesproken: $0! = 1$. Deze wiskundige functie is ook recursief te definiëren², zie (5.2).

$$n! = \begin{cases} 1, & \text{als } n \leq 1 \\ n \cdot (n-1)!, & \text{als } n > 1 \end{cases} \quad (5.2)$$

Dus $4!$ kan ook als volgt berekend worden: $4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Als we deze recursieve definitie coderen in C krijgen we een recursieve functie zoals te zien is in listing 5.14. Dit programma drukt de faculteiten af van 0 t/m 20.

```
1 #include <stdio.h>
2
3 unsigned long long int faculteit(unsigned long long int n) {
4
5     if (n <= 1) {
6         return 1;
7     }
8     return n * faculteit(n - 1);
9 }
10
11 int main(void) {
12
13     for (unsigned long long int i = 0; i <= 20; i++) {
14         printf("%2llu! = %llu\n", i, faculteit(i));
15     }
16     return 0;
17 }
```

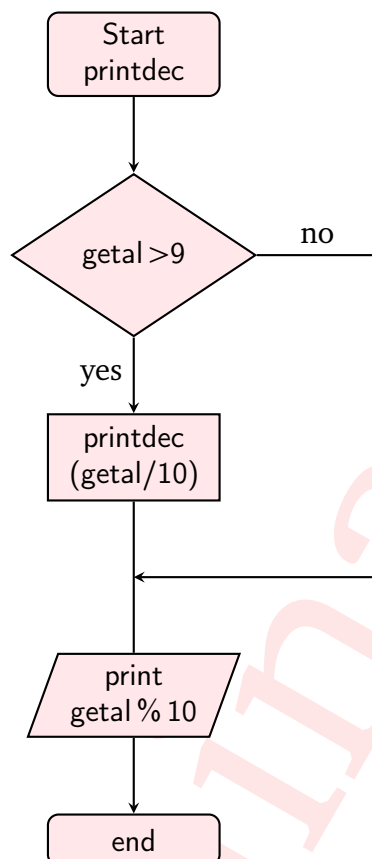
Listing 5.14: Een recursieve functie om de faculteit van een natuurlijk getal te berekenen.

Omdat de faculteit gedefinieerd is voor natuurlijke getallen (gehele getallen groter dan of gelijk aan nul) is als kwalificer van de parameter `unsigned` gekozen (een geheel getal zonder teken). Daarnaast wordt de faculteit van grotere getallen al snel erg groot. Daarom is voor de kwalificer `long long` gekozen. Op de meeste computers is dit een 64-bits getal. Het lukt dan nog net om $20!$ uit te rekenen. Bij grotere getallen treedt *overflow* op; het getal is te groot om in 64 bits op te slaan.

We zouden ook kunnen kiezen voor een `double`. Daar kunnen weliswaar grotere getallen in worden opgeslagen, maar de nauwkeurigheid is eindig, ongeveer 16 significante cijfers. Zo is de exacte waarde van $23! = 25852016738884976640000$ maar gebruik maken van een `double` geeft als resultaat 25852016738884978212864 . We zien dat de eerste 16 cijfers correct zijn, de laatste 7 cijfers zijn echter niet juist. Bij het gebruik van een `double` krijgen we dus slechts een benadering van $n!$ voor grotere waarden van n .

Een recursieve aanpak is vooral handig als er nog code wordt uitgevoerd *na* de recursieve aanroep. Stel dat je een natuurlijk getal wilt afdrukken in het tientallig talstelsel, dan kunnen we gebruik maken van het recursieve algoritme dat gegeven is in figuur 5.5.

² In een recursieve definitie wordt hetgeen gedefinieerd wordt in de definitie zelf gebruikt.



Figuur 5.5: Flowchart van de recursieve functie *printdec*.

Als we de functie binnenkomen, dan controleren we eerst of het meegegeven argument groter is dan 9. Zo ja, dan delen we het getal door 10 (daarmee “verdwijnt” de eenheid) en geven dit als argument mee aan weer een aanroep van *printdec*. Als het argument kleiner is dan 10, dan hebben we een eenheid en drukken we het argument af en keren terug naar de vorige aanroep. In de vorige aanroep drukken we de eenheid van het argument af.

Laten we eens het getal 3961 afdrukken. We roepen *printdec* aan met argument 3961. Dit argument is groter dan 9, dus wordt *printdec* weer aangeroepen maar nu met 396. Ook dat is groter dan 9 en zo wordt *printdec* weer aangeroepen met 39, enzovoorts. Uiteindelijk wordt het argument 3 aan *printdec* meegegeven en dat is kleiner dan 10, dus wordt dat argument afgedrukt. De functie wordt beëindigd en keert terug naar de vorige aanroep en daar was het argument 39. Met behulp van de modulus operator wordt alleen de 9 afgedrukt. Ook deze aanroep keert terug. Daar was het argument 396. De 6 wordt afgedrukt en de functie keert terug. We zijn nu teruggekeerd bij de eerste aanroep van *printdec* en daar was het argument 3961. Nu wordt de 1 afgedrukt en keert de functie terug naar *main*. Het programma voor de recursieve functie is te vinden in listing 5.15.

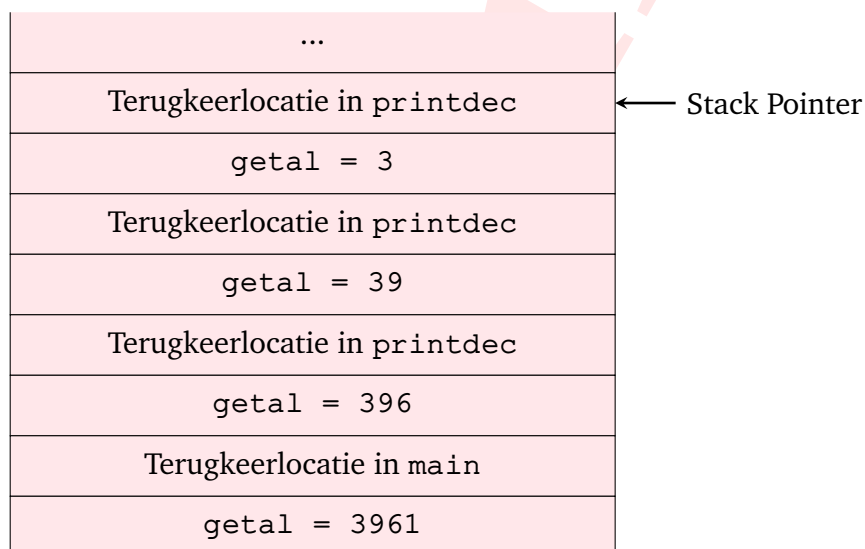
We maken in dit geval slim gebruik van het feit dat functies die achtereenvolgens aangeroepen worden in de omgekeerde volgorde terugkeren (Last In First Out, zie kader op pagina 51). Merk op dat ook het argument voorafgaande aan de functieaanroep op de stack wordt gezet. Dat is nodig omdat het argument steeds verandert. We kunnen het argument dus niet in een register van de processor opslaan. Nadat de functie voor de vierde keer is aangeroepen ziet de stack eruit zoals te zien is in figuur 5.6.

```

1 #include <stdio.h>
2
3 void printdec(int getal) {
4     if (getal > 9) {
5         printdec(getal / 10);
6     }
7     printf("%d", getal % 10);
8 }
9
10 int main(void) {
11     int i = 3961;
12
13     printdec(i);
14     return 0;
15 }

```

Listing 5.15: Een recursieve functie om een natuurlijk getal af te drukken.



Figuur 5.6: De stack op het moment dat de functie *printdec* vier maal is aangeroepen.

* 5.6 Complexiteit van recursieve functies

Recursieve functies zijn vaak eenvoudig van aard. We spreken dan van een lage complexiteit. Maar we kunnen ook op andere manieren naar de recursieve functies kijken. Twee belangrijke eigenschappen van recursieve functies zijn de *Call Complexity* (het aantal aanroepen van de functie) en de *Time Complexity* (hoe lang duurt het om een getal te berekenen).

Een bekend voorbeeld waar veel over geschreven is, is de reeks van Fibonacci. De reeks heeft de recursieve definitie voor een natuurlijk getal n :

$$F(n) = \begin{cases} 0, & \text{als } n = 0 \\ 1, & \text{als } n = 1 \\ F(n-1) + F(n-2), & \text{als } n > 1 \end{cases} \quad (5.3)$$

Dus een Fibonacci-getal $F(n)$ is uit te rekenen door de som te nemen van de twee eerdere Fibonacci-getallen. Rekenen we een aantal Fibonacci-getallen uit dan komen we tot de reeks:

$$F(0) = 0, F(1) = 1, F(2) = 1, F(3) = 2, F(4) = 3, F(5) = 5, F(6) = 8, F(7) = 13 \quad (5.4)$$

Een recursieve functie hiervoor is vrij eenvoudig. Dit is te zien in listing 5.16.

```
1 int fib(int n) {  
2  
3     if (n <= 1) {  
4         return n;  
5     }  
6     return fib(n - 1) + fib(n - 2);  
7 }
```

Listing 5.16: *Recursieve functie voor berekenen van Fibonacci-getallen.*

Stel dat we $F(10)$ willen uitrekenen, dan vinden we via de functie dat $F(10) = F(9) + F(8)$. Maar $F(9) = F(8) + F(7)$ en $F(8) = F(7) + F(6)$ enzovoorts. Het probleem zit niet zozeer in het groter worden van de getallen maar in het aantal (recursieve) *aanroepen* van de functie en in het verlengde daarvan de totale tijd die nodig is een Fibonacci-getal te berekenen.

In listing 5.17 is een programma te zien dat het aantal aanroepen van de functie en de totale tijd om een Fibonacci-getal te berekenen bijhoudt. Omdat de getallen vrij snel groot worden hebben gebruik gemaakt van unsigned long long-getallen dat bij veel computers neerkomt op 64-bits getallen. We houden bij elke berekening bij hoe vaak de functie is aangeroepen en maken gebruik van de clock-functie in C om de rekentijd bij te houden.

We berekenen de getallen $F(37)$ t/m $F(43)$. De uitvoer van het programma is te zien in figuur 5.7. We zien dat voor het berekenen van $F(37)$ 24157817 functieaanroepen nodig zijn en de rekentijd bedraagt 1200³. Voor het berekenen van $F(43)$ zijn meer dan 1,4 miljard aanroepen nodig en de rekentijd bedraagt 20118.

| C:\ Command Prompt | |
|---|--|
| Fib(37) = 24157817, Calls: 78176337, Time: 1200 | |
| Fib(38) = 39088169, Calls: 126491971, Time: 1878 | |
| Fib(39) = 63245986, Calls: 204668309, Time: 3019 | |
| Fib(40) = 102334155, Calls: 331160281, Time: 4817 | |
| Fib(41) = 165580141, Calls: 535828591, Time: 7740 | |
| Fib(42) = 267914296, Calls: 866988873, Time: 12487 | |
| Fib(43) = 433494437, Calls: 1402817465, Time: 20118 | |

Figuur 5.7: *Uitvoer van een Fibonacci-programma.*

³ De tijdschaal is niet zozeer van belang. Het gaat ons meer om de relatieve uitkomsten.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 long long int calls = 0;
6
7 unsigned long long int fib(unsigned long long int n) {
8
9     calls = calls + 1;
10
11     if (n <= 1) {
12         return n;
13     }
14     return fib(n - 1) + fib(n - 2);
15 }
16
17 int main() {
18
19     int i = 1;
20
21     for (i = 37; i < 44; i++) {
22         calls = 0;
23         clock_t starttime = clock(), endtime;
24
25         printf("Fib(%d) = %llu, ", i, fib(i));
26         endtime = clock() - starttime;
27
28         printf("Calls: %lld, ", calls);
29         printf("Time: %d\n", endtime);
30     }
31
32     return 0;
33 }

```

Listing 5.17: *Programma om Fibonacci-getallen af te drukken.*

Het is aan te tonen dat het aantal aanroepen $C(n)$ voor het berekenen van $F(n)$ bedraagt:

$$C(n) = 2F(n + 1) - 1 \quad (5.5)$$

Voor het tijd die een berekening kost is van de orde:

$$T(n) = \mathcal{O}(1,618^n) \quad (5.6)$$

Dit wordt de “Big O-notation” genoemd. Dit zegt ons niet zozeer hoeveel rekentijd een berekening kost maar wel wat de verhouding is met de rekentijd van het volgende getal.

Overigens zijn er geweldig mooie relaties tussen de getallen:

$$\begin{aligned}\frac{F(43)}{F(42)} &= \frac{433494437}{267914296} \approx 1,618 \\ \frac{C(43)}{C(42)} &= \frac{1402817465}{866988873} \approx 1,618 \\ \frac{T(43)}{T(42)} &= \frac{20118}{12487} \approx 1,618\end{aligned}\tag{5.7}$$

Voor steeds groter wordende waarde van n convergeren de verhoudingen naar de exacte waarde $\frac{1 + \sqrt{5}}{2}$. Deze waarde wordt de *gulden snede* genoemd⁴.

5.7 Pointers en functies als parameter

Een pointer is een variabele met als inhoud het *adres* van een (andere) variabelen. Met behulp van pointers kunnen we dus niet een kopie van een variabele als argument meegeven, maar een adres waarop de (originele) variabele te vinden is. Ook een functie, of eigenlijk, het beginadres van een functie kan als argument en parameter dienen. We bespreken dit in hoofdstuk 7.

5.8 Functies die meer dan één waarde teruggeven

Met behulp van het `return`-statement kan in C vanuit een functie slechts één waarde teruggegeven worden. Willen we meerdere waarden teruggeven, dan kan dat op twee manieren. We kunnen:

- de waarden “inpakken” in een *structure*, dit is vooral handig als verschillende datatypes moeten worden teruggegeven, zie hiervoor hoofdstuk 8;
- de waarden teruggeven via zogenoemde *Call by Reference*-parameters, dit is vooral handig bij dezelfde datatypes (array's), zie hiervoor hoofdstuk 6.

⁴ De waarde is bij benadering 1,61803398874989484820458683436563811772.

6

Array's

De enkelvoudige datatypes hebben een beperking in de zin dat een groot aantal variabelen niet gemakkelijk kan worden verwerkt. Stel dat we de som van een aantal variabelen willen bepalen. We moeten dan voor elke variabele een declaratie geven en in het programma moeten we de som bepalen door alle variabelen op te tellen. Willen nu meer variabelen optellen, dan moeten we door het hele programma aanpassingen maken. Dat is nog wel te doen als het aantal variabelen beperkt is maar naar mate dat aantal groeit, wordt het programma groter en complexer. We kunnen veel van dit soort vraagstukken elegant oplossen met behulp van een *array*.

We bespreken array's op vele manieren, onder andere hoe een array wordt gedeclareerd en het wordt gebruikt. We laten zien dat het mogelijk is om “langs een array te lopen” met behulp van een lus. We kunnen tweedimensionale array's gebruiken waarbij een array uit rijen en kolommen bestaat. We zien dat een *string* een array is van karakters en dat in C bewerkingen op strings mogelijk zijn. We kunnen een array als parameter aan een functie meegeven met de kanttekening dat de grootte van de array niet berekend kan worden in de functie. Maar er zijn ook zaken die we hier niet bespreken, namelijk de relatie tussen array's en pointers. Dat wordt besproken in hoofdstuk 7.

6.1 Declaratie en selectie

Een *array* is een manier om bij elkaar behorende gegevens onder één naam te groeperen en te bewerken. Een array is een *samengesteld datatype*, een datatype dat bestaat uit een aantal enkelvoudige datatypes. Zo zorgt de declaratie

```
int a[10];
```

ervoor dat we gebruik kunnen maken van de tien `int`-variabelen:

```
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
```

Bij de declaratie wordt het aantal *elementen* opgegeven. In het voorbeeld zijn dat er 10. De nummering van de elementen begint bij 0 en loopt tot het aantal elementen minus 1.

Tevens wordt bij de declaratie het datatype opgegeven. Dit mag elk gangbaar datatype zijn.

We kunnen een element selecteren door gebruik te maken van de blokhaken `[]` met daarin het elementnummer. Dit elementnummer mag ook berekend worden met een expressie als het resultaat van de expressie maar een integer is. Om het achtste element toe te kennen aan de variabele `i` gebruiken we de toekenning:

```
i = a[7];
```

Om de variabele `i` toe te kennen aan het vijfde element gebruiken we de toekenning:

```
a[4] = i;
```

We mogen alleen de elementen gebruiken die gedeclareerd zijn, dus de expressies

```
i = a[-1]; j = a[99];
```

zijn niet geldig¹. Overigens voegen de meeste compilers geen extra programmatuur toe om deze grenzen te bewaken. De programmeur moet het zelf in de gaten houden.

Een prettige manier om een array voor te stellen is om de array als een rij vierkante hokjes te tekenen. Boven de hokjes schrijven we de elementnummers. In de hokjes schrijven we de waarden (of inhouden) van de elementen. Een voorbeeld is te zien in figuur 6.1.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|----|----|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| a | 2 | 5 | 3 | 9 | 8 | 2 | 6 | -1 | 10 | 0 |

Figuur 6.1: Uitbeelding van een array met tien elementen.

Overigens zijn de blokhaken `[]` als geheel een *operator* met een hoge prioriteit.

6.2 Initialisatie

De array `a` met declaratie

```
int a[10];
```

wordt, als *globale* variabele, automatisch geïnitieerd met nullen. Als *lokale* variabele is de inhoud onbekend. We kunnen de array bij declaratie gelijk initialiseren. Om de array in figuur 6.1 direct te initialiseren gebruiken we een rij getallen tussen accolades en gescheiden door komma's:

```
int a[10] = {2, 5, 3, 9, 8, 2, 6, -1, 10, 0};
```

Omdat de compiler de lengte van de lijst kan uitrekenen, mogen we het aantal elementen bij de declaratie weglaten. We kunnen dus ook schrijven:

```
int a[] = {2, 5, 3, 9, 8, 2, 6, -1, 10, 0};
```

¹ Technisch gezien zijn de elementnummers wel geldig. De compiler berekent via het elementnummer de plaats in het geheugen waar het element zich bevindt. Door gebruik te maken van *pointers* kan op zinnige wijze gebruik gemaakt worden van negatieve elementnummers.

Als de lijst korter is dan het aantal opgegeven elementen, dan worden de overige elementen met nullen geïnitieerd:

```
int a[10] = {2, 5 ,3 ,9, 8};    /* the rest are zero */
```

Willen we een lokale array met nullen initialiseren, dan kunnen we volstaan met één nul:

```
int a[10] = {0};    /* all elements are zero */
```

Een *constante array* moet altijd geïnitieerd worden en de elementen mogen niet gewijzigd worden:

```
const int a[] = {2, 5 ,3 ,9, 8, 2, 6, -1, 10, 0};
```

6.3 Eendimensionale array

De array *a* wordt na de declaratie

```
int a[10];
```

een *eendimensionale* array genoemd. Zoals al eerder is geschreven, mogen we het elementnummer van een array met een expressie uitrekenen, op voorwaarde dat het uitkomst een geheel getal is en binnen de grenzen van van array ligt. In listing 6.1 is een programma te zien dat een array vult met kwadraten van 0 t/m 9. We gebruiken *i* als lusvariabele en berekenen voor elk element het kwadraat. Daarna bepalen we de som van twee kwadraten door de juiste elementen uit de array bij elkaar op te tellen. Vervolgens drukken we de gegevens op het scherm af.

```
1 #include <stdio.h>
2
3 int main(void) {
4
5     int kwad[10], i;
6     int a = 3, b = 5, som;
7
8     for (i = 0; i < 10; i = i + 1) {
9         kwad[i] = i * i;
10    }
11
12    som = kwad[a] + kwad[b];
13
14    printf("De_som_van_de_kwadraten_%d_en_%d_is_%d\n", a, b, som);
15
16    return 0;
17 }
```

Listing 6.1: Afdrukken van de som van twee kwadraten kwadraten.

Een interessant geval is dat we de inhoud van een element van een array mogen gebruiken als een elementnummer van een andere array. Dit is te zien in listing 6.2. We declareren een array *rij* en initialiseren de array met de getallen waarvan we de kwadraten willen

optellen. Daarna vullen we de array `kwad` met de kwadraten zoals we dat al eerder deden. In de regels 13 t/m 15 worden een voor een de waarden uit array `rij` opgevraagd en gebruikt als elementnummer voor array `kwad`. Merk op dat `rij[i]` eerst wordt bepaald en de uitkomst daarvan wordt gebruikt om een element uit de array `kwad` te selecteren.

```
1 #include <stdio.h>
2
3 int main(void) {
4
5     int kwad[10], i, som = 0;
6
7     int rij[5] = { 2, 5, 8, 1, 6 };
8
9     for (i = 0; i < 10; i = i + 1) {
10         kwad[i] = i * i;
11     }
12
13     for (i = 0; i < 5; i = i + 1) {
14         som = som + kwad[rij[i]];
15     }
16
17     printf("De_som_van_de_kwadraten_is_%d\n", som);
18
19     return 0;
20 }
```

Listing 6.2: Afdrukken van de som van twee kwadraten kwadraten.

6.4 Aantal elementen van een array bepalen

We kunnen bij de declaratie van een array het aantal elementen opgeven, maar dat hoeft niet als de array gelijk geïnitieerd wordt. De declaratie

```
int a[] = {2, 5, 3, 9, 8, 2, 6, -1, 10, 0};
```

zorgt ervoor dat `a` uit 10 elementen bestaat. In een programma moeten we erop toezien dat we niet buiten de array komen. Als we de initialisatie groter of kleiner maken, dan verandert het aantal elementen en moet het programma hierop aangepast worden. Met behulp van de operator `sizeof` kunnen we het aantal elementen *tijdens compile-time* uitrekenen. Let erop dat `sizeof` de grootte *in bytes* uitrekent. We moeten dus de grootte van de array delen door de grootte van één element. Het aantal elementen van een array wordt berekend met:

```
int siz = sizeof a / sizeof a[0];
```

Merk op dat `sizeof` een hogere prioriteit heeft dan `/` maar lager dan `[]`. We hoeven dus geen haakjes te zetten om `a[0]`. We kunnen het getal 5 regel 13 in listing 6.2 vervangen door een berekening van het aantal elementen. Dit is te zien in listing 6.3.

```

1  ...
2  for (i = 0; i < sizeof rij / sizeof rij[0]; i = i + 1) {
3      som = som + kwad[rij[i]];
4  }
5  ...

```

Listing 6.3: Bepalen van het aantal elementen in een array.

6.5 Tweedimensionale array

Een *tweedimensionale* array van 5×3 elementen wordt gedeclareerd met:

```
int matrix[5][3];
```

Het eerste elementnummer wordt het *rijnummer* genoemd en het tweede elementnummer wordt het *kolomnummer* genoemd. Het aantal rijen en kolommen mag natuurlijk gelijk zijn.

We kunnen de array tegelijk met de declaratie initialiseren. We gebruiken een dubbele array-initialisatie met accolades. De buitenste accolades geven de afbakening van de initialisatie aan. Binnen de accolades vormen we drie groepen die zijn afgebakend door accolades en gescheiden zijn door komma's.

```
int matrix[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

We mogen hierbij het aantal rijen weglaten. De compiler berekent die automatisch. Als we een element uit de array willen gebruiken dan geven we eerst het rijnummer op en daarna het kolomnummer:

```
int i = matrix[2][1]; /* row 2, column 1 */
```

Een veelvoorkomende fout is om het rijnummer en kolomnummer te scheiden door een komma. Dat kan niet want de komma werkt als de komma-operator:

```
int i = matrix[2,1]; /* WRONG */
```

Let erop dat het verwisselen van het rijnummer en kolomnummer een ander element selecteert. Dus

```
matrix[2,1] en matrix[1,2]
```

zijn twee verschillende elementen.

We kunnen het aantal rijen en kolommen van een array bepalen met behulp van `sizeof`. Dit is te zien in listing 6.4. In regel 7 bepalen we het aantal rijen door de grootte van de gehele array te delen door de grootte van een rij. Dus

```
int rows = sizeof twodim / sizeof twodim[0];
```

kent het aantal rijen toe aan `rows`. In regel 8 bepalen we het aantal kolommen door de grootte van een rij te delen door de grootte van één element. Dus

```
int cols = sizeof twodim[0] / sizeof twodim[0][0];
```

kent het aantal kolommen toe aan `cols`. Daarna drukken we de twee variabelen af.

```
1 #include <stdio.h>
2
3 int main(void) {
4
5     int twodim[15][20];
6
7     int rows = sizeof twodim / sizeof twodim[0];
8     int cols = sizeof twodim[0] / sizeof twodim[0][0];
9
10    printf("Rows:_%d\n", rows);
11    printf("Cols:_%d\n", cols);
12
13    return 0;
14 }
```

Listing 6.4: *Bepalen van het aantal rijen en kolommen.*

6.6 Afdrukken van array

Een array kan niet als één eenheid worden afgedrukt. We moeten dat doen met behulp van een lus. Dit is te zien in listing 6.5. In regel 5 declareren en initialiseren we de array met in dit geval 12 elementen. Omdat de array kan wijzigen berekenen we in regel 7 het aantal elementen waaruit de array bestaat en drukken dat af in regel 9. Daarna selecteren we een voor een de elementen uit de array. Dat doen we met een `for`-lus. Binnen de lus, in regel 12, drukken we de waarde van het element af.

```
1 #include <stdio.h>
2
3 int main(void) {
4
5     int rij[] = { 2, 5, 8, 1, 6, 7, 3, 9, 0, 4, 7, 8}, i;
6
7     int len = sizeof rij / sizeof rij[0];
8
9     printf("De_array_heeft_%d_elementen:_", len);
10
11    for (i = 0; i < len; i = i + 1) {
12        printf("%d%c", rij[i], i < len-1 ? '_' : '\n');
13    }
14
15    return 0;
16 }
```

Listing 6.5: *Afdrukken van een array.*

We zullen de aanroep van de `printf`-functie even uitleggen. Het eerste argument is de format string waarin twee format specifications staan: `%d` en `%c`. De eerste geeft aan dat een integer moet worden afgedrukt en de tweede geeft aan dat één karakter moet worden afgedrukt. Het tweede argument is `rij[i]`, het element uit de array. Het derde argument wordt gevormd door een *conditionele expressie*:

```
i < len-1 ? ' ' : '\n'
```

Dit werkt als volgt. Voor het vraagteken staat de relationele expressie `i < len-1`. Als de expressie waar is, dan wordt een *spatie* afgedrukt. Als de expressie onwaar is dan wordt een *newline* afgedrukt. Het gevolg is dat na een element een spatie wordt afgedrukt, behalve na het laatste element, dan wordt een *newline* afgedrukt. De conditionele expressie wordt besproken in paragraaf 2.9.4.

6.7 Array's en functies

Een array kan als argument van een functie dienen. Een functie kan *geen* array teruggeven. Dat heeft te maken met de manier waarop array's overgedragen worden. Bij enkelvoudige datatypes wordt een *kopie* gemaakt van de originele variabelen en die kopieën wordt doorgegeven. Bij overdragen van een array wordt geen kopie gemaakt, maar wordt de plaats in geheugen waar de array begint overgedragen². Omdat niet een complete kopie wordt gemaakt, kan de functie de grootte van de array niet uitrekenen; het kent immers de declaratie niet. Er moet dus altijd een parameter met de grootte van de array worden overgedragen.

In listing 6.6 is een functie te zien die de positie bepaalt van een getal dat in een array voorkomt. De waarde `-1` wordt teruggegeven als het getal niet in de array voorkomt. De aanroepende functie moet hierop testen. Als het getal meerdere keren voorkomt, wordt de eerste positie teruggegeven. Om aan te geven dat de eerste parameter een array is, wordt de declaratie `int ary[]` gegeven. De tweede parameter is de grootte van de array in elementen en de derde parameters is het getal dat gezocht wordt.

```
1 int findnumber(int ary[], int siz, int num) {
2     int pos;
3
4     for (pos = 0; pos < siz; pos = pos + 1) {
5         if (ary[pos] == num) {
6             return pos;
7         }
8     }
9     return -1;
10 }
```

Listing 6.6: Functie voor het vinden van een getal in een array.

² Dit is gedaan uit efficiëntieoverwegingen. Stel dat een array uit 10000 elementen bestaat, dan zou er een kopie moeten worden gemaakt van al die elementen. Niet alleen de geheugenruimte is een probleem, ook de snelheid waarmee de kopie wordt gemaakt vormt een obstakel. En wat te denken als een functie weer een functie aanroept met dezelfde array. Dan moet er weer een kopie gemaakt worden.

In listing 6.7 is te zien hoe de functie wordt aangeroepen. Zie regel 9. Als eerste argument wordt de *naam* van de array opgegeven. De compiler weet aan de hand van de functiedefinitie en de declaratie van de array dat het om een array gaat. Het tweede argument is het aantal elementen van de array dat berekend wordt met behulp van `sizeof`. Het derde argument is het getal waarnaar gezocht wordt.

```
1 #include <stdio.h>
2
3 int findnumber(int ary[], int siz, int num);
4
5 int main(void) {
6
7     int rij[] = { 4, 8, 3, 8, 2, 3, 4, 5, 1, 2 };
8
9     int plek = findnumber(rij, sizeof rij / sizeof rij[0], 5);
10
11     printf("Plek_%d\n", plek);
12
13     return 0;
14 }
```

Listing 6.7: Aanroepen van de functie *findnumber*.

Omdat de plaats van de array in het geheugen als argument wordt meegegeven, kunnen we de array *in de functie* wijzigen. De functie `patcharray` in listing 6.8 vervangt alle elementen die gelijk zijn aan parameter `what` met de parameter `with`. De functie geeft aan het einde het aantal vervangingen terug. Omdat er geen kopie wordt meegegeven maar het adres van het eerste element, kan de functie bij de originele array komen en zodoende de elementen veranderen.

```
1 int patcharray(int ary[], int siz, int what, int with) {
2
3     int pos;
4     int count = 0;
5
6     for (pos = 0; pos < siz; pos = pos + 1) {
7         if (ary[pos] == what) {
8             ary[pos] = with;
9             count = count + 1;
10        }
11    }
12    return count;
13 }
```

Listing 6.8: Functie om een getal te vervangen door een ander getal.

Doorgewinterde C-programmeurs zouden het trouwens zo willen oplossen (listing 6.9):

```

1 int patcharrayunreadable(int ary[], int siz, int what, int with) {
2     int count = 0;
3
4     while (--siz > 0) {
5         ary[siz] == what ? ary[siz] = with, count++ : 0;
6     }
7     return count;
8 }

```

Listing 6.9: *Functie om een getal te vervangen door een ander getal.*

6.8 Array's vergelijken

Een array is een samengesteld datatype en kan niet zonder meer met een andere array vergeleken worden. Het is verleidelijk om de twee namen van de array's te vergelijken zoals te zien is in listing 6.10. Door de manier waarop C met de namen omgaat zorgt ervoor dat niet de inhoud van de array's vergeleken worden, maar de adressen van het eerste element. Dit is vergelijkbaar met de array als argument van een functie.

```

1 #include <stdio.h>
2
3 int main(void) {
4
5     int ary1[10] = { 0 }, ary2[10] = { 0 };
6
7     printf("De_array's_zijn_%sgelijk\n", ary1 == ary2 ? "" : "on");
8 }

```

Listing 6.10: *Vergelijken van twee array's (foutief).*

De juiste manier om twee array's te vergelijken is om de elementen paarsgewijs te vergelijken. We doen dit met behulp van de functie `arraycompare`, te zien in listing 6.11.

```

1 int arraycompare(int ary1[], int ary2[], int siz) {
2
3     int i;
4
5     for (i = 0; i < siz; i++) {
6         if (ary1[i] != ary2[i]) {
7             return 0;
8         }
9     }
10    return 1;
11 }

```

Listing 6.11: *Vergelijken van twee array's.*

De twee array's en het aantal elementen worden als argumenten meegegeven. De array's moeten natuurlijk uit een gelijk aantal elementen bestaan. Met een `for`-lus lopen we langs alle elementen van de array's. We vergelijken telkens twee elementen op dezelfde positie. Als twee elementen ongelijk zijn dan weten we dat de array's niet aan elkaar gelijk zijn en geven een 0 terug. Als alle elementen vergeleken zijn en we uit de `for`-lus komen dan weten we dat alle elementen gelijk zijn en geven we een 1 terug. In listing 6.12 is te zien hoe we de functie moeten aanroepen.

```

1 #include <stdio.h>
2
3 int arraycompare(int ary1[], int ary2[], int siz);
4
5 int main(void) {
6
7     int ary1[10] = { 0 }, ary2[10] = { 0 };
8
9     printf("De_array's_zijn_%sgelijk\n", arraycompare(ary1, ary2,
10                                     sizeof ary1 / sizeof ary1[0]) ? " " : "on");
11 }

```

Listing 6.12: Vergelijken van twee array's.

De standard library bevat een aantal functies waarmee we gemakkelijk array's kunnen vergelijken, onafhankelijk van het datatype. Dit wordt besproken in hoofdstuk 7.

6.9 Strings

Een *string* is een eendimensionale array van karakters afgesloten met een nul-karakter. In tegenstelling tot veel andere talen is een string in C dus geen enkelvoudig datatype. Dat betekent dat we niet op eenvoudige wijze strings kunnen manipuleren zoals het bepalen van de lengte, inkorten en veranderen. Gelukkig zijn er functies beschikbaar die het programmerwerk enigszins verlichten.

We definiëren een string door de rij karakters te beginnen en af te sluiten met aanhalingstekens. De declaratie

```
char str[] = "Ik ben een string";
```

initialiseert een string van 18 karakters. Het aantal elementen van de array wordt automatisch bepaald. In figuur 6.2 is te zien hoe de string in het geheugen ligt. Het einde van de string wordt gekenmerkt door een *nul-karakter* (`\0`). Een nul-karakter is een geheel getal, meestal een byte, waarvan alle bits 0 zijn. Er is dus altijd één karakter meer nodig dan de karakters die opgegeven zijn in de string.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| str | I | k | | b | e | n | | e | e | n | | s | t | r | i | n | g | \0 |

Figuur 6.2: Uitbeelding van een stringarray.

Door middel van dit nul-karakter kan een programma het einde van de string vinden in het geheugen. Het berekenen van het aantal karakters kan heel eenvoudig met een lus uitgevoerd worden. Dit is te zien in listing 6.13. In regel 3 declareren en initialiseren we de `str` met een string constante. In de `while`-lus lopen we een voor een langs de elementen en zolang we het nul-karakter niet gevonden hebben, verhogen we een teller. Na het uitvoeren van de lus is het aantal karakters beschikbaar in variabele `len`.

```

1 #include <stdio.h>
2
3 int main() {
4
5     char str[] = "Hallo_wereld!";
6     int len = 0;
7
8     while (str[len] != '\0') {    /* while not end of string ... */
9         len = len + 1;           /* point to the next character */
10    }
11
12    printf("Lengte_is_%d\n", len);
13 }

```

Listing 6.13: Berekenen van de lengte van een string.

6.9.1 Stringfuncties

De standard library kent een groot aantal functies waarmee strings kunnen worden gemanipuleerd. Om ze te gebruiken moet het header-bestand `string.h` worden ingelezen. We hebben de vier belangrijkste functies op een rijtje gezet. Zie tabel 6.1.

Tabel 6.1: Enkele standaard functies voor stringmanipulaties.

| | |
|-------------------------------|--|
| <code>strlen(s)</code> | Geeft de lengte van <code>s</code> in karakters exclusief het nul-karakter. |
| <code>strcpy(to, from)</code> | Kopieert de string <code>from</code> naar string <code>to</code> . |
| <code>strcat(to, from)</code> | Voegt de string <code>from</code> toe aan het einde van string <code>to</code> . |
| <code>strcmp(s1, s2)</code> | Vergelijkt <code>s1</code> met <code>s2</code> . |

De functie `strlen` geeft het aantal karakter in de string terug exclusief het nul-karakter. De functie `strcpy` kopieert de string `from` naar de string `to`. Let erop dat de ruimte van `to` groot genoeg is. De functie test dat niet en een *buffer overflow* is dan het gevolg. De functie `strcat` voegt de string `from` toe aan het einde van string `to`. In dit verband wordt vaak het Engelse woord *concatenation* gebruikt dat aaneenschakelen betekent. Ook hier moet erop gelet worden dat de ruimte in `to` groot genoeg is.

De functie `strcmp` vergelijkt `s1` met `s2` op *alfabetische ordening* met inachtneming van de karakterset van de computer. Alfabetische ordening wil min of meer zeggen: zoals de woorden in een woordenboek. Dat betekent dat de string "Jan" voorgaat op "Janus". De functie geeft `-1` terug als `s1` "kleiner" is dan `s2`, geeft `0` terug als de strings identiek

zijn en geeft 1 terug als `s1` “groter” is dan `s2`. Let erop dat de karakterset van de computer gebruikt wordt dus “123” is kleiner dan “Jan” en “Janus” is kleiner dan “janus”. Zie bijlage A over de ordening van de ASCII-karakterset.

Dat deze functies niet zo ingewikkeld zijn is te zien listing 6.14 waar we overigens gebruik hebben gemaakt van enkele juweeltjes van C-snelschrift. We dagen de lezer uit om de functies te analyseren.

```
1 int strlen(char str[]) {
2     int len;
3
4     for (len = 0; str[len] != '\0'; len++);
5
6     return len;
7 }
8
9 void strcpy(char to[], char from[]) {
10
11     int len;
12
13     for (len = 0; (to[len] = from[len]) != '\0'; len++);
14 }
15
16 int strcmp(char str1[], char str2[]) {
17
18     int len;
19
20     for (len = 0; str1[len] == str2[len]; len++) {
21         if (str1[len] == '\0') {
22             return 0;
23         }
24     }
25     return str1[len] < str2[len] ? -1 : 1;
26 }
27
28 void strcat(char to[], char from[]) {
29     int i, j;
30
31     for (i = j = 0; to[i] != '\0'; i++);
32
33     while ((to[i++] = from[j++]) != '\0');
34 }
```

Listing 6.14: Een implementatie van de stringfuncties.

7

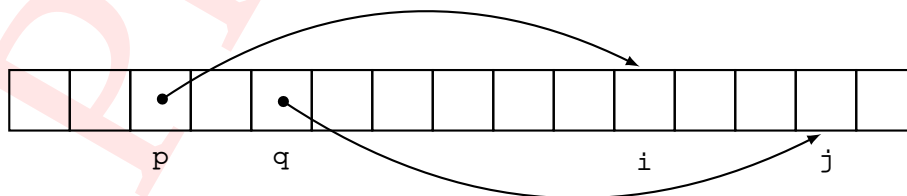
Pointers

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 efficiënt 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 7.1 is dat te zien. Elk vakje stelt een geheugenplaats voor¹. 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 7.1: 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 7.3.

7.1 Pointers naar enkelvoudige datatypes

In listing 7.1 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 7.1: Enkele declaraties van pointers.

We kunnen pointers uitbeelden door middel van vakjes, zoals te zien is in figuur 7.2. 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.



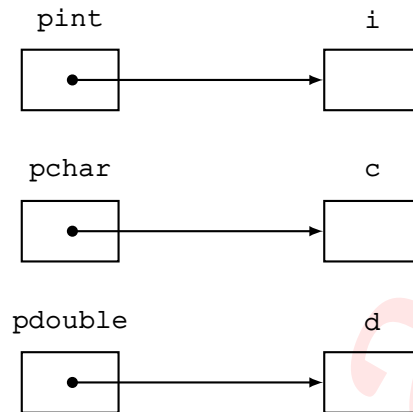
Figuur 7.2: Voorstelling van drie pointers in het geheugen.

Aan een pointer is het adres van een variabele van hetzelfde type toe te kennen. Hiervoor gebruiken we de *adres-operator* `&` (ampersand). In listing 7.2 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 7.2: Enkele toekenningen van adressen aan pointers.

Nu de pointers geïnitieerd zijn, kunnen we een voorstelling maken van de relatie tussen de pointers en de variabelen. Dit is te zien in figuur 7.3. Pointer `pint` wijst naar variabele `i`, pointer `pchar` wijst naar variabele `c` en pointer `pdouble` wijst naar variabele `d`.



Figuur 7.3: 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 *dereferentie* operator `*`. Deze operator heeft voorrang op de rekenkundige operatoren.

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

Listing 7.3: Gebruik van een pointer bij een toekenning.

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

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

Listing 7.4: Gebruik van een pointer bij het afdrukken van een variabele.

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

```

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

```

Listing 7.5: Declaratie en initialisatie van een pointer.

7.2 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-geïnitieerde pointer. Een NULL-pointer is een pointer waarin alle bits 0 zijn². Een niet-geïnitieerde pointer heeft een willekeurige waarde³. In C is een preprocessor-macro genaamd NULL te gebruiken om een pointer als NULL-pointer te initialiseren. De C-standaard schrijft 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 7.6: 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 7.7: Dereferentie van een NULL-pointer.

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

```

1 int *p = NULL, *q = NULL;
2 ...
3 if (p == q) { /* true */ }

```

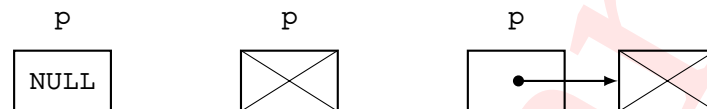
Listing 7.8: Vergelijken van twee NULL-pointers.

² In het algemeen 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.

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 7.4: Drie voorstellingen van NULL-pointers.

7.3 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 `&`.

```

1 int i;
2 char c;
3 double d;
4
5 void *p; /* void-pointer */
6
7 p = &i; /* valid */
8 p = &c; /* valid */
9 p = &d; /* valid */

```

Listing 7.9: 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 dereferentie operator `*` kunnen we bij de inhoud van variabele `i`. De constructie `*(int *)` is dus een expliciete dereferentie naar een integer.

7.4 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.

```

1 int i = 2;
2 void *p = &i;                                /* void-pointer */
3
4 ...
5 *(int *) p = *(int *) p + 1;    /* explicit type cast */
6
7 ...
8 printf("De_waarde_is_%d\n", *(int *) p);

```

Listing 7.10: Declaratie, initialisatie en deference van een void-pointer.

```

1 #include <stdio.h>
2
3 int main() {
4
5     int i = 2, *p = &i;
6
7     printf("Pointer:_%p\n", (void *) p);
8
9     return 0;
10 }

```

Listing 7.11: 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.

7.5 Pointers naar array's

We kunnen een pointer ook laten wijzen naar het eerste element van een array. Dit is te zien in listing 7.12. 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.

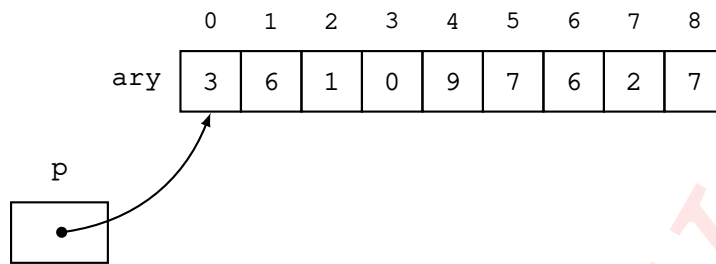
```

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

```

Listing 7.12: Een pointer naar het eerste element van een array.

De uitbeelding hiervan is te zien in figuur 7.5. 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 7.13.



Figuur 7.5: Uitbeelding van een pointer naar het eerste element van een array.

```

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 7.13: 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 7.14: 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.

```

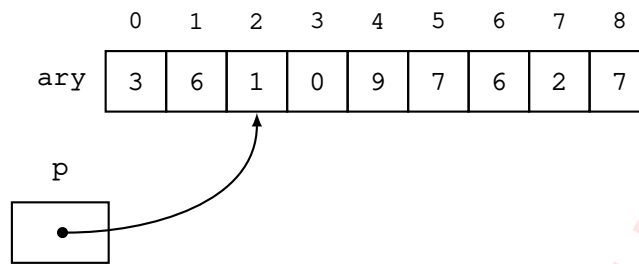
1 int ary[] = {3, 6, 1, 0, 9, 7, 6, 2, 7};
2
3 int *p = &ary[2]; /* p points to third element of array */

```

Listing 7.15: Een pointer naar het derde element van een array.

Een uitbeelding is te zien in figuur 7.6. 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 7.8.

We kunnen de lengte van ary berekenen 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 7.6: Uitbeelding van een pointer naar het derde element van een array.

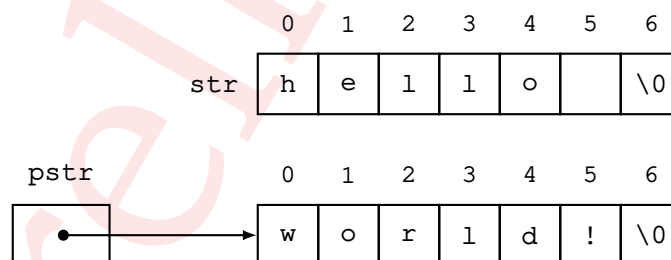
7.6 Strings

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 7.16 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 7.16: 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 7.7.



Figuur 7.7: 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` is 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.

7.7 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.

```
1 int ary[] = {3,6,1,0,9,7,6,2,7};
2 int *p = ary;  /* p pointe to ary[0] */
3
4 p = p + 1;      /* p points to ary[1] */
5 ...
6 p = p + 1;      /* p points to ary[2] */
7 ...
8 p = p + 3;      /* p points to ary[5] */
```

Listing 7.17: Rekenen met pointers.

Een mooi voorbeeld van het rekenen met pointers is het bepalen van de lengte van een C-string. In listing 7.18 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.

```
1 #include <stdio.h>
2
3 int main() {
4
5     char *str = "Hallo_wereld!";
6     char *begin = str;
7
8     while (*str != '\0') { /* while not end of string ... */
9         str = str + 1;    /* point to the next character */
10    }
11
12    printf("Lengte_is_%d\n", str-begin);
13 }
```

Listing 7.18: 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 7.18 kunnen vervan-

```

1 int ary[] = {3,6,1,0,9,7,6,2,7};
2 int *p = &ary[2];
3 int *q = &ary[5];
4
5 printf("Verschil_is_%d\n", q-p);

```

Listing 7.19: Het berekenen van het verschil van twee pointers.

gen door de onderstaande programmafragment. Uiteraard moeten de twee pointers naar hetzelfde datatype wijzen en heeft de vergelijking alleen zin als de pointers naar elementen binnen dezelfde array wijzen.

```

1 if (str>begin) {
2     printf("Lengte_is_%d\n", str-begin);
3 } else {
4     printf("De_string_is_leeg\n");
5 }

```

Listing 7.20: Vergelijken van twee pointers.

7.8 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 7.21 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.

```

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

```

Listing 7.21: 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 echt `p` ook lezen als de naam van een array. Om 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:

```

1 int ary[] = {3,6,1,0,9,7,6,2,7};
2 int *p = ary;
3
4 int sum = p[0]+p[1]+p[2]+p[3]+p[4]+p[5]+p[6]+p[7]+p[8];

```

Listing 7.22: 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 dereferentie-operator voorgaat op de optelling. Om de som van de array te bepalen mogen we dus schrijven:

```
1 int ary[] = {3,6,1,0,9,7,6,2,7};
2 int *p = ary;
3
4 int sum = *ary + *(ary+1) + *(ary+2) + *(ary+3) + ... ;
```

Listing 7.23: *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 7.24.

```
1 int ary[] = {3,6,1,0,9,7,6,2,7};
2
3 int *p = ary+2;                      /* p points to third element */
4
5 printf("Contents_is_%d\n", *p);      /* prints third element */
6 printf("Contents_is_%d\n", p[0]);    /* prints third element */
```

Listing 7.24: *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 7.25 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 array-grenzen ligt.

7.9 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 7.26 is te zien hoe we een functie `swap` definiëren die de inhoud van twee variabelen verwisseld. Bij het aanroepen van de functie geven we de adressen mee van

```

1 int ary[] = {3,6,1,0,9,7,6,2,7};
2
3 int *p = ary+2;    /* p points to third element */
4
5 int a = p[-1];     /* legal: points to second element */
6 int b = ary[-1];   /* ILLEGAL: points outside array */

```

Listing 7.25: Pointer die naar een element in een array wijst.

de te verwisselen variabelen. In de functie gebruiken we pointers om de inhoud te verwisselen.

```

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

```

Listing 7.26: 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 7.27. 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 7.27: 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 7.28.

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

```

1 void string_copy(char *to, char *from) {
2
3     if (from == NULL) {          /* sanity check */
4         return;
5     }
6
7     while (*from != '\0') {      /* while not end of string ... */
8         *to = *from;             /* copy character */
9         to = to + 1;             /* point to the next character */
10        from = from + 1;
11    }
12    *to = '\0';                  /* terminate string */
13 }
14
15 int main() {
16
17     char stra[] = "Hello_world!";
18     char strb[100];
19
20     string_copy(strb, stra);     /* copy stra to strb */
21
22     return 0;
23 }
24

```

Listing 7.28: Functie voor het kopiëren van een string.

C-bibliotheek heeft een functie `strcpy` die een efficiënte implementatie is van het kopiëren 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 7.28 wordt 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 7.29. 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.

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 7.30. 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

```

1 void string_copy(char to[], char from[]) {
2
3     int i=0;
4
5     while (from[i] != '\0') { /* while not end of string */
6         to[i] = from[i];      /* ... copy character */
7         i = i + 1;            /* point to next character */
8     }
9     to[i] = '\0';             /* .. and terminate string */
10 }

```

Listing 7.29: Functie voor het kopiëren van een string.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void printarray(int ary[], int len) {
5
6     int i;
7
8     for (i=0; i<len; i++) {
9         printf("%d_", ary[i]);
10    }
11 }
12
13 int main(void) {
14
15     int list[] = {1,2,3,4,5,6,7,8,9};
16     int length = sizeof list / sizeof list[0];
17
18     printarray(list, length);
19
20     return 0;
21 }

```

Listing 7.30: Meegeven van de grootte van een array.

7.10 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.

Let goed op de definitie van de functie `find_token`. Voor de functienaam is de dereferentie-

```

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 7.31: Pointer als return-waarde.

operator geplaatst. Dit betekent dat de functie een pointer naar een karakter teruggeeft. Deze vorm wordt vaak verward met pointers naar functies. Zie paragraaf 7.15.

7.11 Pointers naar pointers

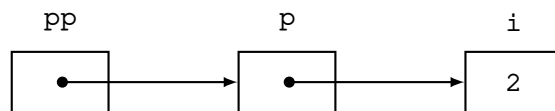
Een pointer kan ook gebruikt worden om naar een andere pointer te wijzen. In figuur 7.8 is te zien dat pointer *p* wijst naar variabele *i*. Met behulp van de dereferentie **p* kunnen we bij de inhoud van variabele *i* komen. De pointer *pp* wijst naar *p*. We hebben nu een *dubbele dereferentie* de nodig om via pointer *pp* bij de inhoud van variabele *i* te komen.

Voor de dubbele dereferentie gebruiken we twee keer de dereferentie-operator ***. Om

CALL BY REFERENCE...

In de programmeerwereld is het gebruikelijk om onderscheid te maken tussen twee manieren van argumentenoverdracht: *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.



Figuur 7.8: Uitbeelding van een pointer naar een pointer naar een *int*.

toegang te krijgen tot de inhoud van variabele *i* via pointer *pp* gebruiken we dus ***pp*. Dit is te zien in de onderstaande listing.

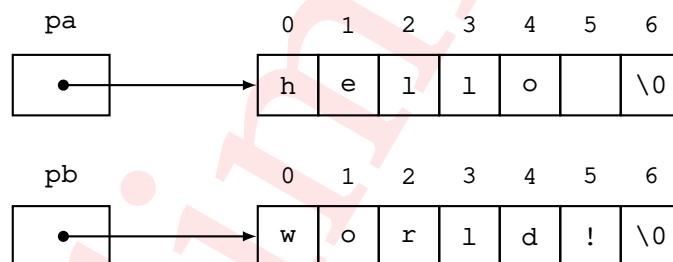
```

1 int i = 2;          /* the integer */
2 int *p = &i;       /* p points to i */
3 int **pp = &p;     /* pp points to p */
4
5 printf("De_waarde_is_%d\n", **pp);

```

Listing 7.32: Voorbeeld van een pointer naar een pointer.

Met behulp van pointers naar pointers kunnen de inhoud van twee pointers verwisselen. In figuur 7.9 is te zien dat de pointers *pa* en *pb* wijzen naar twee strings in het geheugen. Als we nu de strings willen “verwisselen”, hoeven we alleen maar de pointers er naartoe te verwisselen.



Figuur 7.9: Uitbeelding van pointers naar strings.

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

```

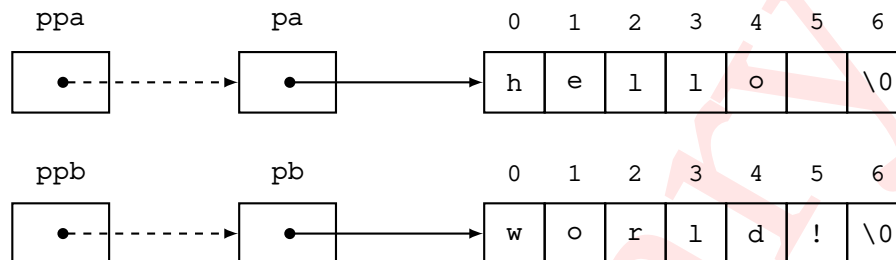
1 char *pa = "hello_";
2 char *pb = "world!";
3 char *temp;
4
5 temp = pa;          /* swap pa and pb */
6 pa = pb;
7 pb = temp;

```

Listing 7.33: 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 7.10. 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 inhoud van `pa` en `pb` verwisselen.



Figuur 7.10: Uitbeelding van pointers naar pointers naar strings.

In listing 7.34 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 dereferentie `*ppa` kopiëren we de inhoud van pointer `pa` naar `temp`. Daarna kopiëren we `pb` naar `pa` en als laatste kopiëren we `temp` naar `pb`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void swapstr(char **ppa, char **ppb) {
5
6      char *temp;
7
8      temp = *ppa;      /* copy pa into temp */
9      *ppa = *ppb;      /* copy pb into pa */
10     *ppb = temp;      /* copy temp into pb */
11 }
12
13 int main()
14 {
15     char *pa = "hello_";
16     char *pb = "world!";
17
18     printf("%s%s\n", pa, pb);
19     swapstr(&pa, &pb);
20     printf("%s%s\n", pa, pb);
21     return 0;
22 }

```

Listing 7.34: Functie voor het verwisselen van twee pointers.

De uitvoer van dit programma is te zien in de figuur 7.11. Te zien is dat de in de eerste regels de string op originele volgorde worden afgedrukt en in de twee regel in omgekeerde

volgorde. Deze manier van herschikken van array's van strings is veel efficiënter dan strings kopiëren.

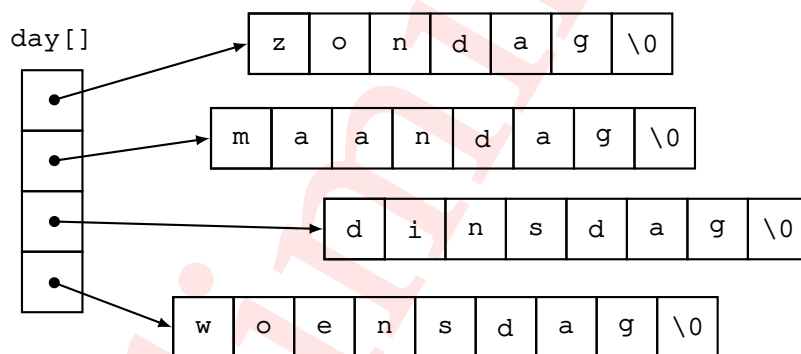
```
Command Prompt

hello world!
world!hello
```

Figuur 7.11: Verwisselen van twee stings.

7.12 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 7.12. 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 7.12: Voorstelling van een array van pointers naar strings.

We declareren een array van vier pointers naar karakters. Daarna laten we de pointers naar strings wijzen. Zie listing 7.35.

```
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 7.35: 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 7.36.

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

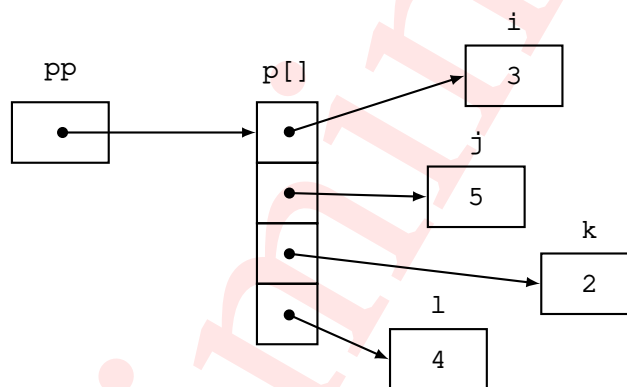

```
1 char *day[4] = {"zondag", "maandag", "dinsdag", "woensdag"};
```

Listing 7.36: Een array van pointers naar strings met initialisatie.

dit soort constructies meestal niet veranderen, kan het keyword `const` voor de declaratie gezet worden, waarmee wordt aangegeven dat de strings niet veranderen. C++-compilers accepteren deze constructie.

7.13 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 7.13.



Figuur 7.13: Voorstelling van een pointer naar een array van pointers naar integers.

De array `p` wordt gedeclareerd als:

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

De rechte haken hebben een hogere prioriteit dan de dereferentie-operator. We lezen de declaratie dus als: `p` is een array van vier elementen en elk element is 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 7.37. 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.

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

```

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 7.37: Voorbeeld van het gebruik van pointers.

de pointers af. In de regels 12 t/m 15 drukken we de adressen van de integers en de inhoud van de array-elementen af. Om variabele `i` af te drukken hebben we drie mogelijkheden: `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 figuur 7.14. 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.

7.14 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.

Elk C-programma krijgt per definitie de twee parameters `argc` en `argv` mee, die aan `main` worden meegegeven. Dit is te zien in listing 7.38.

De integer `argc` (**argument count**) geeft aan hoeveel parameters aan het C-programma zijn

```
C:\ 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
```

Figuur 7.14: Uitvoer van enkele pointers.

```
C:\ 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>
```

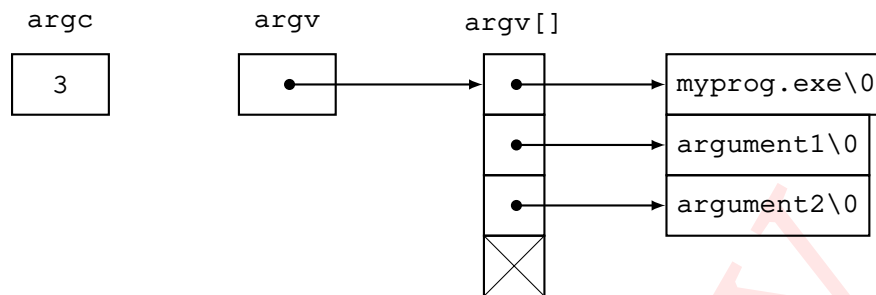
Figuur 7.15: Afdrukken van programmanaam en argumenten.

```
1 int main(int argc, char *argv[]) {
2
3     /* rest of the code */
4
5     return 0;
6 }
```

Listing 7.38: Declaratie van de command line parameters.

meegegeven. De pointer `argv` (**argument vector**) 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 7.16 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 7.16: Voorstelling van de variabelen *argc* en *argv*.

Het programma `myprog.exe` is te zien in listing 7.39. 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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]) {
5
6     int i;
7
8     printf("\nAantal argumenten: %d\n\n", argc);
9     for (i = 0; i < argc; i++) {
10         printf("Argument %d: %s\n", i, argv[i]);
11     }
12     return 0;
13 }
  
```

Listing 7.39: Het programma `myprog.exe`.

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 7.40.

7.15 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 *functie-pointers* genoemd.

In listing 7.41 is te zien hoe een functie-pointer wordt gedeclareerd. In de eerste regel wordt een functie gedefinieerd, een zogenoemde prototype, die een `int` als parameter meekrijgt 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.

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 7.42.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv) {
5
6     printf("\nArguments:");
7     while (argc>0) {
8         printf("%s", *argv);
9         argv = argv + 1;
10        argc = argc - 1;
11    }
12    return 0;
13 }

```

Listing 7.40: Afdrukken van argumenten.

```

1 int func(int a); /* func is a function returning an int */
2 int (*pf)(int a); /* pf is a pointer to a function
3                    returning an int */
4
5 pf = func; /* assign pf */

```

Listing 7.41: Een functie en een pointer naar een functie.

```

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 7.42: Een functie en een pointer naar een functie.

In figuur 7.17 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.

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.

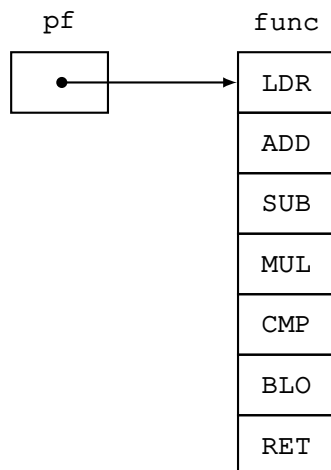
Quicksort

Quicksort is een sorteeralgoritme ontworpen door C.A.R. Hoare in 1962. Het is een van de efficiëntste sorteeralgoritmes voor algemeen gebruik. De exacte werking zullen we niet bespreken, er zijn genoeg boeken die dit beschrijven. De standard library 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*));

```



Figuur 7.17: Voorstelling van een pointer naar een functie.

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 programmeur zelf te schrijven) die twee pointers naar twee elementen in de array meekrijgt. De pointers zijn van het type `void *` want we weten niet wat het datatypes van de elementen van de array zijn. De functie *moet* een getal geven 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 7.43 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 komen. De pointers zijn van het type `void *` dus er is een expliciete type case nodig naar een pointer naar een integer. Dat wordt gerealiseerd 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.

7.16 Dynamische geheugenallocatie

Bij het schrijven van een programma declareren we de variabelen die we nodig hebben. Deze geheugenplaatsen blijven in principe bezet. Wel is het zo dat lokale variabelen worden aangemaakt en afgebroken aan het begin respectievelijk het einde van een functie (tenzij ze als `static` gekwalificeerd zijn), maar er komt geen extra geheugen erbij. Via het besturingssysteem is het mogelijk om extra geheugenruimte aan te vragen als dat nodig is. Is de aangevraagde geheugenruimte niet meer nodig dan geven we het terug aan het besturingssysteem. We noemen dit *dynamische geheugenallocatie*.

De standard library kent een aantal functie op dit gebied. Om deze functies te gebruiken moet het header-bestand `malloc.h` geladen worden. De functie `malloc` vraagt aan het besturingssysteem extra geheugenruimte *in bytes*. Als dat lukt, dan wordt een void-pointer naar het begin van de geheugenruimte teruggegeven. De void-pointer moet dus gecast worden naar het juiste type. Het geheugen wordt niet geïnitieerd, bijvoorbeeld met

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int values[] = { 88, 56, 100, 2, 25 };
5
6 int cmpint (const void * a, const void * b) {
7     /* explicit type case to pointers to integers */
8     return ( *(int*)a - *(int*)b );
9 }
10
11 int main () {
12     int n;
13
14     printf("De_array_voor_sorteren:\n");
15     for( n = 0 ; n < 5; n++ ) {
16         printf("%d_", values[n]);
17     }
18
19     qsort(values, 5, sizeof(int), cmpint);
20
21     printf("\nDe_array_na_sorteren:\n");
22     for( n = 0 ; n < 5; n++ ) {
23         printf("%d_", values[n]);
24     }
25
26     return (0);
27 }

```

Listing 7.43: Sorteren van een array met quicksort.

nul-bytes. Als het niet lukt, wordt een NULL-pointer teruggegeven. Hier moet uiteraard op getest worden. De functie `free` geeft eerder gealloceerd geheugen weer vrij. Als argument wordt een pointer naar het gealloceerde geheugenruimte meegegeven. Let erop dat de geheugenruimte niet eerder is vrijgegeven, want dan stopt het programma met een crash.

Een van de meest voorkomende problemen is dat gealloceerd geheugen niet wordt vrijgegeven. Dit wordt een *memory leak* genoemd. Op zich is dat niet zo erg, want als het programma stopt, ruim het besturingssysteem alle gealloceerde geheugenruimtes op. Maar als een programma alsmatig geheugen vraagt en niets vrijgeeft, kan het zijn dat uiteindelijk geen geheugen meer beschikbaar is.

Een eenvoudig gebruik van `malloc` en `free` is te zien in listing 7.44. De regel 6 definiëren we een *macro* met de grootte van 1000 bytes. Regel 13 zorgt ervoor dat Visual Studio de functie `strcpy` accepteert. In regel 17 declareren we een karakterpointer en initialiseren we die met NULL. Regel 19 roept de functie `malloc` aan. De functie geeft een void-pointer terug, dus we moeten een type cast naar `char *` doen. We testen in de regels 21 t/m 24 of de allocatie gelukt is en zo niet, dan drukken we een foutmelding af en stoppen we het programma.

```

1 #include <stdio.h>
2 #include <malloc.h>
3 #include <string.h>
4
5 /* Define 1000 bytes of space*/
6 #define SIZE 1000
7 /* Define 10 GB of space */
8 /* #define SIZE 10000000000 */
9 /* Define 100 GB of space */
10 /* #define SIZE 100000000000 */
11
12 /* Keep Visual Studio happy */
13 #define strcpy(A,B) strcpy_s(A,SIZE,B);
14
15 int main(void) {
16
17     char *pstr = NULL;
18
19     pstr = (char *) malloc(SIZE);
20
21     if (pstr == NULL) {
22         printf("Kan_geheugen_niet_alloceren!");
23         return 0;
24     }
25
26     strcpy(pstr, "Vul_maar_wat_in");
27
28     printf("%s", pstr);
29
30     free(pstr);
31
32     return 0;
33 }

```

Listing 7.44: Gebruik van *malloc* en *free*.

In regel 26 t/m 28 kopiëren we een string naar het geheugen en drukken het af. In regel 30 geven we het geheugen weer vrij. We hebben twee macro's toegevoegd die we naar believen kunnen activeren: één voor 10 GB en één voor 100 GB. De lezer wordt gevraagd om deze te testen.

We vermelden nog even twee andere functies: *realloc* en *calloc*. De functie *realloc* heralloceert een stuk geheugen. Met de aanroep

```
pnew = realloc(porig, size);
```

wordt het stuk geheugen waar *porig* naar wijst uitgebreid of ingekrompen, *pnew* wijst naar de nieuwe geheugenruimte. Er zijn een paar zaken waar we op moeten letten: 1) als het geheugen wordt ingekrompen, blijft de originele inhoud ongewijzigd; 2) als het

geheugen wordt uitgebreid blijft de originele inhoud ongewijzigd en het nieuwe, extra geheugen wordt niet geïnitieerd, 3) als het niet lukt om het geheugen op de originele plaats uit te breiden, wordt een compleet nieuw geheugenblok gealloceerd en wordt de inhoud van het originele inhoud gekopieerd, de rest wordt niet geïnitieerd, het oude geheugenblok wordt vrijgegeven; 4) als er geen geheugenruimte meer vrij is wordt een NULL-pointer teruggegeven en blijven de originele pointer en geheugen intact.

De functie `calloc` allocceert een aantal veelvoud van een elementair datatype en vult de geheugenruimte met nul-bytes. Dus

```
int *pint = calloc(1000, sizeof(int));
```

allocceert 1000 integers en vult ze met nul-bytes. Als het niet lukt, wordt een NULL-pointer teruggegeven.

* 7.17 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

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

zorgt ervoor dat `p` een pointer is naar een unsigned integer op adres $40FE_{16}$ (hexadecimale notatie). Er is een expliciete type cast nodig om de integer `0x40fe` om te zetten naar een adres. We kunnen nu iets in die geheugenplaats zetten door een dereferentie:

```
*p = 0x3ff;
```

Nu zullen dit soort toekenningen niet voorkomen op systemen waar een besturingssysteem op draait. Het gebruik van de pointer zal hoogst waarschijnlijk een crash van het programma veroorzaken. Maar op kleine computersystemen zonder besturingssysteem, 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 ATmega32-microcontroller van Atmel.

* 7.18 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.

```

1 typedef unsigned char uint8_t;
2
3 int main(void) {
4
5     uint8_t c1, c2, i;
6     volatile int i;
7
8     // Pointers to addresses
9     volatile uint8_t* ddrb = (uint8_t*) 0x37;    // DDRB address
10    volatile uint8_t* portb = (uint8_t*) 0x38;    // PORTB address
11
12    *ddrb = 0xFF; // Set PORTB to all outputs
13
14    while (1) {
15        c1 = 0x80;
16        c2 = 0x01;
17
18        for (i = 0; i < 4; i++) {
19            for (i = 0; i < 30000; ++i); // wait a bit
20            *portb = ~(c1 | c2); // Leds on Port B are active low
21            c1 >>= 1;
22            c2 <<= 1;
23        }
24    }
25
26    return 0;
27 }

```

Listing 7.45: Het gebruik van pointers op een ATmega-microcontroller.

| | |
|----------------------------------|---|
| <code>int *p</code> | <code>p</code> : pointer to int |
| <code>int **pp</code> | <code>pp</code> : pointer to pointer to int |
| <code>int ***ppp</code> | <code>ppp</code> : pointer to pointer to pointer to int |
| <code>int **pp[3]</code> | <code>pp</code> : array[3] of pointer to pointer to int |
| <code>char **argv</code> | <code>argv</code> : pointer to pointer to char |
| <code>char *argv[]</code> | <code>argv</code> : array[] of pointer to char |
| <code>int *list[5]</code> | <code>list</code> : array[5] of pointer to int |
| <code>int (*list)[5]</code> | <code>list</code> : pointer to array[5] of int |
| <code>int *(*list)[5]</code> | <code>list</code> : pointer to array[5] of pointer to int |
| <code>int *pf()</code> | <code>pf</code> : function returning a pointer to int |
| <code>int (*pf)()</code> | <code>pf</code> : pointer to function returning an int |
| <code>int *(*pf)()</code> | <code>pf</code> : pointer to function returning a pointer to int |
| <code>char ((*x())[])()</code> | <code>x</code> : function returning pointer to array[] of pointer to function returning char. |
| <code>char ((*x[3])())[5]</code> | <code>x</code> : array[3] of pointer to function returning pointer to array[5] of char. |

8

Structures

Een structuur (Engels: *structure*)¹ is een verzameling bij elkaar behorende gegevens beschikbaar onder één enkele naam. Dit is vooral handig bij complexe *datastructures* want ze helpen om gerelateerde variabelen als een eenheid te bewerken in plaats van een aantal verschillende variabelen. Structures komen overeen met een *rij* in een relationele database. Een bekend voorbeeld is de loonlijst van werknemers. Een werknemer heeft een naam, een adres, een salaris en een functie.

Structures kunnen gezien worden als één enkele variabele. Zo mogen we structures eenvoudig kopiëren met een toekenning, we mogen ze meegeven als argumenten aan een functie, een functie kan een structure in zijn geheel teruggeven, we kunnen het adres van een structure opvragen met de adres-operator `&` en we mogen de variabelen binnen de structure gebruiken in expressie. We mogen *niet* twee structures vergelijken.

8.1 Definitie en declaratie

Een welhaast klassiek voorbeeld van een structure is een aantal artikelen. Van de artikelen geven we het artikelnummer, de naam, het aantal dat beschikbaar is en de prijs per stuk op. We definiëren de structure `artikel` als volgt:

```
1 struct artikel {  
2     int nummer;        // artikelnummer  
3     char naam[20];     // artikelnaam  
4     int aantal;        // aantal beschikbaar  
5     double prijs;      // prijs per stuk  
6 };
```

Listing 8.1: De structure `artikel`.

¹ We zullen vanaf nu de Engelse naam `structure` gebruiken.

De variabelen binnen de structure worden *leden* genoemd maar we zullen gebruik maken van de Engelse naam *members*.

We kunnen nu de structure gebruiken in declaraties door gebruik te maken van

```
struct artikel variabele;
```

om variabelen te declareren. In listing 8.2 is een aantal declaraties te zien:

```
1 struct artikel floppy;
2
3 struct artikel usbstick = { 1, "USB_stick", 25, 7.84 };
4 struct artikel sdcard = { 2, "SD_card", 63, 4.97 };
```

Listing 8.2: Declaratie van enkele variabelen.

In regel 1 wordt de variabele `floppy` gedeclareerd zonder initialisatie. We kunnen bij declaratie gelijk de members initialiseren zoals te zien is in regels 3 en 4. In de initialisatielijst moet natuurlijk wel de juiste datatypes gebruikt worden, tenzij automatische conversie mogelijk is. Structures mogen zowel globaal als lokaal worden gedeclareerd.

8.2 Toegang tot members

Om gebruik te maken van een member gebruiken we de *member operator* `.` (punt). Om de prijs van het artikel `floppy` in te stellen kunnen we bijvoorbeeld gebruiken:

```
floppy.prijs = 10.73;
```

Om het aantal af te drukken gebruiken we `printf` met de member `aantal`:

```
printf("Aantal beschikbaar is %d", floppy.aantal);
```

Om het aantal beschikbare exemplaren van een artikel te verhogen kunnen we de member `aantal` verhogen:

```
floppy.aantal += 5; // 5 exemplaren erbij
```

Let erop dat de naam van een artikel een *string* is. Om de naam aan te passen moeten we gebruik maken van de functie `strcpy`:

```
strcpy(floppy.naam, "Floppy 3.5 in");
```

8.3 Functies met structures

Stel dat we de gegevens van een artikel willen afdrukken. Dan kunnen we natuurlijk alle members apart afdrukken. Maar is het handiger om een functie te definiëren die dat voor ons doet. Een structure mag gewoon als parameter aan een functie worden weergegeven. In de functie kunnen we dan de members een voor een afdrukken. Dit is te zien in listing 8.3. Binnen de functie drukken we de members een voor een af. We hebben extra spaties toegevoegd om de gegevens netjes van elkaar te scheiden.

```

1 void print_artikel(struct artikel a) {
2     printf("Nummer:_%d_", a.nummer);
3     printf("Naam:_%s_", a.naam);
4     printf("Aantal:_%d_", a.aantal);
5     printf("Prijs:_%%.2f\n", a.prijs);
6 }

```

Listing 8.3: Afdrukken van de gegevens van een artikel.

Als we een nieuw artikel willen toevoegen, kunnen we alle members van de structure een waarde toekennen. Maar het is gemakkelijker om een functie te definiëren die dat voor ons doet. Een mogelijke functie is te zien in listing 8.4.

```

1 struct artikel maak_artikel(int nummer, char naam[], int aantal,
2     double prijs) {
3     struct artikel nieuw;
4
5     nieuw.nummer = nummer;
6     strcpy(nieuw.naam, naam);
7     nieuw.aantal = aantal;
8     nieuw.prijs = prijs;
9
10    return nieuw;
11 }

```

Listing 8.4: Aanmaken van een nieuw artikel.

De parameters krijgen de waarden mee die aan de members moeten worden toegekend. Om een nieuwe structure te maken waar de gegeven inkomen, declareren we in regel 3 een structure met de naam `nieuw`. We moeten daarna een voor een de waarden aan de members toekennen. Vervolgens geven we de gehele structure terug aan de aanroepen

We kunnen de structure `floppy` bijvoorbeeld initialiseren met:

```
floppy = maak_artikel(7, "Floppy", 5, 10.73);
```

8.4 Typedef

Met behulp van het keyword `typedef` kunnen we een structure beschikbaar stellen onder een eigen datatype. Zo kunnen we het nieuwe type `artikel_t` aanmaken met de definitie in listing 8.5. Let erop dat `artikel_t` geen echt nieuw datatype is, het is meer een synoniem. Het blijft onder alle omstandigheden een structure. We mogen dan ook de naam na `struct` achterwege laten. Merk op dat de definitie van de oorspronkelijke structure tussen `typedef` en `artikel_t` staat.

```

1 typedef struct {
2     int nummer;        // artikelnummer
3     char naam[20];     // artikelnaam
4     int aantal;        // aantal beschikbaar
5     double prijs;      // prijs per stuk
6 } artikel_t;

```

Listing 8.5: De definitie van datatype `artikel_t`.

Vervolgens kunnen we variabelen declareren met de `typedef`:

```
artikel_t floppy, sdcard, usbstick;
```

We kunnen `typedef` ook gebruiken bij andere datatypes:

```
typedef unsigned long int ulint;
```

`Typedef's` zijn handig om een programma onafhankelijke te maken van de C-compiler die gebruikt wordt en de computer waarop het programma draait. Als een programma verplaatst wordt naar een andere computer (dat wordt *porteren* genoemd), hoeven alleen de `typedef's` te worden aangepast. Daarna kan het programma gecompileerd worden met de nieuwe `typedef's`.

8.5 Pointers naar structures

Als een structure een groot aantal members bevat, is het niet handig om een hele structure aan een functie mee te geven als parameters. Een argument dat wordt meegegeven wordt namelijk gekopieerd naar de parameters. Het is dan beter om een pointer naar een structure mee te geven. Bijkomend voordeel (of is het een probleem) is dat ook gelijk de members van de structure kunnen worden aangepast. Als we willen dat een argument niet kan worden aangepast, gebruiken we het keyword `const` bij de parameterdeclaratie. De compiler ziet er dan op toe dat in de functie geen members worden aangepast. Zie listing 8.6 voor de functie voor het afdrukken van gegevens van een artikel.

```

1 void print_artikel(const artikel_t *a) {
2     printf("Nummer:_%d_", (*a).nummer);
3     printf("Naam:_%s_", (*a).naam);
4     printf("Aantal:_%d_", (*a).aantal);
5     printf("Prijs:_%%.2f\n", (*a).prijs);
6 }

```

Listing 8.6: Een functie om de gegevens van een artikel af te drukken.

In regel 1 wordt een pointer naar een structure `artikel` gedeclareerd. Let op het keyword `const` dat aangeeft dat in de functie de members niet worden aangepast. In regel 2 is te zien hoe het artikelnummer wordt afgedrukt. Let hierbij op de haakjes om de pointervariabele.

Die zijn nodig omdat de member operator `.` (punt) een hogere prioriteit heeft dan de dereferentie-operator `*`. Dus:

```
(*a).nummer
```

selecteert member `nummer` van de structure die aangewezen wordt door pointer `a` terwijl

```
*a.nummer
```

ervoor zorgt dat `a.nummer` een pointer is naar een `int` (en de variabele `a` is geen pointer). Omdat het gebruik van de haakjes zo vaak voorkomt, is er een speciale notatie mogelijk. We kunnen de member operator `->` gebruiken om een member van een structure te gebruiken aangewezen door een pointer naar een structure. Dus:

```
a->nummer
```

selecteert member `nummer` van een structure aangewezen door pointer `a`. De twee volgende voorbeelden zijn daarom equivalent

```
(*a).nummer
```

```
a->nummer
```

Dus om een functie te gebruiken die de gegevens van een nieuwe artikel invult, kunnen de functie `maak_artikel` in listing 8.7 gebruiken.

```
1 void maak_artikel(artikel_t *a, int nummer, char naam[],
2                   int aantal, double prijs) {
3
4     a->nummer = nummer;
5     strcpy(a->naam, naam);
6     a->aantal = aantal;
7     a->prijs = prijs;
8 }
```

Listing 8.7: Een functie om de gegevens van een artikel in te stellen.

Een functie kan ook een pointer naar een structure teruggeven. We kunnen bijvoorbeeld het grootste aantal beschikbare exemplaren van twee artikelen bepalen. Dit is te zien in listing 8.8

```
1 artikel_t *grootste_voorraad(artikel_t* a, artikel_t* b) {
2     return (a->aantal > b->aantal) ? a : b;
3 }
```

Listing 8.8: Een functie om grootste aantal artikelen te bepalen.

We hebben hier gebruik gemaakt van de *conditionele expressie*. Deze wordt besproken in paragraaf 2.9.4.

8.6 Array van structures

Natuurlijk is het niet handig om voor elk artikel een variabele te declareren. We kunnen dan veel beter gebruik maken van een array. De declaratie

```
artikel_t art[10];
```

zorgt ervoor dat we een array van tien artikelen kunnen gebruiken. Om een element uit de array te selecteren gebruiken we de blokhaken []. Die hebben een lagere prioriteit dan de member operator . (punt) dus er zijn geen haakjes nodig om een member te gebruiken:

```
art[6].aantal += 3;
```

zorgt ervoor dat het aantal van art[6] met 3 wordt verhoogd. Bij het meegeven van de array aan een functie moeten we expliciet het aantal arrayelementen opgeven omdat een array altijd via *Call by Reference* (dus via een pointer) wordt meegegeven. Zie ook hoofdstuk 7.9.

In listing 8.9 is een functie te zien die alle gegevens van de artikelen afdrukt. We gaan hier ervan uit dat als een artikel het nummer 0 heeft, dit artikel niet is ingevuld en drukken we de gegevens niet af.

```
1 void print_artikelen(artikel_t ary[], int siz) {  
2  
3     for (int i = 0; i < siz; i++) {  
4         if (ary[i].nummer != 0) {  
5             print_artikel(&ary[i]);  
6         }  
7     }  
8 }
```

Listing 8.9: Een functie om alle gegevens van artikelen af te drukken.

Het aantal elementen kunnen we laten uitrekenen met behulp van de operator sizeof. We delen de grootte van de totale array door de grootte van één element om het aantal elementen te berekenen. We roepen de functie aan met

```
print_artikelen(art, sizeof art / sizeof art[0]);
```

8.7 Structures binnen structures

We kunnen ons programma uitbreiden met bestellingen. We definiëren daartoe drie structures voor artikelen, klanten en bestellingen. De structures zijn te zien in listing 8.10. We merken op dat de structure bestelling_t naast een uniek bestelnummer ook variabelen hebben van de structure klant_t en van een array van artikel_t (we gaan er gemakshalve vanuit dat een bestelling niet meer dan tien verschillende artikelen kan bevatten). We hebben dus structures binnen een structure. Als we een bestelling willen aanmaken dan moeten we naast het bestelnummer ook de klantgegevens en de artikelen opgeven.


```

1 typedef struct {
2     int nummer;        // artikelnummer
3     char naam[20];     // artikelnaam
4     int aantal;        // aantal beschikbaar
5     double prijs;      // prijs per stuk
6 } artikel_t;
7
8 typedef struct {
9     int klantnr;       // klantnummer
10    char naam[20];      // naam v.d. klant
11 } klant_t;
12
13 typedef struct {
14     int bestelnr;      // bestellingnr
15     klant_t klant;     // de klant
16                     // bestelde artikelen
17     artikel_t artbestel[10];
18 } bestelling_t;

```

Listing 8.10: De structures die horen bij een bestelling.

Om een bestelling aan te maken declareren we eerst een variabele van het type `bestelling_t` en initialiseren we de structure met nullen. Dat kan door alleen het bestelnummer expliciet met een 0 te initialiseren, de rest wordt automatisch op 0 gezet. Daarna vullen we het bestelnummer in via

```
bestel.bestelnr = 123;
```

Om het klantnummer op te geven moeten we via de structure `klant` in `bestelling` een waarde opgeven:

```
bestel.klant.klantnr = 73
```

We kunnen een artikel bij de bestelling voegen door een artikel in zijn geheel te kopiëren naar een element van de variabele `artbestel`

```
bestel.artbestel[0] = art[2];
```

We moeten dan nog alleen het aantal en de prijs berekenen. Dit is te zien in listing 8.11.

```

1     bestelling_t bestel = { 0 };
2
3     bestel.bestelnr = 123;
4     bestel.klant.klantnr = 73;
5     strcpy(bestel.klant.naam, "Jos");
6     bestel.artbestel[0] = art[2];
7     bestel.artbestel[0].aantal = 3;
8     bestel.artbestel[0].prijs *= bestel.artbestel[0].aantal;

```

Listing 8.11: Een functie om alle gegevens van artikelen af te drukken.

Overigens is het niet slim om alle informatie van een artikel in de structuur `bestel` op te nemen. Als de naam van de klant verandert, dan moeten we in twee structures de naam aanpassen. We kunnen beter alleen het klantnummer opnemen. De gegevens van de klant zijn dan via de klantstructures op te vragen. Zie ook het kader onder aan de bladzijde.

8.8 Teruggeven van meerdere variabelen

In C kan een functie slechts één variabele teruggeven. Als we meerdere variabelen willen teruggeven, kunnen we gebruik maken van een structuur. De structuur “verpakt” dan de variabelen onder één noemer.

Een bekend voorbeeld is zijn de oplossingen van een kwadratische vergelijking met de vorm $ax^2 + bx + c = 0$. We kunnen de *wortels* (getallen waarvoor de functie 0 oplevert) berekenen met de wortelformule:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (8.1)$$

De expressie onder het wortelteken heeft de *discriminant* en heeft alleen een geldige waarde als deze groter is dan of gelijk is aan 0. Verder mag a niet 0 zijn, want dan kunnen we niet delen. We moeten nu in feite drie gegevens bepalen: de wortels x_1 en x_2 en of de wortels geldig zijn. We pakken deze drie gegevens in in een structuur. Het volledige programma is te zien in listing 8.12.

We definiëren een structuur met de drie gegevens zoals te zien is in regels 4 t/m 8. In de functie `bereken_wortels` berekenen we de wortels als dat mogelijk is en we vullen de status van de wortels in. Kunnen ze niet berekend worden dan zetten we variabele `geldig` op 0, anders zetten we die variabele op 1. Aan het einde van de functie geven we een nieuw aangemaakte variabele terug. In de functie `print_wortels` drukken we de wortels af of dat de wortels niet geldig zijn (kunnen niet berekend worden).

RELATIONELE DATABASES

Het gebruik van structures is nauw gekoppeld aan *relationele databases*. We kunnen voor onze winkel bijvoorbeeld *tabellen* aanmaken met klantgegevens, artikelen en bestellingen. Bij de klantgegevens voeren we de uniek klantnummer, naam, het adres en telefoonnummer in, bij de artikelen een uniek artikelnummer, de naam, het aantal dat beschikbaar is en de prijs per stuk. Als we een bestelling invoeren, hoeven we naast een uniek bestelnummer alleen het nummer van de klant in te voeren; we kunnen de klantgegevens via de klanttabel vinden. Als het adres van de klant verandert, dan hoeven we alleen de klanttabel te veranderen. Natuurlijk moet een lijst met gekochte artikelen worden ingevoerd, maar we hoeven dan alleen het artikelnummer en het aantal in te voeren, de rest is via de artikeltabel te vinden. Het uit elkaar trekken van al die gegevens wordt *normalisatie* genoemd en zorgt ervoor dat spaarzaam met opslag wordt omgegaan en dat gegevens niet meerdere keren in de database voorkomen (redundantie).

```

1 #include <stdio.h>
2 #include <math.h>
3
4 typedef struct {
5     double x1;
6     double x2;
7     int geldig;
8 } wortels_t;
9
10 wortels_t bereken_wortels(double a, double b, double c) {
11     wortels_t x12;
12
13     double D = b * b - 4 * a * c;
14
15     if (D < 0.0 || a == 0.0) {
16         x12.geldig = 0;
17         return x12;
18     }
19
20     x12.x1 = (-b + sqrt(D)) / (2.0 * a);
21     x12.x2 = (-b - sqrt(D)) / (2.0 * a);
22     x12.geldig = 1;
23     return x12;
24 }
25
26 void print_wortels(wortels_t x12) {
27     if (x12.geldig) {
28         printf("x1:_%f_%%", x12.x1);
29         printf("x2:_%f\\n", x12.x2);
30     } else {
31         printf("Uitkomst_is_niet_geldig\\n");
32     }
33 }
34
35
36 int main(void) {
37
38     wortels_t uitkomst;
39     uitkomst = bereken_wortels(1.0, 2.0, -6.0);
40     print_wortels(uitkomst);
41
42     return 0;
43 }

```

Listing 8.12: Een programma om de wortels van een kwadratische vergelijking te berekenen.

* 8.9 Unions

Stel dat we onze eigen C-compiler willen ontwerpen (begin er trouwens niet aan, dat is heel complex). Dan moeten een lijst bijhouden van gedeclareerde variabelen. Van de variabele moet dan ook het type worden opgeslagen en mogelijk een initiële waarde². We kunnen dan een structure definiëren die alle mogelijke datatypes bevat, maar dat is verkwisten van geheugenruimte; een variabele kan immers maar één datatype hebben. We kunnen dan gebruik maken van een *union*.

Binnen een union wordt ruimte gereserveerd voor het grootste datatype (in het aantal bits). In listing 8.13 is een union variabele gedeclareerd met de naam `value`.

```
1  union {
2      int valint;
3      float valfloat;
4      double valdouble;
5  } value;
```

Listing 8.13: Een union om verschillende datatypes te gebruiken.

De gereserveerde geheugenruimte is gelijk aan de grootste variabele, in dit geval een `double`. We kunnen de union opnemen in een structure met daarin de naam en het type (en natuurlijk de union). Een voorbeeld is te zien in listing 8.14. We hebben tevens gebruik gemaakt van een *enumeratie* voor de verschillende datatypes.

```
1  typedef enum {typeint = 1, typefloat = 2, typedouble = 3} vartype_t
   ;
2
3  typedef struct {
4      char naam[20];
5      vartype_t type;
6      union {
7          int valint;
8          float valfloat;
9          double valdouble;
10     } value;
11 } varinfo_t;
```

Listing 8.14: Een structure om een variabele in een compiler te gebruiken.

We kunnen nu een variabele declareren en initialiseren met gegevens. Dit is te zien in listing 8.15. Te zien is dat de variabele wordt ingesteld als `float` en dat de waarde 3,14 bedraagt.

² Natuurlijk kan de waarde van een variabele tijdens runtime veranderen, maar het zou kunnen dat de compiler erachter komt dat een variabele op een bepaald moment een bekende waarde heeft. Deze waarde kan dan gebruikt worden bij een expressie.

```

1  varinfo_t devariabele;
2
3  strcpy(devariabele.naam, "getal");
4  devariabele.type = typefloat;
5  devariabele.value.valfloat = 3.14f;

```

Listing 8.15: Initialiseren van een variabele.

We kunnen op elk moment het type en de waarde veranderen (dat zou in een C-compiler natuurlijk niet kunnen gebeuren). We kunnen de variabele bijvoorbeeld kenmerken als een integer en er een waarde aan toekennen.

```

1  devariabele.type = typeint;
2  devariabele.value.valint = 1234;

```

Listing 8.16: Initialiseren van een variabele.

Bij het afdrukken van de waarde van de variabele raadplegen we het type. Met behulp van een switch-statement selecteren we hoe de variabele moet worden afgedrukt.

```

1 void print_var(varinfo_t var) {
2
3     printf("Naam:_%s\n", var.naam);
4
5     switch (var.type) {
6     case typeint:
7         printf("Type:_%int, _Value_%d\n", var.value.valint);
8         break;
9     case typefloat:
10        printf("Type:_%float, _Value_%f\n", var.value.valfloat);
11        break;
12    case typedouble:
13        printf("Type:_%double, _Value_%f\n", var.value.valdouble);
14        break;
15    default:
16        printf("Type:_%ONBEKEND\n");
17        break;
18    }
19 }

```

Listing 8.17: Een functie om een variabele af te drukken.

9

Invoer en uitvoer

9.1 Inlezen van een getal met `scanf`

De functie `scanf` is een veelzijdige functie voor het inlezen van variabelen. Zo kan je een getal, een letter of een string inlezen. Maar hoe weet `scanf` nu wat er ingelezen moet worden? Dat geven we op met een format string. Een enkel geheel getal wordt inlezen met:

```
scanf("%d", &getal);
```

De format specification `%d` vertelt `scanf` dat een geheel getal moet worden ingelezen. De variabele waarin het getal moet komen te staan, wordt voorzien van de adres-operator `&` zodat `scanf` de variabele kan vinden. Het is ook mogelijk om meerdere getallen in te lezen:

```
scanf("%d %d %d", &getala, &getalb, &getalc);
```

Bij het invoeren moeten de getallen gescheiden worden door één of meerdere spaties. Het aantal spaties maakt dus niet uit; deze worden overgeslagen.

We kunnen ook andere datatypes inlezen zoals een `float` of een `char`

```
scanf("%f %c", &fl, &ch);
```

9.1.1 Problemen met `scanf`

Als we in een C-programma een getal inlezen met `scanf` dan gebeuren er vreemde dingen als we in plaats van een getal letters intypen. Bijvoorbeeld:


```

1 #include <stdio.h>
2
3 int main(void) {
4     int getal = -12345;
5
6     printf("Geef een geheel getal: ");
7     scanf("%d", &getal);
8
9     printf("Het getal is %d.\n", getal);
10
11     return 0;
12 }

```

Listing 9.1: Getal inlezen met *scanf*.

Als we dit programma uitvoeren en als invoer het woord `Hallo` intypen dan verschijnt de volgende uitvoer:

| |
|--|
|  Command Prompt |
| <pre> Geef een geheel getal: Hallo Het getal is -12345. </pre> |

Figuur 9.1: Invoer van een heel getal.

Het blijkt dat de waarde van variabele `getal` niet is veranderd. Het eerste teken dat `scanf` tegenkomt is de `H` en dat is geen cijfer, dus wordt er gestopt met inlezen van het toetsenbord. Het wordt echt problematisch als er een getal ingelezen moet worden dat aan een voorwaarde moet voldoen:

```

1 int main(void) {
2     int getal = -12345;
3
4     do {
5         printf("Geef een geheel getal groter dan 0: ");
6         scanf("%d", &getal);
7     } while (getal < 1);
8
9     printf("Het getal is %d.\n", getal);
10    return 0;
11 }

```

Listing 9.2: Het inlezen van een geheel getal dat groter is dan 0.

De uitvoer is nu:


```
C:\ Command Prompt

Geef een geheel getal groter dan 0: Hallo
Geef een geheel getal groter dan 0: Geef een geheel getal groter dan
0: Geef een geheel getal groter dan 0: Geef een geheel getal groter
dan 0: Geef een geheel getal groter dan 0: Geef een geheel getal gro
ter dan 0: Geef een geheel getal groter dan 0: Geef een geheel getal
enzovoorts
```

Figuur 9.2: *Uivoer van het programma na invoer van een string.*

Het programma gaat nu als een razende te werk en de enige manier om het te stoppen is het programma af te sluiten. Maar hoe komt dat nou?

Als `scanf` voor de eerste keer iets inleest dan zijn er nog geen karakters vanaf het toetsenbord ingevoerd. Daarom vraagt `scanf` aan het besturingssysteem (Windows, Linux, OS-X) om een karakter. Het besturingssysteem weet dat er geen karakters beschikbaar zijn en gaat via interne routines karakters opvragen. Er wordt echter niet één karakter opgevraagd maar een hele reeks die afgesloten moet worden met een enter-toets. We moeten als gebruiker dus de invoer altijd afsluiten met een enter-toets, ook als we via `scanf` maar één karakter inlezen. De ingelezen karakters worden ergens in het geheugen opgeslagen. Dit wordt *buffering* genoemd en de geheugenruimte wordt *invoerbuffer* genoemd. Er zijn nu karakters beschikbaar en het eerste ingevoerde karakter wordt aan `scanf` gegeven.

Het eerste karakter is de letter H en dat is geen cijfers. Dus stopt `scanf` direct met het inlezen van karakters (die dus alleen cijfers mogen zijn). Het karakter H blijft hierbij in de invoerbuffer staan. De H wordt dus niet verwijderd.

Omdat `scanf` geen cijfers heeft kunnen inlezen, wordt de opgegeven variabele niet veranderd en blijft zijn originele waarde behouden. Daarom drukt het programma in listing ?? het getal -12345 af. Het programma in listing ?? blijft in een `do-while`-lus steeds een getal inlezen als het getal kleiner is dan -1. Aangezien `scanf` geen cijfers inleest en de opgegeven variabele niet aanpast, blijft de waarde -12345 behouden en dat is kleiner dan -1. Dus wordt de lus nog een keer uitgevoerd. De H staat nog steeds in de invoerbuffer en lukt het `scanf` niet om een getal in te lezen.

Dit kun je vrij eenvoudig voorkomen door de returnwaarde van de functie `scanf` te testen. Deze functie geeft het aantal variabelen terug dat succesvol is geconverteerd en ingelezen (dat kan dus ook 0 zijn), anders geeft de functie de integer waarde EOF terug. Je kan dat als volgt doen:

De uitvoer wordt dan:


Er is nog één probleem. De karakters van `Hallo` staan nog steeds in de invoerbuffer. Die zullen we er een voor een uit moeten halen. Dat kan met het onderstaande programma:

```
1 #include <stdio.h>
2
3 int main(void) {
```

```

1 #include <stdio.h>
2
3 int main(void) {
4
5     int getal = -12345;
6     int ret;
7
8     printf("Geef een geheel getal groter dan 0:_");
9     ret = scanf("%d", &getal);
10
11     if (ret == 1) {
12         printf("Het getal is_%d.\n", getal);
13     } else {
14         printf("Geen getal ingevoerd!\n");
15     }
16
17     printf("Druk op de Enter toets om dit window te sluiten.");
18     getchar();
19     return 0;
20 }

```

| |
|---|
|  Command Prompt |
| <pre> Geef een geheel getal groter dan 0: Hallo Geen getal ingevoerd! Druk op de Enter toets om dit window te sluiten. </pre> |

Figuur 9.3: *please fill in*

```

4 int getal, ret;
5 do {
6     printf("Geef een geheel getal:_");
7     ret=scanf("%d", &getal);
8     if (ret == 0) {
9         printf("Dat_was_geen_getal!\n");
10        printf("Maar_het_karakter_%c.\n", getchar());
11    }
12    else if (ret == EOF) {
13        printf("Er_is_een_fout_opgetreden_bij_het_lezen!\n");
14    }
15    } while (ret != 1);
16    printf("Het getal is_%d.\n", getal);
17    printf("Druk op de Enter toets om dit window te sluiten.");
18    getchar();
19    return 0;
20 }

```

De functie `getchar` leest één karakter uit de invoerbuffer. Daarna proberen we `scanf` opnieuw. Dat mislukt telkens als er een letter gevonden wordt. Als je nu `Hallo` invoert, krijg je de volgende uitvoer:

```
1 Geef een geheel getal: Hallo
2 Dat was geen getal!
3 Maar het karakter H.
4 Geef een geheel getal: Dat was geen getal!
5 Maar het karakter a.
6 Geef een geheel getal: Dat was geen getal!
7 Maar het karakter l.
8 Geef een geheel getal: Dat was geen getal!
9 Maar het karakter l.
10 Geef een geheel getal: Dat was geen getal!
11 Maar het karakter o.
12 Geef een geheel getal:
```

Natuurlijk willen niet steeds dat bij elk karakter een melding op het scherm wordt afgedrukt. We kunnen een aantal `printf`-regels verwijderen maar niet de regel waarin de gebruiker wordt gevraagd om een geheel getal in te voeren. Anders weet de gebruiker niet wat hij/zij moet doen. Helaas wordt deze regel herhaaldelijk afgedrukt. We zullen op een iets andere manier de karakters moeten inlezen.

Gelukkig zorgt het operating system ervoor dat ook het end-of-line-karakter in de invoerbuffer terecht komt. We kunnen dus hierop testen. Hoe dat moet, is te zien in de onderstaande code:

```
1 do { ch = getchar(); } while (ch != '\n' && ch != EOF);
```

We lezen een karakter van de invoerbuffer en dat doen we zolang dat karakter ongelijk is aan `\n` (end-of-line-karakter) en ongelijk is aan `EOF` (end-of-file).

Omdat we vaker gebruik willen maken van het legen van de invoerbuffer plaatsen we de code een functie. We kunnen nu aan de gebruiker vragen om een getal in te voeren:

```
1 #include <stdio.h>
2
3 void purge_stdin(void) {
4     int ch;
5     do {
6         ch = getchar();
7     } while (ch != '\n' && ch != EOF);
8 }
```

```

9
10 int main(void) {
11     int getal, ret;
12     do {
13         printf("Geef_een_geheel_getal:_");
14         ret=scanf("%d", &getal);
15         if (ret == 0) {
16             purge_stdin();
17         }
18     } while (ret != 1);
19     printf("Het_getal_is_%d.\n", getal);
20     printf("Druk_op_de_Enter_toets_om_dit_window_te_sluiten.");
21     purge_stdin();
22     getchar();
23     return 0;
24 }

```

Er is nog een andere manier om de invoerbuffer te legen. We kunnen de invoerbuffer legen met de functie `fflush` (file flush):

```

1 fflush(stdin);

```

Als parameter van `fflush` wordt `stdin` opgegeven. Dat staat voor *standard input* en daar wordt in de regel het toetsenbord mee bedoeld¹. Daarnaast kennen we nog `stdout` (*standard output*, het beeldscherm) en `stderr` (*standard error*, meestal ook het beeldscherm).

Deze manier werkt echter niet met alle C-compilers en operating systems. Dat heeft te maken met de definitie van de functie `fflush`. De functie `fflush` is bedoeld om uitvoerbuffers te legen. Als de uitvoer naar het beeldscherm is, wordt de buffer naar het beeldscherm geschreven. Als de uitvoer naar een bestand is, wordt de buffer naar het bestand geschreven. De C-standaard schrijft echter alleen voor dat *flushen* van een uitvoerbuffer gedefinieerd is, niet van een invoerbuffer. Niet zo gek eigenlijk, want wat wordt er nou bedoeld met flushen van de invoerbuffer. Flushen van het toetsenbord is het nog te begrijpen maar flushen van een invoerbestand niet. Moeten we dan helemaal tot einde van het bestand flushen? Of alleen maar de bijbehorende buffer? Dat levert een onvoorspelbaar programma op want we weten immers niet hoeveel karakters in de buffer staan.

Toch zijn er wel C-implementaties die het flushen van een invoerbuffer uitvoeren, bijvoorbeeld de GNU C-compiler op Linux en MinGW op Windows (zit o.a. in Code::Blocks).

Hieronder is de code te vinden van het flushen van de invoerbuffer van het toetsenbord m.b.v. `fflush`:

¹ Op de bekende operating systems is het mogelijk om de inhoud van bestanden door te geven aan de *standard input*. Het programma krijgt dan data uit een bestand i.p.v. het toetsenbord. Dat wordt *redirection* genoemd.

```
1  /* Please note: might not works on all Operating Systems. */
2  #include <stdio.h>
3
4  int main(void) {
5      int getal, ret;
6      do {
7          printf("Geef_een_geheel_getal:_");
8          ret=scanf("%d", &getal);
9          if (ret == 0) {
10             fflush(stdin);
11         }
12     } while (ret != 1);
13     printf("Het_getal_is_%d.\n", getal);
14     printf("Druk_op_de_Enter_toets_om_dit_window_te_sluiten.");
15     fflush(stdin);
16     getchar();
17     return 0;
18 }
```


Bibliografie

Veel materiaal is tegenwoordig (alleen) via Internet beschikbaar. Voorbeelden hiervan zijn de datasheets van ic's die alleen nog maar via de website van de fabrikant beschikbaar worden gesteld. Dat is veel sneller toegankelijk dan boeken en tijdschriften. De keerzijde is dat websites van tijd tot tijd veranderen of verdwijnen. De geciteerde weblinks werken dan niet meer. Helaas is daar niet veel aan te doen. Er is geen garantie te geven dat een weblink in de toekomst beschikbaar blijft.

- [1] B.W. Kernighan en D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988. ISBN: 9780131103627 (blz. iii).
- [2] LaTeX Project Team. *LaTeX – A document preparation system*. URL: <https://www.latex-project.org/> (bezocht op 30-12-2018) (blz. iii).
- [3] Tex User Groups. *TeX Live Website*. URL: <https://www.tug.org/texlive/> (bezocht op 30-12-2018) (blz. iii).
- [4] Benito van der Zander. *TeXstudio, LaTeX made comfortable*. URL: <https://www.texstudio.org/> (bezocht op 04-05-2019) (blz. iii).
- [5] Bitstream Inc. *Charter Fonts*. URL: <https://www.ctan.org/pkg/charter> (bezocht op 30-12-2018) (blz. iii).
- [6] Artifex. *Nimbus 15 Mono*. Okt 2015. URL: <https://www.ctan.org/tex-archive/fonts/nimbus15> (bezocht op 30-12-2018) (blz. iii).
- [7] J. Hoffman. *listings – Typeset source code listings using LaTeX*. URL: <https://www.ctan.org/pkg/listings> (bezocht op 30-12-2018) (blz. iii).
- [8] T. Tantau. *pgf – Create PostScript and PDF graphics in TeX*. Aug 2015. URL: <https://www.ctan.org/pkg/pgf> (bezocht op 30-12-2018) (blz. iii).
- [9] HBO Engineering. *Body of Knowledge and Skills Elektrotechniek*. Nov 2016. URL: http://www.hbo-engineering.nl/_asset/_public/competenties/BOKS_elektrotechniek_met_specialisaties_def_11nov2016.pdf (bezocht op 01-03-2018) (blz. v).
- [10] M. Barr. *Apple's #gotofail SSL Security Bug was Easily Preventable*. 2014. URL: <https://embeddedgurus.com/barr-code/2014/03/apples-gotofail-ssl-security-bug-was-easily-preventable/> (bezocht op 25-04-2020) (blz. 10).
- [11] D.E. Knuth. *Structured Programming with go to Statements*. 1974. URL: https://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf (bezocht op 10-04-2020) (blz. 33).
- [12] V. Eijkhout. *An ASCII wall chart*. Nov 2010. URL: <https://ctan.org/pkg/ascii-chart> (bezocht op 29-07-2018) (blz. 128).
- [13] American Standards Association. *ASA standard X3.4-1963*. URL: <http://worldpowersystems.com/J/codes/X3.4-1963/> (bezocht op 26-07-2018) (blz. 127).

- [14] A.K. Maini. *Digital Electronics: Principles, Devices and Applications*. Wiley, 2007, p. 28. ISBN: 978-0-470-03214-5 (blz. 127).

Preliminary



De ASCII-tabel

We hebben gezien dat binaire coderingen worden gebruikt om numerieke gegevens (of informatie) weer te geven. Maar niet alle gegevens zijn numeriek. Informatie kan ook bestaan uit letters en leestekens. Om er voor te zorgen dat computers informatie kunnen uitwisselen is de ASCII-code bedacht.

De naam ASCII betekent *American Standard Code for Information Interchange* en dat geeft al goed aan waarvoor de code bedoeld is: op een gestandaardiseerde wijze informatie uitwisselen. De code is in 1963 voor het eerst gepubliceerd [13] en in die tijd was er nog geen noodzaak om andere tekens te gebruiken dan de bekende westerse letters, cijfers en leestekens. Vandaar dat het aantal tekens beperkt is.

De ASCII-code bestaat uit 128 7-bits tekens zoals te zien is in tabel A.1. De tekens zijn verdeeld in leesbare tekens, zoals letters, cijfers en leestekens en zogenoemde *besturingstekens*. De besturingstekens zijn nodig om informatie-overdracht af te bakenen en om een bepaalde *handshake* (uitwisselingsprotocol) te regelen. Zo zijn er codes voor de *carriage return* (CR, code 13_{10}) en de *backspace* (BS, code 8_{10}). De eerste 32 tekens zijn besturingstekens waarvan de meeste tegenwoordig niet meer gebruikt worden [14].

De codes zijn niet willekeurig toegekend dat we goed kunnen zien bij de cijfers en letters. De tabel is zo opgesteld dat de cijfers elkaar opvolgen. Dat is handig bij het afdrukken van een (decimaal) getal. Hetzelfde geldt voor de letters, ook die volgen elkaar op. De makers hebben ook nagedacht over de positie van hoofd- en kleine letters. Deze verschillen in de tabel in slechts één bit (bit b_6 in de tabel). Bij het gebruik van de Caps Lock-toets of Shift-toets hoeft dus maar één bit (in combinatie met een letter) gewijzigd te worden.

Een aantal besturingstekens is ook in C direct te gebruiken. Een besturingsteken begint altijd met een *backslash* ('**') gevolgd door een letter, cijfer of teken. Zo is het teken voor een horizontale tab '*\t*' en voor de *line feed* is het teken '*\n*'. Met een line feed gaat de cursor naar het begin van de volgende regel.

Tabel A.1: De ASCII-code [12].

ASCII CONTROL CODE CHART

| b7 b6 b5 BITS b4 b3 b2 b1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---------------------------------------|---------|--------|--------------------|-------|------------|--------|------------|--------|
| | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| | CONTROL | | SYMBOLS NUMBERS | | UPPER CASE | | LOWER CASE | |
| 0 0 0 0 | 0 NUL | 16 DLE | 32 SP | 48 0 | 64 @ | 80 P | 96 ' p | 112 |
| 0 0 0 1 | 1 SOH | 17 DC1 | 33 ! | 49 1 | 65 A | 81 Q | 97 a q | 113 |
| 0 0 1 0 | 2 STX | 18 DC2 | 34 " | 50 2 | 66 B | 82 R | 98 b r | 114 |
| 0 0 1 1 | 3 ETX | 19 DC3 | 35 # | 51 3 | 67 C | 83 S | 99 c s | 115 |
| 0 1 0 0 | 4 EOT | 20 DC4 | 36 \$ | 52 4 | 68 D | 84 T | 100 d t | 116 |
| 0 1 0 1 | 5 ENQ | 21 NAK | 37 % | 53 5 | 69 E | 85 U | 101 e u | 117 |
| 0 1 1 0 | 6 ACK | 22 SYN | 38 & | 54 6 | 70 F | 86 V | 102 f v | 118 |
| 0 1 1 1 | 7 BEL | 23 ETB | 39 ' | 55 7 | 71 G | 87 W | 103 g w | 119 |
| 1 0 0 0 | 8 BS | 24 CAN | 40 (| 56 8 | 72 H | 88 X | 104 h x | 120 |
| 1 0 0 1 | 9 HT | 25 EM | 41) | 57 9 | 73 I | 89 Y | 105 i y | 121 |
| 1 0 1 0 | 10 LF | 26 SUB | 42 * | 58 : | 74 J | 90 Z | 106 j z | 122 |
| 1 0 1 1 | 11 VT | 27 ESC | 43 + | 59 ; | 75 K | 91 [| 107 k { | 123 |
| 1 1 0 0 | 12 FF | 28 FS | 44 , | 60 < | 76 L | 92 \ | 108 l | 124 |
| 1 1 0 1 | 13 CR | 29 GS | 45 - | 61 = | 77 M | 93] | 109 m } | 125 |
| 1 1 1 0 | 14 SO | 30 RS | 46 . | 62 > | 78 N | 94 ^ | 110 n ~ | 126 |
| 1 1 1 1 | 15 SI | 31 US | 47 / | 63 ? | 79 O | 95 _ | 111 o DEL | 127 |
| | F 17 | 1F 37 | 2F 57 | 3F 77 | 4F 117 | 5F 137 | 6F 157 | 7F 177 |

LEGEND:

| | |
|-----|------|
| dec | CHAR |
| hex | oct |

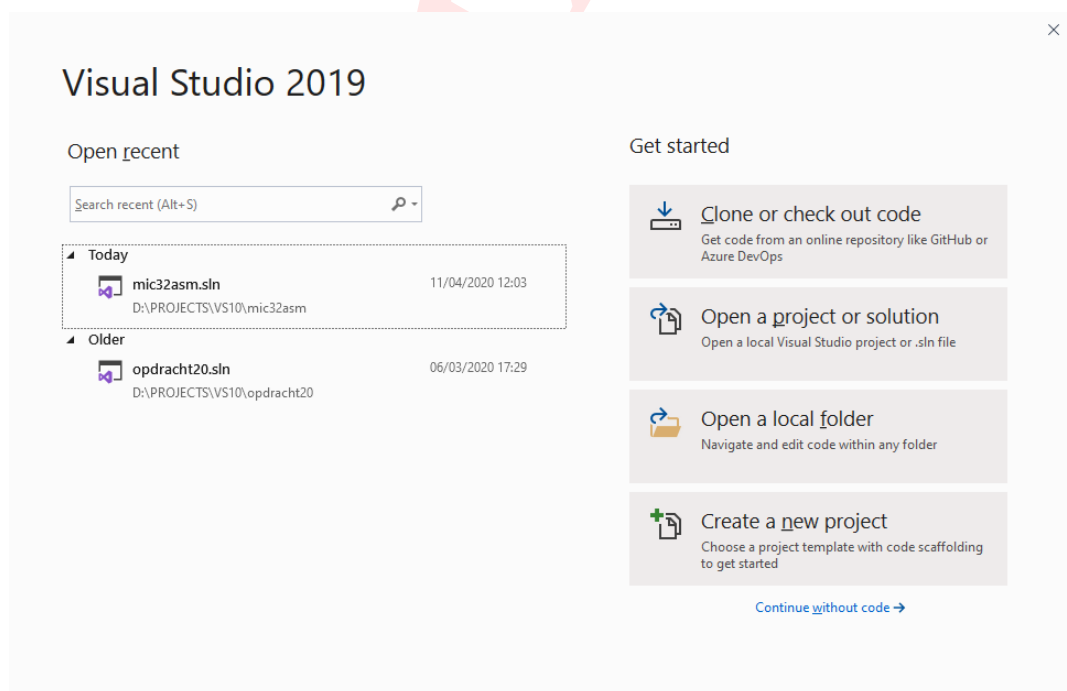
Victor Eijkhout
TACC
Austin, Texas, USA

B

Tutorial Visual Studio

We bespreken in deze bijlage het opzetten van een project in Visual Studio 2019. Visual Studio 2019 kan gedownload worden van de website van Microsoft. De precieze installatie van de software verschilt van computer tot computer, afhankelijk van de al eerder geïnstalleerde software, met name de *runtime libraries*.

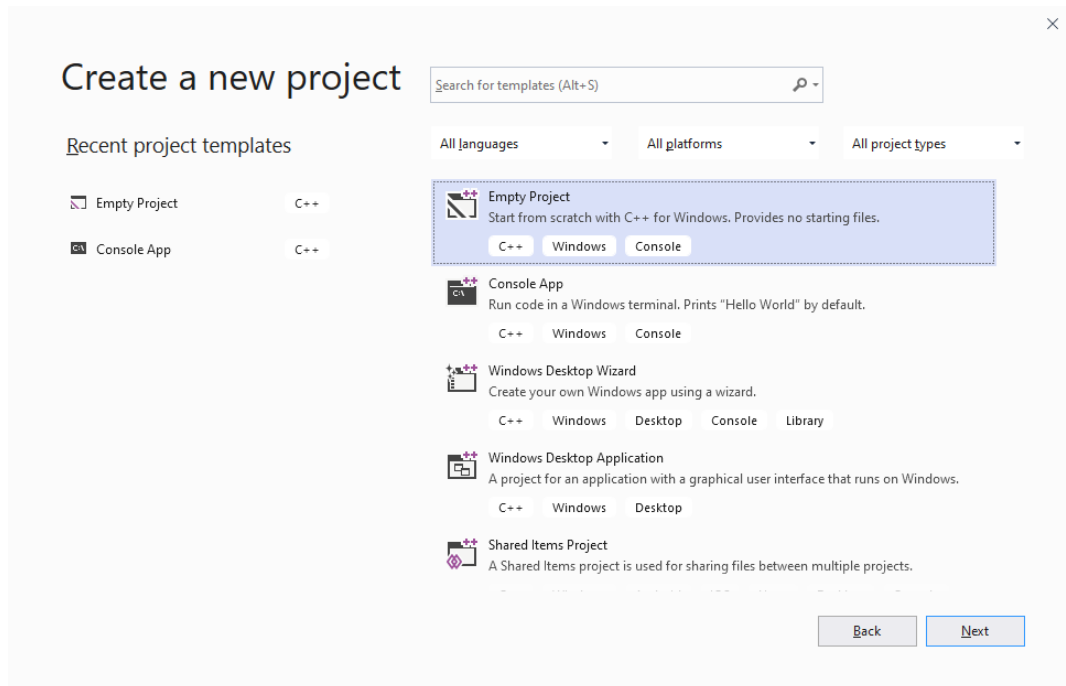
Start Visual Studio door op het icoon te klikken. Visual Studio opent een beginscherm waarin een nieuw project kan worden aangemaakt. Dit is te zien in figuur B.1. Klik op het kader `Create a new project`.



Figuur B.1: Aanmaken van een nieuw project.

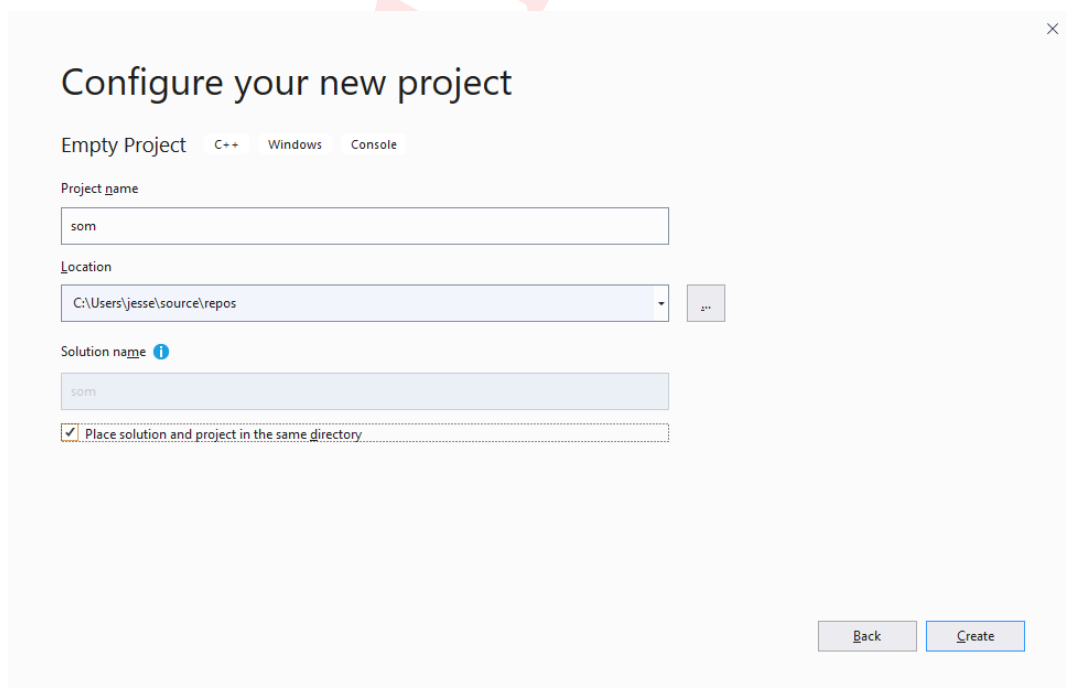
Er wordt een nieuw scherm geopend, zie figuur B.2. Klik daarin op het kader `Empty`

Project. **Klik niet op Console App.**



Figuur B.2: Aanmaken van een leeg project.

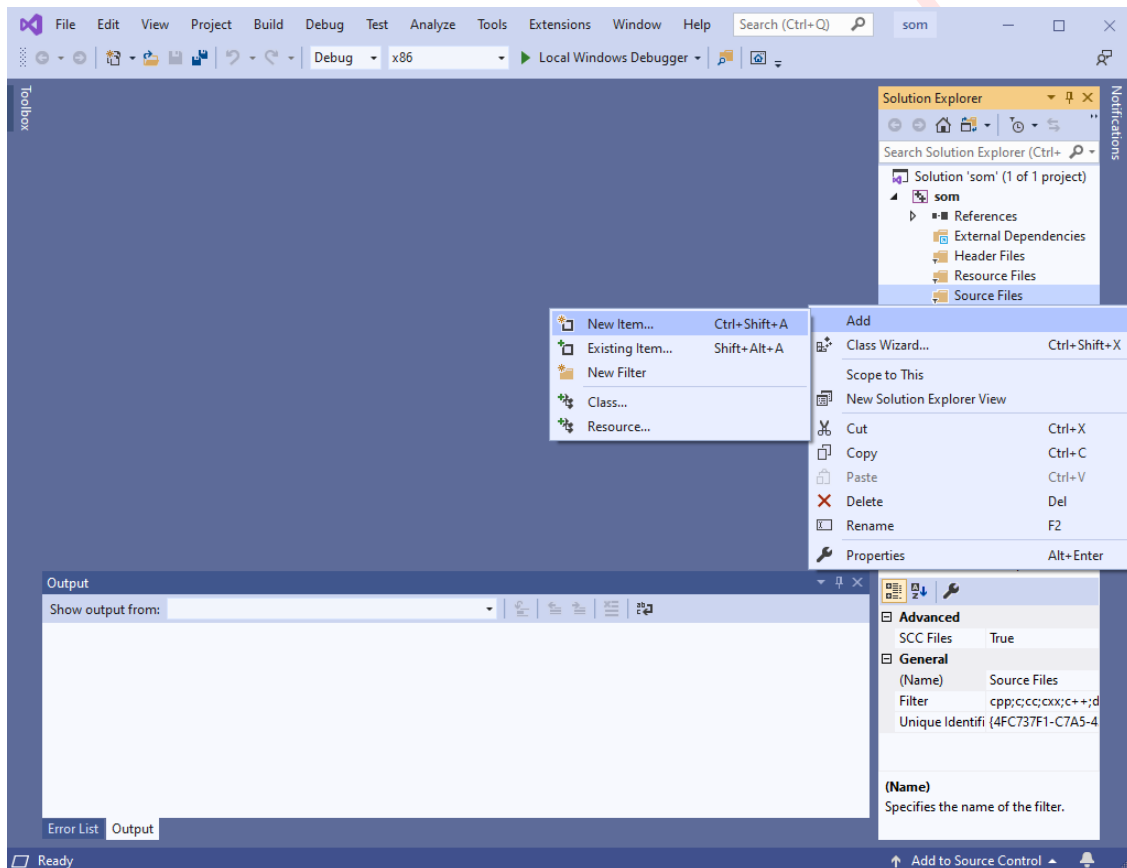
Daarna moeten wat gegevens worden ingevuld. Vul de projectnaam in en de map waarin het project terecht moet komen. Vink de checkbox onderaan aan en klik op de knop Create. Zie figuur B.3.



Figuur B.3: Gegevens van het project invoeren.

Visual Studio komt nu met het hoofdscherm waarin een aantal vensters (Engels: pane) te zien zijn. Er is nog geen C-bestand aangemaakt, dat moeten we zelf doen. In de *Solution Explorer* aan de rechterkant is een map *Source Files* te zien. Ga met de muispointer daar op staan en klik op de **rechter** muisknop.

Selecteer daarna de optie *Add* en daarna *New item...* . . . Zie figuur B.4.



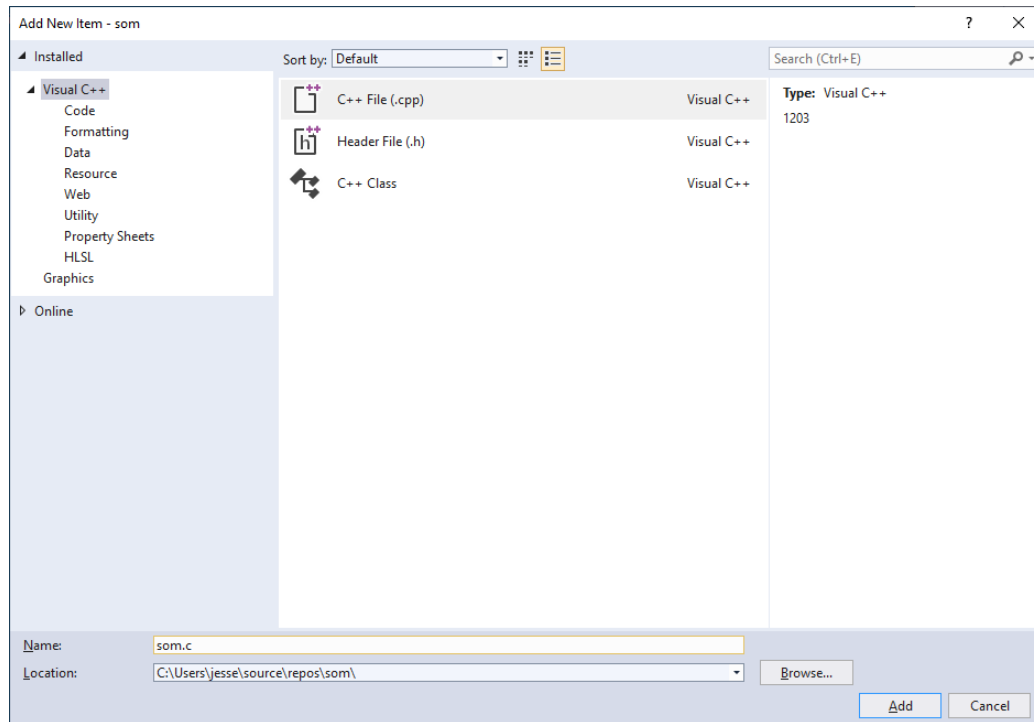
Figuur B.4: Een nieuw bestand aanmaken.

In het volgende scherm moet een bestandstype en een naam worden opgegeven. Klik op *C++ File (.cpp)*. Vul onderin bij *Name* de naam van het bestand in **en zorg ervoor dat de naam eindigt met .c**, anders wordt een C++-bestand aangemaakt. Klik daarna op de knop *Add*. Zie figuur B.5.

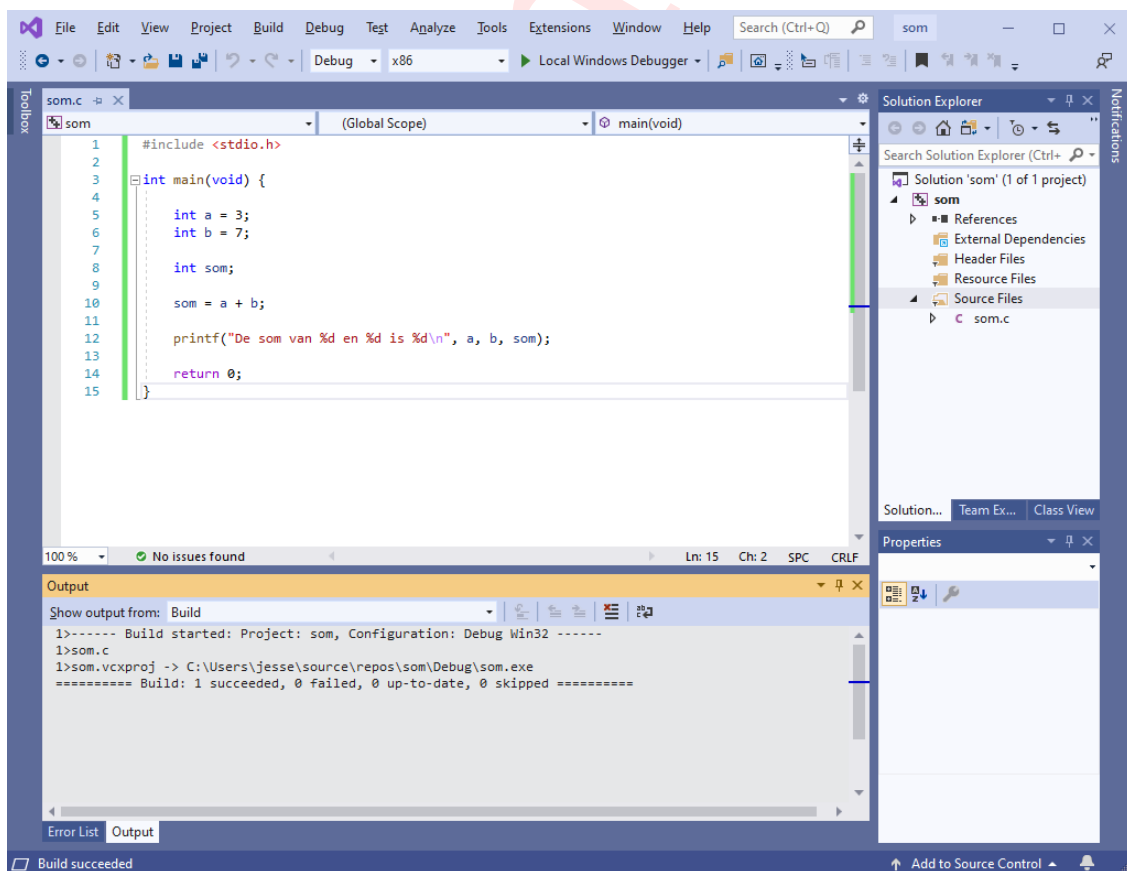
Voer het programma in zoals te zien is in figuur B.6. Klik daarna op de knop *Local Windows Debugger*. Het programma wordt nu gecompileerd en als er geen fouten zijn gevonden, wordt het programma uitgevoerd. Herstel eventuele fouten die door de compiler gevonden worden en herstart de compilatie.

Het programma drukt de regel *De som van 3 en 7 is 10* af. Dit wordt gedaan in een zogenoemde *console*. Dit is te zien in figuur B.7.

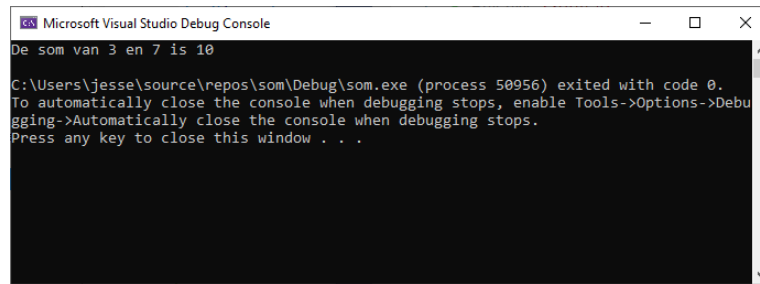
De tutorial is hiermee ten einde.



Figuur B.5: Gegevens van het C-bestand invullen.



Figuur B.6: Compileren en starten van de executable.



```
Microsoft Visual Studio Debug Console
De som van 3 en 7 is 10
C:\Users\jesse\source\repos\som\Debug\som.exe (process 50956) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figuur B.7: *Uitvoer van het programma in een console.*

C

Vorrangsregels van operatoren

In deze tabel zijn alle operatoren in C opgesomd. De voorrang of prioriteit is in aflopende volgorde van hoog naar laag. Twee opmerkelijke operatoren zijn (*type cast*) en `sizeof`. Bij (*type cast*) wordt het casten van een enkelvoudig datatype bedoeld, `sizeof` berekent de grootte in bytes van een datatype of variabele tijdens *compile-time*.

Tabel C.1: Voorrangsregels van alle operatoren.

| Operator | 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 |

Index

Operatoren en constanten

!, logische negatie, 25
!=, operator, 24
-, aftrekken, 23
*, dereferentie, 79
*, vermenigvuldigen, 23
+, operator, 7
+, optellen, 23
->, operator, 109
., operator, 106
/, delen, 23
<, operator, 24
<<, links schuiven, 25
<=, operator, 24
=, operator, 7
==, operator, 11, 24
>, operator, 24
>=, operator, 24
>>, rechts schuiven, 25
?:, operator, 31, 71
[], operator, 13, 66
%, modulus operator, 23, 59
&, adres, 78
&, bitsgewijze AND, 25
&, operator, 117
&&, logische AND, 24
^, bitsgewijze EXOR, 25
~, bitsgewijze inverse, 25
\0, nul-karakter, 21, 74, 84
\, backslash karakter, 21
\n, newline karakter, 21
\r, carriage return karakter, 21
\t, horizontal tab karakter, 21
|, bitsgewijze OR, 25
||, logische OR, 24

A

aantal elementen bepalen, 68
accolades, 6
adres, 20
alfabetische ordening, 75
argument, 8, 47
argumenten

aan een C-programma, 96
array, 13, 65
als argument, 71
als parameter, 71
declaratie van, 65
van pointers, 94
van structures, 110
vergelijken van, 73
array element, 65
assert, functie, 50
assignment, 17
associativiteit, 23
ATmega, 27

B

backslash, 21
backslash karakter, 21
bare metal systeem, 103
bare metal system, 1, 3
bepalen aantal elementen, 68
bestand, 2
bestandssysteem, 2
besturingssysteem, 2
besturingsteken, 127
bibliotheek, 4
bit, 28
bitsgewijze operatoren, 25
buffer overflow, 75
buffering, 119

C

Call by Reference, 91, 110
Call by Value, 56
calloc, functie, 103
carriage return, 21
char, keyword, 19
chip, 3
command line argumenten, 96
compile-time, 3, 68, 135
compiler, 3
compileren, 3
computer, 2
concatenation, 75

conditie, 10
conditionele expressie, 31, 71
conditionele expressie, 109
const, keyword, 22, 108
constante, 20
constante array, 67
cross compiler, 3

D

datatype, 18
datatypeconversie, 27
debuggen, 5
declaratie, 7, 17
double, keyword, 13, 14, 20
dubbele dereferentie, 91
dynamische geheugenallocatie, 100

E

eendimensionaal, 67
element, 65
else, keyword, 10
embedded system, 3
enkelvoudige datatypes, 20
enumeratie, 29, 114
escape sequence, 21
executable, 3
expressie, 7, 23

F

float, keyword, 20
floating point, 13, 20
flowchart, 37
fractie, 24
free, functie, 101
functie, 4, 14, 45
functiedeclaratie, 47
functieprototype, 47

G

geheugen, 2
geheugenallocatie, 100
generieke pointer, 81
getchar, functie, 33
gulden snede, 63

H

header-bestand, 7
horizontal tab, 21

I

I/O, 2
I/O-adressen, 28
if, keyword, 10
initialisatie, 7, 18
instructie, 2
int, keyword, 6, 19
integer, 17
inverteren, 26

invoerbuffer, 119
ISA, 3
iteratie, 12

K

karakterconstante, 21
kast-ongevoelig, 18
keyword, 10
kolomnummer, 69

L

Last In First Out, 51
leap year, 25
library, 4
lifo, 51
linken, 4
logische operatoren, 24
long, keyword, 19
long double, keyword, 20
long long, keyword, 19
lusvariabele, 12

M

machinecode, 3
macro, 101
main, functie, 6, 7
malloc, functie, 100
malloc.h, header-bestand, 100
member, 106
memory leak, 101
microcontroller, 3
modulus operator, 23

N

negatieoperator, 25
newline, 21, 32, 55, 71
niet-nul, 25
nul-bytes, 101, 103
nul-karakter, 21, 74, 84
NULL-pointer, 80, 101

O

one's complement, 26
operating system, 2
overflow, 28, 58

P

parameter, 47
pointer, 20, 77
 als functie-argument, 87
 als return-waarde, 90
 array van, 94
 naar array, 82
 naar array van pointers, 95
 naar functie, 98
 naar pointer, 91
 naar structure, 108
 naar vast adres, 103

- naar void, 81
- NULL, 80
- rekenen met, 85
- postfix, 30
- pragma, 8
- prefix, 21, 30
- printf, functie, 8
- processor, 2
- programma, 2
- programmeren, 3
- promotie, 27

Q

- qualifier, 10, 19, 58
- quicksort, 99

R

- RAM, 2
- realloc, functie, 102
- recursieve functie, 57
- register, 51
- relationele operator, 10
- relationele operatoren, 24
- return, keyword, 6, 52
- returntype, 52
- rijnummer, 69
- ROM, 2
- runtime, 3

S

- samengesteld datatype, 65
- scanf, functie, 8
- schrikkeljaar, 25
- schuifoperator, 26
- secundair geheugen, 2
- short, keyword, 19
- shortcut evaluation, 25
- sizeof, operator, 30, 68, 72, 110
- software, 2
- spatie, 71, 117
- stack, 51
- stack pointer, 51
- standard library, 4
- statement, 5, 6
- static, lokale variabele, 56
- static, qualifier, 56
- stdint.h, header-bestand, 28

- stdio.h, header-bestand, 7, 8
- strcat, functie, 75
- strcmp, functie, 75
- strcpy, functie, 106
- strcpy, functie, 75, 101
- string, 4, 8, 21, 65, 74, 84
- string array, 21
- string constante, 21
- string.h, header-bestand, 75
- strlen, functie, 75
- stroomschema, 37
- struct, keyword, 106
- structure
 - als argument, 106
 - als parameter, 106
 - array van, 110
 - binnen structure, 110
 - member, 106
 - pointer naar, 108
 - typedef, 107

T

- tekenbit, 26
- toekennen, 17
- toekenning, 7
- toolchain, 3
- tweedimensionaal, 69
- type cast, 27
- typedef, keyword, 107

U

- uitvoerbaar bestand, 3, 6
- unaire operator, 23
- underscore, 18
- union, keyword, 114
- unsigned, 19

V

- variabele, 5, 7, 17
- void, keyword, 6, 46
- void-pointer, 81, 100
- voorrangsregels, 32

W

- waarheidstabel, 25
- while, keyword, 12
- wortelformule, 112