De programmeertaal

C

Jesse op den Brouw

Eerste druk

Voor mijn vader Piet op den Brouw, een die-hard C-programmeur.

©2021 Jesse op den Brouw, Den Haag
Versie: 0.577
Datum: 22 juni 2021

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.

De programmeertaal 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 gemaild 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.

De schrijvers geven aan dat hun boek eigenlijk niet bedoeld is om de taal te leren en dat een boek over de beginselen van C helemaal niet zo groot hoeft te zijn. Dat heeft ook te maken met hoe rijk de taal is. C is gebaseerd op een aantal kenmerkende taalconstructies die ook bij vele andere programmeertalen voorkomen. Daarom is dit boek bedoeld als inleiding op de programmeertaal C en is niet toereikend om alle facetten van de taal te leren. Het boek beschrijft ook niet een specifieke standaard van de taal. 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). Zo'n beetje alle programmavoorbeelden van enige omgang zijn getest op Visual Studio. Dit is ook de meest gebruikte IDE. De IDE wordt zeker niet alleen gebruikt voor het ontwikkelen van C-programma's. Bekend is bijvoorbeeld de taal C# die veel wordt gebruikt bij het ontwikkelen van programma's op het Windows-besturingssysteem.

Hoewel C al een oude taal is, wordt het nog volop gebruikt. De reden daarvoor is dat de C-statements door de compiler worden vertaald naar instructies die de microprocessor direct kan uitvoeren. Daarom wordt de taal veel gebruikt bij het schrijven van programma's op *microcontrollers*. Dit zijn kleine computersystemen zonder besturingssysteem. Op deze systemen is C (en C++) de enige taal die gebruikt kan worden. Maar C wordt ook gebruikt op systemen die wel een besturingssysteem draaien. Zo worden bij het Windows-besturingssysteem zogenoemde *drivers* (koppelingen tussen een gebruikersprogramma en de hardware) in C of C++ geschreven. Andere talen zijn daarvoor niet toepasbaar. Bekend is ook dat het Linux-besturingssysteem grotendeels is geschreven in C (en bevat duizenden C-bestanden).

Dit boek is geschreven voor hbo-studenten Elektrotechniek en Informatica. Zij komen vaak in aanraking met *microcontrollers* en dan is, zoals eerder al is vermeld, C vaak de enige taal die gebruikt kan worden. Daarnaast vonden we dat er eigenlijk geen goede boeken zijn op het gebied van C. We hebben getracht het boek kort te houden maar toch voorbeelden te geven die makkelijk gevolgd kunnen worden, ook door de beginnende programmeur.

Dit boek is opgemaakt in LaTeX [2] (LaTeX-engine = XeLaTeX 0.999993). LaTeX leent zich uitstekend voor het opmaken van lopende tekst, tabellen, figuren, programmacode, vergelijkingen en referenties. De gebruikte LaTeX-distributie is TexLive uit 2020 [3]. Als editor is TexStudio [4] gebruikt. Tekst is gezet in Calibri [5], een van de standaard fonts op een bekend besturingssysteem. Code is opgemaakt in Consolas [6] met behulp van de *listings*-package [7]. Voor het tekenen van arrays, 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. Een aantal programmafragmenten is ontwikkeld door collega Harry Broeders, de overige fragmenten zijn door de auteur ontwikkeld.

Natuurlijk zullen docenten ook nu opmerken dat dit boek niet aan al hun verwachtingen voldoet. Dat zal altijd wel zo blijven. Daarom is de broncode van dit boek 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. We doen een klemmend beroep aan alle lezers om fouten en omissies te melden.

Leeswijzer

Hoofdstuk 1 geeft in vogelvlucht enkele belangrijke concepten van C weer. Er wordt gesproken over wat een computer en een compiler zijn 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 behandelen het concept van variabelen en datatypes. Verder wordt even stilgestaan bij beslissen en herhalen en functies. In dit hoofdstuk worden geen pointers, structures en bestandsbewerkingen behandeld.

Hoofdstuk 2 gaat dieper in op variabelen, constanten en expressies. Alle beschikbare datatypes worden behandeld alsmede 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 behandeld 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. We behandelen het gebruik van beslissingen aan de hand van een bepaalde voorwaarde (ook wel conditie genoemd). Een beslissing kan uitgebreid worden met een programmadeel dat uitgevoerd moet worden als een conditie waar of onwaar is. Herhalingen komen zeer vaak voor in een programma. We behandelen drie voorkomende taalconstructies hiervoor.

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 wijze kunnen worden beschreven. Hoewel het gebruik van flowcharts al vrij oud is, wordt het nog volop gebruikt. Het mooie van deze methode is dat het ook bij andere situaties kan worden gebruikt, zoals het beschrijven van de *flow* van processen in een organisatie

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. We behandelen verder het gebruik van lokale en globale variabelen en hoe de zichtbaarheid (scope) geregeld is.

Hoofdstuk 6 gaat over arrays. We tonen hoe een array wordt gedefinieerd en kan worden gebruikt. We laten zien hoe arrays als argument aan functies kan worden meegegeven. We behandelen strings, een array van karakters en laten zien dat voor het bewerken van strings diverse functies beschikbaar zijn.

Hoofdstuk 7 gaat over structures en unions 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. We beschrijven kort de union als een probaat middel om de hoeveelheid gebruikte geheugen in te dammen (alhoewel dat bij moderne PC's eigenlijk geen rol speelt). Ook bitfields worden behandeld, alhoewel bitfields in de praktijk nauwelijks worden gebruikt.

Hoofdstuk 8 gaat over pointers. Er wordt uitgelegd wat een pointer is en wat de relatie met arrays 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. We geven een introductie in het gebruik van dynamische geheugenallocatie, noodzakelijk bij self-referential structures. Het gebruik van self-referential structures (voor onder andere *linked lists*) wordt als verdiepend behandeld, hoewel het geen onderdeel van de taal zelf is.

Hoofdstuk 9 gaat over invoer en uitvoer. Dit zijn geen onderdelen van de taal zelf maar komt zo vaak voor dat er er flink stuk aan wijden. We kijken wat dieper naar het afdrukken op het scherm en het inlezen van het toetsenbord. Natuurlijk behandelen we ook hoe we informatie in bestanden kunnen bewerken: hoe worden bestanden geopend, geschreven, gelezen en gesloten en waar moeten we op letten als we met bestanden werken. Diverse besturingssystemen regelen de naamgeving van bestanden op eigen wijze. We laten zien hoe bestandsnamen op Windows en Linux worden vormgegeven.

Hoofdstuk 10 gaat over de preprocessor. De preprocessor is een faciliteit die vóór de "echte" C-compiler wordt gebruikt. We laten zien hoe we header-bestanden kunnen inlezen, hoe we macro's kunnen definiëren en hoe we macro's kunnen gebruiken voor conditionele compilatie.

Hoofdstuk 11 gaat over het compilatieproces. We behandelen hoe het proces in elkaar steekt en wat er allemaal bij komt kijken voordat een uitvoerbaar programma is gerealiseerd. We laten zien hoe we een C-programma over meerdere bestanden kunnen verdelen en hoe we een bibliotheek kunnen realiseren. Ook behandelen we in dit hoofdstuk het verschil tussen globale

en lokale variabelen en laten zien hoe de zichtbaarheid van globale variabelen en functies kan worden beperkt tot het C-bestand waarin ze zijn gedefinieerd. In dit hoofdstuk wordt ook het verschil tussen definitie en declaratie behandeld.

In bijlage A worden de eerste 128 karakters van de ASCII-code behandeld. Diverse karakters zijn niet afdrukbaar, maar worden gebruikt om besturingscommando's tussen systemen te realiseren. De ASCII-code bevat de in westerse landen gebruikelijke tekens.

In bijlage B is een korte tutorial beschreven voor het ontwikkelen van C-programma's met Visual Studio. Hoewel we ons hier concentreren op Visual Studio, werken alle Integrated Development Environments (IDE) op gelijke wijze.

In bijlage C is de complete lijst te zien met voorrangsregels van operatoren in C. De operatoren zijn in aflopende volgorde van prioriteit gerangschikt.

Een aantal paragrafen is gekenmerkt met een * voor het paragraafnummer. Deze paragrafen bevatten verdiepende stof en kunnen bij een eerste cursus worden overgeslagen.

Studiewijzer

Om de taal goed te doorgronden zouden eigenlijk alle hoofdstukken aan bod moeten komen. Bij een eerste introductie van de taal kunnen hoofdstuk 4 en 11 worden overgeslagen.

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]. Eigenlijk vertelt het boek alle kennis die elke beginnende C-programmeur moet weten. We hebben er naar gestreefd om alleen de taal te behandelen en geen onnodige informatie te verschaffen. Dat resulteert soms in schijnbaar onzinnige voorbeelden. Dat hebben we expres gedaan om de aandacht te houden bij het besproken onderwerp. Toch geven we hier en daar wat uitgebreide voorbeelden die direct aan een C-compiler kunnen worden gevoerd. Het boek gaat niet in op het gebruik van C in microcontroller-omgevingen, zoals Arduino.

Gebruik van Engelse woorden

Het boek is doorspekt met Engelse woorden. Dat heeft te maken met het gangbare gebruik van deze woorden. Zo spreken we in dit boek over pointers in plaats van het Nederlandse woord wijzers. Aan de andere kant gebruiken we het woord lus bij herhalingen en niet van het Engelse woord loop. Het een en ander komt ook voort uit de persoonlijke voorkeur van de auteur.

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.

C-standaard

Dit boek bespreekt niet een specifieke C-standaard. De meeste voorbeelden draaien op C89 en C99.

Afdrukken van dit boek

Dit boek is opgemaakt op A4 met fontgrootte 12 pt. Het boek kan dus gewoon via een printer worden afgedrukt. Het boek kan ook worden afgedrukt op twee pagina's per bladzijde. De fontgrootte is dan 8,5 pt. Een andere optie is het boek af te drukken op 17x24 cm (academisch formaat). Vele online-printbedrijven ondersteunen dit formaat. De fontgrootte is dan 10 pt.

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 onuitputtelijke stroom van correcties en opmerkingen. Daarnaast zijn enkele programmavoorbeelden van hem gebruikt en is hoofdstuk 5 gebaseerd op een schrijven van zijn hand. Collega Kees de Joode wordt bedankt voor zijn bijdrage aan de wiskundige functies die in dit boek beschreven zijn. Jon van den Helder, Gerard Tuk en Jos Knol worden bedankt voor enkele rake opmerkingen en verbeteringen. René Terhorst wordt bedankt voor informatie over de verschillende C-standaarden. Verder wil ik Piet op den Brouw (mijn vader, een die-hard C programmeur) bedanken voor het opsporen van fouten en het geven van suggesties.

De ASCII-tabel op pagina 188 is ontwikkeld door Victor Eijkhout en is met zijn toestemming gebruikt. De tabel kan gevonden worden op http://www.ctan.org/tex-archive/info/ascii-chart.

Jesse op den Brouw, juni 2021.

Inhoudsopgave

Voorwoord iii

1 Un tour de C 1
- 1.1 De computer . . . 2
- 1.2 De compiler . . . 4
- 1.3 Bibliotheken en functies . . . 4
- 1.4 Ontwikkelsystemen . . . 5
- 1.5 Een minimaal C-programma . . . 6
- 1.6 Afdrukken op het scherm . . . 7
- 1.7 Invoer van het toetsenbord . . . 9
- 1.8 Keywords . . . 11
- 1.9 Beslissingen . . . 11
- 1.10 Herhalingen . . . 12
- 1.11 Arrays . . . 13
- 1.12 Functies . . . 15
- 1.13 Programmeerstijlen . . . 16
- 1.14 En verder... . . . 18

2 Variabelen, datatypes en expressies 19
- 2.1 Variabelen . . . 19
- 2.2 Datatypes . . . 21
- 2.3 Constanten . . . 23
- 2.4 Expressies . . . 25
  - 2.4.1 Rekenkundige operatoren . . . 26
  - 2.4.2 Relationele operatoren . . . 27
  - 2.4.3 Logische operatoren . . . 28
  - 2.4.4 Bitsgewijze operatoren . . . 29
- 2.5 Datatypeconversie . . . 31
- 2.6 Overflow . . . 32
- 2.7 Vaste-lenge datatypes . . . 32
- 2.8 Enumeraties . . . 33
- 2.9 Overige operatoren . . . 34
  - 2.9.1 Grootte van een variabele . . . 34
  - 2.9.2 Verhogen of verlagen met 1 . . . 35
  - 2.9.3 Toekenningsoperatoren . . . 35
  - 2.9.4 Conditionele expressie . . . 36

2.9.5 Komma-operator . . . 36
2.9.6 Nog enkele operatoren . . . 37
2.10 Voorrangsregels van operatoren . . . 37
2.11 Volgorde van uitrekenen van expressies . . . 38

3 Beslissen en herhalen **39**
3.1 Blok . . . 39
3.2 Beslissen met het `if`-statement . . . 40
3.3 Beslissen met het `if else`-statement . . . 41
3.4 Beslissen met het `switch`-statement . . . 42
3.5 Herhalen met het `while` statement . . . 45
3.6 Herhalen met het `for` statement . . . 46
3.7 Herhalen met het `do while` statement . . . 48
3.8 Extra lusbewerkingen: de `break` en `continue`-statements . . . 48
3.9 Increment- en decrement-operatoren . . . 49
3.10 Komma-operator . . . 50
3.11 Het `goto`-statement . . . 52
3.12 Op de valreep, het verschil tussen = en == . . . 52

4 Flowcharts **53**
4.1 `if`-statement . . . 54
4.2 `if-else`-statement . . . 55
4.3 `while`-lus en `do-while`-lus . . . 55
4.4 `for`-lus . . . 56
4.5 `switch`-statement . . . 58
4.6 Voorbeeld . . . 59

5 Functies **61**
5.1 Een eenvoudige functie . . . 62
5.2 Functies met parameters . . . 64
5.3 Functies met een returnwaarde . . . 67
5.4 Zichtbaarheid en levensduur van lokale variabelen . . . 71
5.5 Zichtbaarheid en levensduur van globale variabelen . . . 74
5.6 Zichtbaarheid en levensduur van functies . . . 75
5.7 Recursieve functies . . . 75
5.8 Complexiteit van recursieve functies . . . 77
5.9 Pointers en functies als parameter . . . 80
5.10 Functies die meer dan één waarde teruggeven . . . 80
5.11 Wiskundige functies . . . 81
5.12 Classificatie van karakters . . . 82

6 Arrays **85**
6.1 Definitie en selectie . . . 85
6.2 Initialisatie . . . 86
6.3 Eendimensionale array . . . 87
6.4 Aantal elementen van een array bepalen . . . 88
6.5 Tweedimensionale array . . . 88
6.6 Afdrukken van array . . . 90

6.7 Arrays vergelijken . . . 91
6.8 Arrays als argument en parameter . . . 92
6.9 Strings . . . 94
6.9.1 Stringfuncties . . . 95

7 Structures **97**
7.1 Definitie van structures . . . 97
7.2 Toegang tot members . . . 98
7.3 Toekennen van structures . . . 98
7.4 Vergelijken van structures . . . 99
7.5 Functies met structures . . . 99
7.6 Typedef . . . 100
7.7 Array van structures . . . 101
7.8 Structures binnen structures . . . 102
7.9 Teruggeven van meerdere variabelen . . . 103
7.10 Unions . . . 105
7.11 Bitvelden . . . 107

8 Pointers **109**
8.1 Pointers naar enkelvoudige datatypes . . . 110
8.2 De NULL-pointer . . . 112
8.3 Pointer naar `void` . . . 113
8.4 Afdrukken van pointers . . . 113
8.5 Pointers naar arrays . . . 114
8.6 Strings . . . 116
8.7 Rekenen met pointers . . . 117
8.8 Relatie tussen pointers en arrays . . . 118
8.9 Pointers als argumenten . . . 120
8.10 Pointer als returnwaarde . . . 124
8.11 Pointers naar structures . . . 125
8.12 Pointers naar pointers . . . 126
8.13 Array van pointers . . . 129
8.14 Pointers naar een array van pointers . . . 130
8.15 Argumenten meegeven aan een C-programma . . . 132
8.16 Pointers naar functies . . . 133
8.17 Dynamische geheugenallocatie . . . 135
8.18 Generieke geheugenfuncties . . . 138
8.19 Gekoppelde lijsten . . . 139
8.19.1 Dynamisch gekoppelde lijsten . . . 142
8.20 Subtiele verschillen en complexe definities . . . 148

9 Invoer en uitvoer **149**
9.1 Bestandsnamen en het bestandssysteem . . . 149
9.2 Uitvoer naar het beeldscherm . . . 151
9.3 Inlezen van het toetsenbord . . . 152
9.4 Bestanden openen voor gebruik . . . 154
9.5 Bestand sluiten na gebruik . . . 155
9.6 Schrijven naar een bestand . . . 155

9.7 Lezen uit een bestand . . . 157
9.8 Standaard invoer en uitvoer . . . 157
9.9 Andere bestandsfuncties . . . 158
9.10 Lezen uit en schrijven naar een string . . . 159
9.11 Gebruik van command line argumenten en bestanden . . . 160
9.12 Problemen met `scanf` . . . 162

10 De preprocessor **169**
10.1 Invoegen van bestanden . . . 169
10.2 Macrovervanging . . . 170
10.3 Conditionele compilatie . . . 171
10.4 Conditioneel inlezen van een bestand . . . 172

11 Het compilatieproces **173**
11.1 Stappen in de compilatie . . . 174
11.2 Een C-programma in één bestand . . . 174
11.3 Een C-programma in meerdere bestanden . . . 174
11.4 Bibliotheken . . . 176
11.5 Linkage . . . 176
11.6 Assembler-bestanden . . . 177
11.7 Definitie en declaratie . . . 177
11.8 De qualifiers `extern` en `static` . . . 178
11.9 Bibliotheek maken . . . 181
11.10 Dynamic en static linking . . . 181
11.11 Optimalisatie . . . 182

Bibliografie **185**

A De ASCII-tabel **187**

B Tutorial Visual Studio **189**

C Voorrangsregels van operatoren **195**

Index **197**

1

Un tour de C

C is al een oude taal. De taal is rond 1970 ontworpen door Dennis Ritchie[1] en is dus al zo'n 51 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 dan gebruikelijk was voor die tijd. In die tijd werden besturingssystemen geschreven in assembly, een taal die dicht bij 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-generatietaal (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 besturingssysteem. Daarom is de taal geliefd bij programmeurs van kleine computersystemen waarop geen besturingssysteem 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 aan het schrijven van compacte programma's. Dat is te zien aan de soms onduidelijke taalconstructies. C is daarom lastig voor beginnende programmeurs. Maar het is, zoals gezegd, vaak de enige taal die op een een bepaald platform gebruikt kan worden en 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 besturingssystemen). 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 zijn echter

[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 Apple's OS-X.

talen die dit veel beter ondersteunen. Ook het verwerken van bestanden en *strings* (een rij karakters) is mogelijk maar dat 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's[3]. C moet concurreren met vele andere talen zoals C++, C#, Java en Rust [10]. C++ is ontwikkeld als "de betere C" en ondersteunt *objectgeoriënteerd* programmeren. Ook C# ondersteunt objectgeoriënteerd programmeren, is ontwikkeld door Microsoft en is de *de facto* programmeertaal op Windows-systemen. Java is een taal die onafhankelijk van een computersysteem gebruikt kan worden, ondersteunt objectgeoriënteerd programmeren maar kan alleen gebruikt worden als Java ondersteund wordt op het systeem. Rust wordt gezien als de veilige C en C++.

Standaard C staat ook wel bekend als ANSI C of C89 [11]. Momenteel is C17 de standaard voor de C programmeertaal. C2x zal waarschijnlijk in december 2021 worden geratificeerd.

1.1 De computer

Een *computer* is een elektronisch, digitaal systeem dat *data* (gegevens) verwerkt aan de hand van een lijst *instructies*. Het Engelse werkwoord 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 geheugen waarvan de inhoud niet gewijzigd kan worden; RAM (Random Access Memory) is geheugen waarvan de inhoud wel gewijzigd kan worden. We zullen zo meteen zien waarvoor ROM en RAM worden gebruikt.

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 besturingssytemen zijn Windows, Linux en Apple's OS-X.

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: files) gezet. De manier waarop dat wordt gerealiseerd wordt een *bestandssysteem* (Engels: file system) genoemd. Bekende bestandssystemen zijn NTFS op

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

Windows en ext3 op Linux. Het besturingssysteem zorgt ervoor dat de bestanden op ordentelijke wijze kunnen worden benaderd.

In de praktijk komen we een drietal computersystemen tegen. Natuurlijk zijn er de algemeen bruikbare computers waarop een besturingssysteem draait (PC, laptops met Windows, Linux, Apple's OS-X) en die een grote hoeveelheid aan verschillende programma's kan uitvoeren.

Daarnaast zijn er zogenoemde *embedded systems*. Een embedded system in 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 algemeen 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 kant-en-klare ontwikkelsystemen om programma's voor microcontrollers te produceren. Op deze systemen is C (en C++) de enige taal die gebruikt kan worden. We gaan in dit boek hier verder niet op in.

We kunnen natuurlijk niet de *smartphones* vergeten. Zo'n beetje iedereen in Nederland heeft er een. We kunnen een smartphone vergelijken met een algemeen bruikbaar computersysteem, waarop de besturingssystemen Android of iOS draaien. Op deze systemen kunnen programma's worden geïnstalleerd, *apps* genaamd. Deze apps worden over het algemeen *niet* in C geprogrammeerd, en vallen verder buiten het bestek van dit boek.

Bij PC's en laptops is een klein programma in ROM aanwezig. Dit zorgt ervoor dat de PC kan opstarten. Het heeft tot doel om het besturingssysteem te laden (Windows, Linux). Het besturingssysteem en de programma's worden dus in RAM geladen, net als de data waarmee de programma's werken. Op smartphones is het besturingssysteem opgeslagen in ROM die herprogrammeerbaar is. Zo kunnen updates van het besturingssysteem geïnstalleerd worden. De apps en data worden geladen in de RAM. Bij microcontrollers is het complete programma geladen in ROM, die ook herprogrammeerbaar is. De data is geplaatst in RAM.

Bits and bytes, size does matter...

De meeste gangbare computersystemen slaan data op in *bytes*. Een byte is een eenheid van 8 *bits*. Het woord *bit* is een samentrekking van *binary digit*. Binary digit betekent *binair cijfer* en binair betekent *tweewaardig*. Dat betekent dat een bit twee verschillende waarden kan hebben. We noemen die waarden 0 of 1. Met behulp van bits en bytes kunnen we getallen representeren. Het kleinste getal in een byte is 00000000_2 waarbij het subscript 2 aangeeft dat het om een binair getal gaat, en het grootste getal is 11111111_2. Dit komt overeen met het decimale getal 0 respectievelijk het decimale getal 255. Om grotere getallen te representeren, zijn meerdere bytes nodig. Moderne systemen kunnen *van nature* overweg met 32-bits eenheden, ze kunnen met 32 bits als één eenheid rekenen. Daarvoor zijn 4 bytes nodig. We spreken dan van een *32-bits systeem*. Daarnaast kunnen veel processoren ook overweg met 64-bits eenheden, dus zijn er 8 bytes nodig voor zo'n eenheid. We spreken dan van een *64-bits systeem*.

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. De verzameling programma's om een C-programma te compileren wordt een *toolchain* genoemd. Het vertaalde programma wordt een *uitvoerbaar bestand* of *executable* genoemd. Een uitvoerbaar bestand wordt net als een C-programma opgeslagen op de vaste schrijf. Bij het ontwikkelen van programma's voor microcontrollers wordt gebruik gemaakt van PC of laptop met een besturingssysteem en een C-compiler specifiek voor de microcontroller. 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. Al deze functies zijn on-

dergebracht 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 voegt dan de functie aan het programma toe. Dat toevoegen wordt *linken* genoemd.

Bij de C-compiler wordt een vrij beperkte 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 Ontwikkelsystemen

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.

Command Prompt

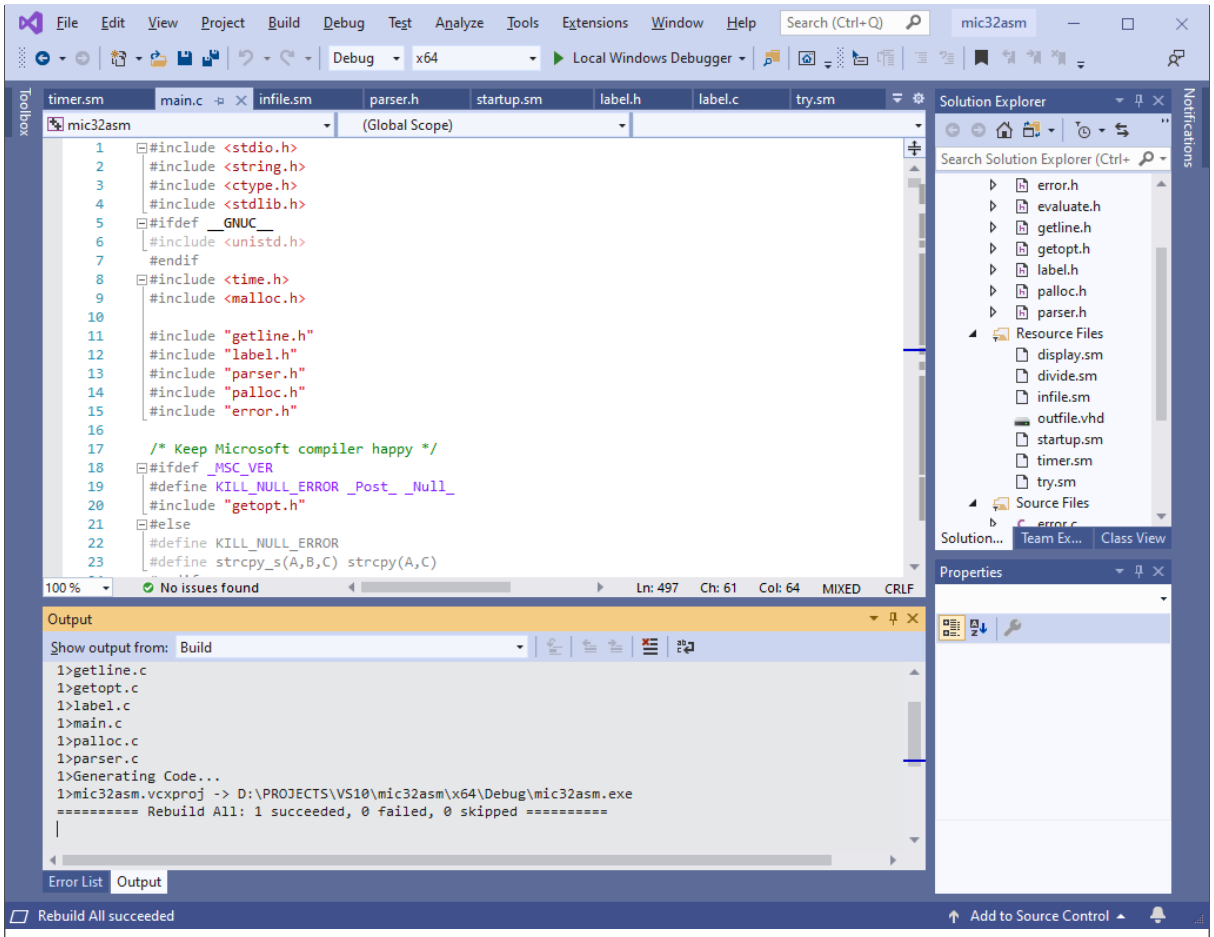
```
C:\Users\Cbook> notepad mooi.c                    (start Notepad)
C:\Users\Cbook> gcc -o mooi.exe mooi.c            (start C-compiler)
C:\Users\Cbook> .\mooi.exe                        (start uitvoerbaar programma)
C is een mooie taal                               (de uitvoer op het scherm)
C:\Users\Cbook>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

Figuur 1.1: *Een voorbeeld van een command line interface.*

Dit is echter niet de werkwijze van veel programmeurs. Gelukkig zijn er goede ontwikkelsystemen (IDE: Integrated Development Environment) die het programmeerwerk verlichten. Bekende systemen zijn Microsoft Visual Studio [12], Code::Blocks [13] en Apple's Xcode [14]. Zulke ontwikkelsystemen zorgen ervoor dat de programmeur gemakkelijk het programma kan invoeren, de compiler kan starten en het programma kan *debuggen*. Dat laatste is vaak nodig als blijkt dat de executie van 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.

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



Figuur 1.2: *Voorbeeld van Microsoft Visual Stdio.*

programma moet gecompileerd worden. Als we een wijziging 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 behandelen. 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 uitleggen 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'.

```
int main(void)
{
    return 0;
}
```

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 uitvoerbare 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 besturingssysteem[4]. 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 *accolades*. Het Engelse woord hiervoor is *curly braces* of gewoon *braces*. De accolades geven het begin en einde aan van *blok*. Een blok begint met een accolade-openen ({) en eindigt met een accolade-sluiten (}). Over de plaats van de accolades 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 uitvoering 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. 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. Vanaf C99 is het niet meer nodig om dit `return`-statement uit te voeren. Dan wordt automatisch het getal 0 teruggegeven.

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.

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 *altijd* de functie `main`. Het (gecompileerde) programma wordt hier gestart. Het programma wordt gestopt na het laatste statement in `main`. Dit is per definitie een `return`-statement.

[4] Dat kan wel, zie hoofdstuk 8.

```
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 5 en 6 worden twee *variabelen* gedefinieerd, 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. Definitie wil zeggen dat een variabele kenbaar wordt gemaakt én dat er geheugenruimte wordt gereserveerd. Bij de defnitie wordt opgegeven wat het type is van de variabele. Dit gebeurt middels het keyword `int`, dat betekent dat a en b alleen gehele getallen kunnen opslaan (Engels: integer). Aan de variabelen worden gelijk waarden toegekend. We noemen dit *initialisatie* van variabelen.

In regel 8 wordt de variabele `som` gedefinieerd 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 en 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. Deze 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 nu 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.

```
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: *Programma om de som van twee getallen te bepalen.*

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 10.

Het programma volgt verder de lijn van listing 1.2. In de functie `main` definiëren we drie variabelen. Daarna drukken we in regel 13 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 specification %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 19 rekenen we de som uit van variabelen a en b en kennen die toe aan variabele `som`. In regel 21 drukken we de drie variabelen af. Het programma wordt afgesloten in regel 23.

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 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.

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.*

We hebben nu gezien hoe we invoer en uitvoer voor een programma kunnen gebruiken. In veel voorbeelden zullen we hiervan gebruik maken.

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*:

```
#pragma warning(disable : 4996)
```

We geven 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 en met behulp van *conditionele compilatie* kan deze pragma overgeslagen worden). 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.

1.8 Keywords

In tabel 1.1 is een lijst te zien met gereserveerde woorden. Een aantal van deze woorden hebben we al gezien zoals `int`, `void` en `return`. Deze woorden worden *keywords* genoemd en vormen de vocabulaire van de taal C.

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

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

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

```
unsigned long int a;
```

Dit bepaalt de grootte (het aantal bits) van variabele a. Zie hoofdstuk 2.

1.9 Beslissingen

Met behulp van de keyswords `if` en `else` kunnen we in het programma beslissingen nemen op basis van een *conditie*. Dit is te zien in listing 1.4. We definiëren 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

```
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.*

in de regels 9 en 11 uitgevoerd. Is de conditie niet waar, dan worden deze 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* [15].

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 *multiway branch* (to branch = vertakken) genoemd. Zie listing 1.5.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a, b;
6
7      printf("Geef twee getallen: ");
8      scanf("%d %d", &a, &b);
9
10     if (a < b)
11     {
12         printf("a is kleiner dan b\n");
13     }
14     else if (a == b)
15     {
16         printf("a is gelijk aan b\n");
17     }
18     else
19     {
20         printf("a is groter dan b\n");
21     }
22     return 0;
23 }
```

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

Technisch gezien hoort de `else` in regel 18 bij de `if` in regel 14. 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*. Een andere, veel gebruikte term is *lus*. Dit is te zien in listing 1.6.

We beginnen het programma met de definitie en initialisatie van een aantal variabelen. Hierin stellen we de ondergrens, bovengrens en stapgrootte in (regel 5). We willen beginnen bij de

```
#include <stdio.h>

int main(void)
{
    int ondergrens = 1, bovengrens = 10, stap = 1;
    int getal, kwadraat;

    getal = ondergrens;
    while (getal <= bovengrens)
    {
        kwadraat = getal * getal;
        printf("Het kwadraad van %3d is %3d\n", getal, kwadraat);
        getal = getal + stap;
    }

    return 0;
}
```

Listing 1.6: *Afdrukken van de kwadraten van 1 t/m 10..*

ondergrens, stoppen bij de bovengrens en bij elke herhaling nieuwe waarden afdrukken. Daarvoor gebruiken we de variabele `getal`. Zo'n variabele wordt een *lusvariabele* genoemd.

In regel 8 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 conditie 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 dat volgt op het `while`-statement.

Binnen de accolades van het `while`-statement zien we drie statements. In regel 11 wordt het kwadraat berekend van de (huidige) waarde van de lusvariabele. Daarna worden de lusvariabele en het kwadraat afgedrukt. In regel 13 wordt de lusvariabele aangepast naar de nieuwe (volgende) waarde door de stapgrootte erbij op te tellen[5].

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 kwadraat ook uitrekenen in de aanroep van de functie `printf`. Als argument van een functie mogen we namelijk een expressie gebruiken. Zo sparen we een variabele uit. Zie listing 1.7.

1.11 Arrays

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

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

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

```
    for (getal = ondergrens; getal <= bovengrens;
                                        getal = getal + stap)
    {
        printf("Het kwadraat van %3d is %3d\n",
                                        getal, getal * getal);
    }
```

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

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 de *elementen* van de array 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] = {3.14159,2.71828,0.57721,1.41421,1.61803};
6      double som = 0.0;
7      int index;
8
9      for (index = 0; index < 5; index = index + 1)
10     {
11         som = som + lijst[index];
12     }
13
14     printf("Gemiddelde: %f\n", som / 5.0);
15
16     return 0;
17 }
```

Listing 1.8: *Gemiddelde van vijf getallen.*

De array mag bij definitie gelijk geïnitialiseerd worden zoals te zien is in regel 5[6]. 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 definiëren, we kunnen als argument gewoon `som / 5.0` gebruiken.

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

Een veel gebruikte arraytype is de *string*. Dit is een array van karakters, afgesloten met een speciaal karakter dat het *null-karakter* of *null-byte* genoemd worden. C heeft geen ingebouwde taalconstructies die direct met strings werken, maar er zijn veel *functies* beschikbaar om strings te verwerken. Een voorbeeld van een string is:

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

Listing 1.9: *Voorbeeld van een string.*

We hoeven de lengte van de string in dit geval niet op te geven, dat wordt automatisch door de C-compiler uitgerekend. Strings worden behandeld in hoofdstuk 6.

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 de berekeningen worden gedaan, we moeten alleen maar weten hoe we de functie moeten gebruiken. We hebben al drie functies gezien: `printf`, `scanf` en `main`. De functies `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).

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

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 1.10. 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 gebruik 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.

```c
#include <stdio.h>

double square_root(double s)
{
    double xiter = s;
    int i;

    if (s == 0.0)
    {
        return 0.0;
    }

    for (i = 0; i < 10; i = i + 1)
    {
        xiter = 0.5 * (xiter + s / xiter);
    }

    return xiter;
}

int main(void)
{
    double i;

    for (i = 0.0; i < 11.0; i = i + 1.0)
    {
        printf("De wortel van %6.3f is %.6f\n", i, square_root(i));
    }

    return 0;
}
```

Listing 1.10: *Programma om de wortels van 1 t/m 10 af te drukken.*

In het `for`-statement worden tien iteraties van de middelste formule uit (1.1) uitgevoerd. Bij elke iteratie 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. Functies worden behandeld in hoofdstuk 5.

1.13 Programmeerstijlen

C is erg coulant in het vormgeven van een programma. We kunnen regels overslaan (een lege regel) en we kunnen statements vooraf laten gaan door een willekeurig aantal spaties of we kunnen statements achter elkaar op een regel plaatsen. Ook het gebruik van variabelenamen

en functienamen staat geheel los van de taal. We kunnen één letter gebruiken of een compleet woord.

Een mooi voorbeeld hoe we onze wortelfunctie onleesbaar kunnen maken, kunnen we zien in listing 1.11. De functie wordt door de C-compiler correct vertaald maar is voor een mens slecht leesbaar. De code in listing 1.10 is veel beter leesbaar.

```
double square_root(double s) {
double xiter = s; int i;
if (s==0.0) return 0.0;
for (i=0;i<10;i=i+1) xiter=0.5*(xiter+s/xiter); return xiter;}
```

Listing 1.11: *Een functie om een wortel van een getal te bepalen.*

Hoe een programma uiterlijk wordt vormgegeven noemen we een *programmeerstijl*. Er zijn behoorlijk wat stijlen. We geven hieronder de meest gebruikte aan.

Layout van het programma

In de K&R-stijl (Kernighan en Ritchie) wordt de accolade-openen *achter* een functie, beslissings- of herhalingstatement geschreven. De statements hierbinnen worden *ingesprongen* (dat wordt in het Engels *indentation* genoemd). Bij gebruik van in elkaar verweven beslissings- of herhalingstatement worden de statements verder of dieper ingesprongen.

```
while (a < b) {
    statements
    if (a < 0) {
        statements
    }
}
```

Listing 1.12: *K&R-stijl.*

In de Allman-stijl wordt de accolade-openen *onder* een functie, beslissings- of herhalingstatement geschreven. De statement hierbinnen worden *ingesprongen*. Bij gebruik van in elkaar verweven beslissings- of herhalingstatement worden de statement verder of dieper ingesprongen.

```
if (a < b)
{
    if (a < 0)
    {
        statements
    }
}
```

Listing 1.13: *Allman-stijl.*

In de eerste hoofdstukken gebruiken we de Allman-stijl. In latere hoofdstukken gebruiken we de K&R-stijl.

Naamgeving van variabelen

Ook voor naamgeving van variabelen zijn diverse gebruiken in omloop. We zullen er drie behandelen.

In de K&R-stijl zijn er eigenlijk geen regels. Variabelennamen mogen uit één letter bestaan, of natuurlijk uit meerdere letters. Als namen syntactisch uit meerdere woorden bestaan, worden de woorden gescheiden door een *underscore*. Het is wel aan te raden dat namen zinvol zijn en dat ze de betekenis vertegenwoordigen. Overigens is het gebruik van eenletterige variabelennamen prima te verantwoorden, bijvoorbeeld binnen het gebruik van een kortdurdende herhaling én als het programma niet al te groot is[7].

In de camelCase-stijl worden variabelen weergegeven met kleine letters en hoofdletters en geven scheidingen aan tussen woorden die syntactisch gescheiden zijn. Er worden geen underscores gebruikt. Deze stijl is afkomsting uit Java.

```
int dayOfTheWeek;
int dayWithinMonth;
int dayOfTheYear;
```

Listing 1.14: *camelCase-stijl.*

Bij de Hungarian Notation worden variabelennamen vooraf gegaan door een omschrijving van het datatype en het semantiek van de variabele. Zo kunnen we integers vooraf laten kunnen gaan door een `i`. Maar dat zien de meeste programmeurs ook wel. We kunnen van een integer ook het doel aangeven. Zo wordt bijvoorbeeld een variabele die dient als teller in een herhaling vooraf gegaan door `c` (van "count"):

```
int iDayOfTheWeek;
int cLoops;
```

Listing 1.15: *Voorbeel Hungarian Notation.*

Deze stijl wordt vooral gebruikt bij grote teams programmeurs die gezamenlijk aan één programma werken. Het nadeel van deze stijl is dat de typering niet gestandaardiseerd is. Overigens kan met ontwikkelomgevingen het type gevonden worden door op de variabele te klikken.

1.14 En verder...

We hebben nu enkele concepten van C uitgelegd. Maar de taal kan meer. Wat we niet behandeld hebben zijn *pointers* en *structures*, onderdelen van de taal. We hebben ook *bestandsverwerking* niet behandeld. Dit is technisch gezien geen onderdeel van de taal, maar wordt vaak gebruikt in programma's. Al deze concepten zullen in de volgende hoofdstukken worden uitgelegd.

[7] In dit boek gebruiken we vooral de K&R-stijl. We gebruiken ook vaak eenletterige variabelen. Dat is te verklaren omdat veel voorbeelden kort zijn en we de lezer niet willen vermoeien met lange variabelennamen. Het gebruik van de accolade-openen achter een taalconstructie levert een besparing op van één programmaregel.

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 weten welke variabelen in een programma gebruikt worden en hoe groot de variabelen zijn. De meeste moderne computersystemen kunnen 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 twee functies gezien: `main` en `square_root`. Functies worden in detail behandeld in hoofdstuk 5.

2.1 Variabelen

Een *variabele* is technisch gezien een plek in het geheugen van een computer waaraan een waarde kan worden toegekend. Een variabele moet in het C-programma kenbaar gemaakt worden. We noemen dat de *definitie*[1] van de variabele. Bij de definitie wordt het *type* van de variabele opgegeven. Zo geeft de definitie

```
int a;
```

aan dat in het programma de variabele a wordt gebruikt van het type `int`. Een `int` is een geheel 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 een onbeperkt aantal keer aanpassen, zie listing 2.1.

[1] Er is een verschil tussen definitie en *declaratie*. Zie hoofdstuk 11.

```
int a;
...
a = 2;
...
a = -3;
...
a = 5;
```

Listing 2.1: *Meerdere toekenningen aan een variabele.*

De naam van een variabele 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 *case sensitive* genoemd. Overigens accepteren de meeste 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;[2]
- Mag niet eerder gedefinieerd zijn;

Enkele goede voorbeelden:

```
Count  val23  loop_counter  ary2to4
```

Enkele foute voorbeelden:

```
_loop  50cent  int  static
```

De meeste C-compilers accepteren de underscore als eerste karakter. Dit wordt echter afgeraden omdat veel *systeemroutines* de underscore als eerste karakter gebruiken en dat kan problemen geven in de latere stadia van de compilatie (zie hoofdstuk 11).

Inhouden en bevatten

Als we spreken over "de variabele" dan bedoelen we van de inhoud, waarde of adres van de variabele. We schrijven de assignment als

```
a = 2;
```

en zeggen dan "de variabele a is 2 geworden". We bedoelen dan dat de inhoud van de variabele 2 is geworden. De term waarde is meestal een synoniem van inhoud. We spreken ook wel van toekennen, dus aan de variabele wordt een waarde toegekend.

[2] 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.

Bij de definitie van variabelen mogen ook gelijk waarden worden toegekend, op voorspraak dat de waarden bekend zijn of ter plaatse berekend kunnen worden. Een voorbeeld is hieronder te zien:

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

2.2 Datatypes

Een variabele is van een bepaald *datatype*. We hebben er al twee gezien, de `int` en de `double`. Een datatype heeft een bepaalde grootte. Zo bestaat een `int` *meestal* uit 32 bits. Er zijn echter ook systemen en C-compilers waar een `int` uit 16 bits bestaat. Dat een `int` uit 32 bits 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. De `int` is een *signed* datatype: zowel positieve als negatieve getallen en 0 zijn mogelijk.

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 gehele datatypes van diverse grootten. Deze zijn weergegeven in tabel 2.1. Gehele getallen worden opgeslagen in de two'scomplement representatie maar voor de gebruiker is dan niet van belang. Zie [16].

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

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

Naast de signed datatypes kent C ook een aantal *unsigned* datatypes (unsigned = zonder teken). We kunnen deze datatypes gebruiken middels de *qualifier* `unsigned`. De bereiken van deze datatypes zijn te zien in tabel 2.2 waarbij we opmerken dat het kleinste getal altijd 0 is.

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

type	bits	bereik
`unsigned char`	8	0 — 255
`unsigned short int`	16	0 — 65535
`unsigned int`	16	0 — 65535
	32	0 — 4294967295
`unsigned long int`	32	0 — 4294967295
	64	0 — 18446744073709551615
`unsigned long long int`	64	0 — 18446744073709551615

Een `char`[3] wordt gebruikt om een karakter op te slaan en is 8 bits groot. De interpretatie van het opgeslagen karakter is afhankelijk van de computer waarop het programma draait. Merk op dat een `char` zowel negatieve als positieve waarden kan bevatten. De meest gebruikte codering is de ASCII-code (zie bijlage A). Om het karakter 'A' op te slaan schrijven we:

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

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

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

Uiteraard moet de waarde passen en we merken terloops op dat negatieve waarden geen echte karakters voorstellen. Een `short int` is meestal 16 bits. Een `int` is 16 of 32 bits groot afhankelijk van de gebruikte C-compiler en de onderliggende hardware. Een `long int` is 32 bits of 64 bits. De C-standaard legt vast dat:

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

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

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

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

```
unsigned char uch;            /* range 0 to 255, a.k.a. byte */
unsigned short int usi;       /* range 0 to 65535 */
unsigned long long int ulli;  /* range 0 to 18446744073709551616 */
```

Naast gehele 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 schijven dus 3.14 en niet 3,14.

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

Voorbeelden van floating point variabelen:

```
float f;
double d;
long double ld;
```

[3] We gaan hier er stilzwijgend vanuit dat integers signed datatypes zijn. Dat is echter niet zo. Een integer kan ook unsigned zijn. Dit is afhankelijk van de gebruikte compiler *en* instellingen van de compiler. Het is echter wel mogelijk om tijdens compilatie de *signedness* in te stellen.

Tabel 2.3: *De floating point dataypes die beschikbaar zijn in C.*

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

[1]) Afhankelijk van de implementatie is een long double 80 of 128 bits.

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 cijfers.

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 bestaan er nog twee andere datatypes: `bool` (vanaf C99) kan de waarde `true` of `false` bevatten en `complex` (vanaf C99) kan een complex getal bevatten.

Een veel gebruikt datatype is de *pointer*. Een pointer is een variabele die het *adres* bevat van een (andere) variabele. Pointers worden in detail behandeld in hoofdstuk 8.

2.3 Constanten

Een gehele constante zoals `123` is van het type `int`. Door er een `L` of `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` of `u` aan het einde. Dat mag in combinatie met de `L`. Om een constante als long long aan te duiden wordt de constante gevolgd door `LL` of `ll` eventueel vooraf gegaan door een `U` of `u` om de constante als unsigned te typeren.

Een floating point-constante bestaat uit cijfers en 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` of `f` aan het einde staat, dan is de constante van het type `float`.

Gehele constanten kunnen als getal 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 0 t/m 7. Een constante die begin met de *prefix* `0x` of `0X` wordt gezien als een hexadecimaal getal. Een constante die begint met de cijfers 1 t/m 9 wordt gezien als een decimaal getal.

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

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

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 (apostrofe of *quote* genoemd). 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 invoeren als `\'`. Om de backslash zelf in te voeren gebruiken we `\\`.

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 escape sequences is vermeld in tabel 2.4.

Tabel 2.4: *Enkele escape sequences in C.*

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

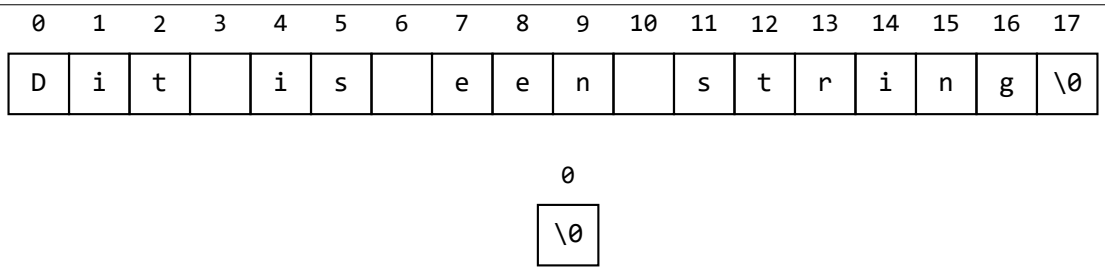
Andere karakters kunnen worden gevormd door de backslash, gevolgd door de letter x en één of twee hexadecimale cijfers. We kunnen de letter A dus ook schrijven als `'\x41'`.

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

```
"Dit is een string"  /* String with characters */
""                   /* 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. De `'\0'` stelt het nul-karakter voor.

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 standard library een groot aantal functies voor het bewerken van strings. We komen hierop terug in hoofdstuk 6.



Figuur 2.1: *Uitbeelding van twee strings.*

Bij de definitie van variabelen mogen ook gelijk waarden worden toegekend, op voorspraak dat de toegekende waarde op dat moment kan worden berekend. Dat betekent dat gebruikte variabelen bekend moeten zijn. Een aantal voorbeelden is hieronder te zien:

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

```
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. De toekenning zelf is ook een expressie:

```
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, alhoewel er met deze waarde niets wordt gedaan. 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 behandelen 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 om de prioriteiten te veranderen:

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

2.4.1 Rekenkundige operatoren

C kent vijf *binaire* rekenkundige operatoren: * voor vermenigvuldigen, / voor delen, % voor de modulus, + voor optellen en - voor aftrekken. 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. Deze operator mag alleen op gehele getallen gebruikt worden. Daarnaast kunnen + en - ook als *unaire operator* gebruikt worden. Een voorbeeld hiervan is

```
int a = -(b+c);
```

Let goed op de prioriteiten van de operatoren. De unaire operatoren gaan voor op vermenigvuldigen (*), delen (/) en modulus (%). De binaire optelling (+) en aftrekking (-) hebben en lagere prioriteit dan *, / en %. Haakjes kunnen de volgorde veranderen. Merk op dat de *associativiteit* van deze operatoren 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 gehele 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 gehele uitkomst hiervan is 0. Daarna wordt de uitkomst met 3 vermenigvuldigd. De uitkomst is nog steeds 0.

Let overigens goed op bij het delen met negatieve getallen. Het quotiënt wordt afgerond naar 0 toe. Het programma in listing 2.2 verduidelijkt dat. De uitvoer van het programma is te zien

```
#include <stdio.h>

int main(void) {

    printf(" 7/ 3 = %2d\n",  7/ 3);
    printf(" 7/-3 = %2d\n",  7/-3);
    printf("-7/ 3 = %2d\n", -7/ 3);
    printf("-7/-3 = %2d\n", -7/-3);
    return 0;
}
```

Listing 2.2: *Delen van positieve en negatieve getallen.*

in figuur 2.2. Als het resultaat van de deling een positief getal oplevert, wordt het resultaat naar beneden (naar de 0 toe) afgerond. Is het resultaat negatief, dan wordt het resultaat naar boven afgerond (ook naar de 0 toe).

Command Prompt

```
 7/ 3 =  2
 7/-3 = -2
-7/ 3 = -2
-7/-3 =  2
```

Figuur 2.2: *Uitvoer van het deelprogramma.*

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 en unaire 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 2.3.

Let op het verschil tussen == en =. De eerste is een *test* of twee expressies gelijk zijn aan elkaar. De tweede is een *toekenning*. Dus `a == b` test of a gelijk is aan b. Bij `a = b` wordt b toegekend aan a.

```c
int main(void)
{
    int a, b;

    printf("Geef twee gehele getallen a en b: ");
    scanf("%d %d", &a, &b);

    if (a < b)
    {
        printf("a is kleiner dan b\n");
    }
    else
    {
        printf("a is groter dan of gelijk aan b\n")
    }

    return 0;
}
```

Listing 2.3: *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

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

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

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

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

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

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

```c
if (b != 0 && a/b > 5) {
    ...
}
```

geen foutmelding oplevert als `b` gelijk aan 0 is.

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 2.4. Overigens leveren de relationele operatoren altijd 0 (niet waar) of 1 (waar) op. Zie paragraaf 2.4.2.

```c
int main(void)
{
    int a;

    printf("Geef een getal tussen 5 en 10: ");
    scanf("%d", &a);

    if ( !(a<5 || a>10) )
    {
        /* do something ... */
    }

    return 0;
}
```

Listing 2.4: *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:

```c
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 gehele variabelen, constanten of expressies. Deze zijn:

&	AND
|	OR
^	EXOR
<<	naar links schuiven
>>	naar rechts schuiven
~	bits inverteren

Opmerking: De bitsgewijze operatoren werken op bits van gehele getallen. Dit is iets anders dan de relationele operatoren. Hierbij worden relaties tussen twee expressies gevormd en daar kan waar (1) of onwaar (0) uit volgen.

De *waarheidstabellen* van de AND-, OR- en EXOR-functies zijn gegeven in tabel 2.5. 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 (een 0 wordt een 1 en een 1 wordt een 0).

Tabel 2.5: *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

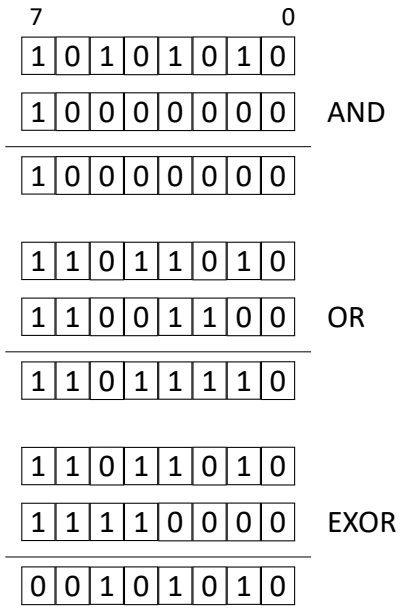
```
1 << 4
```

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. Dit wordt ook wel *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 2.3 zijn drie voorbeelden te zien van bitmanupilaties met AND, OR en EXOR. Hiervoor zijn de gegevens uit tabel 2.5 gebruikt.



Figuur 2.3: *Drie voorbeelden van AND, OR en EXOR.*

Een bekend voorbeeld is het testen of een geheel getal even of oneven is. Een even geheel getal is exact deelbaar door 2. We kunnen dit realiseren met de modulo-operator %:

```c
#include <stdio.h>

int main(void)
{
    int getal;

    printf("Geef een geheel getal: ");
    scanf("%d", &getal);

    if (getal & 1 == 0)
    {
        printf("Getal is even\n");
    }
    else
    {
        printf("Getal is oneven\n");
    }

    return 0;
}
```

Listing 2.5: *Testen of een geheel getal even is.*

```
if (getal % 2 == 0) { ... }
```

We kunnen dit ook realiseren het de &-operator, zie het programma in listing 2.5. De expressie `getal & 1` in regel 10 zorgt ervoor dat alle bits op 0 gezet worden, behalve de minst significante bit. Deze bit geeft aan of het getal even (0) of oneven (1) is.

2.5 Datatypeconversie

Het omzetten of converteren van het ene enkelvoudige datatype naar het andere enkelvoudige 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 cast is het omrekenen van een temperatuur in graden Celsius naar graden Fahrenheit.

Stel we hebben de temperatuur in graden Celsius opgeslagen in de `double` variabele `c` en de temperatuur in Fahrenheit willen berekenen (ook een `double`). Dat kan met het onderstaande statement:

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

We hebben hier de integer 9 gecast naar een `double` want anders gaat bij de deling 9/5 informatie verloren; de *gehele* uitkomst van de deling `9/5` is immers `1`. De constante 32 hoeft niet gecast te worden omdat eerder al de getallen en variabelen naar een `double` gecast zijn. Natuurlijk hadden we ook kunnen schrijven:

```
f = 9.0/5.0 * c + 32.0;
```

2.6 Overflow

Overflow is de situatie wanneer 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 er dus 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 11111111_2). Tellen we daar 1 bij op dat is het resultaat 256 (binair 100000000_2) maar dat kan niet in de variabele worden opgeslagen. Het resultaat is dat de overvloedige bits (de leidende 1 in dit geval) gewoonweg worden geschrapt en dat de uitkomst dus 00000000_2 (0) is. Dit principe geldt ook bij de `int` en `long int`.

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 $+\infty$[4]. 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 zorgen of tot problemen leiden. We weten nu niet of een `int` 16 of 32 bits is. Dit heeft geleid

[4] Volgens de wiskunde is het resultaat van een deling door 0 onbepaald. De specificatie definieert echter de deling als $\pm$oneindig.

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 (−128 t/m +127). Naast de integer met teken zijn er ook 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 er zeker van 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 2.6 en 2.7 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 2.6.

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

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

Tabel 2.7: *De vaste-lengte dataypes die beschikbaar zijn in C (unsigned).*

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

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 gehele constanten in de vorm van

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

De eerste naam binnen de accolades krijgt impliciet 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 in welk integer datatype de enumeratie past.

```c
/* Load new datatypes */
#include <stdint.h>

int main(void)
{
    uint8_t byte = 0;

    /* ... */

    while (byte<128)
    {
        byte = byte + 1;
    }
}
```

Listing 2.6: *Voorbeeld van het gebruik van vaste-lengte datatypes.*

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 in principe alleen tijdens *compile-time* van het C-programma gebeuren. Tijdens het uitvoeren van het programma zijn alle grootten berekend[5]. Er is één restrictie in het gebruik van `sizeof`: bij het gebruik met een datatype *moeten* haakjes gebruikt worden. Zie listing 2.7. Deze operator heeft een hoge prioriteit. We zullen `sizeof` nader behandelen in hoofdstuk 6.

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

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

    return 0;
}
```

Listing 2.7: *Gebruik van `sizeof`.*

[5] Er is één uitzondering. Vanaf C99 kan `sizeof` ook gebruikt worden bij arrays van variabele lengte, waarbij de lengte bij definitie variabel is, bijvoorbeeld bij `int a[b]` en b als parameter van een functie dient.

2.9.2 Verhogen of verlagen met 1

Een toekenning als

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

kan geschreven worden als

```
i++;
```

of als

```
++i;
```

Dit wordt de *increment operator* genoemd. 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 gelijk is aan 5 dan zorgt

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

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

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

zijn k en `i` gelijk aan 6.

Op vergelijkbare wijze werkt de *decrement operator* `--`. De expressie

```
k = --i;
```

verlaagt eerst variabele `i` en kent dan de waarde van `i` toe aan k. We zullen deze operatoren tegenkomen in de hoofdstuk 3 en 6. Deze operatoren hebben een hoge prioriteit. We komen deze operatoren vooral tegen bij herhalingsstatements.

Merk op dat de prefix- en postfix operatoren *ook* gebruikt mogen worden met floating point-getallen, maar in de regel gebruiken we een expliciete optelling: `x = x + 1.0;`.

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);
```

Tabel 2.8: *Toekenningsoperatoren.*

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

Zo'n beetje alle gangbare operatoren kunnen worden gebruikt. Een lijst is te vinden in tabel 2.8. 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 2.8.

Het eerste argument (`%d`) is het aantal documenten dat verwerkt is (`nrdocs`). Het tweede argument 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 als één statement worden gezien. Een voorbeeld van de komma-operator is

```
a = 3, b = 5;
```

```c
#include <stdio.h>

int main(void)
{
    int nrdocs = 0;

    /* ... */

    /*                  +-------------- nrdcos      */
    /*                  |         +---- "" or "en"  */
    /*                  v         v                 */
    printf("In totaal %d document%s verwerkt\n", nrdocs,
                                 (nrdocs == 1) ? "" : "en"));
    return 0;
}
```

Listing 2.8: *Het afdrukken van een meervoudsvorm.*

Bij deze expressie wordt a gelijk aan 3 en b gelijk aan 5. Het resultaat van de gehele expressie is de rechter expressie (`b = 5`). De komma-operator wordt nogal eens gebruikt in het `for`-statement. We behandelen de komma-operator nogmaals in hoofdstuk 3.

Let erop dat de komma's die argumenten in functie-aanroepen en variabelen in definities scheiden *geen* komma-operatoren zijn.

2.9.6 Nog enkele operatoren

De volgende operatoren worden verderop in het boek behandeld:

`[]` `->` `.` `*` (*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 2.9. Let hierbij goed op de *associativiteit* van de operatoren. Die is van links naar rechts óf van rechts naar links. Dat betekent bijvoorbeeld dat `1 / 3 * 3` wordt gelezen als `(1 / 3) * 3`, dus de associatie is van links naar rechts. De deling gaat dus voor op de vermenigvuldiging. Merk op dat de toekenningsoperator = ook in de lijst voorkomt, de associatie is van rechts naar links en dat deze operator een lage prioriteit heeft.

Een in C-programma's gebruikelijke constructie voor het inlezen van een karakter van het toetsenbord én gelijk testen of het een newline-karakter betreft is:

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

Hier gebeuren eigenlijk drie dingen. Eerst wordt de *functie* `getchar` aangeroepen om een karakter van het toetsenbord te lezen, de haakjes `()` direct achter `getchar` hebben de hoogste prioriteit. Daarna wordt dit karakter toegekend aan variabele `ch`. De haakjes zorgen voor de

Tabel 2.9: *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

juiste prioriteit. De uitkomst van de toekenning is de gehele waarde van de toekenning en dat is het ingelezen karakter. Daarna wordt getest of het ingelezen karakter ongelijk is aan het newline-karakter (`\n`).

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

2.11 Volgorde van uitrekenen van expressies

De volgorde van het uitrekenen van expressies ligt niet vast, behalve bij de operatoren `&&`, `||`, `?:` en `,`. Dit heeft voornamelijk consequenties bij het gebruik van argumenten in functies, zoals te zien is in de onderstaande functie-aanroep:

```
printf("{%d %d\n", ++n, --n);
```

De volgorde van het uitrekenen van `++n` en `--n` ligt niet vast. Dit kan verschillende resultaten opleveren bij gebruik van verschillende compilers. De oplossing is het gebruik van extra variabelen:

```
int i = ++n;
int j = --n;
printf("%d %d\n", i, j);
```

3

Beslissen en herhalen

De statements in een C-programma staan onder elkaar geschreven. Na vertaling door de C-compiler worden de statement na elkaar uitgevoerd zoals ze in het C-programma voorkomen. We noemen dat *sequentiële verwerking*. Soms willen we echter dat een reeks van statements wordt uitgevoerd aan de hand van een bepaalde voorwaarde. C biedt een aantal mogelijkheden om dat te beschrijven.

Het is ook mogelijk om een aantal statements herhaaldelijk uit te voeren aan de hand van een bepaalde voorwaarde. C biedt een drietal herhalingsstatements, elk met zijn eigen opzet. Het herhalen zorgt ervoor dat de sequentiële verwerking wordt onderbroken om de serie van statements opnieuw uit te voeren. Binnen een herhaling worden de statements sequentieel uitgevoerd, maar het is natuurlijk mogelijk binnen een herhaling te beslissen of weer te herhalen. Het herhalen van statements wordt een *lus* genoemd (Engels: loop).

3.1 Blok

Een *blok* of *blokstructuur*, in C-terminologie ook wel een *compound statement* genoemd, is een reeks van statements tussen accolades. Door de C-compiler wordt een blok syntactisch als één statement gezien. In listing 3.1 is te zien dat binnen het blok een lokale variabele `i` wordt gedefinieerd. Alleen binnen het blok is deze variabele te gebruiken. Buiten het blok bestaat de variabele niet.

```
// i does not exist
{
    int i = 1 << 2;

    printf("i = %d\n", i);
}
// i does not exist
```

Listing 3.1: *Een blok met een lokale variabele.*

3.2 Beslissen met het `if`-statement

Met behulp van het `if`-statement kunnen we een reeks statements uitvoeren aan de hand van een expressie die waar is (Engels: true). In listing 3.2 is de algemene gedaante van het `if`-statement te zien.

```
if (expressie)
{
    statements
}
```

Listing 3.2: *Algemene opzet van het `if`-statement.*

Tussen de haakjes moet een expressie staan. Als de expressie waar is, dan worden de statements die bij de `if` horen uitgevoerd. Een expressie is waar als de uitkomst *ongelijk* aan 0 is. Een expressie is onwaar als de uitkomst gelijk aan 0 is. Voorbeelden van expressies zijn (zonder de haakjes van de `if`):

```
hoogte < 10
(lengte > 2) && (lengte < 80)
(getal < 1) || (getal > 10)
(year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

Let erop dat een expressie met relationele operator gelijk aan 1 is als de expressie waar is en gelijk aan 0 is als de expressie onwaar is. Dus

```
getal == 25
```

levert een 1 op als de expressie waar is, anders levert de vergelijking een 0 op. Een veel gemaakte fout bij beginnende programmeurs is om de == te schrijven met =. Dit is een toekenning, geen vergelijking.

Een veel gebruikte operator is de negatieoperator `!`. We kunnen deze operator gebruiken zoals te zien is in listing 3.3. De functie `is_date_valid` geeft een 1 als er een geldige datum wordt meegegeven, anders geeft de functie een 0. De negatie-operator draait de waarde van een expressie om, dus een waarde gelijk aan 0 wordt een 1 en een waarde *ongelijk* aan 0 wordt een 1.

```
// ...

valid = is_date_valid("2020/03/21");

if (!valid)     // if date is invalid
{
    printf("Ongeldige datum\n");
}

// ...
```

Listing 3.3: *Gebruik van de negatie-operator.*

3.3 Beslissen met het `if else`-statement

Een `if`-statement mag optioneel een `else`-gedeelte bevatten, zie listing 3.4. Hier worden de *statements*$_1$ direct onder de `if` uitgevoerd als de expressie waar is en de *statements*$_2$ onder de `else` worden uitgevoerd als de expressie niet waar is.

```
if (expressie)
{
    statements1
}
else
{
    statements2
}
```

Listing 3.4: *Algemene opzet van het `if else`-statement.*

Een voorbeeld van het gebruik van `if else` is te zien in listing 3.5. In dit voorbeeld bepalen we of een vak, bestaande uit twee deeltoetsen, is behaald. Daarvoor geldt dat het gemiddelde van de twee vakken groter is dan of gelijk is aan 5,5 en dat voor beide vakken het cijfer groter is dan of gelijk is aan 4,0. We gebruiken voor het gemak gehele getallen. In de regels 7 t/m 9 vragen we de gebruiker om twee cijfers in te voeren. Daarna bepalen we in regel 12 of de som groter is dan of gelijk is aan 11 (2×5,5) én we bepalen of beide cijfers groter zijn dan of gelijk zijn aan 4,0. Als dat zo is, dan drukken we af dat het vak is behaald. Is de voorwaarde niet waar, dan drukken we af dat het vak niet is behaald.

```
#include <stdio.h>

int main(void)
{
    int  toets1, toets2;

    printf("Geef cijfer deeltoets 1: ");
    scanf("%d", &toets1);
    printf("Geef cijfer deeltoets 1: ");
    scanf("%d", &toets2);

    if (toets1 + toets2 >= 11 && toets1 >= 4 && toets2 >= 4)
    {
        printf("Vak is behaald\n");
    }
    else
    {
        printf("Vak is niet behaald\n");
    }

    return 0;
}
```

Listing 3.5: *Voorbeeld van een `if-else`-statement.*

3.4 Beslissen met het switch-statement

Het switch-statement wordt gebruikt om een expressie te testen op meerdere constante waarden. De expressie na switch is meestal een variabele maar er mag ook een expressie staan die een constante waarde oplevert. Binnen de switch wordt de expressie vergeleken met een of meerdere constante expressies met behulp van het case-statement. Is de expressie gelijk aan een constante expressie, dan wordt het statement dat bij de case hoort uitgevoerd. Als geen van de constante expressies gelijk is aan de expressie na de switch, wordt het statement dat bij default hoort uitgevoerd. Het default-statement mag achterwege blijven. De algemene opzet van het switch-statement is te zien in listing 3.6. Het statement achter case hoeft niet tussen accolades gezet te worden en er mogen meerdere statements bij een case-statement geplaatst worden.

```
1  switch (expressie)
2  {
3      case constante-expressie1 : statement1
4                                  break;
5      case constante-expressie2 : statement2
6                                  break;
7      case constante-expressie3 : statement3
8                                  break;
9      // ...
10     default: statementd
11              break;
12 }
```

Listing 3.6: *Opzet van het switch-statment.*

In de algemene opzet hebben we het break-statement gebruikt. Dit break-statement zorgt ervoor dat het switch-statement direct wordt verlaten.

De waarden bij meerdere case-statement kunnen worden samengevoegd, zodat dezelfde statements worden uitgevoerd bij meerdere mogelijkheden. We noemen dit een *fall through*. De algemene opzet van deze vorm is te zien in listing 3.7.

```
1  switch (expressie)
2  {
3      case constante-expressie1:
4      case constante-expressie2:
5      case constante-expressie3: statement
6                                 break;
7      // ...
8      default: statementd
9               break;
10 }
```

Listing 3.7: *Opzet van het switch-statment met fall through.*

Let erop dat fall through met zorg moet worden toegepast. Een break die vergeten wordt, levert automatisch een fall through op.

In listing 3.8 is een compleet programma te zien dat een afstand, opgegeven in millimeter, centimeter of decimeter, wordt omgerekend naar meter. De waarde en de eenheid worden gescheiden door een spatie. We lezen in regel 10 de waarde en de eenheid in. De *prefix* (`m`, `c` of `d`) van de eenheid komt in de karaktervariabele `prefix`. We testen deze variabele met behulp van een `switch`-statement in de regels 15 t/m 26. Bij elke `case` wordt getest op een mogelijke prefix en wordt de variabele `prefixvalue` op de juiste waarde gezet voor een

```
1  #include <stdio.h>
2
3  #pragma warning(disable : 4996)
4
5  int main(void)
6  {
7      double getal, prefixvalue;
8      char space, prefix, unit;
9
10     printf("Geef een afstand in mm, cm of dm (bv. 2.6 dm): ");
11     scanf("%lf%c%c%c", &getal, &space, &prefix, &unit);
12
13     if (space == ' ' && unit == 'm')
14     {
15         switch (prefix)
16         {
17         case 'm':
18             prefixvalue = 0.001;
19             break;
20         case 'c':
21             prefixvalue = 0.01;
22             break;
23         case 'd':
24             prefixvalue = 0.1;
25             break;
26         }
27     }
28
29     if (prefixvalue == 0)
30     {
31         printf("De invoer is ongeldig!\n");
32     }
33     else
34     {
35         printf("Dit is %f meter\n", getal * prefixvalue);
36     }
37
38     return 0;
39 }
```

Listing 3.8: *Voorbeeld van het `switch`-statement.*

latere berekening. Als de prefix een geldige waarde heeft, wordt in regel 35 de geconverteerde waarde afgedrukt.

Een voorbeeld van een fall through is gegeven in listing 3.9. Het programma leest een geheel toetscijfer in tussen 0 en 10. Daar wordt met behulp van een `switch`-statement het ingevoerde cijfer omgezet naar de bijbehorende Amerikaanse toetsletter (A, B, C, D en F).

```
#include <stdio.h>

int main(void)
{

    int cijfer;
    char letter;

    do
    {
        printf("Geef je cijfer: ");
        scanf("%d", &cijfer);
    }
    while (cijfer < 0 || cijfer > 10);

    switch (cijfer)
    {
        case 10:
        case 9:
        case 8: letter = 'A'; break;
        case 7: letter = 'B'; break;
        case 6: letter = 'C'; break;
        case 5: letter = 'D'; break;
        default: letter = 'F'; break;
    }

    printf("Dit komt overeen met een %c.\n", letter);

    return 0;
}
```

Listing 3.9: *Voorbeeld van het gebruik van het `switch`-statment met fall through.*

In het `switch`-statement wordt `cijfer` vergeleken met verschillende waarden. Als `cijfer` de waarde 8, 9 of 10 heeft, wordt variabele `letter` op `'A'` gezet. Het `break`-statement in regel 14 zorgt ervoor dat het `switch`-statement direct wordt verlaten. Als we die zouden weglaten, worden de statements die horen bij de volgende `case` uitgevoerd, dus wordt `'B'` aan `letter` toegekend. In regel 22 worden de statements uitgevoerd als geen van de constante expressies achter de `case`-statement gelijk is aan `cijfer`. Het `break`-statement in regel 22 is logisch gezien overbodig, maar het is *good practice* om deze wel op te nemen. De volgorde van `cases` en `default` is namelijk willekeurig, dus na `default` zou nog een `case` mogen komen. Het is wederom *good practice* om alle `cases` voor de `default` te plaatsen.

3.5 Herhalen met het `while` statement

Een `while`-statement wordt gebruikt als een reeks statements 0 of meer keer moet worden uitgevoerd. De opzet van het `while`-statement is te zien in listing 3.10. Als *expressie* waar is (een uitkomst ongelijk aan 0) dan worden de statements binnen de accolades uitgevoerd. Nadat de statements zijn uitgevoerd, wordt weer opnieuw begonnen met het uitrekenen van *expressie*. Is *expressie* (weer) waar, dan worden de statements nogmaals uitgevoerd. Is *expressie* niet waar (de uitkomst is gelijk aan 0) dan wordt gesprongen naar het statement na de accolade-sluiten. Het kan zijn dat *expressie* bij de allereerste keer niet waar is. Dan worden de statements binnen de `while` niet uitgevoerd.

```
while (expressie)
{
    statement
}
```

Listing 3.10: *Opzet `while`-statement.*

We leggen de werking van het `while`-statement uit aan de hand van een voorbeeld, te zien in listing 3.11. In regel 8 wordt gevraagd om een positief getal in te voeren. In regel 10 wordt het `while`-statement uitgevoerd door de expressie `getal <= 0` uit te rekenen. Is de expressie niet waar dan wordt direct naar regel 16 gesprongen en de statement binnen de `while` worden niet uitgevoerd. Is het getal kleiner dan of gelijk aan 0 dan worden de statements binnen de `while` uitgevoerd. Daar wordt weer gevraagd om een positief getal in te voeren. Vervolgens wordt de expressie opnieuw uitgerekend. Zolang de expressie niet waar is worden de statements binnen de `while` uitgevoerd.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int getal;
6
7      printf("Geef een positief getal: ");
8      scanf("%d", &getal);
9
10     while (getal <= 0)
11     {
12         printf("Helaas! Geef een positief getal: ");
13         scanf("%d", &getal);
14     }
15
16     printf("Het ingevoerde getal = %d\n", getal);
17     return 0;
18 }
```

Listing 3.11: *Voorbeeld van een `while`-statement..*

3.6 Herhalen met het for statement

Het for-statement gebruiken we als het aantal herhalingen bekend en *eindig* is. Het for-statement heeft de gedaante zoals te zien in in listing 3.12.

Vóór de lus wordt *expressie*$_1$ uitgevoerd. Merk op dat de expressie impliciet wordt omgezet in een statement. Bij het herhalen van de lus wordt *expressie*$_2$ uitgevoerd. Als de expressie waar is worden de statements binnen de accolades uitgevoerd. Voor het einde van de lus, net voor de accolade-sluiten, wordt *expressie*$_3$ uitgevoerd. Ook deze expressie wordt impliciet omgezet in een statement. Let erop dat *expressie*$_3$ ná *statement* in regel 3 wordt uitgevoerd. Verder mag vanaf C99 *expressie*$_1$ een definitie van een variabele bevatten. De variabele is dan alleen te gebruiken binnen het for-statement. Let erop dat *expressie*$_1$, *expressie*$_2$ en *expressie*$_3$ van elkaar gescheiden zijn met een punt-komma.

```
1 for (expressie1; expressie2; expressie3)
2 {
3     statement
4 }
```

Listing 3.12: *Opzet van het for-statement.*

Normaal gesproken wordt in *expressie*$_1$ een variabele op een beginwaarde gezet. De *expressie*$_2$ wordt berekend zoals dat in het while-statement ook gebeurt, en *expressie*$_3$ wordt normaal gesproken gebruikt om een variabele aan te passen voor de volgende iteratie. Meestal betekent dit dat de variabele wordt verhoogd of verlaagd.

Het for-statement (zonder een continue-statement, zie paragraaf 3.8) uit listing 3.12 is *semantisch equivalent*[1] aan:

```
1 expressie1;
2 while (expressie2)
3 {
4     statement
5     expressie3;
6 }
```

Listing 3.13: *while-statement als for-statement.*

We kunnen in het for-statement *expressie*$_1$, *expressie*$_2$ of *expressie*$_3$ achterwege laten. Hieronder is de opzet te zien van een eeuwigdurende lus.

```
for (;;) { ... }    /* do forever */
```

Stel dat we de tafels van 1 t/m 10 willen afdrukken. We maken dan gebruikt van twee, in elkaar verweven, for-statements. Dit is te zien in listing 3.14. We concentreren ons op de opzet van beide for-statements. Er zijn diverse printf-statements te zien die de gegevens op een nette manier afdrukken. Zo betekent de 4 in printf("%4d", i * j) de veldbreedte van 4 cijfers van het af te drukken getal, zodat de kolommen netjes onder elkaar komen te staan.

[1] Semantisch equivalent wil zeggen dat de werkingen gelijk zijn aan elkaar. De twee opzetten zijn *niet* syntactisch equivalent. Ze hebben immers een verschillende *syntax*.

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

Listing 3.14: *Programma voor het afdrukken van de tafels.*

De buitenste `for`-statement begint op regel 8, eindigt op regel 16 en definieert variabele `i`. De variabele loopt van 1 tot 11, dus 11 zelf doet *niet* mee. Binnen deze lus gebruiken we een tweede `for`-statement die begint op regel 11 en eindigt op regel 14. Dit keer gebruiken we variabele `j` die in regel 11 wordt gedefinieerd. Ook deze variabele loopt van 1 tot 11 (dus 11 doet niet mee). In regel 13 drukken we het product af van `i` en `j`. De uitvoer van dit programma is te zien in figuur 3.1. Merk op dat op diverse punten in het programma gebruik wordt gemaakt om de gegevens netjes uitgelijnd af te drukken.

Command Prompt

```
  |   1   2   3   4   5   6   7   8   9  10
--+----------------------------------------
 1|   1   2   3   4   5   6   7   8   9  10
 2|   2   4   6   8  10  12  14  16  18  20
 3|   3   6   9  12  15  18  21  24  27  30
 4|   4   8  12  16  20  24  28  32  36  40
 5|   5  10  15  20  25  30  35  40  45  50
 6|   6  12  18  24  30  36  42  48  54  60
 7|   7  14  21  28  35  42  49  56  63  70
 8|   8  16  24  32  40  48  56  64  72  80
 9|   9  18  27  36  45  54  63  72  81  90
10|  10  20  30  40  50  60  70  80  90 100
```

Figuur 3.1: *Uitvoer van het programma in listing 3.14.*

3.7 Herhalen met het do `while` statement

We gebruiken het do `while`-statement als we statements 1 of meer keer willen uitvoeren. Merk op dat de lus dus minstens één keer wordt uitgevoerd. Voor een opzet, zie listing 3.15. Eerst wordt *statement* uitgevoerd en daarna wordt *expressie* uitgerekend. Als *expressie* waar is, wordt de lus nog een keer uitgevoerd, anders wordt de lus verlaten.

```
do
{
    statement
}
while (expressie);
```

Listing 3.15: *Opzet do `while`-statement.*

We kunnen het programma in listing 3.11 ook schrijven met behulp van een do `while`-statement. Zie listing 3.16. De lus wordt minstens één keer uitgevoerd. Als na invoer van getal blijkt dat de expressie `getal <= 0` waar is, wordt de lus nog een keer uitgevoerd, net zolang totdat de expressie niet waar is.

```
#include <stdio.h>

int main(void)
{
    int getal;

    do
    {
        printf("Geef een positief getal: ");
        scanf("%d", &getal);
    }
    while (getal <= 0);

    printf("Het ingevoerde getal = %d\n", getal);
    return 0;
}
```

Listing 3.16: *Gebruik van een do `while`-statement voor het inlezen van een positief getal.*

3.8 Extra lusbewerkingen: de `break` en `continue`-statements

We kunnen een lus voortijdig verlaten met een `break`-statement. Dit is te zien in listing 3.17. De lus wordt vormgegeven door een eeuwig durend `while`-statement (regel 10). We vragen de gebruiker om een positief getal of 0 in te voeren. Als het getal gelijk is aan 0, wordt de lus verlaten met het `break`-statement in regel 16. Alleen de lus waar het `break`-statement bijhoort kan worden verlaten. Als we een lus binnen een lus gebruiken kan dus alleen de binnenste lus worden verlaten.

Met het continue-statement kunnen we binnen een lus springen naar de volgende iteratie van de lus. Bij het while-statement wordt dus gesprongen naar het begin van de lus, bij het do while-statement wordt gesprongen naar het einde van de lus. In beide gevallen wordt de expressie die bij de lus hoort opnieuw uitgerekend.

Bij het for-statement werkt het iets anders. Daarvoor moeten we listing 3.12 nog eens bekijken. Aan het einde van het for-statement wordt *expressie*$_3$ (omgezet in een statement) uitgevoerd, waarin normaal gesproken een variabele wordt aangepast (verhogen of verlagen). Een continue zorgt ervoor dat naar dit statement wordt gesprongen.

```
#include <stdio.h>

#pragma warning(disable : 4996)

int main(void)
{
    int som = 0;
    int i;

    while (1)
    {
        printf("Geef positief getal of 0 om te stoppen: ");
        scanf("%d", &i);
        if (i == 0)
        {
            break;
        }
        if (i < 0)
        {
            continue;
        }
        som = som + i;
    }
    printf("De som is %d\n", som);
}
```

Listing 3.17: *Gebruik van het break- en continue-statement om een lus te verlaten of opnieuw uit te voeren.*

Overigens kunnen we het programma in listing 3.17 geheel herschrijven zonder break en continue. Zie listing 3.18.

We gebruiken break en continue als de code binnen de lus complex is met vele in elkaar verweven lussen en testen.

3.9 Increment- en decrement-operatoren

Bij het opzetten van een for-statement moet de lusvariabele vaak met één verlaagd of verhoogd worden. We kunnen dat doen middels:

```c
#include <stdio.h>

#pragma warning(disable : 4996)

int main(void)
{
    int som = 0;
    int i;

    do
    {
        printf("Geef positief getal of 0 om te stoppen: ");
        scanf("%d", &i);
        if (i > 0)
        {
            som = som + i;
        }
    } while (i != 0);
    printf("De som is %d\n", som);
}
```

Listing 3.18: *Sommeren van positieve getallen.*

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

maar het is in C gebruikelijk om

```
i++;
```

te gebruiken. In de context van het `for`-statement is er geen verschil in werking; beide mogelijkheden zijn correct. Stel dat we een aantal statements 10 keer willen uitvoeren. Dan kunnen we het `for`-statement opzetten zoals te zien is in listing 3.19.

```c
    /* ... */
    for (int i = 0; i < 10; i++)
    {
        /* ... */
    }
    /* ... */
```

Listing 3.19: *Gebruik van de increment-operator in een `for`-statement.*

3.10 Komma-operator

De komma-operator scheidt meerdere expressies van elkaar waarbij de waarde van de meest rechter expressie als antwoord dient. Dus het statement

```
a = ++b, --c, j;
```

zorgt ervoor dat b wordt verhoogd, c wordt verlaagd en dat a gelijk is aan j. De komma-operator bewijst vooral zijn nut in het for-statement waar meerdere variabelen tegelijk moeten worden aangepast. Zie listing 3.20.

```
    int i, j;

    /* ... */

    for (i=0, j=9; i<10; i++, j--) {
        /* ... */
    }
```

Listing 3.20: *Gebruik van de komma-operator.*

Een bekende methode om de tafel van 9 uit het hoofd te leren is dat eerst de getallen 0 t/m 9 van boven naar beneden worden geschreven en daarna de getallen van 9 t/m 0 er achter worden geschreven. Een voorbeeld is gegeven is listing 3.21.

```
#include <stdio.h>

int main(void) {

    for (int i = 0, j = 9; i < 10; i++, j--) {
        printf("%d%d\n", i, j);
    }

    return 0;
}
```

Listing 3.21: *Afdrukken van de tafel van 9.*

Hier worden twee variabelen gebruikt, i en j, die alleen binnen de lus bekend zijn. De uitvoer van het programma is te zien in figuur 3.2.

Command Prompt

```
09
18
27
36
45
54
63
72
81
90
```

Figuur 3.2: *Uitvoer van de tafel van 9.*

3.11 Het goto-statement

Eerst even een opmerking:

> Goto's are the root of all evil.

Er is geen enkele reden om het goto-statement te gebruiken. Het is altijd mogelijk om code te schrijven zonder het goto-statement. In dit boek maken we er ook geen gebruik van. Soms is het echter handig om er gebruik van te maken, bijvoorbeeld als een foutconditie op meerdere plekken in de code kan voorkomen of als we uit een, wat in het Engels *deeply nested structure* genoemd wordt, willen breken, bijvoorbeeld een lus binnen een lus. Een voorbeeld is te zien in listing 3.22.

```
for ( ... )
{
    for ( ... )
    {
        if (panic)
        {
            goto error;
        }
    }
}
error:
    // clean up the mess
```

Listing 3.22: *Gebruik van het goto-statement.*

3.12 Op de valreep, het verschil tussen = en ==

Een van de meest voorkomende fouten in een C-programma is het verschil tussen toekennen aan en vergelijken op is gelijk aan. De expressie:

```
getal = 25
```

kent de waarde 25 aan variabele `getal` toe. De expressie:

```
getal == 25
```

bepaalt of `getal` gelijk is aan 25. Een veel gemaakte fout is

```
if (getal = 25)  /* wrong */
{
    // doe iets
}
```

Hier wordt de waarde 25 toegekend aan `getal` én de uitkomst is ook 25, dus ongelijk aan 0, dus waar en de statements worden uitgevoerd. Veel compilers geven een waarschuwing als zo'n toekenning gebruikt wordt.

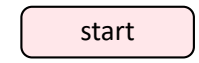
4

Flowcharts

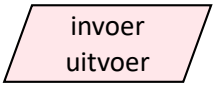
Een *flowchart* (Nederlands: stroomschema) is een grafische weergave van een (deel van een) computerprogramma te documenteren. Flowcharts gebruiken rechthoeken, ovalen, ruiten en mogelijk andere vormen om het programma stap voor stap weer te gegeven, samen met verbindingspijlen die de stroom en volgorde 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".

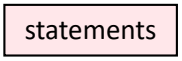
De gebruikte symbolen in een flowchart staan hieronder weergegeven:



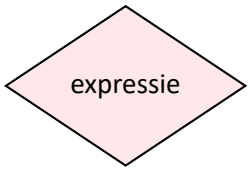
Dit symbool geeft het begin of het einde van de flowchart aan. Naast "start" en "stop" kunnen ook de termen "begin" en "einde" gebruikt worden.



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 $a < 25$”.

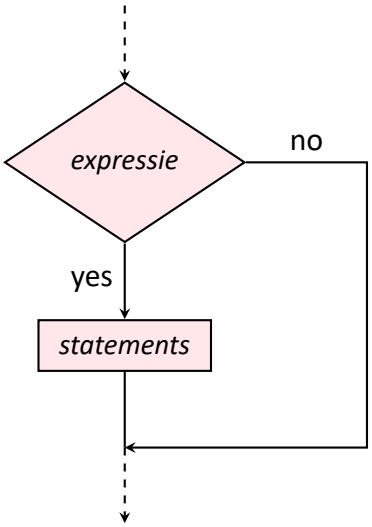
4.1 `if`-statement

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

```
if (expressie) {
    statements
}
```

Listing 4.1: *`if`-statement in C*

De flowchart is gegeven in figuur 4.1.



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

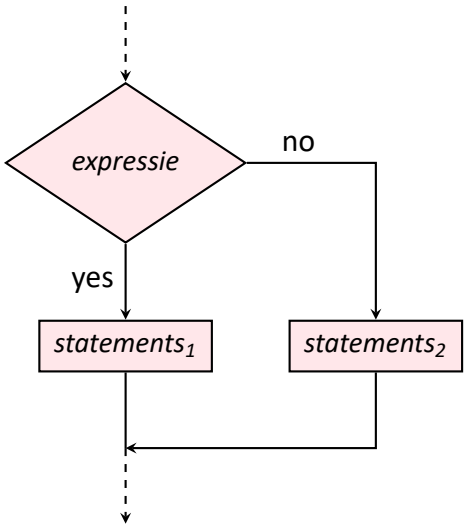
4.2 `if-else`-statement

Een `if-else`-statement van de vorm

```
if (expressie) {
    statements_1
else {
    statements_2
}
```

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:

```
while (expressie) {
    statements
}
```

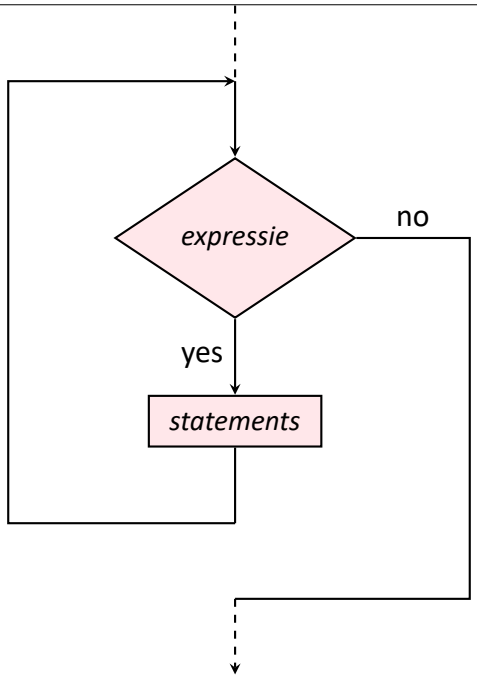
Listing 4.3: *while-lus in C.*

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

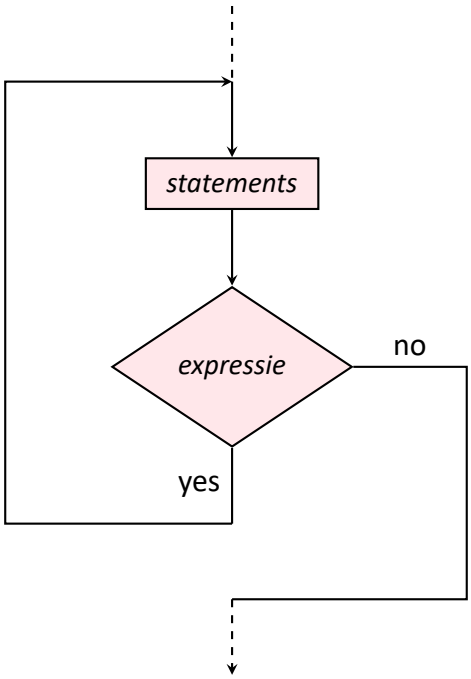
```
do {
    statements
} while (expressie};
```

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

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



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

4.4 `for`-lus

Een `for`-lus van de gedaante (zonder `break`- en `continue`-statemens):

```
for (expressie1; expressie2; expressie3) {
    statements
}
```

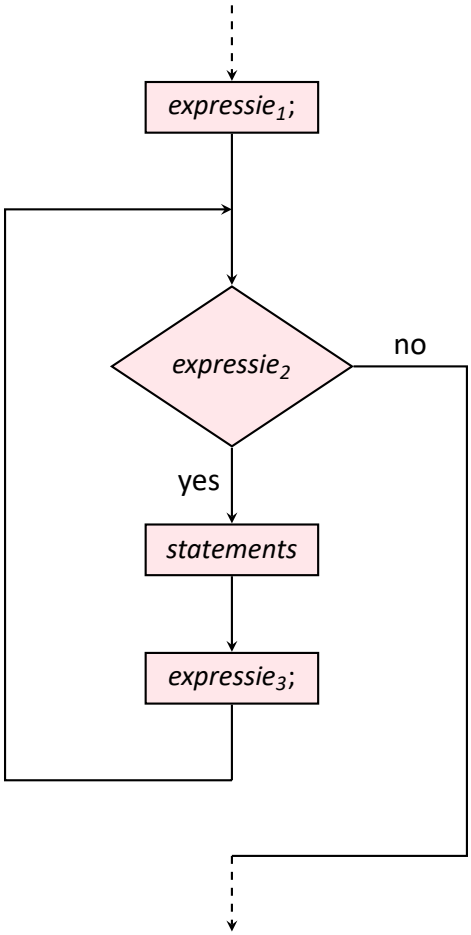
Listing 4.5: *for-lus in C.*

is *semantisch identiek* aan (let op de punt-komma's):

```
expressie1;
while (expressie2} {
    statements
    expressie3;
}
```

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

Let erop dat $expressie_3$ ná *statements* komt. De flowchart is te vinden in figuur 4.5.



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

We hebben hier overigens de statements `break` en `continue` niet opgenomen. Deze statements moeten apart worden ingepast.

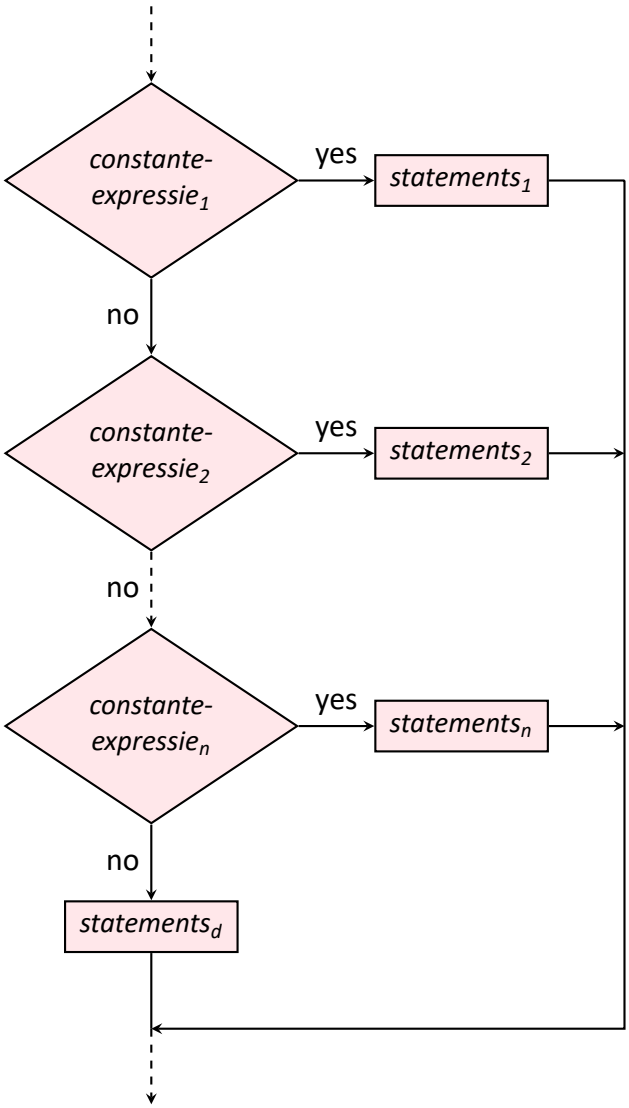
4.5 switch-statement

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

```
switch (variabele) {
    case constante-expressie1: statements1; break;
    case constante-expressie2: statements2; break;
    ...
    case constante-expressienn: statementsn; break;
    default : statementsd; break;
}
```

Listing 4.7: *switch-statement in C.*

Merk op dat *constante-expressie*$_1$, *constante-expressie*$_2$, ..., *constante-expressie*$_n$ een constante moeten opleveren. De flowchart is te zien in figuur 4.6.



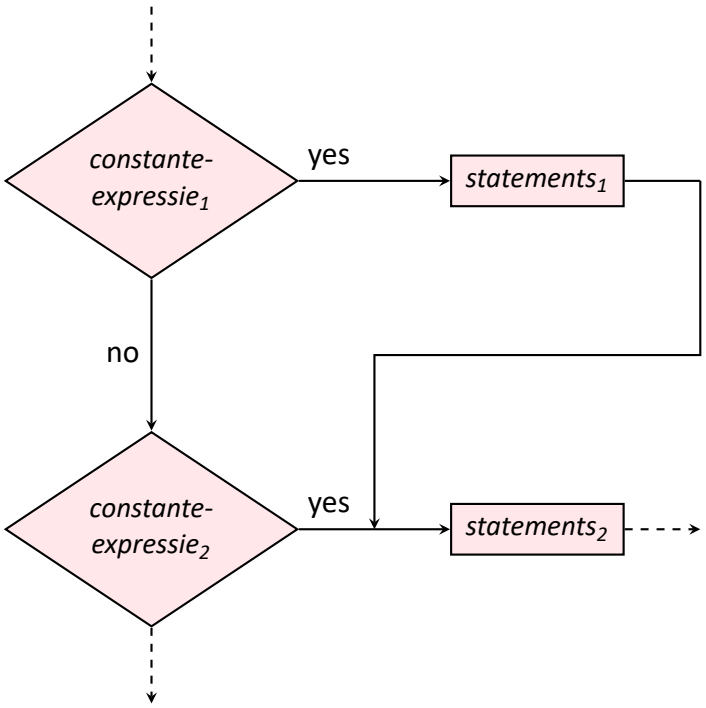
Figuur 4.6: *Flowchart van een switch-statement met break.*

Merk op dat we hier expliciet het `break`-statement hebben toegevoegd. Zonder dit statement wordt verder gegaan met het uitvoeren van het volgende statement. Zie listing 4.8.

```
switch (variabele) {
    ...
    case constante-expressie1: statements1;
    case constante-expressie2: statements2;
    ...
}
```

Listing 4.8: *switch-statement in C.*

De flowchart is te zien in figuur 4.7.



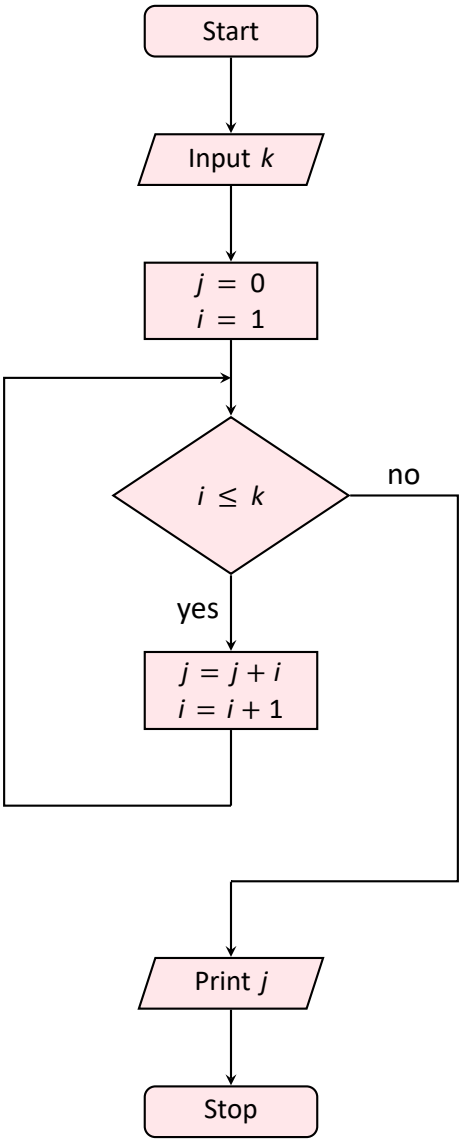
Figuur 4.7: *Flowchart van een `switch`-statement zonder `break`.*

4.6 Voorbeeld

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

$$j = 1 + 2 + 3 + 4 + \cdots + k \tag{4.1}$$

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 we uit bij het einde van het programma. In figuur 4.8 is de flowchart te zien.



Figuur 4.8: *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 kopieën 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 Engelse term van de code van een functie wordt de *body* genoemd.

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 elegant worden opgelost maar we zullen zien dat het wel zijn weerslag heeft op het geheugen en de executietijd.

Binnen een functie kunnen we lokale variabelen definiëren. 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 deze variabele bevat, is dan ook niet meer beschikbaar. Willen we een variabele door meerdere functies laten gebruiken dan moeten we de variabele globaal definiëren.

Gebaseerd op een document van Harry Broeders.

5.1 Een eenvoudige functie

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

```
#include <stdio.h>

int main(void) {
    //  ...
    printf("\n"); // sla 3 regels over
    printf("\n");
    printf("\n");
    //  ...
    printf("\n"); // sla 3 regels over
    printf("\n");
    printf("\n");
    //  ...
    return 0;
}
```

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.

```
#include <stdio.h>

void sla_3_regels_over(void) {
    printf("\n");
    printf("\n");
    printf("\n");
}

int main(void) {
    //  ...
    sla_3_regels_over();
    //  ...
    sla_3_regels_over();
    //  ...
    return 0;
}
```

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 meegeven bij aanroep. Vervolgens wordt deze functie in `main` tweemaal 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. In listing 5.3 zien we hoe een functiedeclaratie kan worden gebruikt [1].

```
#include <stdio.h>

void sla_3_regels_over(void);

int main(void) {
    //  ...
    sla_3_regels_over();
    //  ...
    sla_3_regels_over();
    //  ...
    return 0;
}

void sla_3_regels_over(void) {
    printf("\n");
    printf("\n");
    printf("\n");
}
```

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

[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 noodzakelijk om, als een functie niet boven `main` gedefinieerd is, een functiedeclaratie te gebruiken. Overigens geeft Visual Studio standaard een foutmelding.

5.2 Functies met parameters

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

```
#include <stdio.h>

int main(void) {
    //  ...
    printf("\n"); // sla 3 regels over
    printf("\n");
    printf("\n");
    //  ...
    printf("\n"); // sla 4 regels over
    printf("\n");
    printf("\n");
    printf("\n");
    //  ...
    return 0;
}
```

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

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 variabel 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 geïnitialiseerd wordt met de waarde van het bij aanroep meegegeven argument. De code van de functie wordt vervolgens uitgevoerd. Doordat de parameter `aantal` is geïnitialiseerd 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 geïnitialiseerd met de waarde 4 wordt de code in het `for`-statement vier maal herhaald en daardoor worden in de uitvoer vier regels 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 moe-

```c
#include <stdio.h>

void sla_regels_over(int aantal) {
    for (int teller = 0; teller < aantal; teller++) {
        printf("\n");
    }
}

int main(void) {
    //  ...
    sla_regels_over(3);
    //  ...
    sla_regels_over(4);
    //  ...
    return 0;
}
```

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

ten 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.

Command Prompt

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

Figuur 5.1: *Uitvoer van het programma in listing 5.6.*

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 `assert` wordt gebruikt om te controleren of de meegegeven argumenten aan deze voorwaarden voldoen. Als de expressie die in de `assert` wordt gebruikt `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

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

void print_lijn(int lengte) {

    assert (lengte > 2 && lengte < 80);
    for (int teller = 0; teller < lengte; teller++) {
        printf("+");
    }
    printf("\n");
}

void print_rechthoek(int breedte, int hoogte) {

    assert (hoogte > 2 && hoogte < 40);
    print_lijn(breedte);
    for (int regel = 0; regel < hoogte - 2; regel++) {
        printf("+");
        for (int teller = 0; teller < breedte - 2; teller++) {
            printf(" ");
        }
        printf("+\n");
    }
    print_lijn(breedte);
}

int main(void) {

    print_rechthoek(20, 6);
    return 0;
}
```

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

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 onderaan de pagina.

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 afdrukt 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(hoek));   /* sin(hoek) is printed */
```

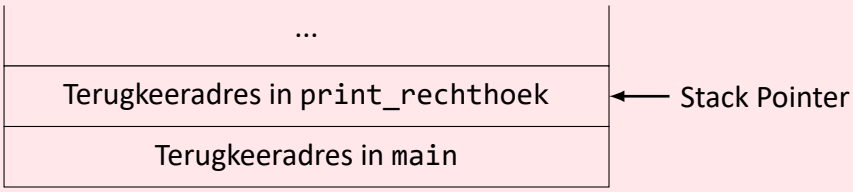
Maar meestal zal de waarde van de sinus gebruikt worden in een berekening:

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

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 het `return`-statement. Er kan slechts één waarde worden teruggegeven via het `return`-statement.

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 afhalen. 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.*

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.

```
#include <stdio.h>

double gemiddelde(int getal1, int getal2, int getal3) {

    double resultaat = (getal1 + getal2 + getal3) / 3.0;
    return resultaat;
}

int main(void) {

    double gem = gemiddelde(27, 29, 33);
    printf("%f\n", gem);
    return 0;
}
```

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.

```
#include <stdio.h>

double gemiddelde(int getal1, int getal2, int getal3) {

    return (getal1 + getal2 + getal3) / 3.0;
}

int main(void) {

    printf("%f\n", gemiddelde(27, 29, 33));
    return 0;
}
```

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

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 we deze functie ook kunnen gebruiken om de maximale waarde van drie getallen te bepalen door de returnwaarde van de ene aanroep te gebruiken als argument van de volgende aanroep.

```
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 afdrukt.*

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

```
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 gedefinieerd.

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

```c
int max(int getal1, int getal2) {

    if (getal1 > getal2) {
        return getal1;
    }
    return getal2;
}
```

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

```c
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(1,
       10));
26     return 0;
27 }
```

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 voor meer over scanf en inlezen van het toetsenbord paragraaf 9.3.

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.

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 en uitvoer van het programma.*

5.4 Zichtbaarheid en levensduur van lokale variabelen

Variabelen die binnen een functie gedefinieerd 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 executie van een functie tot de terugkeer naar de aanroeper. Lokale variabelen kunnen geïnitialiseerd 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 (via de parameters) worden 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 voorbeeld waarbij dit wel gewenst is, is een functie die de inhoud van twee variabelen verwisseld. Dit komt voor in sorteerprogramma's. We definiëren een functie wissel met twee parameters a en b. De functie en de aanroep 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.

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

```c
#include <stdio.h>

void wissel(int a, int b) {

    int hulpje = a;
    a = b;
    b = hulpje;
}

int main(void) {

    int x = 7, y = 8;

    printf("x = %d en y = %d\n", x, y);
    wissel(x, y);
    printf("x = %d en y = %d\n", x, y);
    return 0;
}
```

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

worden alleen de kopieën verwisseld, niet de originele variabelen. Toch is het mogelijk om de inhouden van x en y te verwisselen. Hoe dat moet, zien we in hoofdstuk 8.

Command Prompt

```
x = 7 en y = 8
x = 7 en y = 8
```

Figuur 5.4: *Uitvoer van de functie `wissel`.*

C is een blokgestructureerde taal dat betekent dat we in een *blok* lokale variabelen kunnen definiëren. Dit is te zien in listing 5.13. Variabele `i` is alleen te gebruiken binnen het `if`-statement[2].

```c
    if (n > 0) {
        int i;

        for (i = 0; i < 10; i++) {
            // use variable i
        }
    }
```

Listing 5.13: *Variabele binnen een blok.*

[2] De lokale variabele `i` is alleen binnen het blok te gebruiken. Als buiten het blok nóg een variabele `i` is gedefinieerd dan is die variabele buiten het blok te gebruiken.

Als we de lokale variabele `i` alleen maar binnen het `for`-statement nodig hebben, kunnen we ook gebruik maken van een zeer locale definitie. Dit is te zien in listing 5.14. Voor en na het `for`-statement is lokale variabele `i` niet beschikbaar. De zichtbaarheid van `i` is beperkt tot het `for`-statement.

```
    if (n > 0) {
        // variable i does not exists
        for (int i = 0; i < 10; i++) {
            // use variable i
        }
        // variable i does not exists
    }
```

Listing 5.14: *Variabele binnen een blok.*

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 verwijderd wordt, maar behouden blijft *over aanroepen heen*. Dat kan door bij de definitie van een lokale variabele de qualifier `static` te gebruiken. Zo'n lokale variabele wordt bij het starten van het programma geïnitialiseerd, expliciet door een constante expressie of impliciet met 0, en is beschikbaar zolang het programma draait, maar alleen binnen de functie waar de variabele gedefinieerd is.

In listing 5.15 is een functie te zien met een `static` lokale variabele. De variabele wordt bij het starten van het programma met 0 geïnitialiseerd (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, en wordt de variabele niet verwijderd als we terugkeren naar de aanroeper.

```
#include <stdio.h>

int geefterug(void) {
    static int count = 0;

    count++;

    return count;
}

int main(void) {

    for (int i = 0; i < 10; i++) {
        printf("Aanroep %d\n", geefterug());
    }

    return 0;
}
```

Listing 5.15: *Een functie met een static variabele.*

5.5 Zichtbaarheid en levensduur van globale variabelen

Een globale variabele is beschikbaar en zichtbaar tijdens de hele executie van het programma en binnen het gehele C-programma. Alle functies kunnen deze variabele gebruiken in statements. Het wordt aangemaakt als het programma start en wordt verwijderd als het programma is afgelopen. Een globale variabele kan eenmalig worden geïnitialiseerd, of wordt automatisch op 0 gezet als de initialisatie achterwege is gebleven. Een globale variabele wordt buiten functies gedefinieerd.

Lokale variabelen kunnen globale variabele verbergen. In listing 5.16 is een opzet van een C-programma te zien met een definitie van een globale variabele `i` (regel 1). In de functie `func1` wordt geen lokale variabele `i` gedefinieerd en is de globale variabele `i` zichtbaar. We kunnen in onze statements daar gebruik van maken. In functie `func2` wordt een lokale variabele `i` gedefinieerd. Deze definitie zorgt ervoor dat de globale variabele wordt verborgen. Binnen de functie wordt alleen gebruik gemaakt van de lokale variabele `i`. Er is in C geen mogelijkheid om toch de globale variabele te gebruikten (binnen deze functie natuurlijk).

```
1 int i;
2
3 void func1{int a} {
4     // global i visible
5 }
6
7 void func2{int a} {
8     int i;     // overrules the global i
9 }
```

Listing 5.16: *Zichtbaarheid van globale variabelen.*

Als we een globale variabele alleen maar zichtbaar willen maken binnen het C-bestand waarin de variabele wordt gedefinieerd dan moeten we de variabele voorzien van de qualifier `static`. De globale variabele is dan alleen zichtbaar voor de functies binnen het C-bestand. Een C-programma kan namelijk verdeeld worden over meerdere bestanden (ook wel *translation units* genoemd). Zie hoofdstuk 11.

Call by Value

De methode waarop de inhoud van 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 gedraagt zich als een lokale variabele en 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 8 behandeld.

5.6 Zichtbaarheid en levensduur van functies

Elke functie is in principe in het C-programma te gebruiken. Het is niet mogelijk om een functie alleen maar beschikbaar te stellen voor een andere functie. Willen we een functie alleen maar beschikbaar stellen voor het C-bestand waarin de functie is gedefinieerd, dan moeten we de functie voorzien van de qualifier `static`. Zie ook hoofdstuk 11.

5.7 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).

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \tag{5.1}$$

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

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

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 dan krijgen we een functie zoals te zien is in listing 5.17. Dit programma drukt de faculteiten af van 0 t/m 20.

```
#include <stdio.h>

unsigned long long int faculteit(unsigned long long int n) {

    if (n <= 1) {
        return 1;
    }
    return n * faculteit(n - 1);
}

int main(void) {

    for (unsigned long long int i = 0; i <= 20; i++) {
        printf("%2llu! = %llu\n", i, faculteit(i));
    }
    return 0;
}
```

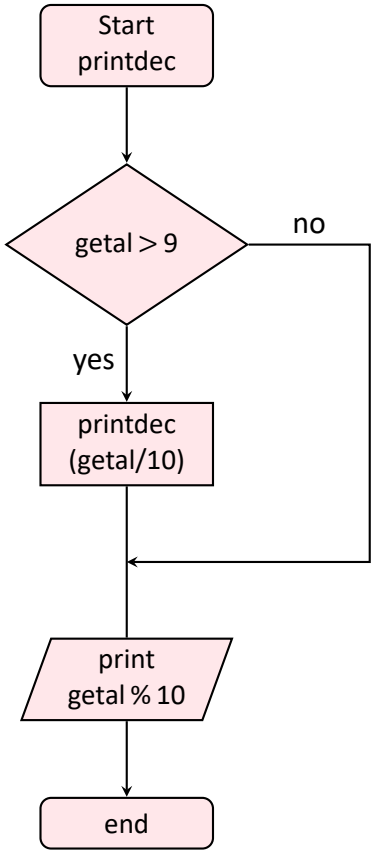
Listing 5.17: *Een recursieve functie om de faculteit van een natuurlijk getal te berekenen.*

[3] In een recursieve definitie wordt hetgeen gedefinieerd wordt in de definitie zelf gebruikt.

Omdat de faculteit gedefinieerd is voor natuurlijke getallen (gehele getallen groter dan of gelijk aan nul) is als qualifier 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 qualifier `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! = 2585201673888497664000 0 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 we een natuurlijk getal willen afdrukken in het tientallig talstelsel zonder gebruik te maken van `printf`, dan kunnen we gebruik maken van het recursieve algoritme dat gegeven is in figuur 5.5.



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 het meest significante cijfer gevonden en drukken dat af en keren terug naar

de vorige aanroep. In de vorige aanroep drukken we de eenheid van het argument af met behulp van de modulus-operator.

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 `prindec` 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 recusieve functie is te vinden in listing 5.18.

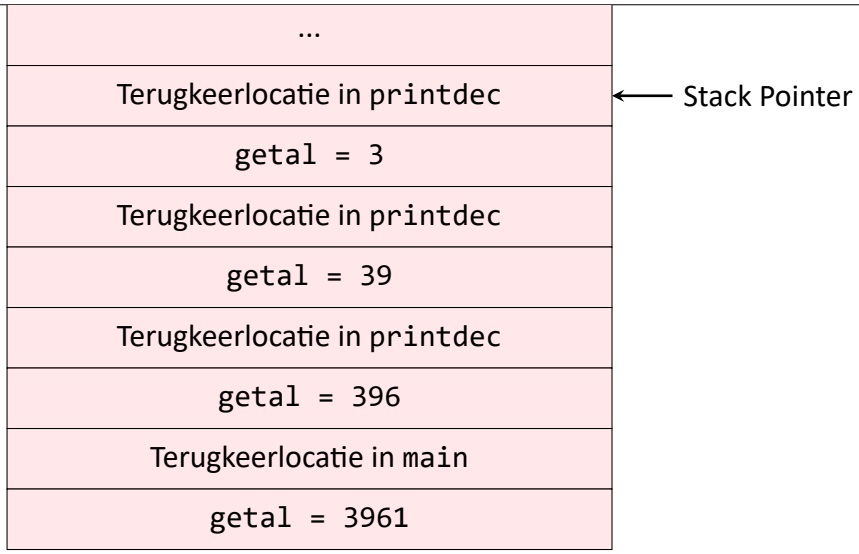
```
#include <stdio.h>

void printdec(int getal) {
    if (getal > 9) {
        printdec(getal / 10);
    }
    printf("%d", getal % 10);
}

int main(void) {
    int i = 3961;

    printdec(i);
    return 0;
}
```

Listing 5.18: *Een recursieve functie om een natuurlijk getal af te drukken.*

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 67). Merk op dat ook het argument voorafgaande aan de functieaanroep op de stack wordt gezet. Dat is nodig omdat het argument steeds verandert. Nadat de functie voor de vierde keer is aangeroepen ziet de stack eruit zoals te zien is in figuur 5.6.

* 5.8 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 dat nodig is om een getal te berekenen) en de *Time Complexity* (hoe lang duurt het om een getal te berekenen).



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

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

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

Dus een Fibonacci-getal $F(n)$ is uit te rekenen door de som te nemen van de twee eerdere Fibonacci-getallen met beginwaarden $F(0) = 0$ en $F(1) = 1$. Rekenen we een aantal Fibonacci-getallen uit dan komen we tot de reeks:

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

Een recursieve functie hiervoor is vrij eenvoudig. Dit is te zien in listing 5.19.

```
int fib(int n) {

    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Listing 5.19: *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 5.20 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 we 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.

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

long long int calls = 0;

unsigned long long int fib(unsigned long long int n) {

    calls = calls + 1;
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}

int main() {

    int i = 1;

    for (i = 37; i < 44; i++) {
        calls = 0;
        clock_t starttime = clock(), endtime;

        printf("Fib(%d) = %llu, ", i, fib(i));
        endtime = clock() - starttime;

        printf("Calls: %lld, ", calls);
        printf("Time: %.3f\n", (double)endtime/CLOCKS_PER_SEC);
    }

    return 0;
}
```

Listing 5.20: *Programma om Fibonacci-getallen af te drukken.*

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 1,534 seconden. Voor het berekenen van $F(43)$ zijn meer dan 1,4 miljard aanroepen nodig en de rekentijd bedraagt 22,381 seconden.

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

$$C(n) = 2F(n+1) - 1 \tag{5.5}$$

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

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

Command Prompt

```
Fib(37) = 24157817, Calls: 78176337, Time: 1.534
Fib(38) = 39088169, Calls: 126491971, Time: 2.085
Fib(39) = 63245986, Calls: 204668309, Time: 3.175
Fib(40) = 102334155, Calls: 331160281, Time: 5.107
Fib(41) = 165580141, Calls: 535828591, Time: 8.315
Fib(42) = 267914296, Calls: 866988873, Time: 13.865
Fib(43) = 433494437, Calls: 1402817465, Time: 22.381
```

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

Dit wordt de "Big O-notation" genoemd. Dit zegt ons niet zozeer hoeveel rekentijd een berekening precies kost maar wel wat de verhouding is met de rekentijd van het volgende getal. Overigens zijn er geweldig mooie relaties tussen de getallen:

$$\begin{aligned}
\frac{F(43)}{F(42)} &= \frac{433494437}{267914296} \approx 1,618 \\
\frac{C(43)}{C(42)} &= \frac{1402817465}{866988873} \approx 1,618 \\
\frac{T(43)}{T(42)} &= \frac{22.381}{13.865} \approx 1,618
\end{aligned} \tag{5.7}$$

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* genoemd[4].

5.9 Pointers en functies als parameter

Een pointer is een variabele met als inhoud het *adres* van een (andere) variabele. 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 behandelen dit in hoofdstuk 8.

5.10 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 7;
- de waarden teruggeven via zogenoemde *Call by Reference*-parameters, dit is vooral handig bij dezelfde datatypes (arrays), zie hiervoor hoofdstuk 6.

[4] De waarde is bij benadering 1,6180339887498948482045868343656381177 2.

5.11 Wiskundige functies

Hoewel C niet specifiek bedoeld is voor het berekenen van wiskundige formules, kunnen met behulp van *mathematical library* toch wiskundige berekeningen uitgevoerd worden. Een aantal veel gebruikte functies is te zien in de onderstaande lijst.

`sin(x)`	berekent de sinus van x (x in radialen)
`cos(x)`	berekent de cosinus van x (x in radialen)
`tan(x)`	berekent de tangens van x (x in radialen)
`asin(x)`	berekent de arcsin x in het bereik $[-\pi/2, \pi/2]$, domein is $[-1, 1]$
`acos(x)`	berekent de arccos x in het bereik $[0, \pi]$, domein is $[-1, 1]$
`atan(x)`	berekent de arctan x in het bereik $[-\pi/2, \pi/2]$
`atan2(x,y)`	berekent de arctan x/y in het bereik $\langle -\pi, \pi]$, $(x, y) \neq (0, 0)$
`sinh(x)`	berekent de sinus hyperbolicus van x
`cosh(x)`	berekent de cosinus hyperbolicus van x
`tanh(x)`	berekent de tangens hyperbolicus van x
`exp(x)`	berekent e^x
`log(x)`	berekent $\ln x$, $x > 0$
`log10(x)`	berekent ${}^{10}\log x$, $x > 0$
`pow(x,y)`	berekent x^y, $x > 0$, $x = 0 \wedge y > 0$, $x < 0 \wedge y \in \mathbb{Z}$
`sqrt(x)`	berekent $\sqrt{x}$, $x \geq 0$
`ceil(x)`	kleinste gehele getal niet kleiner dan x
`floor(x)`	grootste gehele getal niet groter dan x
`fabs(x)`	berekent $\|x\|$
`fmod(x,y)`	berekent de floating point-rest van x/y, met hetzelfde teken als x én $y \neq 0{,}0$

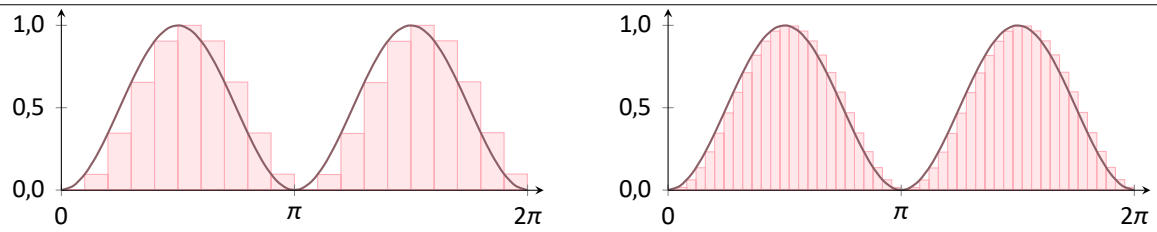
Alle argumenten en returnwaarden zijn van het type `double`. De hoeken in goniometrische functies zijn in radialen. Om de functies te gebruiken, moet header-bestand `math.h` geladen worden. Als de parameters niet in in het bereik zijn van de functies wordt een NaN (not a number) teruggegeven.

Als voorbeeld ontwikkelen we een programma om een numerieke benadering van een Riemann-integraal te programmeren. De te integreren functie is het kwadraat van een sinus van 0 tot 2π. Dit komt onder andere voor bij het bepalen van het gemiddelde elektrisch vermogen over een tijdspanne. De algemene gedaante is:

$$O = \int_a^b f(x)\,\mathrm{d}x = \lim_{n\to\infty} \sum_{k=0}^{n-1} f\left(a + k \cdot \frac{b-a}{n}\right) \cdot \frac{b-a}{n} \tag{5.8}$$

waarbij geldt dat $a = 0$ en $b = 2\pi$. Wat we doen is het volgende. De te integreren functie wordt verdeeld in een aantal rechthoeken zoals te zien is in figuur 5.8. De punt linksboven van de rechthoek raakt de functie. Als we nu de oppervlakten van alle rechthoeken bij elkaar optellen dan vinden we een benadering van de oppervlakte. Dit wordt de Riemann-linkersom genoemd. Als we het aantal rechthoeken steeds groter maken (waarbij de breedte van de rechthoek steeds smaller wordt), dan benaderen we de exacte waarde steeds beter.

Het programma om de numerieke oplossing te berekenen is te vinden in listing 5.21. We stellen het aantal stappen (n) in op 1000, a in op 0, b in op 2π en berekenen de stapgrootte. Daarna



(a) *Met 20 rechthoeken.* **(b)** *Met 50 rechthoeken.*

Figuur 5.8: *Bepalen oppervlakte onder een kromme met behulp van de Riemann-linkersom.*

lopen we met behulp van een `for`-statement van 0 tot $n-1$ en berekenen we de oppervlakte van de huidige rechthoek. Tevens tellen we de oppervlakten van de rechthoeken bij elkaar op. Als we n steeds groter maken, benaderen we de exacte uitkomst steeds beter. De exacte uitkomst is overigens π.

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

double f(double x) {
    return sin(x)*sin(x);
}

int main() {

    int k, n = 1000;

    double a = 0.0;
    double b = 6.28318530717958647692;

    double deltax = (b - a) / n;

    double sum = 0.0;

    for (k = 0; k < n; k++) {
        sum = sum + f(a + k*deltax) * deltax;
        printf("k: %d, sum: %f\n", k, sum);
    }
    printf("Left rule Riemann sum = %.20e\n", sum);
    return 0;
}
```

Listing 5.21: *Programma voor het berekenen van de Riemann-linkersom.*

5.12 Classificatie van karakters

C kent een aantal functies om karakters te classificeren. We gaan hierbij uit van de ASCII-code. De ASCII-code is te vinden in bijlage A.

Om deze functies gebruiken moet header-bestand `ctype.h` geladen worden. Een lijst met functies is te zien in tabel 5.1. Let erop dat de parameter van de functies van het type `int` is. Karakters die als argumenten worden opgegeven, worden dus gecast naar een `int`. De reden hiervoor is dat als het karakter overeenkomt met *end-of-file* (EOF, zie hoofdstuk 9), meestal gedefinieerd als -1 dat dan deze waarde wordt teruggegeven. Een returnwaarde ongelijk aan 0 moet gelezen worden als waar, een returnwaarde gelijk aan 0 moet gelezen worden als niet waar.

Tabel 5.1: *Classificatie van karakters.*

Functie	Test karakter op
`isalnum()`	Alfanumeriek (letter of cijfer)
`isalpha()`	Letter
`iscntrl()`	Control-teken (ASCII 0 – 31, 127)
`isdigit()`	Decimaal cijfer
`isgraph()`	Grafisch
`islower()`	Kleine letter
`isprint()`	Afdrukbaar
`ispunct()`	Leesteken
`isspace()`	Whitespace (spatie, tab, newline ...)
`isupper()`	Hoofdletter
`isxdigit()`	Hexadecimaal cijfer

Er zijn nog twee interessante functies: `tolower()` en `toupper`. De functie `tolower()` geeft een kleine letter terug als het argument van de functie een letter is (hoofdletter of kleine letter). De functie `toupper()` geeft een hoofdletter terug als het argument een letter is (hoofdletter of kleine letter). De functies geven het argument terug als het argument geen letter is.

Dat de functies vrij eenvoudig zijn, kunnen we zien in listing 5.22. De functie `isupper()` hebben we ook echt geïmplementeerd als een functie. Voor `isdigit()` hebben we echter een *macro* gebruikt. Macro's worden beschreven in hoofdstuk 10.

```
int isupper(int arg) {

    if (arg == EOF) {
        return arg;
    }

    return (arg >= 'A' && arg <= 'Z');
}

#define isdigit(A) ((A) == EOF ? EOF : (A) >= '0' && (A) <= '9')
```

Listing 5.22: *Implementatie van de functies `isupper()` en `isdigit()`.*

In listing 5.23 is een programma te zien dat reeks karakters van het toetsenbord inleest, afgesloten met een enter-toets, en deze karakters classificeert.

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

#pragma warning(disable : 4996)

int main(void) {

    char c;

    printf("Enter characters and press <enter>: ");
    scanf("%c", &c);

    while (c != '\n') {
        printf("Character '%c' is a", c);
        if (isprint(c)) {
            printf(" printable");
        }
        if (isdigit(c)) {
            printf(" digit");
        }
        if (isalpha(c)) {
            printf(" letter");
        }
        if (isspace(c)) {
            printf(" whitespace");
        }
        if (isxdigit(c)) {
            printf(" hexadecimal");
        }
        if (ispunct(c)) {
            printf(" punctuation");
        }
        if (islower(c)) {
            printf(" lower case");
        }
        if (isupper(c)) {
            printf(" upper case");
        }
        if (iscntrl(c)) {
            printf(" control");
        }
        printf("\n");
        scanf("%c", &c);
    }

    return 0;
}
```

Listing 5.23: *Classificatie van karakters.*

6

Arrays

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 definitie geven en in het programma moeten we de som bepalen door alle variabelen op te tellen. Willen we 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 behandelen arrays op vele manieren, onder andere hoe een array wordt gedefinieerd en gebruikt. We laten zien dat het mogelijk is om "langs een array te lopen" met behulp van een herhaling. We kunnen tweedimensionale arrays 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 behandelen, namelijk de relatie tussen arrays en pointers. Die wordt behandeld in hoofdstuk 8.

6.1 Definitie 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 definitie

```
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 definitie wordt het aantal *elementen* opgegeven. In het voorbeeld zijn dat er 10. De nummering van de elementen begint bij 0 en loopt tot en met het aantal elementen minus 1. Tevens wordt bij de definitie 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 gedefinieerd zijn, dus de expressies

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

zijn niet geldig[1]. Overigens voegen de meeste compilers geen extra programmacode 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 definitie

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

wordt, als *globale* variabele, automatisch geïnitialiseerd met nullen. Als *lokale* variabele is de inhoud ongedefinieerd. We kunnen de array bij de definitie direct 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 definitie weglaten. We kunnen dus ook schrijven:

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

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

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

[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.

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 geïnitialiseerd 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 definitie

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

een *eendimenationale* array genoemd. Zoals al eerder is geschreven, mogen we het elementnummer van een array met een expressie uitrekenen, op voorwaarde dat de uitkomst een geheel getal is en binnen de grenzen van de 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.*

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 definiëren 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.

```c
#include <stdio.h>

int main(void) {

    int kwad[10], i, som = 0;

    int rij[5] = { 2, 5, 8, 1, 6 };

    for (i = 0; i < 10; i = i + 1) {
        kwad[i] = i * i;
    }

    for (i = 0; i < 5; i = i + 1) {
        som = som + kwad[rij[i]];
    }

    printf("De som van de kwadraten is %d\n", som);

    return 0;
}
```

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

6.4 Aantal elementen van een array bepalen

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

```
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 grootte = 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 in regel 13 in listing 6.2 vervangen door een berekening van het aantal elementen. Dit is te zien in listing 6.3.

6.5 Tweedimensionale array

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

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

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

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

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

We kunnen de array tegelijk met de definitie 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. Om een 5×3 matrix te definiëren en te initialiseren, gebruiken we:

```
int matrix[5][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9},
                     {10, 11, 12}, {13, 14,15} };
```

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 vaak voorkomende 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.

```c
#include <stdio.h>

int main(void) {

    int twodim[15][20];

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

    printf("Rows: %d\n", rows);
    printf("Cols: %d\n", cols);

    return 0;
}
```

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 herhaling. Dit is te zien in listing 6.5. In regel 5 definiëren 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`-statement. Binnen de herhaling, in regel 12, drukken we de waarde van het element af.

```c
#include <stdio.h>

int main(void) {

    int rij[] = {2, 5, 8, 1, 6, 7, 3, 9, 0, 4, 7, 8};

    int len = sizeof rij / sizeof rij[0];

    printf("De array heeft %d elementen: ", len);

    for (int i = 0; i < len; i = i + 1) {
        printf("%d ", rij[i]);
    }

    printf("\n");

    return 0;
}
```

Listing 6.5: *Afdrukken van een array.*

6.7 Arrays vergelijken

Een array is een samengesteld datatype en kan niet zonder meer met een andere array vergeleken worden. Het is verleidelijk om de twee arrays te vergelijken alsof het enkelvoudige variabelen zijn zoals te zien is in listing 6.6. Door de manier waarop C met de namen omgaat zorgt ervoor dat niet de inhouden van de arrays vergeleken worden, maar de adressen van het eerste element. Dit is vergelijkbaar met de array als argument van een functie.

```
#include <stdio.h>

int main(void) {

    int ary1[10] = { 0 }, ary2[10] = { 0 };

    printf("De arrays zijn %sgelijk\n", ary1 == ary2 ? "" : "on");
}
```

Listing 6.6: *Vergelijken van twee arrays (foutief).*

De juiste manier om twee arrays te vergelijken is om de elementen uit de arrays met identiek elementnummer een voor een te vergelijken. We doen dit met behulp van de functie `arraycompare`, te zien in listing 6.7.

```
1  int arraycompare(int ary1[], int ary2[], int siz1, int siz2) {
2
3      /* If sizes are not equal, return 0 */
4      if (siz1 != siz2) {
5          return 0;
6      }
7
8      for (int pos = 0; pos < siz1; pos++) {
9          if (ary1[pos] != ary2[pos]) {
10             /* Elements are not equal, so arrays are not equal */
11             return 0;
12         }
13     }
14
15     /* Array elements are equal, return 1 */
16     return 1;
17 }
```

Listing 6.7: *Vergelijken van twee arrays.*

De twee arrays en het aantal elementen worden als argumenten meegegeven. De arrays moeten natuurlijk uit een gelijk aantal elementen bestaan. Dat testen we in regel 3 en we keren gelijk terug met een waarde 0 als de aantallen verschillend zijn. Met een `for`-statement lopen we langs alle elementen van de arrays. We vergelijken telkens twee elementen op dezelfde

positie. Zodra twee elementen ongelijk aan elkaar zijn, weten we dat de arrays niet aan elkaar gelijk zijn en geven een 0 terug. Als alle elementen vergeleken zijn en we uit het for-statement komen dan weten we dat alle elementen gelijk zijn en geven we een 1 terug. In listing 6.8 is te zien hoe we de functie moeten aanroepen.

```
#include <stdio.h>

int arraycompare(int ary1[], int ary2[], int siz1, int siz2);

int main(void) {

    int ary1[10] = { 0 }, ary2[10] = { 0 };
    int siz1 = sizeof ary1 / sizeof ary1[0];
    int siz2 = sizeof ary2 / sizeof ary2[0];

    printf("De arrays zijn %sgelijk\n", arraycompare(ary1, ary2,
                                        siz1, siz2) ? "" : "on");
}
```

Listing 6.8: *Vergelijken van twee arrays.*

De standard library bevat een aantal functies waarmee we gemakkelijk arrays kunnen vergelijken, onafhankelijk van het datatype. Dit wordt behandeld in hoofdstuk 8.

6.8 Arrays als argument en parameter

Een array kan als argument van een functie worden gebruikt. Een functie kan *geen* array teruggeven. Dat heeft te maken met de manier waarop arrays overgedragen worden. Bij enkelvoudige datatypes wordt een *kopie* gemaakt van de originele variabelen en die kopieën worden doorgegeven. Bij overdragen van een array wordt geen kopie gemaakt, maar wordt de plaats in het geheugen waar de array begint overgedragen[2]. Omdat niet een complete kopie wordt gemaakt, kan de functie de grootte van de array niet uitrekenen; de functie kent immers de definitie niet. Er moet dus altijd een parameter met de grootte van de array worden overgedragen.

In listing 6.9 is een functie te zien die de positie bepaalt van een getal dat in een array voorkomt. Als het getal meerdere keren voorkomt, wordt de eerst gevonden positie teruggegeven. De waarde `-1` wordt teruggegeven als het getal niet in de array voorkomt. De aanroepende functie moet hierop testen. Om aan te geven dat de eerste parameter een array is, wordt de declaratie `int ary[]` gegeven. De tweede parameter is het aantal elementen in de array en de derde parameter is het getal dat gezocht wordt.

[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.

```
int findnumber(int ary[], int siz, int num) {

    for (int pos = 0; pos < siz; pos = pos + 1) {
        if (ary[pos] == num) {
            return pos;
        }
    }

    return -1;
}
```

Listing 6.9: *Functie voor het vinden van een getal in een array.*

In listing 6.10 is te zien hoe de functie wordt aangeroepen. Zie regel 12. Als eerste argument wordt de *naam* van de array opgegeven. De compiler weet aan de hand van de functiedefinitie en de definitie 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.

```
#include <stdio.h>

int findnumber(int ary[], int siz, int num);

int main(void) {

    int rij[] = { 4, 8, 3, 8, 2, 3, 4, 5, 1, 2 }, getal;

    printf("Geef te zoeken getal: ");
    scanf("%d", &getal);

    int plek = findnumber(rij, sizeof rij / sizeof rij[0], getal);

    printf("Getal %d staat op plek %d\n", getal, plek);

    return 0;
}
```

Listing 6.10: *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 6.11 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.

```
int patcharray(int ary[], int siz, int what, int with) {

    int count = 0;

    for (int pos = 0; pos < siz; pos = pos + 1) {
        if (ary[pos] == what) {
            ary[pos] = with;
            count = count + 1;
        }
    }

    return count;
}
```

Listing 6.11: *Functie om een getal te vervangen door een ander getal.*

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 programmeerwerk enigszins verlichten.

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

```
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`). Het nul-karakter is het karakter waarvan alle bits 0 zijn, zie bijlage A. 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 herhaling uitgevoerd worden. Dit is te zien in listing 6.12. In regel 3 definiëren en initialiseren we de `str` met een string constante. In het `while`-statement 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 herhaling is het aantal karakters beschikbaar in variabele `len`.

```c
#include <stdio.h>

int main() {

    char str[] = "Hallo wereld!";
    int len = 0;

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

    printf("Lengte is %d\n", len);
}
```

Listing 6.12: *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 meest gebruikte functies op een rijtje gezet. Zie tabel 6.1.

Tabel 6.1: *Enkele standaard functies voor stringmanipulaties.*

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

De functie `strlen` geeft het aantal karakters in de string terug exclusief het nul-karakter. De functie `stcpy` kopieert de string `from` naar de string `to`. Let erop dat de ruimte van `to` groot genoeg moet zijn. 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 moet zijn.

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 een negatief getal terug als `s1` 'kleiner' is dan `s2`, geeft `0` terug als de strings identiek zijn en geeft een positief getal 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 6.13 waar we overigens gebruik hebben gemaakt van enkele juweeltjes van C-snelschrift. We dagen de lezer uit om de functies te analyseren.

```c
#include <stdio.h>

int stringlength(char str[]) {

    int len;

    for (len = 0; str[len] != '\0'; len++);

    return len;
}

void stringcopy(char to[], char from[]) {

    for (int len = 0; (to[len] = from[len]) != '\0'; len++);
}

int stringcompare(char str1[], char str2[]) {

    int len = 0;

    while (str1[len] == str2[len] && str1[len++] != '\0');

    return str1[len] - str2[len];
}

void stringconcat(char to[], char from[]) {

    int i, j;

    for (i = j = 0; to[i] != '\0'; i++);

    while ((to[i++] = from[j++]) != '\0');
}

int main(void) {

    char van[] = "Ik ben een string";
    char naar[100];

    printf("Lengte is %d\n", stringlength(van));

    stringcopy(naar, van);

    printf("Kopie: %s\n", naar);

    printf("Strings zijn %sgelijk.\n",
                        stringcompare(van, naar) ? "on" : "");

    stringconcat(naar, " en ik ben langer");

    printf("%s\n", naar);

    return 0;
}
```

Listing 6.13: *Een implementatie van de stringfuncties.*

7

Structures

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

Een structure kan 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 en we mogen de variabelen binnen de structure gebruiken in expressie. We mogen *niet* twee structures vergelijken.

7.1 Definitie van structures

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 declareren de structure `artikel` als volgt (listing 7.1).

```
struct artikel {
    int nummer;       // artikelnummer
    char naam[20];    // artikelnaam
    int aantal;       // aantal beschikbaar
    double prijs;     // prijs per stuk
};
```

Listing 7.1: *Declaratie van de structure `artikel`.*

Hierin wordt `artikel` de *structure identier* genoemd. De variabelen binnen de structure worden *leden* genoemd maar we zullen gebruik maken van de Engelse naam *members*.

[1] We zullen vanaf nu de Engelse naam structure gebruiken.

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

```
struct artikel variabele;
```

om variabelen te definiëren. In listing 7.2 is een aantal definities 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 7.2: *Definitie van enkele variabelen.*

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

7.2 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");
```

7.3 Toekennen van structures

Een structure mag toegekend worden aan een andere structure middels de =-operator. We mogen dus schrijven

```
struct artikel a, b;
...
a = b;
```

Hierbij worden de members van structure b naar de overeenkomstige members van structure a gekopieerd. In feite is a nu een kopie van b.

7.4 Vergelijken van structures

Een structure mag niet zomaar met een andere structure vergeleken worden, ook niet als de structures van hetzelfde type zijn. Dus

```
struct artikel a, b;
..
if (a == b) {        /* WRONG */
    ...
}
```

genereert een foutmelding door de compiler. De juiste manier is door alle members van de structures een voor een (*member-wise*) te vergelijken. Dus

```
1 struct artikel a, b;
2 ..
3 if (a.nummer == b.nummer && strcmp(a.naam, b.naam) == 0 &&
4     a.aantal == b.aantal && a.prijs == b.prijs) {
5     /* true */
6 } else {
7     /* false */
8 }
```

Listing 7.3: *Vergelijken van twee structures.*

7.5 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 argument aan een functie worden weergegeven of dienen als parameter in een functie. In de functie kunnen we dan de members een voor een afdrukken. Dit is te zien in listing 7.4. We hebben extra spaties toegevoegd om de gegevens 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 7.4: *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 7.5.

De parameters krijgen de waarden mee die aan de members moeten worden toegekend. Om een nieuwe structure te maken waar de gegevens inkomen, definiëren we in regel 3 een struc-

```
struct artikel maak_artikel(int nummer, char naam[], int aantal,
   double prijs) {

    struct artikel nieuw;

    nieuw.nummer = nummer;
    strcpy(nieuw.naam, naam);
    nieuw.aantal = aantal;
    nieuw.prijs = prijs;

    return nieuw;
}
```

Listing 7.5: *Aanmaken van een nieuw artikel.*

ture 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 aanroeper.

We kunnen de structure `floppy` bijvoorbeeld initialiseren met:

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

7.6 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 7.6. 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.

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).

```
typedef struct {
    int nummer;       // artikelnummer
    char naam[20];    // artikelnaam
    int aantal;       // aantal beschikbaar
    double prijs;     // prijs per stuk
} artikel_t;
```

Listing 7.6: *De definitie van datatype `artikel_t`.*

Vervolgens kunnen we variabelen definiëren 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.

7.7 Array van structures

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

```
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 `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 middels een adres naar het eerste element wordt meegegeven. Zie ook hoofdstuk 8.9.

In listing 7.7 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.

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

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

```c
void print_artikelen(artikel_t ary[], int siz) {

    for (int i = 0; i < siz; i++) {
        if (ary[i].nummer != 0) {
            print_artikel(&ary[i]);
        }
    }
}
```

Listing 7.7: *Een functie om alle gegevens van artikelen af te drukken.*

7.8 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 7.8. 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.

```c
typedef struct {
    int nummer;       // artikelnummer
    char naam[20];    // artikelnaam
    int aantal;       // aantal beschikbaar
    double prijs;     // prijs per stuk
} artikel_t;

typedef struct {
    int klantnr;      // klantnummer
    char naam[20];    // naam v.d. klant
} klant_t;

typedef struct {
    int bestelnr;     // bestellingnr
    klant_t klant;    // de klant
                      // bestelde artikelen
    artikel_t artbestel[10];
} bestelling_t;
```

Listing 7.8: *De structures die horen bij een bestelling.*

Om een bestelling aan te maken definiëren we eerst een variabele van het nieuwe datatype `bestelling_t` en initialiseren we de structure met nullen. Dit is te zien in listing 7.9. Dat kan door alleen het bestelnummer expliciet met een 0 te initialiseren, de andere members wordt automatisch op 0 gezet. Daarna vullen we het bestelnummer in via

```c
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 alleen nog het aantal en de prijs berekenen. Dit is te zien in listing 7.9.

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

Listing 7.9: *Het vormgeven van een bestelling.*

Overigens is het niet slim om alle informatie van een artikel in de structure `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 pagina.

7.9 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 structure. De structure 'verpakt' dan de variabelen onder één noemer.

Een bekend voorbeeld is het berekenen van 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:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{7.1}$$

De expressie onder het wortelteken heet de *discriminant* en de wortel van de discriminant heeft alleen een reële 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 structure. Het volledige programma is te zien in listing 7.10.

We definiëren een structure 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

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

typedef struct {
    double x1;
    double x2;
    int geldig;
} wortels_t;

wortels_t bereken_wortels(double a, double b, double c) {
    wortels_t x12;

    double D = b * b - 4 * a * c;

    if (D < 0.0 || a == 0.0) {
        x12.geldig = 0;
        return x12;
    }

    x12.x1 = (-b + sqrt(D)) / (2.0 * a);
    x12.x2 = (-b - sqrt(D)) / (2.0 * a);
    x12.geldig = 1;
    return x12;
}

void print_wortels(wortels_t x12) {
    if (x12.geldig) {
        printf("x1: %f   ", x12.x1);
        printf("x2: %f\n", x12.x2);
    } else {
        printf("Uitkomst is niet geldig\n");
    }

}

int main(void) {

    wortels_t uitkomst;
    uitkomst = bereken_wortels(1.0, 2.0, -6.0);
    print_wortels(uitkomst);

    return 0;
}
```

Listing 7.10: *Een programma om de wortels van een kwadratische vergelijking te berekenen.*

variabele terug. In de functie `print_wortels` drukken we de wortels af of dat de wortels niet geldig zijn (kunnen niet berekend worden).

Merk overigens op dat als de discriminant 0 is, de wortels x_1 en x_2 dezelfde waarde hebben.

* 7.10 Unions

Stel dat we onze eigen C-compiler willen ontwerpen (begin er trouwens niet aan, dat is heel complex). Dan moeten we een lijst bijhouden van gedefinieerde variabelen. Van de variabele moet dan ook het type worden opgeslagen en mogelijk een initiële waarde[2]. 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 7.11 is een union variabele gedefinieerd met de naam `value`.

```
    union {
        int valint;
        float valfloat;
        double valdouble;
    } value;
```

Listing 7.11: *Een union om verschillende datatypes te gebruiken.*

De gereserveerde geheugenruimte is gelijk aan de geheugenruimte van 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 7.12. We hebben tevens gebruik gemaakt van een *enumeratie* (zie paragraaf 2.8) voor de verschillende datatypes.

```
typedef enum {typeint = 1, typefloat, typedouble} vartype_t;

typedef struct {
    char naam[20];
    vartype_t type;
    union {
        int valint;
        float valfloat;
        double valdouble;
    } value;
} varinfo_t;
```

Listing 7.12: *Een structure om een variabele in een compiler te gebruiken.*

[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 of initialisatie.

We kunnen nu een variabele definiëren en initialiseren met gegevens. Dit is te zien in listing 7.13. Te zien is dat de variabele wordt ingesteld als `float` en dat de waarde 3,14 bedraagt.

```
    varinfo_t devariabele;

    strcpy(devariabele.naam, "getal");
    devariabele.type = typefloat;
    devariabele.value.valfloat = 3.14f;
```

Listing 7.13: *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.

```
    devariabele.type = typeint;
    devariabele.value.valint = 1234;
```

Listing 7.14: *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. Zie listing 7.15.

```
void print_var(varinfo_t var) {

    printf("Naam: %s\n", var.naam);

    switch (var.type) {
    case typeint:
        printf("Type: int, Value %d\n", var.value.valint);
        break;
    case typefloat:
        printf("Type: float, Value %f\n", var.value.valfloat);
        break;
    case typedouble:
        printf("Type: double, Value %f\n", var.value.valdouble);
        break;
    default:
        printf("Type: ONBEKEND\n");
        break;
    }
}
```

Listing 7.15: *Een functie om een variabele af te drukken.*

* 7.11 Bitvelden

Bitvelden (bit fields) maken het mogelijk om delen van een geheugenplaats (of meerdere geheugenplaatsen gegroepeerd als één geheel), onder te verdelen in afzonderlijke bits. Dit is vooral handig bij het aansturen van hardware.

Stel dat we een (fictieve) hardwarematige *teller* (Engels: counter) willen aansturen. De gegevens van de teller worden met 32 bits weergegeven. In listing 7.16 is de indeling van de 32 bits te zien. De tellerwaarde bestaat uit 16 bits, dus de teller kan tellen van 0 t/m 65535. Dit wordt weergegeven met het veld `value`. De teller telt *cyclisch* dus na de hoogste waarde volgt weer de laagste waarde. De teller heeft dan een *overflow* gehad. Dit wordt weergegeven met het veld `overflow`. De teller heeft acht mogelijke manieren van tellen (omhoog, omlaag etc.). Dit wordt weergegeven met het veld `mode`. Als een speciale *interruptfunctie* gestart moet worden, geven we dat aan met het veld `interrupt_enable`.

```
struct {
    unsigned int interrupt_enable : 1; /* Bit 31 */
    unsigned int padding : 0;          /* Bits 30 down to 21 */
    unsigned int mode : 3;             /* Bits 20 down to 18 */
    unsigned int enable : 1;           /* Bit 17 */
    unsigned int overflow : 1;         /* Bit 16 */
    unsigned int value : 16;           /* Bits 15 down to 0 */
} counter;
```

Listing 7.16: *Voorbeeld van bitvelden.*

Het veld `padding` heeft de speciale breedte 0 om ervoor te zorgen dat de velden worden *opgelijnd* naar de breedte die van nature door de computer worden verwerkt. Een veel gebruikte term hiervoor is *machinewoord*. De bit `interrupt_enable` komt daardoor op positie 31 terecht, de andere bits lopen van 0 t/m 20.

Om te teller te starten gebruiken we:

```
counter.enable = 1;
```

Om de tellerwaarde te kopiëren naar een variabele gebruiken we:

```
unsigned int value = counter.value;
```

Om te testen of een overflow heeft plaatsgevonden, gebruiken we:

```
if (counter.overflow == 1) { ... }
```

Alles aan het gebruik van bitvelden is afhankelijk van de hardware waarop het programma draait en de C-compiler die gebruikt wordt. Dat betekent dat programma's met bitvelden moeilijk overdraagbaar zijn naar andere computers en andere C-compilers. Bij systemen die van nature met 32 bits werken worden de bits (hoogst waarschijnlijk) samengepakt in 32 bits eenheden.

Bitvelden werden vroeger gebruikt om geheugenruimte te besparen. Tegenwoordig is dat niet meer nodig omdat de meeste computers genoeg geheugen hebben. Het gebruik van bitvelden wordt daarom ook afgeraden.

8

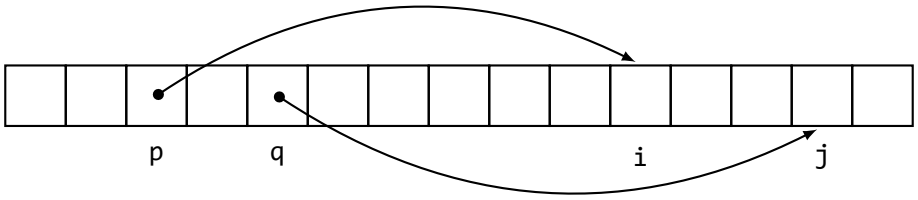
Pointers

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 argumenten 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 geven de programmeur een krachtig instrument in handen waar verantwoord mee omgegaan moet worden. Onzorgvuldig gebruik kan ervoor zorgen dat een programma voortijdig crasht. Het maakt het ook moeilijk om aan te tonen dat een programma correct functioneert. 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 8.1 is dat te zien. Elk vakje stelt een geheugenplaats voor[1]. In de figuur zijn de variabelen `i` en `j` te zien. De twee pointers `p` en `q` wijzen respectievelijk naar variabele `i` en `j`. Dit is weergegeven met de twee pijlen. De inhoud van pointer `p` is dus het adres van variabele `i` en de inhoud van pointer `q` is het adres van variabele `j`.



Figuur 8.1: *Uitbeelding van twee pointers die naar variabelen in het geheugen wijzen.*

[1] We gaan er gemakshalve van uit 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 object in het geheugen zoals variabelen, array-elementen, een structure (zie hoofdstuk 7) of (het begin van) een functie. Een pointer kan niet wijzen naar een expressie of een `register` variabele (zie hoofdstuk 11).

Het is ook mogelijk om een pointer naar het datatype `void` te laten "wijzen". Deze pointers worden generieke pointers genoemd. We zullen dit behandelen in paragraaf 8.3.

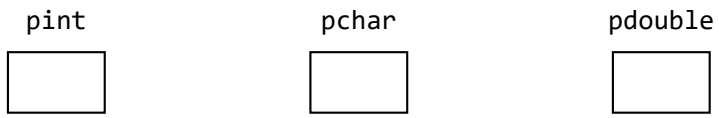
8.1 Pointers naar enkelvoudige datatypes

In listing 8.1 is de definitie van enkele pointers naar enkelvoudige datatypes te zien. Bij de definitie moet het type variabele waarnaar de pointer wijst worden opgegeven. De asterisk (*) geeft aan dat het de definitie van een pointer betreft.

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

Listing 8.1: *Enkele definities van pointers.*

We kunnen pointers uitbeelden door middel van vakjes, zoals te zien is in figuur 8.2. Elk vakje stelt een pointer voor. Na de definitie hebben de pointers een willekeurige inhoud. Er wordt dan wel gezegd dat de pointer nergens naar toe wijst, maar dat is feitelijk onjuist. Een pointer heeft altijd een adres met inhoud, maar het kan zijn dat de pointer niet naar een bekende variabele wijst.



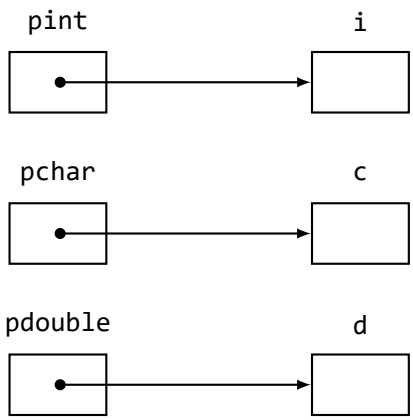
Figuur 8.2: *Voorstelling van drie pointers in het geheugen.*

Aan een pointer is het adres van een variabele van hetzelfde type toe te kennen. Hiervoor gebruiken we de *adresoperator* & (ampersand). In listing 8.2 is een aantal toekenningen van adressen te zien.

```
int i;              /* the variables */
char c;
double d;

int *pint;          /* the pointers */
char *pchar;
double *pdouble;

pint = &i;          /* pint points to variable i */
pchar = &c;         /* pchar points to variable c */
pdouble = &d;       /* pdouble points to variable d */
```

Listing 8.2: *Enkele toekenningen van adressen aan pointers.*

Nu de pointers geïnitialiseerd zijn, kunnen we een voorstelling maken van de relatie tussen de pointers en de variabelen. Dit is te zien in figuur 8.3. Pointer `pint` wijst naar variabele `i`, pointer `pchar` wijst naar variabele `c` en pointer `pdouble` wijst naar variabele `d`.



Figuur 8.3: *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.

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

Listing 8.3: *Gebruik van pointers bij toekenningen.*

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

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

Listing 8.4: *Gebruik van een pointer bij het afdrukken van een variabele.*

Overigens kan tijdens definitie ook gelijk de initialisatie van een pointer plaatsvinden. Deze definitie kan verwarrend zijn. In listing 8.5 wordt de pointer `pint` gedefinieerd en geïnitialiseerd met het adres van variabele `i`. Het betreft hier dus *geen* dereferentie.

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

Listing 8.5: *Definitie en initialisatie van een pointer.*

8.2 De NULL-pointer

In principe wijst een pointer naar een variabele (of anders: bevat het geheugenadres van een variabele). Om aan te geven dat een pointer niet naar een variabele wijst, kunnen we de *NULL-pointer* gebruiken. Let erop dat de NULL-pointer niet hetzelfde is als een niet-geïnitialiseerde pointer. Een NULL-pointer is een pointer waarin alle bits 0 zijn[2]. Een niet-geïnitialiseerde pointer heeft een willekeurige waarde[3]. In C is een preprocessor-macro (zie hoofdstuk 10) genaamd NULL te gebruiken om een pointer als NULL-pointer te initialiseren. De C-standaard schrijft voor voor dat NULL wordt gedefinieerd in `locale.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, `time.h`, en `wchar.h`. Slechts een van deze header-bestanden is noodzakelijk om NULL te definiëren.

```
#include <stdio.h>

int *p = NULL;   /* p is initialized as NULL-pointer */
```

Listing 8.6: *Definitie 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.

```
#include <stdio.h>

int *p = NULL;   /* p is initialized as NULL-pointer */

*p = *p + 1;     /* Oops, dereference of NULL-pointer! */
```

Listing 8.7: *Dereferentie van een NULL-pointer.*

Een vergelijking van twee NULL-pointers zal altijd `true` opleveren.

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

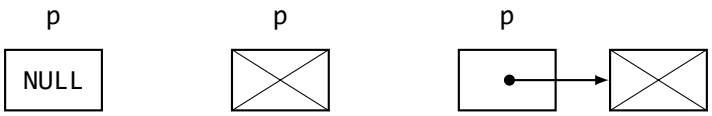
Listing 8.8: *Vergelijken van twee NULL-pointers.*

[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 gedefinieerd, zorgt de compiler ervoor dat de inhoud op NULL 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 (zie figuur 8.4). Bij de linker voorstelling wordt er 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 8.4: *Drie voorstellingen van NULL-pointers.*

8.3 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 &.

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

void *p;  /* void-pointer */

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

Listing 8.9: *Definitie 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 listing 8.10 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.

8.4 Afdrukken van pointers

Het afdrukken van de waarde van een pointer kan met de `printf`-functie en de format specification %p. De pointer mag elk type zijn, want hij wordt automatisch gecast naar `void *`. Zie listing 8.11 en paragraaf 9.2.

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

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

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

Listing 8.10: *Definitie, initialisatie en deference van een void-pointer.*

```c
#include <stdio.h>

int main() {

    int i = 2, *p = &i;

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

    return 0;
}
```

Listing 8.11: *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.

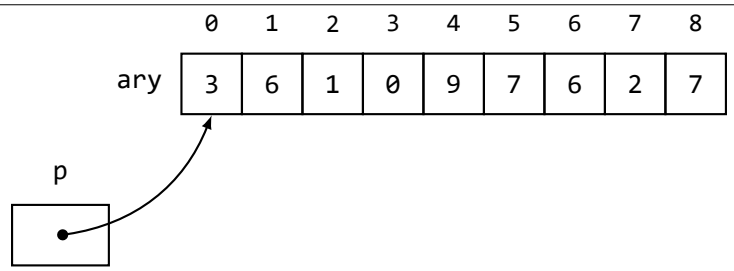
8.5 Pointers naar arrays

We kunnen een pointer ook laten wijzen naar het eerste element van een array. Dit is te zien in listing 8.12. 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.

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

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

Listing 8.12: *Een pointer naar het eerste element van een array.*

De uitbeelding hiervan is te zien in figuur 8.5. 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 8.13.



Figuur 8.5: *Uitbeelding van een pointer naar het eerste element van een array.*

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

int *p = ary; /* p points to first element of array */
```

Listing 8.13: *Een pointer naar het eerste element van een array.*

Omdat de naam van een array een synoniem is, mag de naam niet zomaar gebruikt worden aan de linkerkant van een toekenning. Zie listing 8.14.

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

p = ary;     /* correct use of pointer and array name */
ary[2] = *p; /* correct use of pointer and array element */

ary = p;     /* ERROR: array name cannot be used in this context */
```

Listing 8.14: *Een pointer naar het eerste element van een array.*

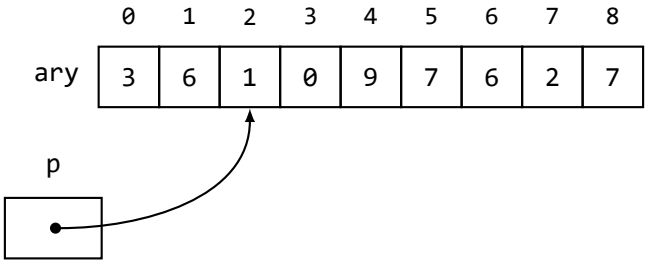
Het is ook mogelijk om een pointer naar een ander element van een array te laten wijzen. De compiler test niet of de toekenning binnen de array-grenzen ligt. Dit moet door de programmeur in de gaten worden gehouden. Zie listing 8.15.

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

int *p = &ary[2]; /* p points to third element of array */
```

Listing 8.15: *Een pointer naar het derde element van een array.*

Een uitbeelding is te zien in figuur 8.6. 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 8.8.

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

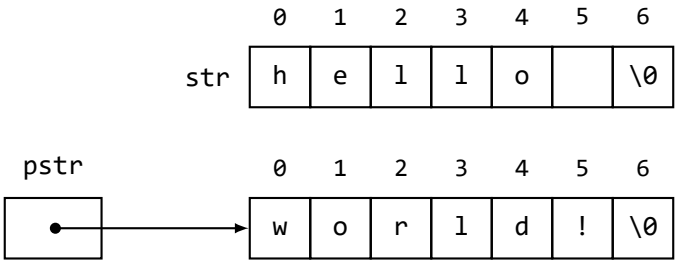
8.6 Strings

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 8.16 zijn twee definities met strings te zien, een echte array met een string als inhoud en een pointer naar een string in het geheugen.

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

Listing 8.16: *Definitie 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 8.7.



Figuur 8.7: *Voorstelling van twee C-strings.*

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

Toch is het mogelijk om tijdens het executie 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.

8.7 Rekenen met pointers

Pointers kunnen rekenkundig worden aangepast, dat vooral nuttig is bij het gebruik van arrays. 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. Door de pointer met 1 te verhogen wordt dus naar het volgende element gewezen.

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

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

Listing 8.17: *Rekenen met pointers.*

Een mooi voorbeeld van het rekenen met pointers is het bepalen van de lengte van een C-string. In listing 8.18 wordt pointer `str` gedefinieerd en wijst naar het begin van de string. Pointer `begin` wijst ook naar het begin van de string. Daarna verhogen we pointer `str` totdat het einde van de string is bereikt. Daarna drukken we het verschil van de twee pointers af.

```
#include <stdio.h>

int main() {

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

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

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

Listing 8.18: *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 programmafragment in listing 8.19 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 8.18 kunnen vervangen

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

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

Listing 8.19: *Het berekenen van het verschil van twee pointers.*

door het programmafragment in listing 8.20. Uiteraard moeten de twee pointers naar hetzelfde datatype wijzen en heeft de vergelijking alleen zin als de pointers naar elementen binnen dezelfde array wijzen.

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

Listing 8.20: *Vergelijken van twee pointers.*

8.8 Relatie tussen pointers en arrays

De relatie tussen pointers en arrays is zo sterk in C, dat we er een aparte paragraaf aan wijden. In listing 8.21 zijn de definitie en initialisatie van een array en een pointer te zien. De pointer p wijst na initialisatie naar het eerste element van de array.

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

Listing 8.21: *Definitie en initialisatie van een array en een pointer.*

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

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

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

Listing 8.22: *Bepalen van de som 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 dereferentie. 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:

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

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

Listing 8.23: *Bepalen van de som 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 we het derde element afdrukken door p als pointer of als array te beschouwen. Zie listing 8.24.

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

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

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

Listing 8.24: *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 8.25 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.

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

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

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

Listing 8.25: *Pointer die naar een element in een array wijst.*

8.9 Pointers als argumenten

Pointers zijn gewone variabelen en kunnen dus 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 8.26 is te zien hoe we een functie swap definiëren die de inhouden van twee variabelen verwisselt. Bij het aanroepen van de functie geven we de *adressen* mee van de te verwisselen variabelen. In de functie gebruiken we pointers om de inhouden van de variabelen te verwisselen.

```
1  #include <stdio.h>
2
3  void swap(int *pa, int *pb) {
4      int temp;
5
6      temp = *pa;
7      *pa = *pb;
8      *pb = temp;
9  }
10
11 int main(void) {
12
13     int a = 2, b = 3;
14
15     printf("a = %d, b = %d\n", a, b);
16
17     swap(&a, &b);
18
19     printf("a = %d, b = %d\n", a, b);
20
21     return 0;
22 }
```

Listing 8.26: *Een functie om de inhouden van twee variabelen te wisselen.*

In de functie swap is pa het adres van variabele a en is pb het adres van variabele b. In regel 17 van het programma geven we de adressen van de variabele mee middels de adres-oparator & (ampersand). De uitvoer van het programma is te zien in figuur 8.8.

Command Prompt

```
a = 2, b = 3
a = 3, b = 2
```

Figuur 8.8: *Verwisselen van twee variabelen.*

Een 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 variabelenamen van de twee strings als argumenten meegegeven. De variabelenaam van een string is immers een pointer naar het eerste karakter van de string. Het programma is te zien in listing 8.27.

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

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

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

int main() {

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

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

    return 0;
}
```

Listing 8.27: *Functie voor het kopieren van een string met behulp van pointers.*

Merk op dat de geheugenruimte voor de kopie groot genoeg moet zijn om de kopie op te slaan en dat de twee strings elkaar in het geheugen niet mogen overlappen. De standard library 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 8.27 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 arrays beschouwen. Zie listing 8.28. 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-karakter (`\0`).

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

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

    int i=0;

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

Listing 8.28: *Functie voor het kopieren van een string met behulp van arrays.*

listing 8.29. Let erop dat in `main` wél de grootte van de array kan worden uitgerekend, want daar wordt de array gedefinieerd. 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.

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

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

    int i;

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

int main(void) {

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

    printarray(list, length);

    return 0;
}
```

Listing 8.29: *Meegeven van de grootte van een array.*

Als we pointers gebruiken als argumenten, kunnen we ook gegevens *teruggeven* via die pointers. We geven een voorbeeld in listing 8.30. In de functie `gemiddeldeminmax` lezen we een aantal getallen in (aangegeven door de parameter `aantal`) en bepalen we het gemiddelde van de getallen. Tevens geven we via de pointers `min` en `max` het minimale en maximale ingevoerde getal terug. Op deze manier is het mogelijk om een functie meerdere gegevens terug te geven.

Merk op dat we in `main` wel variabelen definiëren maar niet initialiseren. Dat hoeft ook niet want de functie vult de gegevens in, enerzijds via de returnwaarde, anderzijds via de meegegeven pointers.

```c
#include <stdio.h>

#pragma warning(disable : 4996)

double gemiddeldeminmax(int aantal, double *min, double *max) {

    int teller;

    double som, getal;

    printf("Geef getal 1: ");
    scanf("%lf", &getal);
    som = *min = *max = getal;

    for (teller = 2; teller <= aantal; teller++) {
        printf("Geef getal %d: ", teller);
        scanf("%lf", &getal);

        som += getal;

        if (getal > *max) {
            *max = getal;
        } else if (getal < *min) {
            *min = getal;
        }
    }

    return som / aantal;
}

int main(void) {

    double mainmin, mainmax, maingem;

    printf("Geef 5 getallen.\n\n");

    maingem = gemiddeldeminmax(5, &mainmin, &mainmax);

    printf("Gemiddelde %lf, minimum %lf, maximum %lf\n",
                                    maingem, mainmin, mainmax);

    return 0;
}
```

Listing 8.30: *Bepalen van een gemiddelde, het minimum en het maximum.*

8.10 Pointer als returnwaarde

Een pointer kan ook als returnwaarde dienen. In listing 8.31 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 van de functie 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.

```
char *find_token(char* str, char ch) {

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

    while (*str != '\0') {   /* while not end of string */
        if (*str == ch) {    /* if character found ... */
            return str;      /* return pointer */
        }
        str++;
    }

    return NULL;             /* character not found */
}
```

Listing 8.31: *Pointer als return-waarde.*

Let goed op de definitie van de functie `find_token`. Vóór de functienaam is de dereferentie-operator geplaatst. Dit betekent dat de functie een pointer naar een karakter teruggeeft. Deze vorm wordt vaak verward met pointers naar functies. Zie paragraaf 8.16. De standard library bevat een implementatie van de functie onder de naam `strchr`.

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 inhoud van 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.

8.11 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 parameter. 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 parameterdefinitie. De compiler ziet er dan op toe dat in de functie geen members worden aangepast en als dat toch gebeurt, volgt een foutmelding. Zie listing 8.32 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 8.32: *Een functie om de gegevens van een artikel af te drukken.*

In regel 1 wordt een pointer naar een structure `artikel_t` gedefinieerd. 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
```

ervan uitgaat dat `a.nummer` een pointer is naar een `int` (en de variabele `a` is geen pointer). Omdat het gebruik van de haakjes niet zo duidelijk overkomt, is er een andere 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 we de functie `vul_artikel` in listing 8.33 gebruiken.

```
void vul_artikel(artikel_t *a, int nummer, char naam[],
                               int aantal, double prijs) {

    a->nummer = nummer;
    strcpy(a->naam, naam);
    a->aantal = aantal;
    a->prijs = prijs;
}
```

Listing 8.33: *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 8.34

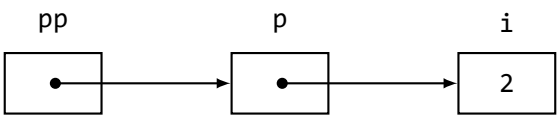
```
artikel_t *grootste_voorraad(artikel_t* a, artikel_t* b) {
    return (a->aantal > b->aantal) ? a : b;
}
```

Listing 8.34: *Een functie om grootste aantal artikelen te bepalen.*

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

8.12 Pointers naar pointers

Een pointer kan ook gebruikt worden om naar een andere pointer te wijzen. In figuur 8.9 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.



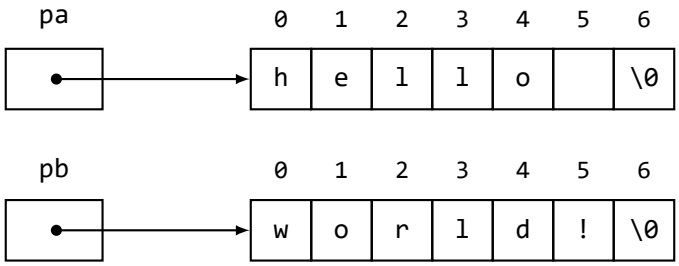
Figuur 8.9: *Uitbeelding van een pointer naar een pointer naar een `int`.*

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

Met behulp van pointers naar pointers kunnen de inhouden van twee pointers verwisselen. In figuur 8.10 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.

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

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

Listing 8.35: *Voorbeeld van een pointer naar een pointer.*



Figuur 8.10: *Uitbeelding van pointers naar strings.*

We kunnen dat doen met het programmafragment in listing 8.36. We definiëren drie pointers en laten pa en pb wijzen naar strings. We gebruiken de pointer `temp` om de verwisseling tot stand te brengen.

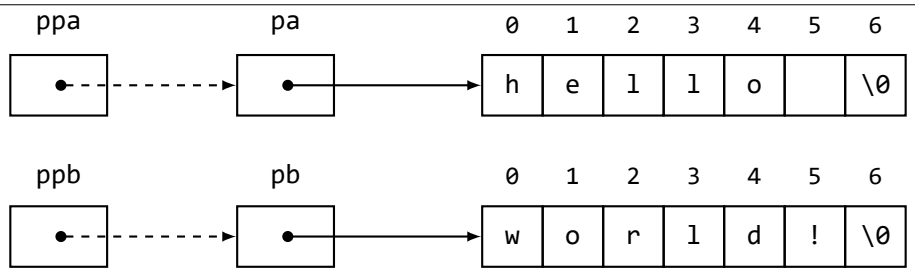
```c
char *pa = "hello ";
char *pb = "world!";
char *temp;

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

Listing 8.36: *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 8.11. De pointers pa en pb wijzen naar de strings. In de functie zijn twee pointers gedefinieerd die wijzen naar pointers naar strings. Dus ppa wijst naar pointer pa en ppb wijst naar pointer pb. We kunnen nu in de functie de inhouden van pa en pb verwisselen.

In listing 8.37 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 definiëren 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.

De uitvoer van dit programma is te zien in de figuur 8.12. 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 strings is veel efficiënter dan strings kopiëren.



Figuur 8.11: *Uitbeelding van pointers naar pointers naar strings.*

Command Prompt

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

Figuur 8.12: *Verwisselen van twee stings.*

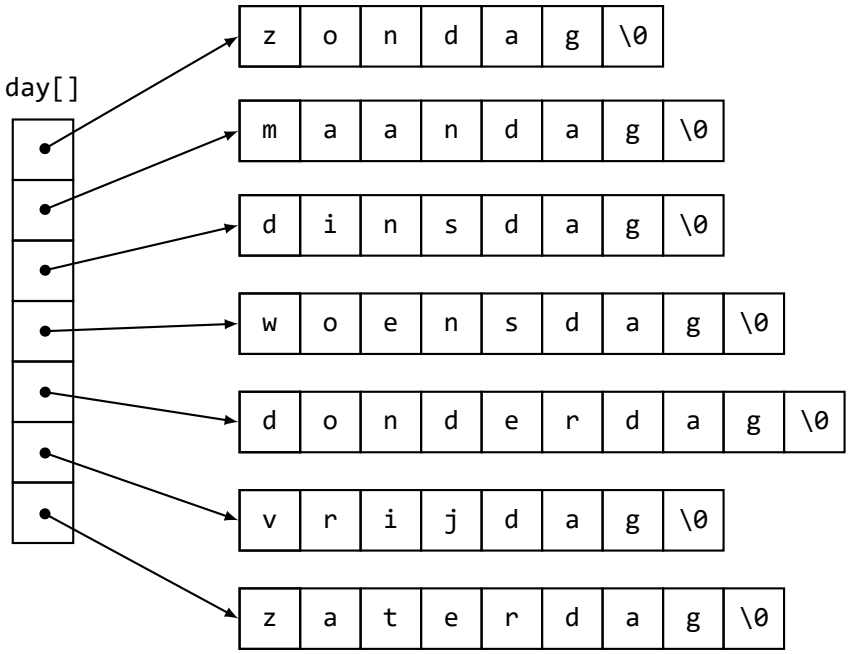
Merk op dat we in de functiedefinitie een *dubbele derefentie* opgeven maar in de functie zelf gebruiken we een enkele derefentie. De enkele derefentie is nodig om bij de pointers pa en pb te komen.

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

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

    char *temp;

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

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

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

Listing 8.37: *Functie voor het verwisselen van twee pointers.*

8.13 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 8.13. Merk op dat de zeven pointers naar karakters wijzen. Dat daar toevallig zeven strings aan gekoppeld zijn, is vanuit het perspectief van de pointers niet te zien (en in principe ook niet belangrijk).



Figuur 8.13: *Voorstelling van een array van pointers naar strings.*

We definiëren een array van zeven pointers naar karakters. Daarna laten we de pointers naar strings wijzen. Zie listing 8.38. Zoals gebruikelijk worden de strings afgesloten met een `\0`.

```
char *day[7];           /* array of seven pointers to char */

day[0] = "zondag";      /* points to the 'z' */
day[1] = "maandag";     /* points to the 'm' */
day[2] = "dinsdag";     /* points to the 'd' */
day[3] = "woensdag";    /* points to the 'w' */
day[4] = "donderdag";   /* points to the 'd' */
day[5] = "vrijdag";     /* points to the 'v' */
day[6] = "zaterdag";    /* points to the 'z' */
```

Listing 8.38: *Een array van pointers.*

Omdat dit soort toekenningen veel voorkomen, mogen de strings ook bij definitie aan de pointers worden toegekend. Dit is te zien in listing 8.39. Ook hier worden de strings afgesloten met een `\0`.

We kunnen de pointers nu gewoon gebruiken bijvoorbeeld bij het afdrukken van de strings:

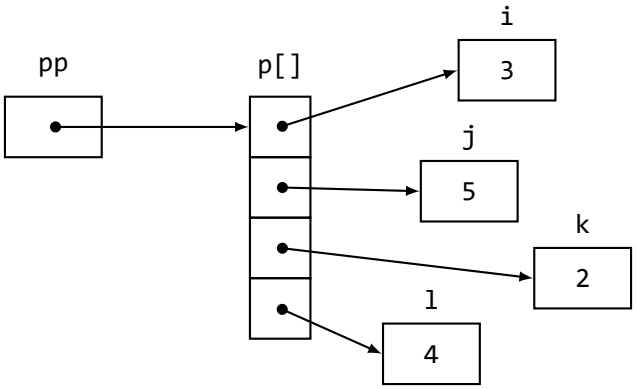
```
printf("The Dutch for Saturday is %s\n", day[6]);
```

```
char *day[] = {"zondag", "maandag", "dinsdag", "woensdag",
               "donderdag", "vrijdag", "zaterdag"};
```

Listing 8.39: *Een array van pointers naar strings met initialisatie.*

8.14 Pointers naar een array van pointers

We bekijken nu een wat complexer voorbeeld van het gebruik van pointers. Dit doen we ter voorbereiding op de volgende paragraaf. We definiëren een array p van vier pointers naar integers. We vullen de array met de adressen van de integers `i`, `j`, k en `l`. Daarna definiëren we een pointer pp die wijst naar (het eerste element van) de array. Een voorstelling van de variabelen is te zien in figuur 8.14.



Figuur 8.14: *Voorstelling van een pointer naar een array van pointers naar integers.*

De array p wordt gedefinieerd als:

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

De rechte haken hebben een hogere prioriteit dan de dereferentie-operator. We lezen de definitie dus als: p is een array van vier elementen en elk element is een een pointer die wijst naar een integer. De pointer pp wordt gedefinieerd 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 8.40. In regel 6 worden de vier integers gedefinieerd. In regel 7 definiëren 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 definiëren we de pointer pp (let op het gebruik van de dubbele dereferentie-operator) en initialiseren pp met het adres van p. In regel 10 en 11 drukken alle adressen van de pointers af. In de regels 13 t/m 16 drukken we de adressen van de integers en de inhouden van de array-elementen af. Om variabele `i` af te drukken hebben we drie mogelijkheden: `i` (de

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

int main(void) {

    int i=3, j=5, k=2, l=4;
    int *p[] = {&i, &j, &k, &l};
    int **pp = p;

    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]);

    printf("&i: %p, p[0]: %p\n", &i, p[0]);
    printf("&j: %p, p[1]: %p\n", &j, p[1]);
    printf("&k: %p, p[2]: %p\n", &k, p[2]);
    printf("&l: %p, p[3]: %p\n", &l, p[3]);

    printf("\ni: %d, *p[0]: %d, **pp: %d\n", i, *p[0], **pp);

    return 0;
}
```

Listing 8.40: *Voorbeeld van het gebruik van pointers.*

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

Command Prompt

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

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

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

Figuur 8.15: *Uitvoer van enkele pointers.*

Bij gebruik van een 64-bits compilers worden de pointers afgedrukt met 16 hexadecimale cijfers.

8.15 Argumenten meegeven aan een C-programma

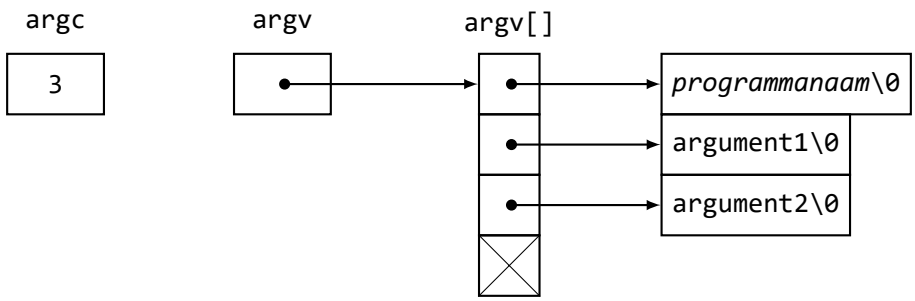
In besturingssystemen die C ondersteunen zoals Windows, Linux en OS-X, is het mogelijk om een C-programma *command line argumenten* mee te geven. Bij het starten van een programma kunnen we (optioneel) gegevens invoeren en overdragen aan een gecompileerd C-programma.

Elk C-programma krijgt per definitie de twee parameters `argc` en `argv` mee, die aan `main` worden meegegeven. Dit is te zien in listing 8.41.

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

    /* rest of the code */

    return 0;
}
```

Listing 8.41: *Definitie van de command line parameters.*

De integer `argc` (**arg**ument **c**ount) geeft aan hoeveel argumenten aan het C-programma zijn meegegeven. De pointer `argv` (**arg**ument **v**ector) is een pointer naar een lijst van pointers naar strings, gedefinieerd als `*argv[]`. Een voorstelling is gegeven in figuur 8.16.



Figuur 8.16: *Voorstelling van de variabelen argc en 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. De C-standaard schrijft voor dat de lijst van pointers naar strings wordt afgesloten met een NULL-pointer.

Omdat alle argumenten strings zijn, is het eenvoudig om deze argumenten af te drukken. Een voorbeeld is gegeven in listing 8.42. Het programma drukt eerst de variabele `argc` af. Met behulp van een `for`-statement worden de argumenten een voor een 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.

Als voorbeeld starten we het programma `commandlinearguments.exe` met de argumenten `hallo` en `wereld!`. Het programma drukt eenvoudigweg alle argumenten op het beeldscherm af. Te zien is dat ook de programmanaam als argument wordt meegegeven.

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

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

    printf("\nAantal argumenten: %d\n\n", argc);

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

    return 0;
}
```

Listing 8.42: *Afdrukken van de command line argumenten.*

Command Prompt

```
C:\Users\Cbook>commandlineargument s.exe hallo wereld!

Aantal argumenten: 3

Argument 0: commandlineargument s.exe
Argument 1: hallo
Argument 2: wereld!

C:\Users\Cbook>
```

Figuur 8.17: *Afdrukken van programmanaam en argumenten.*

In het voorbeeldprogramma is `argc` dus 3 en zijn `argv[0]`, `argv[1]` en `argv[2]` pointers naar respectievelijk `commandlineargument s.exe`, `hallo` en `wereld!`. De strings worden, zoals gebruikelijk in C, afgesloten met een nul-karakter.

Merk op dat `argv` een echte pointer is en niet de naam van een array. We mogen `argv` dus aanpassen. Dit is te zien in listing 8.43.

8.16 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 8.44 is te zien hoe een functie-pointer wordt gedefinieerd. In de eerste regel wordt een functie gedefinieerd, een zogenoemde prototype, die een `int` als argument meekrijgt en een `int` teruggeeft. In de tweede regel wordt `pf` gedefinieerd als een pointer naar een functie die een `int` meekrijgt en een `int` teruggeeft.

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

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

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

Listing 8.43: *Afdrukken van argumenten.*

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

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

Listing 8.44: *Een functie en een pointer naar een functie.*

Let op het gebruik van de haakjes. Die zijn nodig om prioriteiten vast te leggen. Zonder de haken staat er `int *pf(int a)` en dan is `pf` een functie die een `int` als parameters meekrijgt en een pointer naar een `int` teruggeeft. Zie listing 8.45.

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

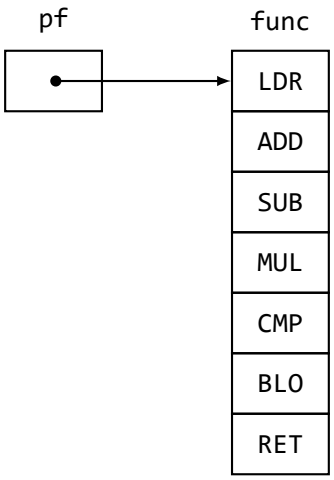
Listing 8.45: *Een functie en een pointer naar een functie.*

In figuur 8.18 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 een interessante functie de revue laten passeren waarbij een functie-pointer 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 behandelen, er zijn genoeg boeken die dit beschrijven. De standard library bevat een implementatie onder de naam `qsort`. Het prototype van `qsort` is:



Figuur 8.18: *Voorstelling van een pointer naar een functie.*

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

Hierin is `base` een pointer naar het eerste element van de array, `nitems` het aantal elementen in de array en `size` de grootte (in bytes) van één element. Het type `size_t` is in de regel gelijk aan een `int`. 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 `qsort` weet niet op voorhand wat het datatypes van de elementen van de array zijn. De functie `compar` *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 8.46 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 cast 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.

8.17 Dynamische geheugenallocatie

Bij het schrijven van een programma definiëren we de variabelen die we nodig hebben. Deze geheugenplaatsen blijven in principe bezet. Wel is het zo dat lokale variabelen worden aangemaakt en verwijderd aan het begin respectievelijk het einde van een functie (tenzij ze als `static` gekwalificeerd zijn), maar er komt geen extra geheugen bij. 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 functies 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

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

/* Function to sort elements */
int cmpint(const void *a, const void *b) {

    /* explicit type cast to pointers to integers */
    return ( *(int*)a - *(int*)b );
}

int main() {

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

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

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

    printf("\nDe array na sorteren: \n");

    for (int n = 0; n < 5; n++) {
        printf("%d ", values[n]);
    }

    return(0);
}
```

Listing 8.46: *Sorteren van een array met quicksort.*

WHAT'S IN THE NAME...

De schrijfwijze bij definities van pointers kan op twee manieren:

```c
int *p;
```

en

```c
int* p;
```

In beide gevallen is p en pointer naar een integer. De eerste manier sluit aan bij de gedachte "p in een pointer" (naar een `int`). De tweede manier sluit aan bij: p is een "pointer naar een int". Het gebruik van de tweede variant komt goed tot zijn recht bij listing 8.46. Daarin wordt een expliciete cast gedaan naar een pointer van het type `int` en die wordt weer gebruikt in een dereferentie naar een `int`.

begin van de geheugenruimte teruggegeven. De void-pointer moet dus gecast worden naar het juiste type. Het geheugen wordt niet geïnitialiseerd, de inhoud van de geheugenplaatsen is onbekend. 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 dat veroorzaakt over het algemeen een crash van het programma.

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, ruimt het besturingssysteem alle gealloceerde geheugenruimtes op. Maar als een programma alsmaar 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 8.47. Regel 5 zorgt ervoor dat Visual Studio de functie strcpy accepteert. In regel 8 definiëren we de constante SIZE met de waarde van 1000. De constante is van het type size_t. In regel 12 definiëren we een karakterpointer en initialiseren we die met NULL. Regel 14 roept de functie malloc aan om 1000 bytes te alloceren. De functie geeft een void-pointer terug, dus we moeten een type cast naar char * doen. We testen in de regels 16 t/m 19 of de allocatie gelukt is en zo niet, dan drukken we een foutmelding af en stoppen we het programma.

In regel 21 t/m 23 kopiëren we een string naar het geheugen en drukken het af. In regel 25 geven we het geheugen weer vrij. De lezer wordt gevraagd om het programma te testen met verschillende waarden voor SIZE.

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 geïnitialiseerd, 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 geïnitialiseerd, 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 alloceert een aantal veelvouden van een elementair datatype en vult de geheugenruimte met nul-bytes. Dus

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

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

```c
#include <stdio.h>
#include <malloc.h>
#include <string.h>

#pragma warning(disable : 4996)

/* Define 1000 bytes of space*/
const size_t SIZE = 1000;

int main(void) {

    char *pstr = NULL;

    pstr = (char *) malloc(SIZE);

    if (pstr == NULL) {
        printf("Kan geheugen niet alloceren!");
        return 0;
    }

    strcpy(pstr, "Vul maar wat in");

    printf("%s", pstr);

    free(pstr);

    return 0;
}
```

Listing 8.47: *Gebruik van `malloc` en `free`.*

8.18 Generieke geheugenfuncties

We hebben gezien dat strings met behulp van diverse functies zijn te manipuleren. Deze zijn echter bedoeld voor strings, dus karakters in het geheugen. C kent ook een aantal generieke geheugenfuncties die ook gebruikt kunnen worden bij andere datatypes. Zie tabel 8.1.

Tabel 8.1: *Enkele standaard functies voor geheugenmanipulaties.*

`memcpy(to, from, n)`	Kopieert vanaf `from` naar `to` met precies `n` bytes.
`memmove(to, from, n)`	Kopieert vanaf `from` naar `to` met precies `n` bytes, maar kan ook gebruikt worden als `to` en `from` elkaar overlappen.
`memcmp(s1, s2, n)`	Vergelijkt `s1` met `s2` met `n` bytes, returnwaarde als met `strcmp`.
`memset(s, c, n)`	Zet byte `c` op de eerste `n` bytes vanaf `s`.

Let erop dat de pointers van het type `void *` zijn en dat `n` het aantal geheugenplaatsen in bytes is.

* 8.19 Gekoppelde lijsten

We kunnen structures en pointers gebruiken voor het ontwikkelen van *gekoppelde lijsten*. We zullen in het vervolg echter gebruik maken van de Engelse term *linked lists*. De basis van een linked list wordt gevormd door een structure met daarin een pointer member die kan wijzen naar een andere structure van hetzelfde type. Zo'n structure wordt doorgaans een *node* genoemd. Laten we eens een lijst maken met leeftijden en namen. We declareren een node met daarin de member `age` voor de leeftijd en een member `name` voor de naam. Daarnaast definiëren we een *pointer* naar zo'n zelfde node met de naam `next`. De declaratie van de node is hieronder te zien.

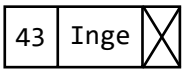
```
struct node {
    int age;
    char name[20];
    struct node *next;
};
```

Naast de twee eerder opgegeven members is nu ook een member `next` te zien die een pointer is naar zo'n zelfde node. In het boek van K&R wordt dit ook wel een *self-referential structure* genoemd. Het is met deze node mogelijk om te wijzen naar een *andere* node van hetzelfde type (maar in principe ook naar zichzelf).

We kunnen nu een node van dit type initialiseren:

```
struct node a = { 43, "Inge", NULL };
```

De member `next` wordt geïnitialiseerd met NULL, dat betekent dat `next` naar niets wijst. Een uitbeelding van de node is te zien in figuur 8.19. Het kruis geeft aan dat de member `next` naar niets wijst.



Figuur 8.19: *Uitbeelding van de node.*

We kunnen meerdere variabelen (nodes) aanmaken, zie hieronder:

```
struct node a = { 43, "Inge", NULL };
struct node b = { 18, "Jesscia", NULL };
struct node c = { 37, "Karin", NULL };
struct node d = { 14, "Renate", NULL };
```

We kunnen dit voorstellen met de afbeelding in figuur 8.20.

Figuur 8.20: *Uitbeelding van de vier nodes.*

Door middel van het statement

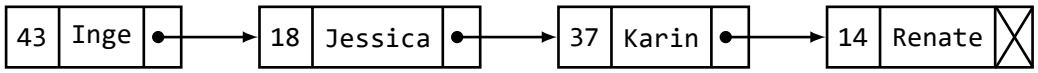
```
a.next = &b;
```

kunnen we de member `next` (de pointer) laten wijzen naar node `b`. De uitbeelding van dit statement is te zien in figuur 8.21.

Figuur 8.21: *Uitbeelding van de vier nodes, a wijst naar b.*

Natuurlijk kunnen we op dezelfde wijze de andere nodes ook naar de volgende in de linked list laten wijzen. Dit is te zien in de onderstaande listing. De uitbeelding is te zien in figuur 8.22.

```
a.next = &b;
b.next = &c;
c.next = &d;
```



Figuur 8.22: *Uitbeelding van de vier nodes, a wijst naar b, b wijst naar c en c wijst naar d.*

We hebben nu een *linked list* gemaakt waarbij we van een node naar een andere node kunnen "springen".

Laten we de linked list nu eens afdrukken. Daarvoor hebben we een pointer variabele nodig die we `current` noemen. Natuurlijk moet `current` in het begin wijzen naar het adres van `a`. We definiëren én initialiseren deze variabele als

```
struct node *current = &a;
```

Nu kunnen we de leeftijd en naam in `a` ook afdrukken met

```
current->age  en  current->name
```

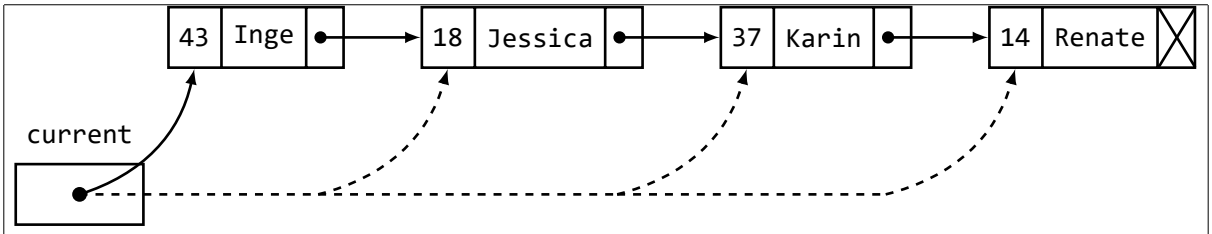
Om naar de volgende node te wijzen (node `b`) gebruiken we het statement

```
current = current->next;
```

Maar dat mag natuurlijk alleen als `current` ongelijk aan `NULL` is want dan is er geen volgende in de lijst. Om de hele linked list nu af te drukken gebruiken we een `while`-statement. Dit is te zien in de onderstaande listing.

```
/* While we're not at the end of the list */
while (current != NULL) {
    /* Print the data */
    printf("Age: %2d, name: %s\n", current->age, current->name);
    /* Point to the next in the list */
    current = current->next;
}
```

Met dit `while`-statement lopen we van het begin tot aan het einde van de linked list. Dit wordt in het Engels *traversal* (to traverse: langslopen) genoemd. In figuur 8.23 is de traversal van de pointer `current` te zien. Pointer `current` begint bij node `a`, gaat daarna naar `b` en daarna naar `c` en eindigt vervolgens bij `d`. Dat is het einde van de linked list, want `d` heeft geen opvolger, en dus stopt de `while`-lus. Het complete programma is te zien in listing 8.48.



Figuur 8.23: *Traversal van de pointer* `current`.

```
/* Needed includes */
#include <stdio.h>

/* The nodes in the list*/
struct node {
    int age;
    char name[20];
    struct node *next;
};

int main(void) {

    /* Populate the list */
    struct node a = { 43, "Inge", NULL };
    struct node b = { 18, "Jesscia", NULL };
    struct node c = { 37, "Karin", NULL };
    struct node d = { 14, "Renate", NULL };

    /* Let each item point to the next */
    a.next = &b;
    b.next = &c;
    c.next = &d;
    /* Except for d, which points to NULL */

    /* Variable to traverse the list */
    /* At first, point to structure a */
    struct node *current = &a;

    /* Loop trough the complete list and print data */
    while (current != NULL) {
        printf("Age: %2d, name: %s\n", current->age,
                                       current->name);
        /* Point to the next in the list */
        current = current->next;
    }

    return 0;
}
```

Listing 8.48: *Aanmaken van enkele nodes en de traversal van de nodes.*

* 8.19.1 Dynamisch gekoppelde lijsten

Nu biedt het opzetten van een linked list op deze manier niet veel meerwaarde. We hadden de nodes ook in een array kunnen plaatsen. Het aantal nodes staat namelijk vast.

Het wordt echter interessant wanneer we van te voren niet weten hoeveel nodes er gebruikt gaan worden. Met *dynamische geheugenallocatie* (zie paragraaf 8.17) is het mogelijk om tijdens de executie van een programma nieuwe nodes aan te maken en in de linked list op te nemen.

We gaan nu een programma ontwerpen dat de leeftijd en naam van personen inleest en deze gegevens in een linked list plaatst. Daarbij onderscheiden we de volgende functionaliteiten:

- Inlezen van leeftijd en naam;
- Aanmaken van een nieuwe node en vullen met de gegevens. Tevens wordt de nieuwe node opgenomen aan het einde van de linked list;
- Afdrukken van de complete linked list;
- Verwijderen van de complete linked list.

Voor het aanmaken van een nieuwe node ontwerpen we de functie `create_node`, het afdrukken van de linked list doen we met de functie `print_all_nodes` en voor het verwijderen van de linked list gebruiken we de functie `remove_all_nodes`. Het inlezen van de gegevens en het aanmaken van een nieuwe node gebeurt in een `do while`-lus. Als we klaar zijn met inlezen, drukken we de linked list af en verwijderen we de linked list.

Omdat we enkele functies uit de standard library gebruiken, laden we eerst de benodigde header-bestanden. Dit is te zien in het onderstaande programmafragment.

```
/* The includes needed */
#include <stdio.h>
#include <string.h>
#include <malloc.h>
```

De node die we gebruiken is hieronder gedefinieerd.

```
/* The node structure */
struct node {
    int age;             /* The age */
    char name[20];       /* The name */
    struct node *next;   /* Pointer to the next */
};
```

De linked list wordt dynamisch opgebouwd en om het begin van de linked list aan te geven, moeten we bijhouden waar het begin is. Dit wordt doorgaans de *head* van de linked list genoemd. We gebruiken hiervoor een pointer `head`. Deze pointer wijst dus altijd naar het begin van de linked list. In eerste instantie wijst de pointer naar niets, vandaar de initialisatie met `NULL`. Merk op dat `head` globaal gedefinieerd is zodat de functies de pointer kunnen gebruiken.

```
/* Pointer to the first node, commonly known as the head */
struct node *head = NULL;

/* Function to create and fill a new node */
struct node* create_node(int age, char name[]);
/* Function to print all nodes */
void print_all_nodes(void);
/* Function to remove all nodes */
void remove_all_nodes(void);
```

Omdat we drie functies gebruiken, declareren we de prototypes. Dit is te zien in onderstaande programmafragment.

Laten we nu eens kijken naar de opzet van het programma. We bekijken eerst de functie `main`. Dit is te zien in listing 8.49. Het inlezen van de gegevens wordt gedaan met behulp van een `do while`-lus die loopt van regel 8 t/m 20. Het stopcriterium is een leeftijd kleiner dan of gelijk

```
1  int main(void) {
2
3      int age;
4      char name[20];
5
6      /* Read in the data */
7      do {
8          printf("Give age (> 0): ");
9          scanf("%d", &age);
10         if (age <= 0) { break; }
11         printf("Give name: ");
12         scanf("%s", name);
13
14         /* Create new node, bail out if error */
15         if (create_node(age, name) == NULL) {
16             printf("Node creation failed, bailing out!\n");
17             return -1;
18         }
19
20     } while (age > 0);
21
22     /* Print all nodes */
23     print_all_nodes();
24
25     /* Remove all nodes */
26     remove_all_nodes();
27     return 0;
28 }
```

Listing 8.49: *De body van het programma.*

aan 0. Het inlezen wordt gedaan in de regels 8 t/m 12 middels de bekende `scanf`-functie. In regel 14 wordt de functie `create_node` aangeroepen met de leeftijd en naam als argumenten. Als het aanmaken *niet* lukt, dan wordt het programma met een foutmelding verlaten. Nadat de lus is verlaten, wordt de linked list afgedrukt met de functie `print_all_nodes`. Als laatste wordt de linked list verwijderd met `remove_all_nodes`.

De functie `create_node` is te zien in listing 8.50. De functie moet drie zaken afhandelen: dynamisch alloceren van geheugen om een nieuwe node aan te maken, het plaatsen van de node aan het einde van de linked list en het vullen van de gegevens in de nieuwe structure.

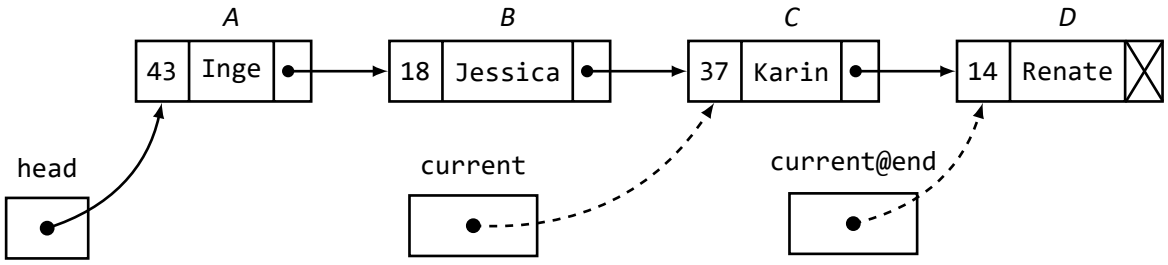
```
/* Function to create and fill a new node */
struct node *create_node(int age, char name[]) {

    struct node *created, *current;

    /* Create a new node */
    created = (struct node*) malloc(sizeof(struct node));

    /* Could we create a new node? No, return signalling NULL */
    if (created == NULL) {
        return NULL;
    }

    /* At first, there are no nodes, so head points to NULL */
    /* We test for it and set the head to the first node */
    if (head == NULL) {
        head = created;
    }
    else {
        /* There are already nodes, so find the last one */
        current = head;
        while (current->next != NULL) {
            current = current->next;
        }

        /* Let the last one point to the new node */
        current->next = created;
    }

    /* Fill the new node with data */
    created->age = age;
    strcpy(created->name, name);
    created->next = NULL;

    /* Return address of new node */
    return created;
}
```

Listing 8.50: *De functie* `create_node`.

In regel 7 wordt via de functie `malloc` een stuk geheugen gealloceerd. De grootte wordt bepaald met behulp van de operator `sizeof` zodat er precies genoeg geheugen wordt opgevraagd om de node te plaatsen. In regels 10 t/m 12 wordt getest of het alloceren gelukt is. Dat kan mislukken als er al teveel geheugen is opgevraagd. Dan geven we als returnwaarde `NULL` terug ten teken van het mislukken van de allocatie.

Als het alloceren van het geheugen gelukt is, bevat de pointer `created` het adres van de node. We moeten nu alleen nog bepalen of dit de eerste node in de linked list is. Zo ja, dan moet `head` gaan wijzen naar de nieuwe node. Dit wordt gerealiseerd in regels 16 t/m 18. Als er al één of meerdere nodes zijn aangemaakt, dan moeten we het einde van de linked list zien te vinden. Dit wordt gerealiseerd in regels 20 t/m 28.

Om goed te begrijpen hoe het vinden van het einde van de linked list werkt, hebben we een uitbeelding gemaakt van vier nodes. Dit is te zien in figuur 8.24. Hierbij hebben we de nodes de namen *A*, *B*, *C* en *D* gegeven. Node *A* ligt aan het begin van de linked list en node *D* ligt aan het einde van de linked list. In de figuur is te zien dat `head` wijst naar node *A*, de eerste in de linked list. De pointer `current` gebruiken we om langs de linked list te lopen. Uiteindelijk wordt het einde van de linked list gevonden. Dit is aangegeven met de (pseudo)-pointer `current@end`.



Figuur 8.24: *Het vinden van het einde van de linked list.*

In eerste instantie wordt pointer `current` gelijk gesteld aan `head`. Dat is namelijk het begin van de linked list. We schrijven dus:

```
current = head;
```

Zowel `head` als `current` wijzen dus naar node *A*. In de conditie van de `while`-lus kijken we of er een opvolger is. Om te bepalen of er een opvolger is, testen we de member `next` van de pointer `current`. Dus we schrijven:

```
while (current->next != NULL) {
    ...
}
```

Als er een opvolger is dan springen we naar de opvolger. Dit doen we door gebruik te maken van member `next`:

```
current = current->next;
```

Dus als `current` wijst naar *A* wordt de nieuwe waarde *A*`->next` oftwel *B*. De pointer `current` wijst na het uitvoeren van het statement naar *B*. Uiteindelijk is de laatste node in de linked list gevonden en wijst `current` naar node *D*. Dit is weergegeven met de (pseudo)-pointer `current@end`. Daarna laten we deze laatste node *D* wijzen naar de nieuwe node. Dat doen we met het statement

```
current->next = created;
```

Vervolgens moeten we de nieuwe node vullen met gegevens. We kopiëren de leeftijd en de naam naar de nieuwe node en zetten gelijk de member `next` op `NULL` want de nieuwe node heeft geen opvolger.

```
created->age = age;
strcpy(created->name, name);
created->next = NULL;
```

Als laatste statement geven we als returnwaarde het adres van `created` terug. Op zich is dat niet zo belangrijk maar we moeten onderscheid maken tussen `NULL` en een geldig adres, want in `main` wordt getest of de returnwaarde `NULL` is of niet.

```
return created;
```

De functie `print_all_nodes` is kort en krachtig. De functie is te zien in listing 8.51. De functie heeft geen parameters en geeft niets terug. We gebruiken hier een `while`-lus om langs alle nodes te gaan totdat we het einde van de linked list hebben bereikt. We beginnen met het initialiseren van de pointer `current` met `head`. Daar moeten we tenslotte beginnen. In de `while`-lus wordt getest of `current` niet naar `NULL` wijst. Dan is er nog een node waarvan de gegevens moeten worden afgedrukt. In de lus laten we `current` naar de volgende node wijzen.

```
/* Function to print all nodes in the list */
void print_all_nodes(void) {

    struct node *current;

    /* Traverse all nodes */
    current = head;
    while (current != NULL) {
        printf("Age: %2d, name: %s\n", current->age, current->name);
        current = current->next;
    }
}
```

Listing 8.51: *De functie `print_all_nodes`.*

Voordat het programma wordt afgesloten, moeten de gealloceerde nodes eerst weer worden vrijgegeven. Dat kan met de functie `remove_all_nodes`. De functie is te zien in listing 8.52. We beginnen met het verwijderen bij het begin van de linked list, dus waar de pointer `head` naar wijst. We verwijderen de node met de functie `free` en laten `head` naar de volgende in de linked list wijzen. We doen dit totdat de laatste node is verwijderd.

Er zit nog wel een bepaalde volgorde in het verwijderen van een node en het laten wijzen van `head` naar de volgende in de linked list. We moeten `head` eerst laten wijzen naar de volgende node en dan pas de (eerdere) node verwijderen. Dat kan natuurlijk niet met `head` want die wijst al naar de volgende node. Daarom gebruiken we een extra variabele `remnode` die naar de te verwijderen node wijst.

```c
/* Function to remove all nodes */
void remove_all_nodes(void) {

    struct node *remnode;

    while (head != NULL) {
        /* Make sure we point to the node to be deleted */
        remnode = head;
        /* Goto next in the list */
        head = head->next;
        /* Now we can safely remove the node */
        free(remnode);
    }
}
```

Listing 8.52: *De functie* `remove_all_nodes`.

Overigens kunnen we het verwijderen van de nodes in dit programma achterwege laten. Als het programma is afgesloten, wordt al het gealloceerde geheugen automatisch door het besturingssysteem vrijgegeven.

Een mogelijke invoer en uitvoer is hieronder gegeven.

Command Prompt

```
Give age (> 0): 54
Give name: Harry
Give age (> 0): 23
Give name: Klara
Give age (> 0): 45
Give name: Gerard
Give age (> 0): 28
Give name: Joyce
Give age (> 0): 0
Age: 54, name: Harry
Age: 23, name: Klara
Age: 45, name: Gerard
Age: 28, name: Joyce
```

Figuur 8.25: *Mogelijke invoer en uitvoer van het programma.*

We kunnen nog wat zeggen over de efficiëntie van enkele acties op de linked list. Als we een node toevoegen moeten we eerst het einde van de linked list bepalen. We moeten dus alle nodes aflopen. We noemen deze *tijdcomplexiteit* "van de orde n" en wordt geschreven $O(n)$. Als we een node net een specifieke inhoud moeten verwijderen (overigens niet in het voorbeeld gegeven) dan moeten we ten hoogste alle nodes aflopen. Ook deze actie is van de orde $O(n)$. Om dezelfde reden is opzoeken of wijzigen van een node ook van de orde $O(n)$.

Actie	Orde
Toevoegen	$O(n)$
Verwijderen	$O(n)$
Opzoeken	$O(n)$
Wijzigen	$O(n)$

We kunnen het toevoegen ook anders realiseren. In plaats van het einde van de linked list steeds opnieuw op te zoeken, kunnen een nieuwe node ook aan het *begin* van de linked list toevoegen. Dat begin is direct te vinden want daarvoor gebruiken we de pointer `head`. We maken een nieuwe node aan, laten de member `next` wijzen naar `head` en laten daarna `head` wijzen naar de nieuwe node. Nu is deze functie van de orde $O(1)$.

Overigens is de linked list niet gesorteerd. Dat heeft ook alleen maar zin als we de linked list gesorteerd willen afdrukken. Voor het opzoeken en verwijderen van een node maakt het niet veel uit.

Als we gebruik willen maken van een gesorteerde linked list, bijvoorbeeld van woorden, dan kunnen we beter gebruik maken van een *binaire boom* (Engels: binary tree). Tijdens het toevoegen van een node kan de sortering dan efficiënt worden gerealiseerd. De eerder gegeven acties zijn dan *gemiddeld* van de orde $O(\log n)$ en in het slechtste geval $O(n)$. Het valt echter buiten de scope van dit boek. Er is veel literatuur te vinden over binaire bomen.

8.20 Subtiele verschillen en complexe definities

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 definities.

`int *p`	`p`: pointer to int
`int **pp`	`pp`: pointer to pointer to int
`int ***ppp`	`ppp`: pointer to pointer to pointer to int
`int **pp[3]`	`pp`: array[3] of pointer to pointer to int
`char **argv`	`argv`: pointer to pointer to char
`char *argv[]`	`argv`: array[] of pointer to char
`int *list[5]`	`list`: array[5] of pointer to int
`int (*list)[5]`	`list`: pointer to array[5] of int
`int *(*list)[5]`	`list`: pointer to array[5] of pointer to int
`int *pf(void)`	`pf`: function (no parameters) returning a pointer to int
`int (*pf)(void)`	`pf`: pointer to function (no parameters) returning an int
`int *(*pf)(void)`	`pf`: pointer to function (no parameters) returning a pointer to int
`char (*(*x())[])(int)`	`x`: function returning pointer to array[] of pointer to function (with an int as parameter) returning char.
`char (*(*x[3])(int *))[5]`	`x`: array[3] of pointer to function (with a pointer to int as parameter) returning pointer to array[5] of char.

Via de website [17] kunnen complexe definities omgezet worden in leesbaar Engels.

9

Invoer en uitvoer

Invoer en uitvoer van gegevens is geen onderdeel van de taal C, maar C biedt via de standard library wel faciliteiten hiervoor. Twee functies zijn we al eerder tegengekomen: `printf` voor het afdrukken van gegevens op het beeldscherm en `scanf` voor het inlezen van gegevens van het toetsenbord. We zullen deze functies in detail bekijken.

In dit hoofdstuk kijken we verder dan het toetsenbord en beeldscherm. We zullen demonstreren hoe we *bestanden* op het bestandssysteem kunnen benaderen. Een bestand is een logische eenheid van informatie in het bestandssysteem. Zo kunnen we bestanden openen voor gebruik, we kunnen eruit lezen en we kunnen erin schrijven. Bestanden kunnen na gebruik gesloten worden.

Het gebruik van bestanden vergt veel interactie met het besturingssysteem. Dat wordt voor de gebruiker verhuld door het gebruik van een aantal functies. Die maken het mogelijk om in een C-programma eenvoudig met bestanden te werken.

9.1 Bestandsnamen en het bestandssysteem

Als we een bestand willen benaderen dan doen we dat door de *naam* van het bestand te gebruiken. De bestanden zijn hiërarchisch georganiseerd door middel van *mappen* (Engels: folders). Deze term is in gebruik geraakt op het Windows-besturingssysteem. Op Unix-achtige systemen, bijvoorbeeld Linux en OS-X, wordt gesproken van *directories*. Elk bestand valt onder een map en een map kan ook weer onder een map vallen. Willen we een bestand openen, dan moeten we de naam en eventuele mapnamen gebruiken. Het samenstel van een bestandsnaam en mapnamen wordt een *padnaam* genoemd. Een padnaam kan *absoluut* of *relatief* zijn.

Het bestandssysteem van een computer wordt gevormd door alle *schijven* die beschikbaar zijn. Dat kunnen naast vaste schijven ook SSD's, USB-sticks, SD-cards en externe schijven zijn. De term 'schijf' komt nog uit de tijd dat een disk werd opgebouwd uit cirkelvormige platte schijven (*platters genaamd*) waarop met behulp van een magnetisch materiaal gegevens (nullen en enen) werden opgeslagen.

Een padnaam is op Windows anders dan op Unix. Windows maakt gebruik van zogenoemde *schijven*. Een schijf wordt aangegeven met een letter tussen A t/m Z. Historisch gezien werden A en B gebruikt voor *floppy disks*. De eerste vaste schijf heeft de naam C. Een voorbeeld van een padnaam naar een map is te zien in figuur 9.1.

Command Prompt

```
Microsoft Windows [Version 10.0.18363.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Cbook>
```

Figuur 9.1: *Starten van een DOS-box.*

In de figuur zien we dat de *command line interpreter* is gestart. Op Windows is dat het programma `cmd.exe`. Dit programma is te starten door in de zoekbalk onderaan het scherm de naam van het programma in te typen. Na starten van het programma komen we in de *huidige map.* In dit geval is dat de map `C:\Users\Cbook`. De schijfletter wordt afgesloten met een dubbele-punt en we bevinden ons in de map Cbook die valt onder de map `Users`. De namen van de mappen worden gescheiden door een *backslash* (\). Dit is een voorbeeld van een *absolute padnaam*. Vanaf de schijf worden alle tussenliggende mappen en de bestandsnaam opgegeven.

We kunnen de inhoud van de map zien (de bestanden die eronder vallen) door het commando `dir` uit te voeren. Een mogelijke uitvoer is te zien in figuur 9.2.

Command Prompt

```
C:\Users\Cbook>dir
 Volume in drive C is Windows
 Volume Serial Number is 7C7B-7257

 Directory of C:\Users\Cbook

02/05/2020  08:59    <DIR>          .
02/05/2020  08:59    <DIR>          ..
02/05/2020  06:56    <DIR>          Desktop
14/04/2020  11:14    <DIR>          Documents
02/05/2020  09:28    <DIR>          Downloads
18/09/2018  19:12             1.113 bestand.txt
02/05/2020  08:59            36.714 _viminfo
               2 File(s)          37.827 bytes
               5 Dir(s)   68.727.808.000 bytes free

C:\Users\Cbook>
```

Figuur 9.2: *Uitvoer van het `dir`-commando.*

Om de inhoud van het bestand `bestand.txt` op het scherm af te drukken, gebruiken we het commando `type` en de bestandsnaam. Dit is te zien in figuur 9.3. We hebben nu niet de hele padnaam opgegeven, maar slechts een deel. Dit wordt een *relatieve padnaam* genoemd. De bestandsnaam wordt gevormd door de huidige map en de bestandsnaam.

Command Prompt

```
C:\Users\Cbook>type bestand.txt
Dit is wat tekst.
C:\Users\Cbook>
```

Figuur 9.3: *Afdrukken van een bestand.*

Let op het volgende: onder Windows worden mappen en bestanden gescheiden door de backslash, maar dat is C het begin van een *escape sequence*. Om een padnaam *in een string* op te geven moeten we een *dubbele backslash* (`\\`) gebruiken. We komen hier in dit hoofdstuk nog op terug.

Unix-systemen maken geen gebruik van schijfletters. In plaats daarvan kent Unix de *root directory* die wordt aangegeven met een *forward slash* (`/`) aan het begin van een bestandsnaam. Een bestandsnaam heeft dan het uiterlijk van

```
/home/cbook/bestand.txt
```

De mappen worden dus gescheiden door de *forward slash* en de eerste forward slash stelt de root directory voor. Dit geeft problemen als we een C-programma willen schrijven dat op beide besturingssystemen moet draaien. Gelukkig kunnen de bestandsfuncties *meestal* overweg met beide naamgevingen.

9.2 Uitvoer naar het beeldscherm

We hebben de functie `printf` al vaak gebruikt in onze voorbeelden. We zullen nu wat dieper ingaan op de aanroep. Er zijn heel veel opties mogelijk, we zullen alleen de meest gebruikte behandelen. De functie wordt aangeroepen met een *format string* gevolgd door 0 of meer (extra) argumenten. Het prototype is

```
int printf(char *formatstring, arg1, arg2, arg3, ... );
```

Technisch gezien is de format string het eerste argument maar we zullen `arg1` als eerste argument beschouwen. In de format string kunnen we gebruik maken van zogenoemde *format specifications* die informatie verschaffen over de argumenten die meegegeven worden. Een lijst van mogelijke format specifications is te zien in tabel 9.1.

De karakters `d`, `i`, `o`, `x`, `X` en `u` kunnen nog extra informatie meekrijgen die tussen `%` en het karakter staan. Een minteken (`-`) zorgt ervoor dat het getal tegen de linkerkant wordt afgedrukt. Een nul (`0`) zorgt ervoor dat voorloopnullen worden afgedrukt. Een getal anders dan 0 zorgt ervoor dat het argument met een minimum breedte wordt afgedrukt. Dus de specification `%04d` zorgt ervoor dat het argument met een minimum breedte van 4 karakters, rechts uitgelijnd en met voorloopnullen wordt afgedrukt. De karakters kunnen nog worden vooraf gegaan door `h` (`short int`), `l` of `L` (`long int`) of `ll` (`long long int`).

Tabel 9.1: *Lijst van format specifications voor `printf`.*

Karakter	Argumenttype
d, i	afdrukken van een `int`, decimale notatie
o	afdrukken van een `int`, octale notatie
x, X	afdrukken van een `int`, hexadecimale notatie, x levert a…f, X levert A…F
u	afdrukken van een `unsigned int`, decimale notatie
c	afdrukken van een `char` als karakter
s	afdrukken van een string, argument is een pointer naar een string
f	afdrukken van een `double` en `float` (zie tekst)
e, E	afdrukken van een `double`/`float` in wetenschappelijke notatie ($10^x \rightarrow$Ex)
p	afdrukken van een pointer, weergave hangt af van de implementatie
%	het karakter % wordt afgedrukt

De karakters `f`, `e` en `E` kunnen worden voorafgegaan door een minteken (`-`) (links uitgelijnd), een getal dat aangeeft hoeveel cijfers er moeten worden afgedrukt, een punt `.` dat een scheiding vormt tussen de breedte en de precisie, en een getal dat de precisie aangeeft. De specification `%6.2f` geeft dus aan: totaal 6 karakters (inclusief de punt) en 2 decimalen (cijfers na de punt). Afronding is zoals gebruikelijk in de wetenschappelijke wereld. Merk op dat `%f` gebruikt wordt voor het afdrukken van een `double` én een `float`. Dat komt omdat een `float` gepromoot wordt naar een `double` en de `double` wordt doorgegeven aan `printf`.

Het karakter `s` geeft aan dat het argument een *string* is. Het argument is dus een pointer naar een string. Het karakter mag worden voorafgegaan door een minteken (`-`) (links uitgelijnd) en een getal dat de breedte aangeeft.

Het karakter `p` geeft aan dat het argument een *pointer* is. De representatie op het scherm is afhankelijk van de implementatie. Een 32-bits pointer wordt afgedrukt als 8 hexadecimale cijfers, een 64-bits pointer wordt afgedrukt als 16 hexadecimale cijfers.

De functie `printf` geeft een return-waarde terug. Dit is het aantal karakters dat naar het beeldscherm is geschreven. Meestal doen we daar niets mee en negeren we de return-waarde. Let erop dat `printf` een complexe, grote functie is en niet geschikt is voor gebruik op microcontrollers. Een microcontroller heeft over het algemeen geen besturingssysteem en geen beeldscherm. De functie `printf` kan dan niet zinnig gebruikt worden.

9.3 Inlezen van het toetsenbord

De functie `scanf` is een veelzijdige functie voor het inlezen van variabelen. We zullen wat dieper ingaan op het gebruik hiervan. Een bekend voorbeeld is het inlezen van een decimaal geheel getal 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 *whitespaces* (spaties, tabs of newlines (enter-toets)). Het aantal whitespaces maakt dus niet uit; deze worden overgeslagen. De spaties in de format specifications mogen ook weggelaten worden. We kunnen ook andere datatypes inlezen zoals een `float` of een `char`:

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

In tabel 9.2 zijn de meeste format specifications te zien. Als er andere karakters in de format string voorkomen, dan moeten die letterlijk ingevoerd worden, behalve whitespaces want die worden overgeslagen.

Tabel 9.2: *Lijst van format specifications voor `scanf`.*

Karakter	Argumenttype
d	inlezen van een `int`, decimale notatie
o	inlezen van een `int`, octale notatie
x, X	inlezen van een `int`, hexadecimale notatie
u	inlezen van een `unsigned int`, decimale notatie
c	inlezen van een `char` als karakter
s	inlezen van een string, tot aan de eerste whitespace
f, e	inlezen van een `float` floating point getal
lf, le	inlezen van een `double` floating point getal
[...]	inlezen van een string, met karakters uit de set tussen de blokhaken
[^...]	inlezen van een string, met karakters uit de set *niet* tussen de blokhaken

De karakters d, o, u en x mogen voorafgegaan worden door een h (`short int`) of een `l` (`long int`). De karakters `f` en `e` moeten voorafgegaan worden door een `l` als een `double` moet worden ingelezen. Zonder de `l` wordt een `float` ingelezen.

Het inlezen van een string is problematisch. De specification `%s` leest een string in tot aan de eerste whitespace, dus ook tot de eerste spatie. Willen we een string inlezen met spaties dan gebruiken we de specification `[^\n]`. Let erop dat de string waarin de karakters moeten komen groot genoeg is. Functie `scanf` houdt daar geen rekening mee en een *buffer overflow* kan het gevolg zijn. Dit kan worden voorkomen door bij de format specification een maximale lengte op te geven:

```
scanf("%99s", str);
```

We kunnen de gebruiker dwingen om bepaalde karakters is te voeren. Stel dat we een datum willen inlezen en de onderdelen van de datum zijn gescheiden door een forward slash. Dat kunnen we doen met

```
scanf("%d/%d/%d", &day, &month, &year);
```

en moeten de getallen in de invoer ook gescheiden worden door forward slashes.

De functie `scanf` geeft als return-waarde het aantal correct ingelezen argumenten terug. Daar kunnen we zinnig gebruik van maken zoals te zien is in paragraaf 9.12.

De functie `scanf` kan voor problemen zorgen bij het inlezen van gegevens. We zullen hier nader op ingaan in paragraaf 9.12.

9.4 Bestanden openen voor gebruik

Zoals al eerder is geschreven, kent C een aantal functies om met bestanden in het bestandssysteem te werken. Voordat we met een bestand kunnen werken moeten we het *openen*. Openen wil zeggen dat we aan het besturingssysteem melden dat we het bestand willen gebruiken. In C is dat te realiseren met de functie `fopen`. De functie heeft het prototype

```
FILE *fopen(char *filename, char *filemode);
```

De functie krijgt een bestandsnaam en een open-modus mee en geeft een pointer naar een *file handle* terug. De `filename` is de naam van het bestand in een string. Een voorbeeld van een *absolute* bestandsnaam is:

```
C:\Users\Cbook\bestand.txt
```

De karakters na de punt worden de *extensie* genoemd. C maakt geen onderscheid tussen verschillende extensies. Het is meer voor de gebruiker een indicatie wat voor bestand het betreft. De extensie `txt` geeft aan dat het (vermoedelijk) om een tekstbestand gaat.

De *filemode* geeft aan hoe en waarvoor het bestand moet worden geopend. Let erop dat de filemode een string is. Er zijn diverse opties mogelijk. Zie tabel 9.3.

Tabel 9.3: *De open-modi van bestanden.*

mode	betekenis
`"r"`	bestand wordt geopend voor alleen lezen
`"w"`	bestand wordt geopend voor alleen schrijven; de inhoud wordt verwijderd
`"a"`	bestand wordt geopend voor alleen schrijven; schrijven gebeurt aan het einde van het bestand, originele inhoud wordt niet verwijderd
`"r+"`	bestand wordt geopend voor lezen en schrijven; originele inhoud wordt niet verwijderd
`"w+"`	bestand wordt geopend voor lezen en schrijven; de inhoud wordt verwijderd
`"a+"`	bestand wordt geopend voor lezen en schrijven; schrijven gebeurt aan het einde van het bestand, originele inhoud wordt niet verwijderd

Een filemode kan gevolgd worden door een b die aangeeft dat het om een *binary* bestand gaat. Op Unix-systemen heeft dat eigenlijk geen effect maar op Windows-systemen wel. Windows slaat standaard het einde van de tekstregel op met twee karakters: een *carriage return* en een *linefeed*. Bij het openen van een bestand zonder de b worden deze twee karakters in C vertaald naar één karakter: de newline. Bij gebruik van b wordt er geen vertaling gedaan en worden dus twee karakters ingelezen bij het einde van een regel. De b heeft dus vooral zin als we geen tekstbestanden openen.

Als return-waarde geeft `fopen` een pointer naar een *file handle* terug. Dit is een structure ergens in het geheugen van de computer waar de boekhouding van het bestand wordt bijgehouden. De structure is gedefinieerd in headerbestand `stdio.h`. De precieze indeling en werking is voor ons niet van belang, maar we moeten de pointer wel gebruiken als we met het bestand willen werken. Als het niet lukt om een bestand te openen, krijgen we een NULL-pointer terug. Redenen waarom het openen niet lukt zijn:

- Bestandsnaam is onbekend (bestand bestaat niet);

- Geen rechten om bestand te openen voor lezen;
- Geen rechten om bestand te openen voor schrijven;
- Bestandsnaam bevat illegale karakters;
- Bestand kan niet aangemaakt worden (schrijven, bijvoorbeeld vaste schijf is vol).

Noot: Windows en Linux heeft de onhebbelijke eigenschap om spaties toe te staan in bestandsnamen. Let daar goed op. Programma's kunnen hierdoor in de war raken omdat spaties ook kunnen worden gezien als scheider tussen karakterreeksen.

9.5 Bestand sluiten na gebruik

Nadat de acties op een bestand zijn uitgevoerd, moet een bestand gesloten worden. Dit geeft aan het besturingssysteem aan dat verdere acties op het bestand niet zullen worden gedaan. Sluiten van een bestand kan met

```
fclose(fp);
```

waarbij `fp` een file handle is van een eerder geopend bestand. Let erop dat `fp` naar een bestaande file handle moet wijzen anders volgt een foutmelding (of in het ergste geval een crash van het programma). Als een programma is afgelopen worden geopende bestanden automatisch afgesloten.

9.6 Schrijven naar een bestand

Schrijven naar een bestand kan met de functie `fprintf`. De functie heeft het prototype:

```
int fprintf(FILE *fp, char *format, arg1, arg2, arg3, ... );
```

MICROSOF DOET HET ANDERS ...

De functie `fopen` wordt door Microsoft gezien als onveilig. Compilatie met Visual Studio eindigt dan met een foutmelding. In plaats daarvan biedt Visual Studio de functie `fopen_s`. De argumenten die meegegeven moeten worden zijn anders dan bij `fopen`. De functie heeft het prototype:

```
errno_t fopen_s(**pfp, char *name, char *mode);
```

Waarbij `pfp` een *pointer* is naar een pointer naar een `FILE`-structure. De parameter `name` is de bestandsnaam en `mode` is de bestandsmodus. De modus kent meer mogelijkheden dan bij `fopen`. De return-waarde is van het type `errno_t` en geeft de conditie aan waarmee het bestand is geopend (of niet).

Helaas wordt `fopen_s` niet ondersteund door andere compilers en besturingssystemen. We kunnen de foutmelding uitschakelen met een *pragma*:

```
#pragma warning(disable : 4996)
```

aan het begin van de C-programma zorgt ervoor dat `fopen` gebruikt kan worden.

Deze functie lijkt erg veel op `printf` en heeft, naast de `FILE`-pointer, exact dezelfde opzet. Bij het schrijven moeten we dus een `FILE`-pointer meegeven en een format string, precies zoals ook gebruikt wordt in `printf`, en natuurlijk de argumenten. In listing 9.1 is een compleet programma te zien voor het openen, schrijven en sluiten van een bestand.

```
1  #include <stdio.h>
2
3  // Keep Visual Studio happy
4  #pragma warning(disable : 4996)
5
6  int main(void) {
7
8      FILE *fp;
9      int a = 3;
10
11     fp = fopen("C:\\Users\\Cbook\\bestand.txt", "w");
12
13     if (fp == NULL) {
14         printf("Bestand kan niet worden geopend!");
15         return -1;
16     }
17
18     fprintf(fp, "Dit is wat tekst: %d\n", a);
19
20     fclose(fp);
21
22     printf("Bestand is geschreven\n");
23
24     return 0;
25 }
```

Listing 9.1: *Een bestand openen voor schrijven.*

In regel 8 wordt de pointer `fp` naar een `FILE`-structure gedefinieerd. Deze pointer wordt gebruikt in opvolgende bestandshandelingen. In regel 11 openen we het bestand voor schrijven. De return-waarde is een adres naar een `FILE`-structure en die kennen we toe aan de pointer `fp`. In regel 13 testen we of de pointer gelijk is aan `NULL`. Dat betekent namelijk dat het bestand niet kon worden geopend. We eindigen dan met een melding naar de gebruiker en sluiten het programma af met een `return`-statement. We geven als return-waarde `-1` mee, zodat het besturingssysteem hier iets mee kan doen (over het algemeen negeert het besturingssysteem die waarde). In regel 18 schrijven we een stukje tekst en de waarde van een variabele naar het bestand. In principe geeft `fprintf` het aantal karakters dat geschreven is terug. We kunnen dan testen of het schrijven ook daadwerkelijk gelukt is. In dit geval negeren we de return-waarde. In regel 20 wordt het bestand vervolgens gesloten. Na deze regel is het niet meer mogelijk om het bestand via pointer `fp` te benaderen.

Let erop dat op Windows-systemen de namen van de mappen en het bestand gescheiden worden door een dubbele backslash `\\`. Een enkele backslash wordt gezien als het begin van escape sequence zoals `\n`. Dat zal problemen opleveren bij het openen van een bestand.

9.7 Lezen uit een bestand

Nu we een bestand hebben geschreven, kunnen we het ook openen voor lezen. Dit is te zien in listing 9.2. Aan de functie `fopen` geven we weer de bestandsnaam mee en het argument `"r"` om aan te geven dat we willen lezen. In de regels 11 t/m 14 testen we of het openen is gelukt.

```
1  #include <stdio.h>
2
3  #pragma warning(disable : 4996)
4
5  int main(void) {
6
7      FILE* fp;
8      char ch;
9
10     fp = fopen("C:\\Users\\Cbook\\bestand.txt", "r");
11
12     if (fp == NULL)     {
13         printf("Bestand kan niet geopend worden\n");
14         return -1;
15     }
16
17     while (fscanf(fp, "%c", &ch) == 1) {
18         printf("%c", ch);
19     }
20
21     fclose(fp);
22
23     return 0;
24 }
```

Listing 9.2: *Een bestand openen voor lezen.*

Het inlezen en afdrukken van afzonderlijke karakters gebeurt in regels 16 t/m 18. Met behulp van de functie `fscanf` lezen we steeds één karakter in de variabele `ch` in, en drukken dat vervolgens af op het scherm. Aan het einde sluiten we het bestand weer.

De functie `fscanf` werkt hetzelfde als de functie `scanf` maar ook hier moet weer een pointer naar een `FILE`-structure worden meegegeven. De return-waarde van `fscanf` geeft aan hoeveel gegevens correct zijn geconverteerd. In het geval van het programma is dat de waarde 1. Een ander getal geeft aan dat het niet gelukt is om een karakter in te lezen en wordt de lus afgebroken. Daarna sluit het programma het bestand.

9.8 Standaard invoer en uitvoer

Elk uitvoerbaar (C-)programma krijgt per definitie drie geopende bestanden mee:

- `stdin`: Dit is een pointer naar het geopende bestand dat gekoppeld is aan de *standaard invoer*, in de regel het toetsenbord;
- `stdout`: Dit is een pointer naar het geopende bestand dat gekoppeld is aan de *standaard uitvoer*, in de regel het beeldscherm;
- `stderr`: Dit is een pointer naar het geopende bestand dat gekoppeld is aan de *standaard foutuitvoer*, in de regel het beeldscherm.

Deze bestanden hoeven niet expliciet geopend te worden; ze bestaan al als een uitvoerbaar (C-)programma wordt gestart. We kunnen de variabelen `stdin`, `stdout` en `stderr` gewoon gebruiken om karakters in te lezen of weg te schrijven. Sterker nog, `printf` en `scanf` zijn niets anders dan aanroepen naar de bestandsvarianten:

```
scanf(format, arg1, ...)   →   fscanf(stdin, format, arg1, ...)
printf(format, arg1, ...)   →   fprintf(stdout, format, arg1, ...)
```

9.9 Andere bestandsfuncties

De standard library kent een groot aantal functies met betrekking tot het afhandelen van bestanden. We zullen hier een aantal van behandelen:

- `int fflush(FILE *fp)`: zorgt ervoor dat de *uitvoerbuffers* in het geheugen van de computer worden geschreven naar het bestandssysteem (to flush = doorspoelen). Geeft een EOF terug als er een schrijffout optreedt. Op een bestand dat geopend is voor lezen is de werking van deze functie *ongedefinieerd*.
- `int fgetc(FILE *fp)`: leest één karakter in van het bestand dat gekoppeld is aan `fp`, of een EOF als het einde van een bestand is bereikt.
- `char *fgets(char *str, int n, FILE *fp)`: leest vanuit het bestand dat gekoppeld is aan `fp` ten hoogste `n` karakters en plaatst dat in de string `str`. Het inlezen stopt als een newline is gevonden (de newline komt in de string terecht), en de string wordt afgesloten met een nul-karakter (`'\0'`). Geeft een NULL-pointer terug als een fout is opgetreden.
- `int fputc(int ch, FILE *fp)`: schrijft één karakter naar het bestand dat gekoppeld is aan `fp`, geeft EOF terug als er een fout is opgetreden.
- `int fputs(char *str, FILE *fp)`: schrijft naar het bestand dat gekoppeld is aan `fp` de string `str`. Geeft EOF terug als er een fout is opgetreden.
- `int puts(char *str)`: is equivalent aan `fputs(str, stdout)`.
- `int getchar(void)`: is equivalent aan `fgetc(stdin)`.
- `int putchar(int ch)`: is equivalent aan `fputc(ch, stdout)`.
- `int fseek(FILE *fp, long offset, int origin)`: zet voor het lezen of schrijven de positie binnen een bestand. De `origin` mag een van de drie waarden hebben: `SEEK_SET` (begin van het bestand), `SEEK_CUR` (huidige positie binnen het bestand) of `SEEK_END` (einde van het bestand). Geeft een getal anders dan 0 terug bij een fout.

- `void rewind(FILE *fp)`: equivalent aan `fseek(fp, 0L, SEEK_SET)` maar heeft geen return-waarde. Zet de positie op het begin van een bestand. Eventuele foutindicatoren worden gewist.

Veel van deze functies krijgen een karakter mee als een `int`. Deze `int` wordt voor gebruik eerst nog geconverteerd naar een `unsigned char`. Diverse functies geven een `int` terug als karakter. De reden daarvoor is dat de functie ook een *end-of-file* (EOF) kunnen teruggeven. De EOF is meestal gedefinieerd als het getal -1.

Er bestaat ook nog een functie `gets` die een regel van het toetsenbord inleest. Het wordt afgeraden deze functie te gebruiken omdat er geen lengte van de string kan worden opgegeven, dat tot gevolg kan hebben dat meer karakters worden ingelezen dan er in de string beschikbaar zijn. Dat kan leiden tot een *buffer overflow*.

9.10 Lezen uit en schrijven naar een string

Naast de bekende `printf`- en `scanf`-functies kent de standard library nog drie andere nuttige functies: `sscanf`, `sprintf` en `snprintf`. Dit zijn varianten die een string gebruiken als invoer- en uitvoermedium.

De functie `sscanf` leest, net als `scanf`, gegevens in, maar nu uit een string. Om deze functie te gebruiken moet een pointer naar een string worden opgegeven. Dus om een integer uit een string in te lezen moeten we het volgende statement gebruiken.

```
sscanf(string, "%d", &getal);
```

De functie `sprintf` drukt, net als `printf`, karakters af, maar de karakters worden nu geplaatst in een string. Om een integer in een string te plaatsen gebruiken we

```
sprintf(string, "%d", getal);
```

Bij `sprintf` kan een probleem optreden als de grootte van de string te klein is voor het resultaat. Het aantal te schrijven karakters is dan groter dan de grootte van de string. Dit leidt tot een *buffer overflow*. Om ervoor te zorgen dat de string niet overschreven wordt, geven we bij de functie `snprintf` de lengte van de string op

```
snprintf(string, sizeof buffer, "%d", getal);
```

Bij deze lengte is het afsluitende nul-karakter meegenomen.

In listing 9.3 is een programma te zien dat een datum inleest. De datum moet bestaan uit een dag, een maand en een jaar, allemaal als decimaal getal ingegeven. De onderdelen van een datum mogen gescheiden zijn door spaties, slashes (`/`) of mintekens (`-`). In regel 13 lezen we een regel tekst met behulp van `fgets`. We gebruiken als invoerbestand `stdin`, een `FILE`-pointer die gekoppeld is aan het toetsenbord. We geven tevens de grootte van de string mee, zodat er niet teveel karakters worden ingelezen. Daarna wordt met behulp van `sscanf`-functies getest of aan een van de drie formaten wordt voldaan. Als aan een van de formaten wordt voldaan, dan worden de gegevens een in string geplaatst met behulp van `sprintf`. Daarna wordt de string met behulp van `puts` afgedrukt.

Overigens kan het gebruik van `puts` vermeden worden door `printf` te gebruiken in plaats van `sprintf`.

```c
#include <stdio.h>

// Keep Visual Studio happy
#pragma warning(disable : 4996)

int main(void) {

    char regel[100];
    int dag, maand, jaar;
    int valid = 0;

    printf("Geef een datum op: ");
    fgets(regel, sizeof regel, stdin);

    if (sscanf(regel, "%d %d %d", &dag, &maand, &jaar) == 3) {
        valid = 1;
    }
    else if (sscanf(regel, "%d/%d/%d", &dag, &maand, &jaar) == 3) {
        valid = 1;
    }
    else if (sscanf(regel, "%d-%d-%d", &dag, &maand, &jaar) == 3) {
        valid = 1;
    }

    if (valid) {
        sprintf(regel, "Geldige datum: %d %d %d\n",
                                                dag, maand, jaar);
    }
    else {
        sprintf(regel, "Ongeldige datum opgegeven\n");
    }

    puts(regel);

    return 0;
}
```

Listing 9.3: *Inlezen van een geldige datum met behulp van* `sscanf`.

9.11 Gebruik van command line argumenten en bestanden

We kunnen command line argumenten (zie paragraaf 8.15) gebruiken om aan het programma bestandsnamen op te geven. Daarmee kunnen we, als voorbeeld, een programma ontwerpen dat een bestand kopieert naar een ander bestand. Het complete programma is te zien in listing 9.4. Een groot gedeelte van het programma wordt gebruikt om te testen of de gebruiker wel correcte argumenten heeft meegegeven. Zo moeten er minstens twee (extra) argumenten worden meegegeven: het invoerbestandsnaam en het uitvoerbestandsnaam. De test hiervoor is te zien in regels 16 t/m 19. Als er geen argumenten zijn meegegeven dan sluiten we af met

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

#pragma warning(disable : 4996)

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

    /* The input and output file handlers */
    FILE* infile, * outfile;

    /* Character to read and character count */
    int ch, count = 0;

    /* We need exactly 3 arguments */
    if (argc != 3) {
        printf("Usage: %s infile outfile\n", argv[0]);
        return -1;
    }

    /* Test if input file and output file have the same name */
    if (strcmp(argv[1], argv[2]) == 0) {
        printf("Filenames cannot be the same\n");
        return -2;
    }

    /* Open the input file in binary mode, exit if error */
    infile = fopen(argv[1], "rb");
    if (infile == NULL) {
        printf("Cannot open input file %s\n", argv[1]);
        return -3;
    }

    /* Open the output file in binary mode, exit if error */
    outfile = fopen(argv[2], "wb");
    if (outfile == NULL) {
        printf("Cannot open output file %s\n", argv[2]);
        fclose(infile);
        return -4;
    }

    /* Copy characters from input file to output file */
    while ((ch = fgetc(infile)) != EOF) {
        count++;
        fprintf(outfile, "%c", ch);
    }

    printf("Copied %d characters\n", count);

    fclose(infile);
    fclose(outfile);

    return 0;
}
```

Listing 9.4: *Programma voor het kopiëren van een bestand.*

een korte beschrijving van het gebruik van het programma en geven we als return-waarde −1 terug.

De bestandsnamen mogen ook niet hetzelfde zijn want anders overschrijven we het invoerbestand. Dit wordt getest in regels 22 t/m 25. We geven de return-waarde −2 terug als de bestandsnamen hetzelfde zijn.

In regel 28 openen we het invoerbestand. We gebruiken hiervoor de modus `"rb"` zodat het bestand wordt geopend als *binary* bestand. Dat betekent dat er geen conversie plaatsvindt tussen *end-of-line* karakters. Als het bestand niet geopend kan worden, sluiten we het programma af met een foutmelding (return-waarde −3).

In regel 35 openen we het uitvoerbestand. Lukt dat niet dan stoppen we het programma en geven als return-waarde −4 terug. We sluiten tevens het invoerbestand.

Vanaf regel 43 wordt het daadwerkelijke kopiëren geregeld. We lezen met behulp van de functie `fgetc` een karakter uit het invoerbestand en schrijven dat naar uit het uitvoerbestand. We houden bij hoeveel karakters zijn ingelezen en weggeschreven. Aan het einde worden het invoerbestand en het uitvoerbestand afgesloten en het aantal gekopieerde karakters wordt afgedrukt.

9.12 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:

```
#include <stdio.h>

int main(void) {
    int getal = -12345;

    printf("Geef een geheel getal: ");
    scanf("%d", &getal);

    printf("Het getal is %d.\n", getal);

    return 0;
}
```

Listing 9.5: *Getal inlezen met `scanf`.*

Als we dit programma uitvoeren en als invoer het woord `Hallo` intypen dan verschijnt de volgende uitvoer (zie figuur 9.4).

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. Zie listing 9.6.

Command Prompt

```
Geef een geheel getal: Hallo
Het getal is -12345.
```

Figuur 9.4: *Invoer van een heel getal.*

```c
int main(void) {
    int getal = -12345;

    do {
        printf("Geef een geheel getal groter dan 0: ");
        scanf("%d", &getal);
    } while (getal < 1);

    printf("Het getal is %d.\n", getal);
    return 0;
}
```

Listing 9.6: *Het inlezen van een geheel getal dat groter is dan 0.*

Als we nu `Hallo` invoeren dan wordt de uitvoer zoals te zien in in figuur 9.5.

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 9.5: *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 cijfer. 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 9.6 het getal −12345 af. Het programma blijft in een `do-while`-statement 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 kunnen we 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). We kunnen dat als volgt doen:

```
#include <stdio.h>

int main(void) {

    int getal = -12345;
    int ret;

    printf("Geef een geheel getal groter dan 0: ");
    ret = scanf("%d", &getal);

    if (ret == 1) {
        printf("Het getal is %d.\n", getal);
    } else {
        printf("Geen getal ingevoerd!\n");
    }

    return 0;
}
```

Listing 9.7: *Inlezen van een getal groter dan 0.*

De uitvoer wordt dan:

Command Prompt

```
Geef een geheel getal groter dan 0: Hallo
Geen getal ingevoerd!
```

Figuur 9.6: *Uitvoer van het programma.*

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 programma in listing 9.8. Nog iets: als

er iets misgegaan is met het inlezen anders dan conversieproblemen, dan geeft de functie de integer waarde EOF (*end of file*) terug. We hebben EOF nader behandeld in paragraaf 9.9.

```
#include <stdio.h>

int main(void) {
    int getal, ret;

    do {
        printf("Geef een geheel getal: ");
        ret=scanf("%d", &getal);
        if (ret == 0) {
            printf("Dat was geen getal!\n");
            printf("Maar het karakter %c.\n", getchar());
        }
        else if (ret == EOF) {
            printf("Er is een fout opgetreden bij het lezen!\n");
        }
    } while (ret != 1);

    printf("Het getal is %d.\n", getal);
    return 0;
}
```

Listing 9.8: *Inlezen van een geheel getal.*

De functie `getchar` leest één karakter uit de invoerbuffer. Daarna proberen we `scanf` opnieuw. Dat mislukt telkens als er een letter gevonden wordt. Als we nu `Hallo` invoeren, krijgen we de volgende uitvoer zoals te zien is in figuur 9.7.

Command Prompt

```
Geef een geheel getal: Hallo
Dat was geen getal!
Maar het karakter H.
Geef een geheel getal: Dat was geen getal!
Maar het karakter a.
Geef een geheel getal: Dat was geen getal!
Maar het karakter l.
Geef een geheel getal: Dat was geen getal!
Maar het karakter l.
Geef een geheel getal: Dat was geen getal!
Maar het karakter o.
Geef een geheel getal:
```

Figuur 9.7: *Uitvoer van het programma.*

Natuurlijk willen we 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 gebrui-

ker wordt gevraagd om een geheel getal in te voeren. Anders weten we niet wat we moeten doen. Helaas wordt deze regel herhaaldelijk afgedrukt. We zullen op een iets andere manier de karakters moeten inlezen.

Gelukkig zorgt het besturingssysteem ervoor dat ook het end-of-line-karakter (\n, de enter-toets) in de invoerbuffer terecht komt. We kunnen dus hierop testen. Hoe dat moet, is te zien in de onderstaande code:

```
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, zie listing 9.9.

```c
#include <stdio.h>

void purge_stdin(void) {
    int ch;

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

int main(void) {
    int getal, ret;

    do {
        printf("Geef een geheel getal: ");
        ret = scanf("%d", &getal);
        if (ret == 0) {
            purge_stdin();
        }
    } while (ret != 1);

    printf("Het getal is %d.\n", getal);
    return 0;
}
```

Listing 9.9: *Inlezen van een geheel getal.*

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

```
fflush(stdin);
```

Als parameter van `fflush` wordt `stdin` opgegeven. Dat staat voor *standard input* en daar wordt in de regel het toetsenbord mee bedoeld (zie 9.8).

Deze manier werkt echter niet met alle C-compilers en besturingssystemen. 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). Het werkt *niet* met de Microsoft C-compiler.

Het programma is te vinden in listing 9.10.

```
/* Please note: might not work on all Operating Systems. */

#include <stdio.h>

int main(void) {
    int getal, ret;

    do {
        printf("Geef een geheel getal: ");
        ret = scanf("%d", &getal);
        if (ret == 0) {
            fflush(stdin);
        }
    } while (ret != 1);

    printf("Het getal is %d.\n", getal);
    printf("Druk op de Enter toets om dit window te sluiten.");
    fflush(stdin);
    getchar();
    return 0;
}
```

Listing 9.10: *Leegmaken van de invoerbuffer van `stdin`.*

10

De preprocessor

De C-compiler is uitgerust met een *preprocessor*. Dit kan een apart programma zijn dat door de C-compiler zelf gestart wordt, of het kan ingebed zijn in de C-compiler.

De preprocessor is *geen* compiler in de zin van het genereren van uitvoerbare code voor een computer. De preprocessor verwerkt het C-bestand aan de hand van bepaalde opdrachten die allemaal beginnen met een *hash*[1] (#). Die worden *directives* genoemd. De uitvoer van de preprocessor is een bestand waarin geen directive meer voorkomt en dat bestand wordt aan de C-compiler doorgegeven. De C-compiler heeft dus geen weet van directives.

10.1 Invoegen van bestanden

Een bestand kan in een C-programma worden ingevoegd door de *include directive*. Deze heeft de vorm:

```
#include <bestandsnaam>
```

of

```
#include "bestandsnaam"
```

Invoegen gebeurt op het punt waar `#include` voorkomt. De include directive zelf wordt verwijderd. Een bestandsnaam kan tussen < en > staan. Dan wordt gezocht in zogenoemde *systeemmappen*. Welke dat zijn, hangt van de toolchain af. Een bestandsnaam kan ook tussen aanhalingstekens staan. Dan wordt eerst in de map gezocht waarin ook het C-programma geplaatst is en, als het bestand niet gevonden is, in de systeemmappen. De include directive wordt typisch gebruikt om een *header-bestand* aan het begin van een C-programma in te lezen. We hebben er al een aantal gebruikt:

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

[1] Tegenwoordig wordt dit ook wel een hashtag genoemd.

```
#include <string.h>
#include <math.h>
```

De extensie `.h` geeft aan dat het om een header-bestand gaat. Dit is niet verplicht maar wel *good practice*. Het is *niet* de bedoeling om andere C-bestanden via de include directive in te lezen. Dat kan wel, maar is *bad practice*.

Header-bestanden kunnen zelf ook weer include directives bevatten, wat ook vaak het geval is. Daarbij moet erop gelet worden dat header-bestanden niet meerdere keren geladen worden. Hoe dat vermeden kan worden, zien we in paragraaf 10.4.

10.2 Macrovervanging

Een *macro* in de eenvoudigste vorm, ziet eruit als:

`#define` *macronaam vervangende-tekst*

Overal waar *macronaam* voorkomt wordt dit vervangen door *vervangende-tekst* behalve in C-strings. De *macronaam* moet exact overeenkomen, dus als `MAX` gedefinieerd is, dan wordt er geen vervanging gedaan in `printf("MAX")` en `MAX_GETAL`. Macro's mogen zelf ook weer macro's bevatten en het vervangen gaat net zolang door totdat er geen vervangingen meer worden gedaan. We noemen dit *macro-expansie*.

Een macro kan argumenten bevatten zodat de vervangende tekst voor verschillende doeleinden gebruikt kan worden. Een klassiek voorbeeld is:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

De macro `max(A, B)` ziet eruit als een functie maar dat is het niet. Overal waar `max` voorkomt wordt het vervangen door de vervangende tekst. Merk op het gebruik van de vele haakjes. Dat is nodig om de argumenten correct uit te rekenen:

```
x = max(p+q, r+s);
```

wordt dan vervangen door

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Een veel gemaakte fout is het als volgt definiëren van een kwadraat:

```
#define sqr(x) x * x     /* WRONG */
```

en dat geeft een verkeerd antwoord als `x` wordt vervangen door `y+1`. De correcte manier is om extra haakjes te gebruiken:

```
#define sqr(x) ((x) * (x))     /* correct */
```

Het kan alsnog verkeerd gaan als we de increment- of decrement-operator gebruiken:

```
j = sqr(++i);
```

Dit wordt geëxpandeerd tot `j = ((++i) * (++i))` dus i wordt twee keer verhoogd. Een voordeel van een macro is dat de argumenten onafhankelijk zijn van het datatype. Als we een floating point-getal willen kwadrateren, dan kan dat ook:

```
double j = sqr(2.16);
```

We kunnen de preprocessor ook een macro laten vergeten:

`#undef` *macronaam*

10.3 Conditionele compilatie

Het gebruik van macro's is bijzonder handig als we delen van een C-bestand wel of niet willen laten compileren. Een veel gebruikt voorbeeld is het afdrukken van debug-informatie:

```
#include <stdio.h>

#define DEBUG 1

int main(void) {
    // ...
#if defined(DEBUG)
    printf("Variabele: %d\n", a);
#endif
    // ...
}
```

Listing 10.1: *Conditionele compilatie.*

In plaats van `defined` mogen we ook `#ifdef` gebruiken. We kunnen ook header-bestanden aan de hand van macro's laten invoegen. De diverse C-compilers definiëren bepaalde macro's zodat deze in een C-bestand te gebruiken zijn, de zogenoemde *predefined macros*. Een voorbeeld is te zien in listing 10.2.

```
#include <stdio.h>

#ifdef __GNUC__                    /* GNU C-compiler */
#define VSTRING "GNU C-compiler"
#include <gnuc.h>
#endif
#ifdef _MSC_VER                    /* Microsoft C-compiler
#define VSTRING "Visual Studio C-compiler"
#include <ms.h>
#endif
#ifdef __clang__                   /* Clang C-compiler
#define VSTRING "Clang C-compiler"
#include <clang.h>
#endif

int main(void) {

    printf("Gecompileerd met " VSTRING);
    return 0;
}
```

Listing 10.2: *Conditioneel inlezen van header-bestanden.*

Welke predefined macros gedefinieerd zijn, is afhankelijk van de C-compiler. De documentatie moet hiervoor geraadpleegd worden. Sommige macro's hebben ook nuttige waarden zoals te zien is in listing 10.3.

```
#ifdef _MSC_VER
#if _MSC_VER == 1926
#define VSTRING "Visual Studio 2019 versie 16.6"
#elif _MSC_VER == 1925
#define VSTRING "Visual Studio 2019 versie 16.5"
#elif _MSC_VER == 1924
#define VSTRING "Visual Studio 2019 versie 16.4"
#else
#define VSTRING "Visual Studio 2019 versie 16.3 of ouder"
#endif
#endif
```

Listing 10.3: *Gebruik van de _MSC_VER-macro.*

Meerdere macro's mogen in een logische expressie voorkomen. De onderstaande regels zorgen ervoor dat er conditioneel gecomplieerd wordt als de macro `__linux__` wel bestaat en de macro `__ANDROID__` niet bestaat.

```
#if defined(__linux__) && !defined(__ANDROID__)
```

In principe mag de expressie willekeurig complex zijn, als het maar resulteert in een constante integer. Typische C-constructies kunnen niet gebruikt worden, zoals (C-)variabelen, `sizeof` en expliciete type casts want de preprocessor is geen C-compiler.

10.4 Conditioneel inlezen van een bestand

Om ervoor te zorgen dat tijdens een compilatie van een C-bestand een header-bestand niet meer dan één keer wordt ingelezen, kunnen we een macro gebruiken. Aan het begin van het header-bestand testen we of een bepaalde macro bestaat. Als dat *niet* zo is, dan definiëren we de macro en laden de rest van het header-bestand. De volgende keer dat het header-bestand (in dezelfde compilatieslag) wordt geladen is de macro gedefinieerd en wordt de code overgeslagen. Zie listing 10.4.

```
/* File myheader.h */

#ifndef _MYHEADER_H
#define _MYHEADER_H

    // ... contents goes here

#endif
```

Listing 10.4: *Gebruik van een macro om een header-bestand één keer in te lezen.*

11

Het compilatieproces

De compiler is een ingewikkeld stuk gereedschap. Hij compileert C-bestanden, voegt ze samen, voegt er bibliotheken met vooraf geprogrammeerde functies aan toe en zorgt ervoor dat uiteindelijk een uitvoerbaar bestand of *executable* wordt gegenereerd. Dit uitvoerbaar bestand bevat de instructies die door de computer kunnen worden uitgevoerd. We moeten direct opmerken dat de "compiler" uit meerdere programma's bestaat. Het is eigenlijk een verzameling van programma's. Dit wordt een *toolchain* of *compiler suite* genoemd.

In dit hoofdstuk zullen we enige aspecten van het compilatieproces behandelen. Hoewel er veel verschillende C-compilers bestaan, volgen de C-compilers min of meer dezelfde lijn als het om compilatie van bestanden gaat. We zullen ons richten op een veelgebruikte C-compiler, namelijk de GNU C-compiler [18]. Deze compiler is geschikt voor onder andere de PC of laptop (Windows en Linux) en een keur aan microcontrollers, zoals de ATmega-, STM32, PIC- en MSP430-microcontrollers. Natuurlijk wordt alles geregeld door de Integrated Development Environment (IDE) en hoeft de gebruiker niet zelf alle commando's te geven. We doen dat in dit hoofdstuk wel, om te laten zien wat er nou eigenlijk gebeurt.

De GNU C-compiler wordt gestart met het programma `gcc`. Het `gcc`-programma is niet de echte compiler maar een programma dat een vertaler is tussen de gebruiker en andere programma's. Het verzamelt bestandsnamen en opties en start dan de benodigde programma's met opties. Om het type van de bestanden te onderscheiden worden *extensies* gebruikt. Het Engelse woord hiervoor is *suffix*. De belangrijkste extensies zijn te zien in tabel 11.1.

Tabel 11.1: *Enkele bestandsextensies in de C-omgeving.*

`.c`	een C-bestand, dient als invoer voor de preprocessor
`.i`	uitvoer van de preprocessor, dient als invoer voor de "echte" C-compiler
`.s`	een assembler-bestand, dient als invoer voor de assembler
`.o`	uitvoer van de C-compiler en assembler, dient als invoer voor de linker
`.a`	een bibliotheek (archive), dient als invoer voor de linker
`.exe`	uitvoer van de linker, uitvoerbaar bestand (Windows)

11.1 Stappen in de compilatie

De compilatie van een programma gebeurt in een aantal stappen. Deze zijn in een schematisch overzicht in figuur 11.1 te zien. We geven in deze paragraaf een korte introductie. De stappen worden in detail behandeld in de volgende paragrafen.

C-bestanden worden in drie stappen gecompileerd. Eerst worden de C-bestanden aan de *preprocessor* (PRE) doorgegeven (zie hoofdstuk 10). De uitvoer zijn tijdelijk bestanden. Dit is eigenlijk geen echte compilatie want er worden alleen *macro's* verwerkt. Include-bestanden worden ook door de preprocessor verwerkt. Daarna worden de tijdelijke bestanden aan de "echte" C-compiler (C) doorgegeven. De C-compiler genereert *assembler-bestanden* met daarin de instructies voor de gekozen processor. De *assembler* (AS) assembleert deze bestanden en genereert *object-bestanden*. In een object-bestand zijn de bitpatronen van de instructies opgeslagen, maar het kan zijn dat bepaalde *referenties* (zie paragraaf 11.5) naar functies en globale variabelen nog niet zijn ingevuld. Denk hierbij aan de `printf`-functie die we niet zelf geschreven hebben maar in een *bibliotheek* is opgeslagen.

Alle gegenereerde objectbestanden worden aan de *linker* doorgegeven, die de objectbestanden samenvoegt met functies en globale variabelen uit de bibliotheken. Daarna wordt een uitvoerbaar programma gegenereerd.

Als we een programma ontwikkelen voor een microcontroller (dat wordt *cross-compiling* genoemd, omdat de compiler instructies genereert voor een andere processor), volgt vaak nog een *post-processing*-stap. Deze stap is nodig om een bestand te genereren dat met behulp van een *programmer* in de microcontroller kan worden geladen.

11.2 Een C-programma in één bestand

De meeste kleine C-programma's worden in één bestand georganiseerd. We compileren het en daarna kan het op de computer worden uitgevoerd. Laten we eens kijken hoe de C-compiler gestart wordt. We geven de opdracht (op de command line):

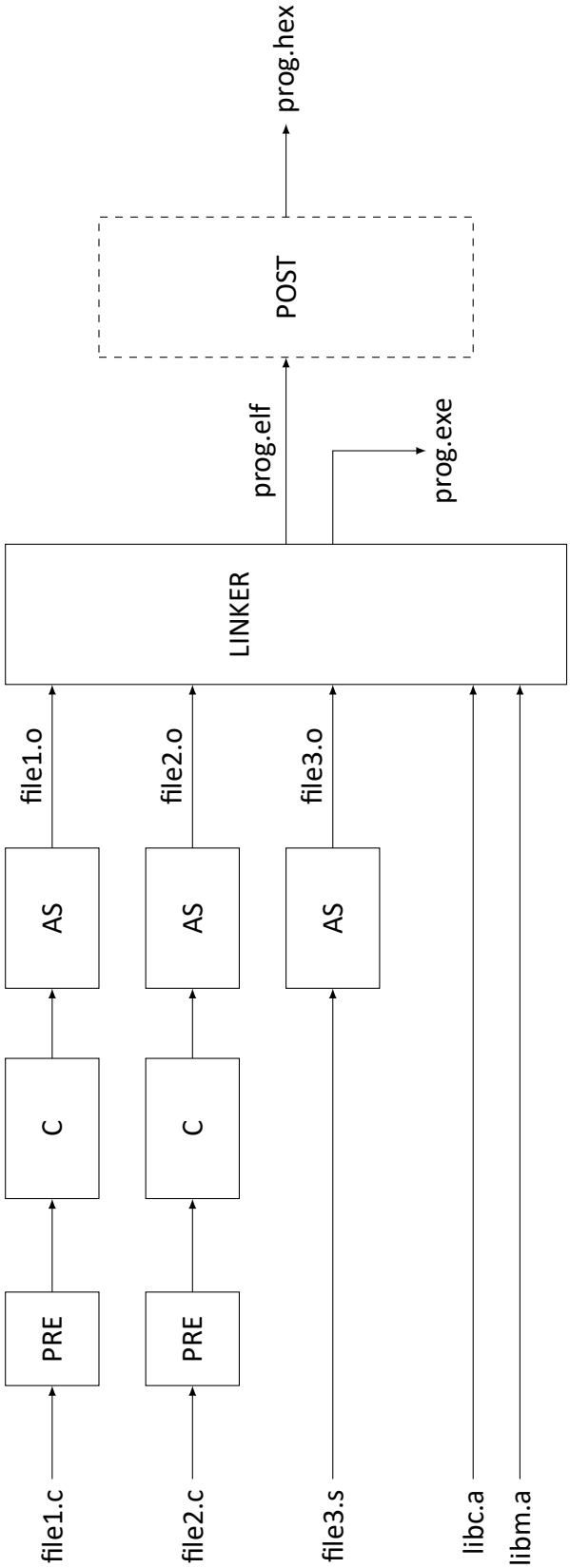
```
gcc mooi.c -o mooi.exe
```

Als alles goed verloopt, wordt er een uitvoerbaar bestand `mooi.exe` gegenereerd[1]. We kunnen dit bestand uitvoeren met `.\mooi.exe`. Soms kan de `.\` weggelaten worden, afhankelijk van de instellingen van Windows. Hoewel het uit dit voorbeeld niet blijkt, worden alle stappen uit figuur 11.1 doorlopen.

11.3 Een C-programma in meerdere bestanden

Een groot C-programma kunnen we onderverdelen in meerdere C-bestanden. Dit heeft een aantal voordelen. Ten eerste kunnen we veel gebruikte programmadelen, denk hierbij aan functies, afzonderen van het grote geheel. We kunnen deze functies zo opstellen dat ze door meerdere programma's gebruikt kunnen worden. Dit is goed voor de *herbruikbaarheid*. Ten tweede kunnen we de ontwikkeling van het programma makkelijk verdelen onder meerdere personen. Dit is goed voor de *verantwoordelijkheid* van onderdelen van het programma.

[1] Op Unix-systemen, dus ook Linux en OS-X, heeft een uitvoerbaar bestand geen extensie. De compilatie-opdracht wordt dan `gcc mooi.c -o mooi`, waarna we `mooi` kunnen uitvoeren.



Figuur 11.1: *Schematisch overzicht van een compilatie.*

Tijdens compilatie moeten alle C-bestanden gecompileerd worden om uiteindelijk een uitvoerbaar bestand te krijgen. Stel dat we ons C-programma verdelen over meerdere bestanden (dit worden *translation units* genoemd). Dan kunnen we de C-compiler aanroepen met:

```
gcc file1.c file2.c file3.c -o programma.exe
```

Alle C-bestanden worden gecompileerd en uiteindelijk samengevoegd tot het uitvoerbare bestand `programma.exe`. Als er in een van de C-bestanden een fout zit, moet die hersteld worden en moeten alle C-programma's opnieuw gecompileerd worden op de bovenstaande wijze.

Dat is natuurlijk niet handig als een een C-programma verdeeld is over een groot aantal bestanden. Het is daarom mogelijk om een C-bestand te compileren naar een *object-bestand* en daarna de object-bestanden te linken. We kunnen de compilatie ook realiseren met

```
gcc -c file1.c -o file1.o
gcc -c file2.c -o file2.o
gcc -c file3.c -o file3.o
gcc file1.o file2.o file3.o -o programma.exe
```

Als nu in een C-bestand een fout zit, moet alleen dat ene C-bestand opnieuw naar een object-bestand worden gecompileerd. Natuurlijk moet het laatste commando uitgevoerd om een uitvoerbaar bestand te krijgen.

11.4 Bibliotheken

Een bibliotheek is eigenlijk een verzameling van object-bestanden. Tijdens het linken worden uit de bibliotheek alleen de object-bestanden meegenomen die nodig zijn. De overige object-bestanden worden dus niet meegenomen. Het heeft namelijk geen zin om de functie `strlen` erbij te voegen als de functie niet aangeroepen wordt.

11.5 Linkage

Het samenvoegen van object-bestanden en bibliotheken wordt gedaan door de *linker* en het proces wordt *linkage* genoemd. Tijdens het linken worden onbekende *referenties* (verwijzingen naar functies of globale variabelen) door de linker ingevuld. Een voorbeeld hiervan is de functie `printf` die niet door ons is geprogrammeerd maar die in een bibliotheek is opgeslagen. Als alle referenties kunnen worden ingevuld, volgt een uitvoerbaar bestand.

Tijdens het linken worden bibliotheken gebruikt om de referenties in te vullen. Twee veelgebruikte bibliotheken zijn de *standard library* en de *mathematical library*. De standard library heeft de naam `libc.a` (bij de GNU C-compiler onder de naam `glibc.a`). De extensie `.a` (*archive*) geeft aan dat het om een bibliotheek gaat. De mathematical library heeft de naam `libm.a` en moet in veel gevallen expliciet worden opgegeven:

```
gcc file1.c -l m -o programma.exe
```

Alleen de `m` moet worden opgegeven, de rest wordt automatisch door de linker ingevuld. Merk op dat de standard library niet expliciet moet worden opgegeven. De C-compiler regelt dat automatisch.

11.6 Assembler-bestanden

Soms is het nodig om bepaalde code te programmeren in *assembly*. Assembly is een taal die zeer dicht tegen de gebruikte processor ligt. Te denken valt aan code die niet in C geschreven kan worden, zoals opstartcode van het C-programma, geoptimaliseerde programmafragmenten en Interrupt Service Routines.

Een assembler-bestand heeft meestal de extensie `.s`, soms ook wel `.asm`. We kunnen `gcc` vragen om een assembler-bestand te compileren:

```
gcc file.s -o programma.exe
```

Meestal wordt een assembler-bestand samen met C-bestanden gecompileerd:

```
gcc file1.c file2.c startup.s -o programma.exe
```

11.7 Definitie en declaratie

Er is een verschil tussen definitie en declaratie van variabelen en functies. Grosso mode komt het hier op neer: bij definitie wordt geheugenruimte gereserveerd, bij declaratie niet. Dus

```
int a;
```

definieert een variabele; de variabele wordt kenbaar gemaakt aan de C-compiler én er wordt geheugenruimte gereserveerd. Maar

```
extern int a;
```

is een declaratie; de variabele wordt kenbaar gemaakt aan de C-compiler, maar er wordt geen geheugenruimte gereserveerd. Dat wordt op een andere plaats gedaan. De C-compiler heeft echter genoeg informatie om bijvoorbeeld het statement `a = a + 3` te realiseren, ook al is de exacte geheugenplaats nog niet bekend. Een functie wordt gedefinieerd als er ook geheugen wordt gebruikt; de functie heeft een *body*. Dus

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

definieert de functie `max` en maakt deze tevens kenbaar aan de C-compiler. Voor de parameters `a` en `b` wordt ook geheugenruimte gerealiseerd want er wordt immers een kopie van de argumenten gemaakt. Een functiedeclaratie heeft geen body. Dus

```
int max(int a, int b);
```

is een declaratie (ook wel functie-prototype genoemd). De C-compiler heeft dan genoeg informatie om de aanroep van de functie te realiseren. Natuurlijk moet er ook een functiedefinitie beschikbaar zijn.

Declaraties kunnen ook gebruikt worden bij structures en unions. Zo geeft

```
struct punt {
    int x;
    int y;
};
```

de declaratie van de *structure identifier* `punt`. Maar

```
struct punt oorsprong;
```

definieert de structure `oorsprong`, er wordt namelijk geheugen gereserveerd.

Er is één uitzondering:

```
extern int a = 4;
```

zorgt voor definitie van de variabele `a` en kent daar direct de waarde 4 aan toe.

11.8 De qualifiers `extern` en `static`

Een *globale* variabele is zichtbaar in het gehele programma en de levensduur is tijdens de gehele executie van het programma. Een groot C-programma is echter verdeeld over meerdere bestanden. We kunnen deze globale variabele zichtbaar maken in meerdere C-bestanden. Hiervoor gebruiken de qualifier `extern`. In één bestand geven we de definitie van de variabele en in de andere bestanden geven we de declaratie van de variabele middels `extern`. Dit is te zien in listings 11.1 en 11.2.

```c
int counter;

void func1(void) {
    counter++;
}
```

Listing 11.1: *file1.c*

```c
extern int counter;

void func2(void) {
    counter--;
}
```

Listing 11.2: *file2.c*

In bestand `file1.c` wordt variabele `counter` gedefinieerd. Hier wordt ook gelijk geheugenruimte gereserveerd. In bestand `file2.c` wordt `counter` gedeclareerd; de variabele wordt kenbaar gemaakt maar er wordt geen geheugenruimte gereserveerd. In beide bestanden is `counter` beschikbaar. Tijdens het linken wordt de referentie (geheugenadres) naar `counter` in bestand `file2.c` ingevuld.

Een functie is per definitie globaal beschikbaar en dus zichtbaar in het gehele programma. We definiëren de functie add in `file3.c`. Daar wordt ook geheugenruimte gereserveerd. In `file4.c` volgt een functie-declaratie; de functie wordt kenbaar gemaakt, er wordt geen geheugenruimte gereserveerd maar kan wel aangeroepen worden. Tijdens het linken wordt de referentie naar de functie add in `file4.c` ingevuld.

```c
int add(int a, int b) {
    return a + b;
}

void func2(void) {
    int a = add(2, 3);
}
```

Listing 11.3: *file3.c*

```c
int add(int a, int b);

void func4(void) {
    int a = add(2, 3);
}
```

Listing 11.4: *file4.c*

Soms is het handig om een globale variabele alleen beschikbaar te stellen in het C-bestand waarin hij gedefinieerd is. We voorzien de variabele dan van de qualifier `static`. Daarmee is de variabele afgeschermd van de andere C-bestanden en kan de variabele niet in andere C-bestanden (per ongeluk) worden aangepast. We laten dit zien aan de hand van de implementatie van een *software stack*, listing 11.5.

```
#include <stdio.h>

#define STACK_DEPTH (100)

static int stack_pointer = 0;
static int stack[STACK_DEPTH];

void push(int val) {
    if (stack_pointer < STACK_DEPTH) {
        stack[stack_pointer] = val;
        stack_pointer++;
    } else {
        printf("Stack overflow! Nothing pushed.\n");
    }
}

int pop(void) {

    if (stack_pointer > 0) {
        stack_pointer--;
        return stack[stack_pointer];
    } else {
        printf("Stack underflow! Returning 0.\n");
        return 0;
    }
}
```

Listing 11.5: *Implementatie van een software stack.*

De stack is opgebouwd rond een *stack pointer* en de feitelijke stack. De stack wordt gerealiseerd met een array van een bepaalde grootte. De stack pointer houdt bij op welke positie het laatste waarde op de stack beschikbaar is. Twee functies manipuleren de stack: de functie `push` zet een nieuwe waarde op de stack, de functie `pop` haalt de laatst ingebrachte waarde van de stack. Beide functie passen de stack pointer aan. Het is mogelijk om teveel waarden op de stack te *pushen*; dan volgt een *stack overflow*. Het is ook mogelijk om teveel te *poppen*; dan volgt een *stack underflow*. De functies `push` en `pop` zijn zichtbaar in het hele programma, de globale variabelen niet. We plaatsen de code in de listing in een apart C-bestand.

We kunnen de stack nu gebruiken in ons programma. Een voorbeeld is te zien in listing 11.6. In regels 3 en 4 geven we de declaraties van de functies `push` en `pop` zodat de compiler weet hoe de functies aangeroepen moeten worden. In regels 8 en 9 worden twee getallen op de stack geplaatst. In regels 11 en 12 wordt de functie `pop` gebruikt om een waarde van de stack te halen. De `pop` in regel 13 (what's in the number) zorgt ervoor dat de stack een underflow

```c
#include <stdio.h>

void push(int val);
int pop(void);

int main(void) {

    push(5);
    push(3);

    printf("Pop from stack: %d\n", pop());
    printf("Pop from stack: %d\n", pop());
    printf("Pop from stack: %d\n", pop());
}
```

Listing 11.6: *file4.c*

krijgt en wordt een foutmelding afgedrukt. Opmerking: elke processor implementeert een *hardware stack*. Die wordt gebruikt om argumenten over te dragen aan functies. Tevens wordt op de hardware stack het terugkeeradres van de aanroepende functie geplaatst.

Het is ook mogelijk om een functie alleen beschikbaar te stellen in het C-bestand waarin hij gedefinieerd is. We voorzien de functie dan van de qualifier `static`. Daarmee is de functie afgeschermd van de andere C-bestanden. Een voorbeeld is te zien in listing 11.7.

```c
static char *searchtok(char *str, char tok) {
    // ...
}

int evaluate(char *str) {

    char *p;

    if ((p = seachtok(str, '*')) != NULL) {
        // ...
    } else if ((p = seachtok(str, '/')) != NULL) {
        // ...
    } else if ((p = seachtok(str, '%')) != NULL) {
        // ...
    }
}
```

Listing 11.7: *Gebruik van een static globale variabele.*

De functie `searchtok` zoekt naar de positie van een karakter in een string. Als het karakter gevonden is, wordt een pointer naar dit karakter teruggegeven. Zo niet, dan wordt een NULL-pointer teruggegeven. De functie is alleen nodig in het C-bestand waarin hij gedefinieerd is dus wordt `searchtok` voorzien van de qualifier `static`. De functie `evaluate` kan deze functie wel gebruiken want die is hetzelfde bestand gedefinieerd.

De qualifier `static` heeft nog een andere nuttige functie. Wanneer dit wordt toegepast bij de definitie van een *lokale variabele* dan blijft de inhoud van deze variabele behouden bij het binnengaan en verlaten van een functie. Een voorbeeld is te zien in listing 11.8.

```
int countup(void) {

    static int counter = 0;

    count++;

    return count;
}
```

Listing 11.8: *Gebruik van een static lokale variabele.*

De lokale variabele `counter` wordt bij het starten van het programma eenmaal geïnitialiseerd en blijft dan zijn toegekende waarde behouden. In tegenstelling tot een gewone lokale variabele wordt `counter` dus niet steeds opnieuw aangemaakt en verwijderd bij het binnengaan van en terugkeren uit de functie.

11.9 Bibliotheek maken

Het is ook mogelijk om een bibliotheek te maken. Daarvoor moet een C-bestand gecompileerd worden naar een object-bestand:

```
gcc -c file1.c -o file1.o
gcc -c file2.c -o file2.o
gcc -c file3.c -o file3.o
```

Daarna moet de *archiver* gestart worden:

```
ar rcs libmy.a file1.o file2.o file3.o
```

De bibliotheek is nu te linken met:

```
gcc file.c -l my -o programma.exe
```

11.10 Dynamic en static linking

Er bestaat op veel besturingssystemen zoiets als *dynamic linking*. Bibliotheken worden dan eenmalig in het geheugen van de computer worden geladen waarna ze gebruikt kunnen worden door programma's. Pas op het moment dat een programma wordt gestart, worden de referenties naar functies en externe variabelen ingevuld. Dit wordt gedaan door de *loader*. Het starten kost dus extra tijd omdat bepaalde referenties nog moeten worden ingevuld. Het voordeel is wel duidelijk: de omvang van uitvoerbare programma's wordt geminimaliseerd en dat zorgt weer voor het sneller laden van een programma. Een ander voordeel is dat een veelgebruikte functie, bijvoorbeeld `printf`, slechts één keer in het geheugen voorkomt. Voorbeelden zijn *dynamic link libraries* (bestanden met de extensie `.dll`) op Windows en *shared objects* (bestanden met de extensie `.so`) op Linux.

Soms is het nodig of wenselijk om object-bestanden uit een bibliotheek op te nemen in het uitvoerbare programma, bijvoorbeeld bij een zeer uitgekleed besturingssysteem dat dynamic linking niet ondersteunt. Dit wordt *static linking* genoemd. Standaard zullen de meeste C-compilers dynamic linking gebruiken maar static linking is te forceren middels een optie aan de compiler. Zo kunnen we de `gcc`-compiler aanroepen met

```
gcc mooi.c -o mooi.exe -static
```

om static linking te forceren. Het uitvoerbare programma wordt wel groter omdat bepaalde functies nu aan het programma worden toegevoegd.

11.11 Optimalisatie

Zoals al eerder is vermeld, is de C-compiler een ingewikkeld programma[2]. De huidige compilers zijn zeer pienter in het herkennen van de C-taal. Laten we eens kijken naar het programma in listing 11.9.

```c
#include <stdio.h>

int main(void) {

    int ary[1000], i;

    for (i = 0; i < 1000; i++) {
        ary[i] = 0;
    }
    // ...

    return 0;
}
```

Listing 11.9: *Een C-programma.*

In principe alloceert de compiler 1000 integers voor array `ary` en een integer voor `i`. Maar de compiler heeft door dat `i` gebruikt wordt als lusvariabele. Het is dan beter om deze variabele niet in het geheugen van de computer op te slaan maar vast te houden in een *register*. Een register is een speciale geheugenplaats in de processor. Alle processoren hebben een aantal van deze registers. Als variabele `i` nu in een register wordt opgeslagen dan hoeft de processor `i` niet steeds uit het geheugen te halen en weer terug te zetten. Daardoor wordt de executie van het `for`-statement sneller. Het gevolg is dat voor variabele `i` helemaal geen geheugenplaats wordt gereserveerd. We kunnen aan de C-compiler opgeven dat een bepaalde variabele beter in een register kan worden opgeslagen met het keyword `register`:

```
register int i;
```

[2] De vraag is altijd: in welke taal is de C-compiler geschreven? Het antwoord is: in C zelf. Dit levert natuurlijk het kip-en-ei-probleem. Het antwoord is dat de C-compiler in eerste instantie in assembly is geschreven en vandaar uit is een echte C-compiler gebouwd. Opvolgende generaties van C-compilers kunnen dus gecompileerd worden door (oudere) C-compilers.

Overigens staat het de compiler vrij om `register` te negeren. Over het algemeen weet de compiler beter dan de programmeur welke variabelen in registers moeten worden gehouden. Het gebruik van `register` wordt daarom niet aanbevolen.

Een ander voorbeeld is een `for`-statement waarin ogenschijnlijk niets gebeurt. Dit is te zien in listing 11.10. Dit soort herhalingen worden nog wel eens gebruikt als er in een programma een tijdje moet worden gewacht. Als we naar het `for`-statement kijken, gebeurt er eigenlijk niets, behalve dat variabele `i` wordt opgehoogd. De C-compiler kan nu beslissen om het hele `for`-statement te verwijderen want, zo is de gedachte, er gebeurt toch niets. Willen we ervoor zorgen dat het `for`-statement toch wordt gebruikt dan kunnen we een variabele voorzien van het keyword `volatile`:

```
#include <stdio.h>

int main(void) {

    int i;

    for (i = 0; i < 30000; i++) {
        // do nothing, just waste processor time
    }

    return 0;
}
```

Listing 11.10: *Een C-programma.*

```
volatile int i;
```

Het keyword `volatile` betekent zoiets als: gebruik variabele `i` ook al lijkt er niets zinnigs mee te gebeuren. De C-compiler zal hierdoor over het algemeen wel een geheugenplaats voor `i` realiseren; `i` wordt dan niet in een register vastgehouden.

De `gcc`-compiler kent nog vele optimalisatiemogelijkheden. Dit moeten aan de C-compiler opgegeven worden middels opties. In tabel 11.2 zijn de *algemene* opties gegeven.

Tabel 11.2: *Optimalisaties bij de `gcc`-compiler.*

Optie	Betekenis
`-O0`	geen optimalisaties
`-O1`	enkele optimalisaties
`-O2`	nog meer optimalisaties
`-O3`	heel veel optimalisaties (beïnvloedt de nauwkeurigheid van floating point)
`-Og`	optimalisatie voor debuggen
`-Os`	optimalisatie voor de grootte van de executable
`-Ofast`	optimalisatie voor de snelheid van de executable

Daarnaast zijn er nog specifieke optimalisaties maar die zijn veelal afhankelijk van de gebruikte processor. Alle optimalisaties zijn te vinden in [19].

Bij *loop unrolling* worden de statements in een herhaling omgezet in een aantal sequentiële statements zodat het opzetten van de herhaling vermeden wordt. Zie listing 11.11.

```
#include <stdio.h>

int main(void) {

    int ary[4], i;

    for (i = 0; i < 4; i++) {
        ary[i] = 0;
    }

    return 0;
}
```

Listing 11.11: *Een C-programma.*

Na analyse van het `for`-statement kan de compiler beslissen of het wenselijk is om het `for`-statement om te zetten naar vier toekenningen; de compiler is in staat om het *bereik* (range) van variabele `i` te bepalen. Dat is sneller dan een lus op te zetten. Dus wordt het `for`-statement omgezet naar:

```
ary[0] = 0; ary[1] = 0; ary[2] = 0; ary[3] = 0;
```

Een sterk staaltje van loop unrolling *by hand* (dus door de programmeur geschreven) is *Duff's Device* [20], ontwikkeld in 1983. Het is een slimme manier om een array van integers zeer snel te kopiëren naar één enkel adres. Dit komt onder andere voor bij microcontrollers.

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. iv).

[3] Tex User Groups. *TeX Live Website*. URL: https://www.tug.org/texlive/ (bezocht op 30-12-2018) (blz. iv).

[4] Benito van der Zander. *TeXstudio, LaTeX made comfortable*. URL: https://www.texstudio.org/ (bezocht op 04-05-2019) (blz. iv).

[5] Lucas de Groot. *Calibri font family*. URL: https://docs.microsoft.com/en-us/typography/font-list/calibri (bezocht op 04-05-2019) (blz. iv).

[6] Lucas de Groot. *Consolas font family*. URL: https://docs.microsoft.com/en-us/typography/font-list/consolas (bezocht op 04-05-2019) (blz. iv).

[7] J. Hoffman. *listings – Typeset source code listings using LaTeX*. URL: https://www.ctan.org/pkg/listings (bezocht op 30-12-2018) (blz. iv).

[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. iv).

[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. vi).

[10] The Rust Foundation. *Rust, A language empowering everyone to build reliable and efficient software*. URL: https://www.rust-lang.org/ (bezocht op 22-06-2021) (blz. 2).

[11] *Programming Language C: American National Standard for Information Systems : ANSI X3.159-1989*. A.N.S.I., 1989 (blz. 2).

[12] Microsoft. *Visual Studio*. 2020. URL: https://visualstudio.microsoft.com/ (bezocht op 05-06-2020) (blz. 5).

[13] The Code::Blocks Team. *Code::Blocks*. 2020. URL: http://www.codeblocks.org/ (bezocht op 05-06-2020) (blz. 5).

[14] Apple. *Xcode*. 2020. URL: https://developer.apple.com/xcode/ (bezocht op 05-06-2020) (blz. 5).

[15] 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. 12).

[16] Jesse op den Brouw. *Digitale techniek*. 1ste ed. Delft Academic Press, 2020. ISBN: 978-90-6562-4468 (blz. 21).

[17] Ridiculous Fish. *cdecl C gibberish ↔ English*. URL: https://cdecl.org/ (bezocht op 23-08-2020) (blz. 148).

[18] Free Software Foundation. *GNU Compiler Colletion*. URL: https://gcc.gnu.org/ (bezocht op 20-01-2021) (blz. 173).

[19] Free Software Foundation. *Options That Control Optimization*. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (bezocht op 21-01-2021) (blz. 183).

[20] Tom Duff. *The amazing Duff's Device by Tom Duff!* 1983. URL: https://www.lysator.liu.se/c/duffs-device.html (bezocht op 20-01-2021) (blz. 184).

[21] American Standards Association. *ASA standard X3.4-1963*. URL: http://worldpowersystems.com/J/codes/X3.4-1963/ (bezocht op 26-07-2018) (blz. 187).

[22] A.K. Maini. *Digital Electronics: Principles, Devices and Applications*. Wiley, 2007, p. 28. ISBN: 978-0-470-03214-5 (blz. 187).

[23] V. Eijkhout. *An ASCII wall chart*. Nov 2010. URL: https://ctan.org/pkg/ascii-chart (bezocht op 29-07-2018) (blz. 188).

A

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 [21] 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 [22].

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. Ze worden in C *escape sequences* genoemd.. Een escape sequence 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. De code SP staat voor één spatie.

Tabel A.1: *De ASCII-code [23].*

ASCII CONTROL CODE CHART

BITS b7 b6 b5	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4 b3 b2 b1	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL 0	16 DLE 20	32 SP 40	48 0 60	64 @ 100	80 P 120	96 ‘ 140	112 p 160
0 0 0 1	1 SOH 1	17 DC1 21	33 ! 41	49 1 61	65 A 101	81 Q 121	97 a 141	113 q 161
0 0 1 0	2 STX 2	18 DC2 22	34 ” 42	50 2 62	66 B 102	82 R 122	98 b 142	114 r 162
0 0 1 1	3 ETX 3	19 DC3 23	35 # 43	51 3 63	67 C 103	83 S 123	99 c 143	115 s 163
0 1 0 0	4 EOT 4	20 DC4 24	36 $ 44	52 4 64	68 D 104	84 T 124	100 d 144	116 t 164
0 1 0 1	5 ENQ 5	21 NAK 25	37 % 45	53 5 65	69 E 105	85 U 125	101 e 145	117 u 165
0 1 1 0	6 ACK 6	22 SYN 26	38 & 46	54 6 66	70 F 106	86 V 126	102 f 146	118 v 166
0 1 1 1	7 BEL 7	23 ETB 27	39 ’ 47	55 7 67	71 G 107	87 W 127	103 g 147	119 w 167
1 0 0 0	8 BS 10	24 CAN 30	40 (50	56 8 70	72 H 110	88 X 130	104 h 150	120 x 170
1 0 0 1	9 HT 11	25 EM 31	41) 51	57 9 71	73 I 111	89 Y 131	105 i 151	121 y 171
1 0 1 0	10 LF 12	26 SUB 32	42 * 52	58 : 72	74 J 112	90 Z 132	106 j 152	122 z 172
1 0 1 1	11 VT 13	27 ESC 33	43 + 53	59 ; 73	75 K 113	91 [133	107 k 153	123 { 173
1 1 0 0	12 FF 14	28 FS 34	44 , 54	60 < 74	76 L 114	92 \ 134	108 l 154	124 | 174
1 1 0 1	13 CR 15	29 GS 35	45 – 55	61 = 75	77 M 115	93] 135	109 m 155	125 } 175
1 1 1 0	14 SO 16	30 RS 36	46 . 56	62 > 76	78 N 116	94 ^ 136	110 n 156	126 ~ 176
1 1 1 1	15 SI 17	31 US 37	47 / 57	63 ? 77	79 O 117	95 _ 137	111 o 157	127 DEL 177

LEGEND: dec CHAR hex oct

Victor Eijkhout
TACC
Austin, Texas, USA

B

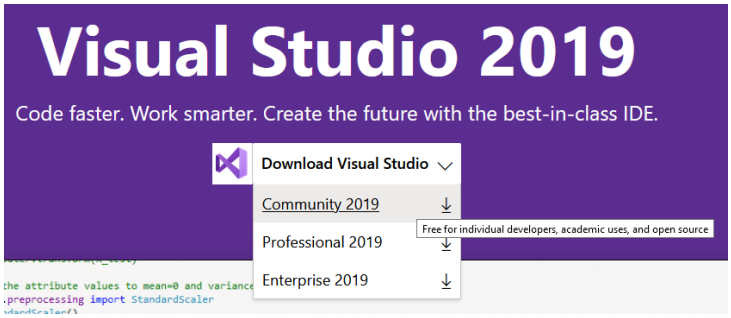
Tutorial Visual Studio

We behandelen 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*.

Download Visual Studio Community via de link:

https://visualstudio.microsoft.com/vs/

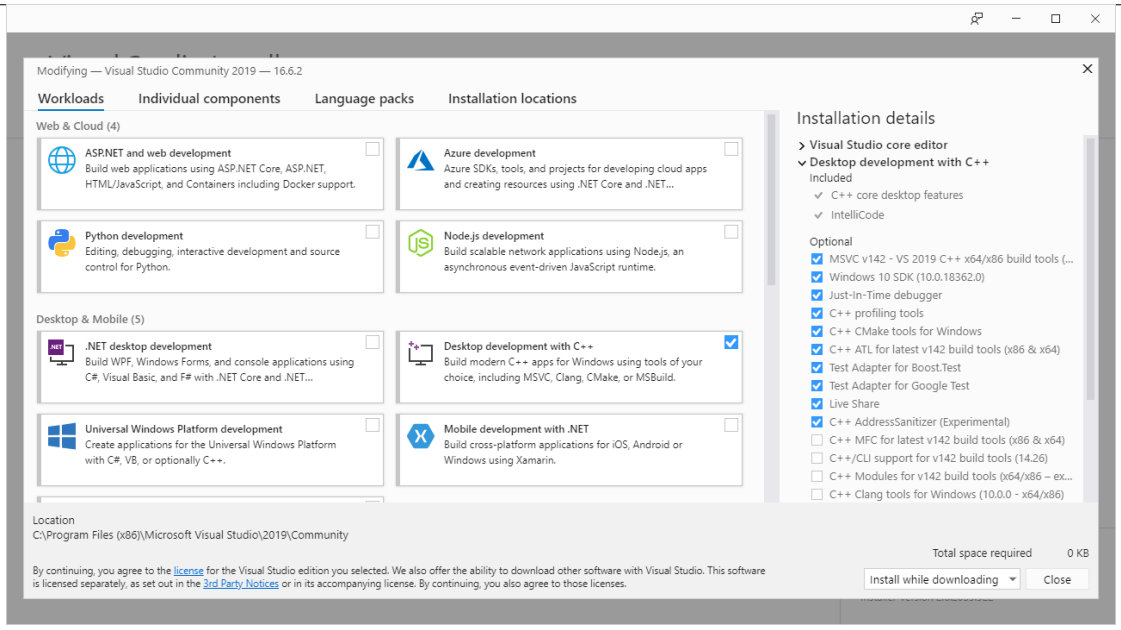
Let op: download de Community-editie. Zie figuur B.1.



Figuur B.1: *Downloaden van Community-editie.*

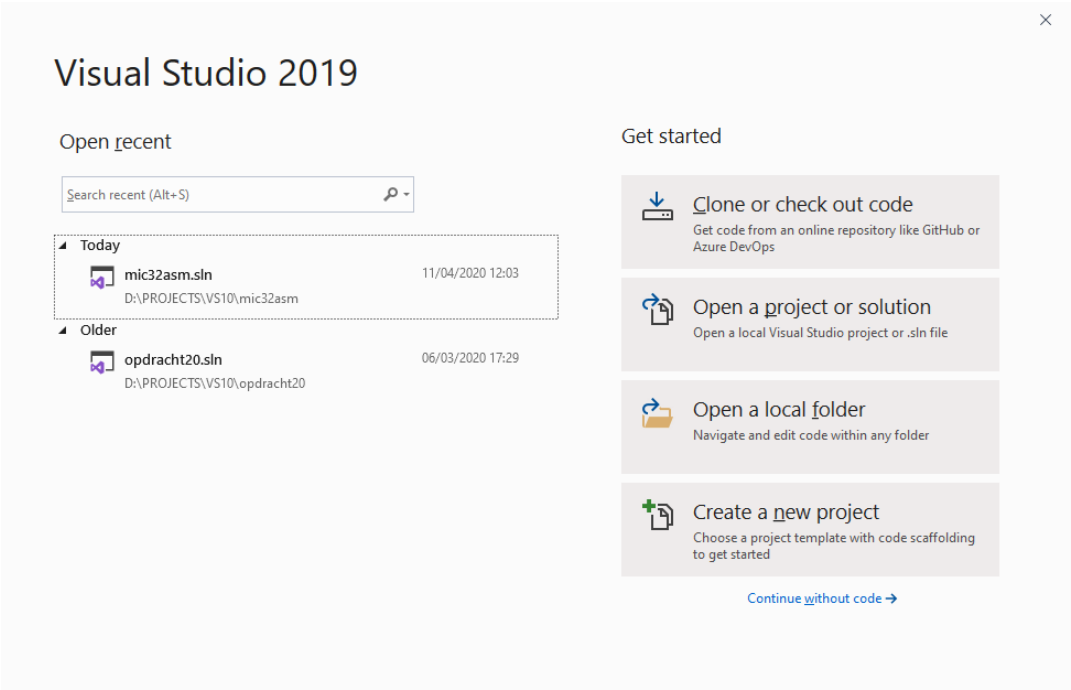
Er wordt nu een kleine installer gedownload. Start daarna de installer. Visual Studio is een groot programma, dus het kost nogal wat tijd om Visual Studio te installeren.

Visual Studio ondersteunt het ontwikkelen van applicaties met een veelzijdigheid aan mogelijkheden. Wij gebruiken de C/C++-compiler voor Desktop Development. Om te kunnen ontwikkelen moeten we een *workload* installeren. Selecteer de workload zoals te zien is in figuur B.2. Selecteer de opties zoals is aangegeven aan de rechterkant van de figuur. Klik daarna op `Install while downloading`.



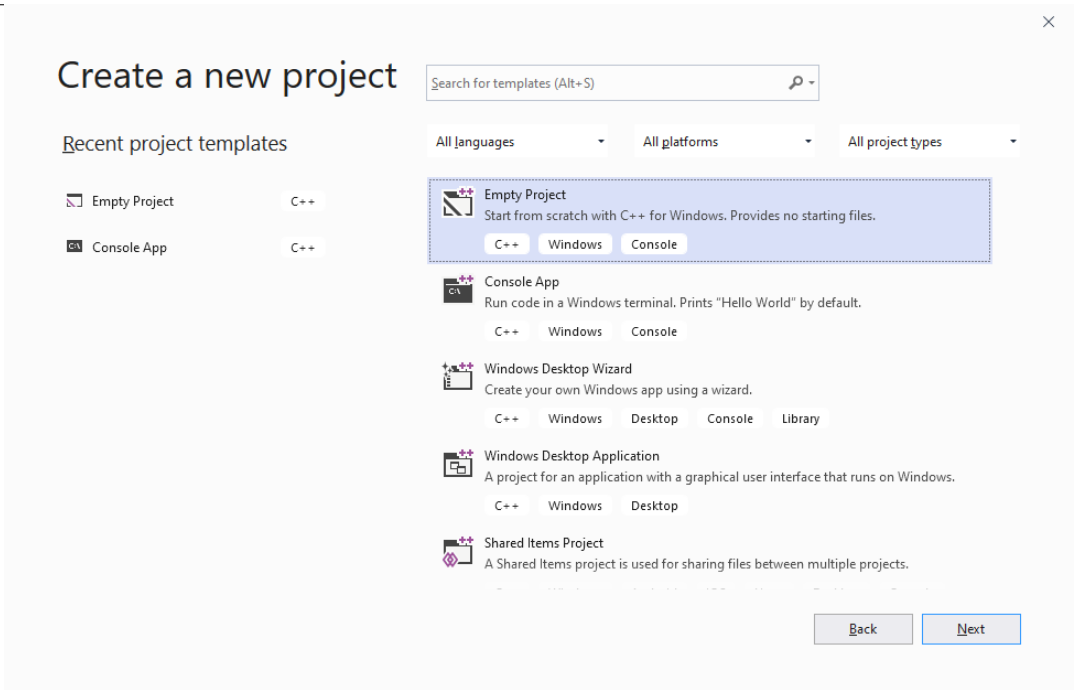
Figuur B.2: *Selecteren van de C++-workload.*

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.3. Klik op het kader `Create a new project`.



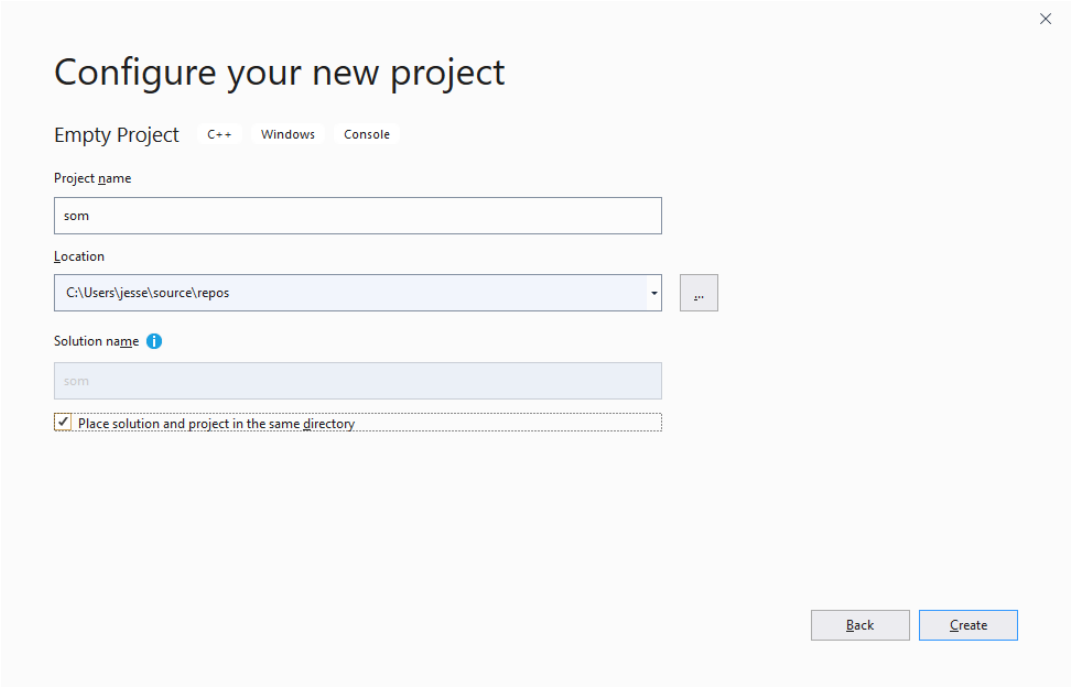
Figuur B.3: *Aanmaken van een nieuw project.*

Er wordt een nieuw scherm geopend, zie figuur B.4. Klik daarin op het kader `Empty Project`. **Klik niet op `Console App`.**



Figuur B.4: *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.5.

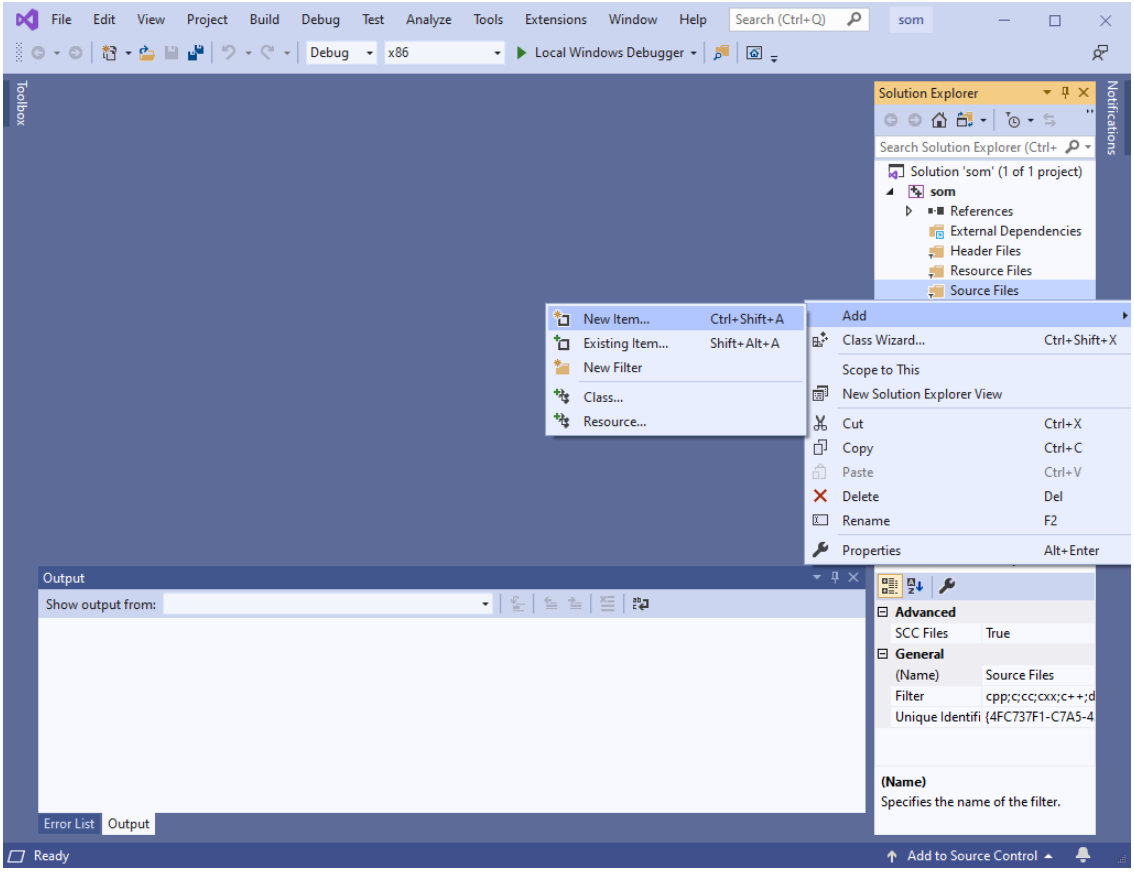


Figuur B.5: *Gegevens van het project invoeren.*

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 muis-pointer daar op staan en klik op de **rechter** muisknop.

Selecteer daarna de optie Add en daarna `New item...` Zie figuur B.6.



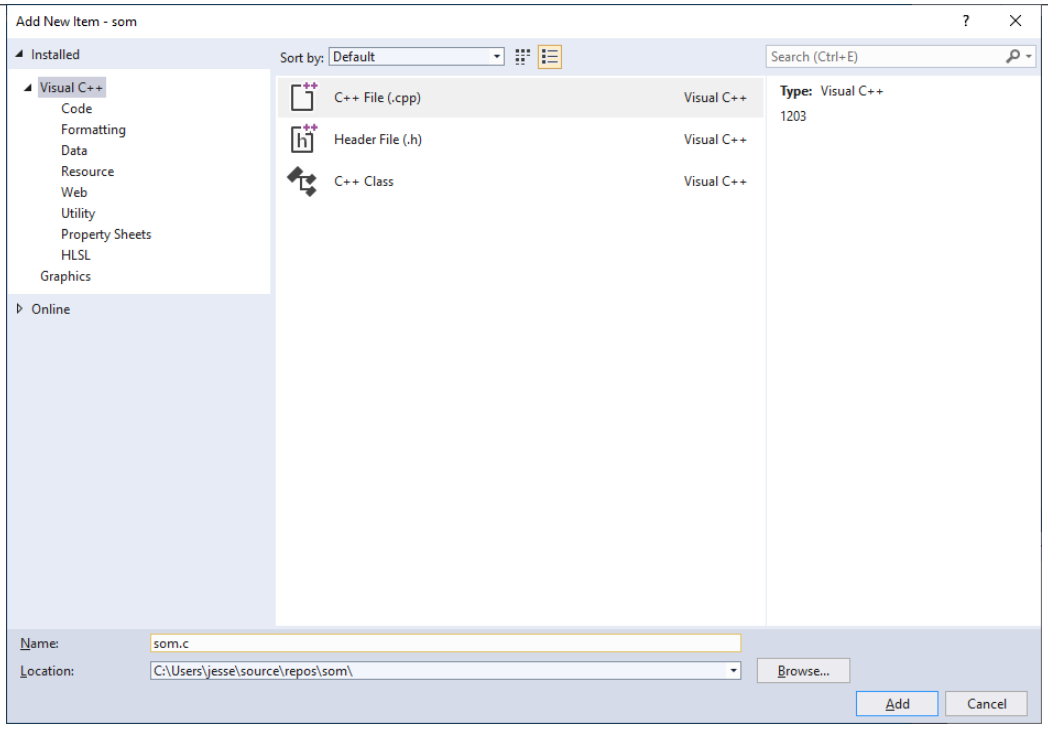
Figuur B.6: *Een nieuw bestand aanmaken.*

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.7.

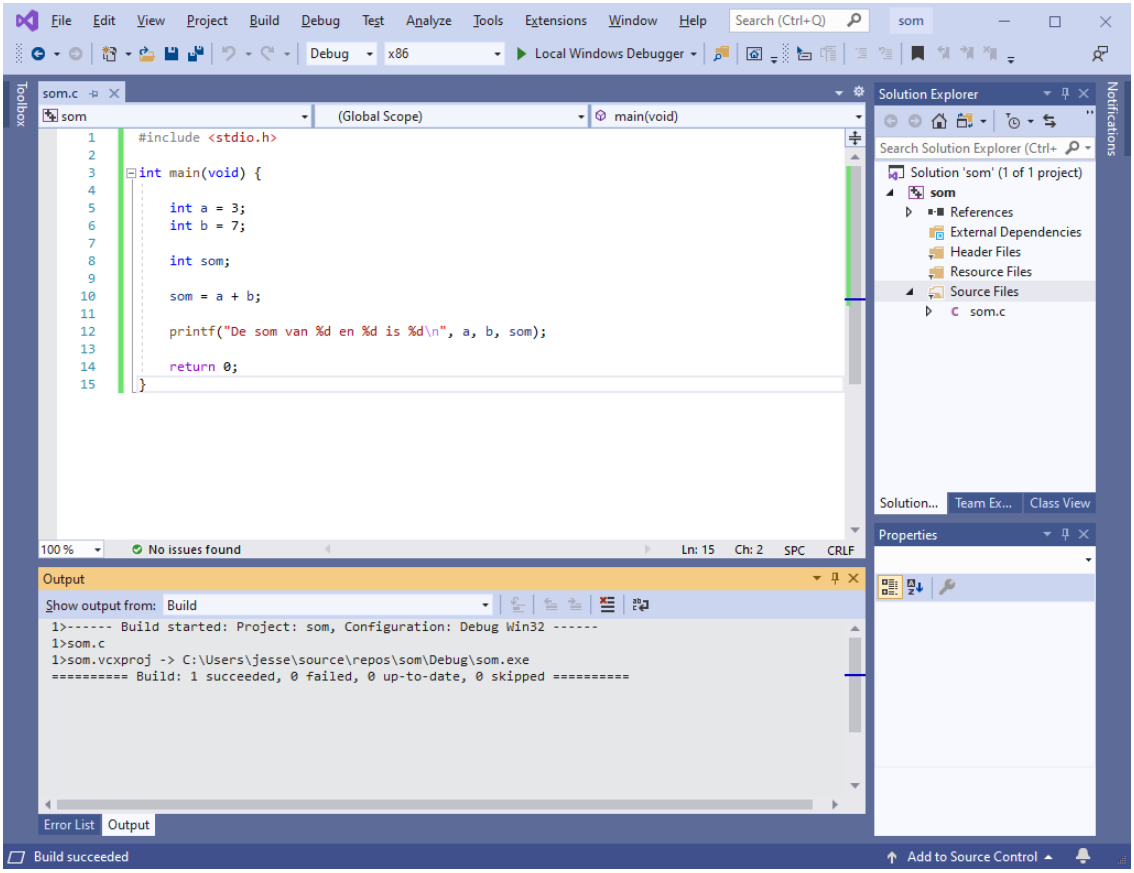
Voer het programma in zoals te zien is in figuur B.8. Klik daarna op de knop `Local Windows Debugger`. Het programma wordt nu gecompileerd en als er geen fouten zijn gevonden, wordt het programma uitgevoerd. Herstel eventuele fouten die door de compiler gevonden worden en herstart de compilatie.

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.9.

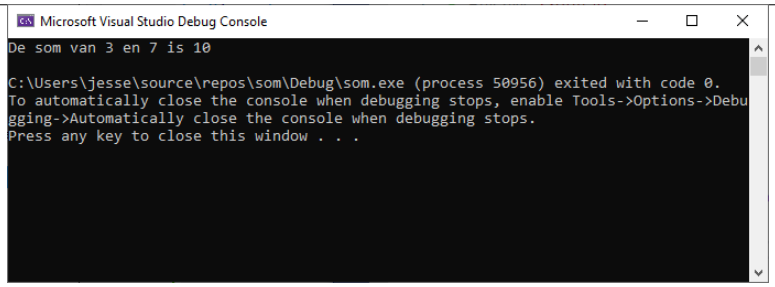
De tutorial is hiermee ten einde.



Figuur B.7: *Gegevens van het C-bestand invullen.*



Figuur B.8: *Compileren en starten van de executable.*



Figuur B.9: *Uitvoer van het programma in een console.*

C

Voorrangsregels 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 en dat kan ook een pointer (naar een variabele) zijn, `sizeof` berekent de grootte in bytes van een datatype of variabele.

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

Let erop dat `sizeof` een compile-time operator is, maar kan vanaf C99 ook gebruikt worden bij een dynamisch gedefinieerde array, bijvoorbeeld:

```
int a[b];
```

en b is niet bekend tijdens compile-time maar wel tijdens run-time.

Index

Operatoren en constanten
!, logische negatie, 28
!=, operator, 27
–, aftrekken, 26
*, dereferentie, 111
*, vermenigvuldigen, 26
+, optellen, 8, 26
++, operator, 35
,, functie, 50
,, operator, 36
--, operator, 35
->, operator, 125
., operator, 98
/, delen, 26
<, operator, 27
<<, links schuiven, 29
<=, operator, 27
=, toekennen, 8
==, operator, 12, 27
>, operator, 27
>=, operator, 27
>>, rechts schuiven, 29
?:, operator, 36
[], operator, 14, 86
%, modulus operator, 26, 77
&, adres, 110, 120, 152
&, bitsgewijze AND, 29
&&, logische AND, 28
^, bitsgewijze EXOR, 29
~, bitsgewijze inverse, 29
\0, nul-karakter, 24, 94, 116, 121
\\, backslash karakter, 24
\n, newline karakter, 24, 38
\r, carriage return karakter, 24
\t, horizontal tab karakter, 24
|, bitsgewijze OR, 29
||, logische OR, 28

A
aantal elementen bepalen, 88
accolades, 7
adres, 23
adres-operator, 110, 120
alfabetische ordening, 95
Allman-stijl, 17
AND, 29
argument, 8, 61, 64
argumenten
    aan een C-programma, 132
array, 13, 85
    als argument, 92
    als parameter, 92
    declaratie van, 85
    van pointers, 129
    van structures, 101
    vergelijken van, 91
array element, 85
assembler, 177
assembler-bestand, 177
assembly, 177
assert, functie, 65
assignmemt, 19
associativiteit, 26

B
backslash, 24, 150, 187
backslash karakter, 24
bare metal system, 1, 3
bepalen aantal elementen, 88
bestand, 2
    lezen, 157
    openen, 154
    schrijven, 155
    sluiten, 155
bestandsextensie, 154
bestandsextensies, 173
bestandsnaam, 149, 154
bestandssysteem, 2
besturingssyteem, 2
besturingsteken, 187
bibliotheek, 5, 174, 176
bit, 32
bit fields, 107
bitsgewijze operatoren, 29
blok, 39
blokstructuur, 39, 72

body, 61
body (functie), 177
bool, 23
break, keyword, 44
buffer overflow, 95, 153, 159
buffering, 163

C
Call by Reference, 124
Call by Value, 74
calloc, functie, 137
carriage return, 24, 154
case, keyword, 42
case sensitive, 20
char, keyword, 21
chip, 3
clock, functie, 79
command line argumenten, 132
compile-time, 4, 88, 195
compiler, 4
compiler suite, 173
compileren, 4
complex, 23
compound statement, 39
computer, 2
concatenation, 95
conditie, 11
conditionele expressie, 36
condtionele expressie, 126
const, keyword, 25, 125
constante, 23
constante array, 87
continue, keyword, 49
cross compiler, 4, 174

D
datatype, 21
datatypeconversie, 31
debuggen, 5
declaratie, 177
decrement operator, 35
default, keyword, 42
#define, preprocessor directive, 170
defined, preprocessor directive, 171
definitie, 8, 19, 177
directory, 149
do, keyword, 48
double, keyword, 14, 15, 22
dubbele dereferentie, 126
dynamische geheugenallocatie, 135

E
eendimensionaal, 87
element, 85
#elif, preprocessor directive, 172
else, keyword, 11, 41
#else, preprocessor directive, 172
embedded system, 3
end of file, 159, 165
end of line, 162
#endif, preprocessor directive, 171, 172
enkelvoudige datatypes, 23
enumeratie, 33, 105
EOF, 83, 158, 159, 165, 166
escape sequence, 24, 151, 187
executable, 4, 173
EXOR, 29
expressie, 8, 25
extensie, bestand, 154, 173
extern, keyword, 178

F
fall through, 42
fclose, functie, 155
fflush, functie, 158, 166
fgetc, functie, 158
fgets, functie, 158
file handle, 154
filemode, 154
filename, 154
float, keyword, 22
floating point, 13, 22
flowchart, 53
folder, bestand, 149
fopen, functie, 154
fopen_s, functie, 155
for, keyword, 46
format specification, 151
format string, 151
fprintf, functie, 155
fputc, functie, 158
fputs, functie, 158
fractie, 26
free, functie, 137
fscanf, functie, 157
fseek, functie, 158
functie, 4, 15, 61
functie-prototype, 177
functiedeclaratie, 63, 177
functieprototype, 63

G
geheugen, 2
geheugenallocatie, 135
gekoppelde lijsten, 139
generieke pointer, 113
getchar, functie, 37, 158, 165
globale variabele, 74, 178
goto, keyword, 52
gulden snede, 80

H
header-bestand, 7, 169
horizontal tab, 24

I
I/O, 2
I/O-adressen, 33
IDE, 173
`if`, keyword, 11, 40
`#if`, preprocessor directive, 171
`#ifdef`, preprocessor directive, 172
`#ifndef`, preprocessor directive, 172
`#include`, preprocessor directive, 169
increment operator, 35
indentation, 17
initialisatie, 8, 21
inspringen
    van code, 17
instructie, 2
`int`, keyword, 7, 21
integer, 19
Integrated Development Environment, 173
inverteren, 30
invoerbuffer, 163
ISA, 4
iteratie, 13, 46

K
K&R-stijl, 17
karakterconstante, 24
keyword, 11
kolomnummer, 89
komma-operator, 36, 50

L
Last In First Out, 66
leap year, 29
lezen uit een bestand, 157
library, 5, 174
lifo, 66
linefeed, 154
linked lists, 139
linken, 5
linker, 174, 176
logische operatoren, 28
lokale variabele, 71
`long`, keyword, 21
`long double`, keyword, 22
`long long`, keyword, 21
loop, 39
loop unrolling, 184
lus, 12, 39
lusvariabele, 13

M
machinecode, 4
macro, 83, 170, 174
macro-expansie, 170
`main`, functie, 7
`malloc`, functie, 135
`malloc.h`, header-bestand, 135
map, bestand, 149
`math.h`, header-bestand, 81
mathematical libray, 176
member, 97
member-wise, 99
memory leak, 137
microcontroller, 3
`modulo`, operator, 30
modulus operator, 26

N
naam, bestand, 154
nan, 32, 81
negatieoperator, 28, 40
newline, 24, 37, 38, 71, 154, 158, 166
niet-nul, 28
node, 139
Not a Number, *zie* nan
nul-bytes, 137
nul-karakter, 24, 94, 116, 121, 158, 159
NULL-pointer, 112, 137, 154

O
object-bestand, 174, 176
one's complement, 30
openen van een bestand, 154
operating system, 2
OR, 29
overflow, 32, 76

P
padnaam, 149
parameter, 64
pointer, 23, 109
    afdrukken van, 113, 152
    als functie-argument, 120
    als return-waarde, 124
    array van, 129
    naar array, 114
    naar array van pointers, 130
    naar functie, 133
    naar pointer, 126
    naar structure, 125
    naar void, 113
    NULL, 112
    rekenen met, 117
pop, software stack, 179
postfix, 35
pragma, 9
predefined macros, 171
prefix, 23, 35
preprocessor, 169
`printf`, functie, 8, 151
processor, 2
programma, 2
programmeerstijl, 17
programmeren, 4

promotie, 31
prototype (functie), 177
push, software stack, 179
putchar, functie, 158
puts, functie, 158

Q
qualifier, 11, 21, 22, 76
quicksort, 134

R
RAM, 2
realloc, functie, 137
recursieve functie, 75
register, 67
register, keyword, 182
relationele operator, 11
relationele operatoren, 27
return, keyword, 7, 67
returntype, 67
rewind, functie, 159
rijnummer, 89
ROM, 2
runtime, 4

S
samengesteld datatype, 85
scanf, functie, 9, 152
schrijven naar een bestand, 155
schrikkeljaar, 29
schuifoperator, 30
secundair geheugen, 2
self-referential structure, 139
short, keyword, 21
shortcut evaluation, 28
sizeof, operator, 34, 88, 93, 101
sluiten van een bestand, 155
software, 2
software stack, 179
stack, 67
stack pointer, 67, 179
stack, in software, 179
standard library, 5, 176
statement, 5, 7
static, keyword, 73–75, 179
stderr, 157
stdin, 157
stdint.h, header-bestand, 33
stdio.h, header-bestand, 7, 9, 154
stdout, 157
strcat, functie, 95
strcmp, functie, 95
strcpy, functie, 95, 98, 137
string, 5, 8, 24, 85, 94, 116, 152
string array, 24
string constante, 24
string.h, header-bestand, 95
strlen, functie, 95
stroomschema, 53
struct, keyword, 97
structure
    als argument, 99
    als parameter, 99
    array van, 101
    binnen structure, 102
    member, 97
    pointer naar, 125
    typedef, 100
structure identifier, 97, 178
suffix, 173
switch, keyword, 42

T
tekenbit, 30
toekenning, 8, 19
toolchain, 4, 173
translation unit, 176
tweedimensionaal, 88
type cast, 32, 135
typedef, keyword, 100

U
uitvoerbaar bestand, 4, 7, 173
unaire operator, 26
#undef, preprocessor directive, 171
underscore, 18, 20
union, keyword, 105
unsigned, 21, 22

V
variabele, 5, 8, 19
    globaal, 74
    lokaal, 71
void, keyword, 7, 62
void-pointer, 113, 135
volatile, keyword, 183
voorrangsregels, 37, 195

W
waarheidstabel, 29
while, keyword, 13, 45
whitespace, 153
wortelformule, 103