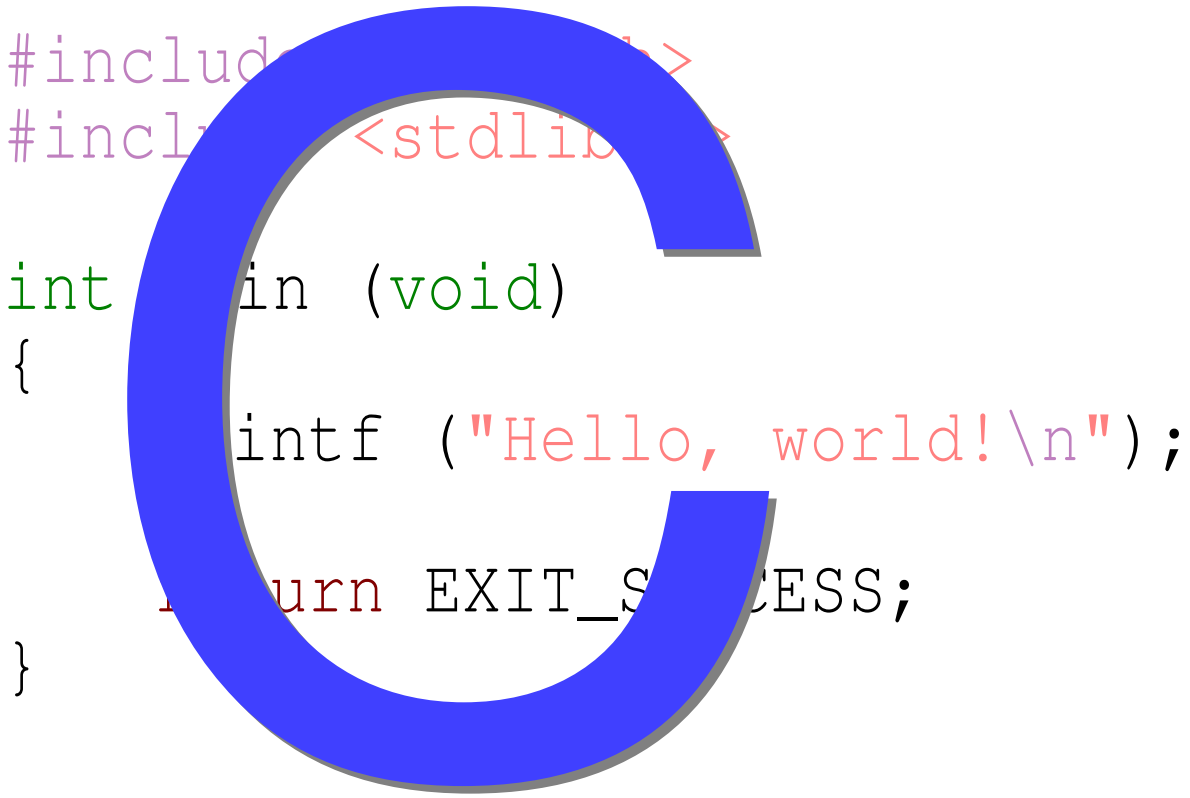
Code EE1400

Practicum Handleiding

Programmeren in C

Delft University of Technology



B. Jacobs, X. van Rijnsoever, Dr. ir. A. J. van Genderen

TU Delft Delft University of Technology

Challenge the future

Practicum Handleiding Programmeren in C V2.1 (10 december 2012)

Deze handleiding is getypeset met LaTeX in Nimbus Roman 10 pt. De schematische afbeeldingen zijn geproduceerd met Dia, de foto's en screenshots zijn bewerkt met GIMP en ImageMagick.

Een woord vooraf

Welkom!

Allereerst welkom bij het practicum Programmeren in C! Programmeren is iets dat niet enkel uit een boek geleerd kan worden, voor het verkrijgen van vaardigheid in het programmeren is het vooral noodzakelijk om in de praktijk te oefenen. In dit practicum zal je de stof uit de colleges en uit het boek in de praktijk brengen.

Het niveau van de handleiding is gericht op de beginnende C-programmeur. We hebben geprobeerd de stof van de colleges en het boek op een interessante manier te verwerken in de opdrachten. We hopen dat je vindt dat we daarin zijn geslaagd, maar mocht dat niet zo zijn, dan is onderbouwde kritiek altijd welkom!

Succes met het practicum!

Delft, december 2012

B. Jacobs (S.Jacobs@TUDelft.nl)
X. van Rijnsoever (X.vanRijnsoever@TUDelft.nl)
Dr. ir. A.J. van Genderen (A.J.vanGenderen@TUDelft.nl)

De handleiding

De opdrachten beschreven in deze practicumhandleiding dienen in 6 weken te worden afgerond. Een hoofdstuk begint met een overzicht van de leerdoelen en de vereiste voorbereiding voor de practicumsessie, daarna volgen de opdrachten. De practicum-opdrachten volgen de stof van de colleges en het boek. Hoofdstuk 1 gebruikt de kennis van het eerste college en geeft een introductie in C, het compileer-proces en de gebruikte Integrated Development Environment (IDE). De stof van het tweede college komt aan bod in de hoofdstukken 2 en 3. Hoofdstuk 2 behandelt datatypen en programmabesturing. In hoofdstuk 3 worden functies behandeld. In hoofdstuk 4 is de stof van het derde college verwerkt en worden arrays en pointers geïntroduceerd, en worden strings nader besproken. Hoofdstuk 5 behandelt de stof van het laatste college: structures en datastructuren.

In de handleiding komen een aantal kaders voor met een verschillend icoontje. Deze kaders bevatten extra informatie die niet direct noodzakelijk is voor het practicum, maar wel handig kan zijn om te weten.

Let op!
Dit kader waarschuwt voor vaak voorkomende fouten en problemen.

Tip!
Dit kader geeft een tip die handig kan zijn bij het aanpakken van een probleem uit de opdrachten.

Achtergrond informatie!
Dit kader geeft achtergrond informatie over bijvoorbeeld praktijk toepassingen, of extra functionaliteit die niet noodzakelijk is voor dit practicum, maar misschien wel van toepassing kan zijn bij andere practica.

Voor zover er sprake is van belangrijke verschillen tussen een implementatie van een opdracht onder Windows of Linux, wordt dit aangegeven middels de volgende kaders:

Windows
Dit kader bevat informatie die alleen van belang is voor het implementeren van de opdracht onder Windows.

Linux
Dit kader bevat informatie die alleen van belang is voor het implementeren van de opdracht onder Linux.

Het practicum

Het practicum moet individueel worden uitgevoerd. Het is van groot belang om een goede voorbereiding te hebben, anders is het niet mogelijk om binnen de gestelde tijd het practicum af te ronden. De meeste studenten zullen meer tijd nodig hebben voor de opdrachten dan de ingeroosterde practicumtijd, het is dan ook uitdrukkelijk de bedoeling buiten de practicumtijd alvast aan de opdrachten te werken.

Het practicum mag worden uitgevoerd onder Windows of Linux, onder beide systemen is de practicum-software beschikbaar. De gebruikte software (Code::Blocks en GCC) is open source en dus ook thuis te downloaden en te installeren. De uiteindelijk code *moet* ANSI C zijn en werken onder de GCC compiler.

Tijdens het practicum zijn assistenten beschikbaar om te helpen bij problemen. Zorg ervoor dat je tenminste de tips hierna zelf hebt doorlopen alvorens assistentie in te roepen. Wanneer je deze stappen niet hebt doorlopen, zullen de assistenten je niet kunnen helpen.

Van een aantal opdrachten moet de C-code ingeleverd worden. Dit wordt bij de desbetreffende opdrachten aangegeven met het icoon dat hiernaast staat afgebeeld. Meer informatie hierover kun je vinden op pagina v.

Opdrachten die niet van zo'n icoon zijn voorzien hoeven niet te worden ingeleverd. Voor het begrip en voor het verkrijgen van de noodzakelijke oefening is het echter wel raadzaam om deze opdrachten uit te voeren.

Om duidelijk weer te geven wat de invoer en de uitvoer van een programma is, wordt gebruik gemaakt van twee gescheiden blokken. Bij de meeste programma's dient eerst alle invoer te worden ingelezen en dient daarna de uitvoer te worden gegenereerd. Bij andere programma's dient onmiddelijk nadat een regel invoer is gegeven, uitvoer te worden gegeneerd. In het laatste geval komen in de praktijk dan de invoer en uitvoer door elkaar op het scherm te staan. Wat precies de bedoeling is staat bij de opdracht aangegeven.

Van verschillende opdrachten is (een deel van) de code beschikbaar. Deze files zijn te vinden op Blackboard.

Tips bij het programmeren

In het ideale geval doen de programma's precies wat de bedoeling is en zijn alle vragen direct duidelijk. Uiteraard is de werkelijkheid vaak wat minder ideaal, om je in die gevallen een beetje op weg te helpen, volgen hier wat tips:

Programmeren en C

Dit vak gaat over programmeren in C. Programmeren is echter een vak dat voor een groot deel los staat van de gebruikte programmeertaal. Om de focus op het oplossen van het gegeven probleem te houden, kan het vaak handig zijn om niet direct in de uiteindelijke programmeertaal te werken. De methode die in deze handleiding hiervoor wordt gebruikt, is het Nassi-Shneiderman diagram (NSD). Door gebruik te maken van NSD's ligt de focus niet op de eigenaardigheden van de programmeertaal, maar op het op een structurele manier oplossen van een programmeerprobleem. In appendix D worden NSD's besproken.

Werk netjes

Hoewel de uiteindelijke code door een compiler wordt verwerkt, moet de code gelezen en geschreven kunnen worden door mensen. Dit betekent dat het belangrijk is om de code op een duidelijke en consistente manier op te maken. Er zijn verschillende methodes om C-code te indenteren en op internet zijn verhitte dicussies te vinden over wat nou echt de beste manier is. In de praktijk maakt dit niet veel uit, als je maar duidelijk structuur kan onderscheiden en de gekozen methode consistent gebruikt. Let erop dat studentassistenten je niet zullen helpen als de code geen structuur heeft! De gebruikte IDE heeft mogelijkheden om de code volgens verschillende methodes automatisch te indenteren:Plugins → Source code formatter (AStyle)). Maak daar gebruik van !

Het programma `indent`
Degenen die via de commandline in Linux werken kunnen gebruik maken van het programma `indent`. Bekijk de man-page voor een overzicht van de (vele) mogelijke opties.

Gebruik (zinnig) commentaar

Commentaar is onmisbaar om een programma te kunnen onderhouden, maar het is dan wel van belang dat het commentaar zinnige informatie bevat. Gebruik voldoende en

terzake doend commentaar in je programma's, op deze manier is het ook voor assistenten veel eenvoudiger te volgen wat de code moet doen.

Compileer met warnings enabled

Compileer de code met het waarschuwingsniveau van de compiler zo hoog mogelijk (`-Wall`). De code die geschreven moet worden, moet voldoen aan de ANSI-C standaard. Door dit aan de compiler op te geven kan deze waarschuwen voor constructies die hier niet aan voldoen (`-ansi`, voor striktere controle samen met `-pedantic`). Hoewel bij warnings de code wel wordt gecompileerd, is het (zeker voor de eenvoudige opdrachten van dit practicum) een duidelijk teken dat er iets niet helemaal in orde is. Zorg ervoor dat je begrijpt wat de warning betekent en pas de code aan om deze warning te voorkomen.

Rebuild het gehele programma als warnings onverwachts verdwenen zijn

Zolang het programma uit een enkele C-file bestaat, wordt uiteraard deze C-file telkens weer gecompileerd. Indien een programma echter uit meerdere files bestaat, dan is het mogelijk dat bepaalde C-files reeds gecompileerd zijn. Bij het bouwen van het uiteindelijke programma worden deze files (indien ze niet gewijzigd zijn) dan niet nogmaals gecompileerd. Dit zorgt voor een hogere compileer-snelheid, maar verbergt tevens warnings die eerder tijdens het compileren van die C-files optraden. Rebuild het gehele programma om te controleren of er bij geen enkele file meer warnings optreden.

Denk na over het probleem als een programma niet werkt

Het lijkt logisch dat je moet nadenken over het probleem als een programma niet werkt, maar in de praktijk is de neiging om even gauw een kleine ad-hoc aanpassing aan het programma te maken erg groot. Helaas is dat in de meeste gevallen niet de juiste methode. Het is gebruikelijk dat een algoritme in één of misschien twee gevallen een uitzondering in de code vereist, maar als de code uit meer uitzonderingen dan gewone gevallen bestaat, dan ligt het probleem ergens anders.

Gebruik print statements om het programma te "debuggen"

Wanneer je programma niet werkt, zou je een *debugger* kunnen gebruiken, maar een eenvoudiger manier is vaak om op de juiste plaatsen enkele `printf`-statements te plaatsen. Op die manier kun je zien hoe het programma verloopt en zo de fout achterhalen.

Test programma's in delen

Vooral bij wat grotere programma's geldt: Probeer niet het hele programma te maken en het dan pas te testen, maar test telkens kleine delen apart. Begin bijvoorbeeld eerst met het inlezen van de gegevens en print dan de ingelezen gegevens weer om te zien of alles goed is ingelezen. Voeg dan de eerste bewerking toe en print het tussenresultaat. Ga op deze manier verder totdat alle functionaliteit is geimplementeerd.

Regels met betrekking tot het inleveren van de code

De code van een aantal opdrachten moet worden ingeleverd. Bij de desbetreffende opdrachten is het hiernaast afgebeelde icoon opgenomen. Om ervoor te zorgen dat de code eenvoudig en snel kan worden nagekeken, moet je je aan de volgende regels houden:

- Inleveren gaat via CPM (`https://cpm.ewi.tudelft.nl/`). Wil je snel weten of je code wordt goedgekeurd, vraag dan een assistent om er naar te kijken, maar lever daarna je code alsnog in via CPM.
- Lever alléén source-code in (.c-files en .h-files), dus *geen* project-files en executables en dergelijke.
- Als je meer dan één source-file moet inleveren, pak dan de source-files in in een .zip-file (Onder Windows: files selecteren en dan rechts klikken om bijvoorbeeld 7-Zip → Add to "naam.zip" uit te voeren) en lever die in.
- Lever de code in bij de juiste opdracht.
- Voorzie elke source-file van je naam, studienummer en het nummer van de opdracht. Doe dit exact zoals in dit voorbeeld:

```
1 /*
2  * Student: A. Programmer
3  * Nummer: 1234567
4  * Opdracht: 2.1
5  */
```

- Zorg ervoor dat je programma ANSI-C is en compileert zonder warnings en errors. De code wordt op de volgende manier gecompileerd:
  `gcc -ansi -pedantic -Wall -lm c_source.c`
  Bij Code::Blocks kun je de compiler opties `-ansi`, `-pedantic` en `-Wall` zetten in het menu Settings → Compiler and debugger....
  Bij compilatie met warnings of errors wordt het programma afgewezen en verder niet nagekeken.
- Voorzie je programma van voldoende (en zinnig) commentaar.
- Zorg dat het programma is voorzien van een consistente indentatie. Bij Code::Blocks kun je hiervoor het volgende commando gebruiken: Plugins → Source code formatter (AStyle).

- Zorg ervoor dat je programma exact dezelfde uitvoer specificaties heeft als in de opdracht vermeld staan. Maak dus geen mooie interface om je programma (welkomstmelding ofzo), dit leidt ertoe dat het programma niet door de automatische controle heen kan komen.

Het niet voldoen aan een van deze regels, leidt tot het afkeuren van je inzending! Als een programma is afgekeurd, dan zal de reden daarvoor worden vermeld. Vergeet niet dat een afgekeurd programma als verbeterde versie nogmaals ingeleverd moet worden.

Bespaar tijd
Bespaar tijd van jezelf en van de assistenten door alleen je code in te leveren wanneer deze aan bovengenoemde eisen voldoet. Lukt het niet om een bepaald probleem op te lossen, vraag het de assistenten of stuur een e-mail naar de docent (A.J.vanGenderen@TUDelft.nl).

Inhoudsopgave

Hoofdstuk 1

Hello, world!

Theorie

- Hoofdstuk 0 – 1.6 en 2 uit [1]
- Hoorcollege 1

Leerdoelen

De leerdoelen voor deze sessie zijn:

- Kennismaken met het gebruikte platform en de gebruikte software.
- Een eenvoudig programma kunnen compileren en uitvoeren.
- Een eenvoudig programma schrijven en testen.

Inleiding

Van harte welkom bij de eerste sessie van het C practicum! Deze sessie begint met een overzicht van wat er allemaal nodig is om van een programma geschreven in C een uitvoerbaar bestand te maken. Daarna volgt een korte tutorial voor Code::Blocks, een *integrated development environment (IDE)*. Tot slot volgen twee programmeeropdrachten die aansluiten bij de behandelde stof uit het boek.

Voor het doorlopen van dit hoofdstuk (en de de volgende hoofdstukken uit deze handleiding) is het noodzakelijk dat je de beschikking hebt over een PC waarop Code::Blocks geinstalleerd is. Voor het installeren Code::Blocks, zie de Blackboard site van dit vak, onder Assignments.

1.1 Hello, world!

Opdracht 1.1: Compilatie en executie

Lees bijlage B.

Opdracht 1.2: Code::Blocks

Lees bijlage C en maak de bijbehorende opdrachten.

1.2 Verschillende temperatuurschalen

De buitentemperatuur wordt (hier) meestal uitgedrukt in °C. Temperaturen kunnen ook op andere manieren worden uitgedrukt, zoals bijvoorbeeld in K of in °F. Om een gegeven temperatuur uit te drukken in °C kunnen de formules 1.1 en 1.2 gebruikt worden. Om een temperatuur in °C uit te drukken in een andere schaal, kunnen de formules 1.3 en 1.4 gebruikt worden.

$$T_{Celcius} = T_{Kelvin} - 273.15 \tag{1.1}$$

$$T_{Celcius} = \frac{T_{Fahrenheit} - 32}{1.8} \tag{1.2}$$

$$T_{Kelvin} = T_{Celcius} + 273.15 \tag{1.3}$$

$$T_{Fahrenheit} = 1.8 \cdot T_{Celcius} + 32 \tag{1.4}$$

Opdracht 1.3: Temperatuurschalen

Schrijf een programma dat voor een temperatuur in °C uitrekent wat de bijbehorende waarden in K en in °F zijn. Gebruik als type variabele hiervoor een double. Print vervolgens de resultaten naar standaard uitvoer.

input:

Een temperatuur in °C (double). Bijvoorbeeld:

```
10
```

output:

Gescheiden door tabs: op de eerste regel de letters C, K en F, en op de tweede regel de temperatuur in de verschillende schalen, nauwkeurig tot op 2 decimalen achter de komma en ook gescheiden door tabs. Bijvoorbeeld:

```
C       K       F
10.00   283.15  50.00
```

Lezen en printen van doubles

Een variabele van het type double kan worden ingelezen met scanf met de format specifier `%lf`. De letter `l` is hierbij nodig omdat het double betreft en geen float. Vergeet verder niet om het karakter `&` te plaatsen voor de naam van de variabele. Dus bijvoorbeeld `scanf ("%lf", &d);`
Bij het printen wordt ook bij een double bij de format specifier de letter `l` niet gebruikt. Een punt gevolgd door een getal n kunnen worden gebruikt om een variabele op n decimalen nauwkeurig achter de komma te printen. Een double kan dan bijvoorbeeld worden geprint als een getal met 2 decimalen achter de komma met `printf ("%.2f", d);`

Tab in C

Net als een newline een speciaal teken is in C (`'\n'`), heeft ook tab een speciale code: `'\t'`

1.3 Geometrische reeksen in C

Een bekend probleem in de wiskunde is het berekenen van de som van een geometrische rij: de geometrische reeks. Deze reeks is gedefiniëerd volgens vergelijking 1.5.

$$S(N) = \sum_{n=0}^{N} ar^n = a + ar + ar^2 + ar^3 + \ \ldots\ + ar^{N-1} + ar^N \tag{1.5}$$

waarbij a en r parameters zijn met een reële waarde en N en positief geheel getal is.

Opdracht 1.4: Geometrische reeks (theorie: §1.5 en §1.6 uit [1])

Schrijf zelf een programma dat de som in vergelijking 1.5 uitrekent. Eerst moeten de parameters a, N en r van standaard invoer worden ingelezen. Vervolgens rekent het programma de som met behulp van een for-lus uit (gebruik een hulp-variabele om het tussenresultaat van de som bij te houden). Print het resultaat naar standaard uitvoer en rondt daarbij af op 2 cijfers achter de komma.

input:

De parameters a (double), N (integer) en r (double). Bijvoorbeeld:

```
5   7   0.7
```

output:

Het resultaat:

```
15.71
```

Wiskundige functies in C

Een aantal wiskundige functies zijn in C gedefiniëerd in de math-library, zoals de functie `pow()` voor machtsverheffen. Om deze functies te kunnen gebruiken moet de regel `#include<math.h>` aan de lijst met ingevoegde headerbestanden worden toegevoegd. Ook moet soms (wanneer geen Code::Blocks wordt gebruikt) de optie `-lm` nog aan de linker worden meegegeven. Zie voor meer informatie bijlagen B en C en §3.10 uit [1].

Hoofdstuk 2

Datatypen en programmabesturing

Theorie

- Hoofdstuk 3 – 4 van [1]
- Practicum Handleiding Bijlage D
- Hoorcollege 2

Leerdoelen

In deze sessie zal je:

- kennis maken met de verschillende fundamentele datatypen
- leren werken met de verschillende statements voor programmabesturing

Inleiding

Deze practicum-sessie behandelt een tweetal onderwerpen. Als eerste worden datatypen behandeld. In het algemeen is het gebruik van de verschillende datatypen niet moeilijk, maar er zijn verschillende randgevallen waarvan de programmeur zich bewust moet zijn bij het kiezen van een specifiek datatype. De opdrachten in dit eerste deel focussen dan ook op deze randgevallen.

Het tweede onderwerp wordt gevormd door statements voor programmabesturing. Zo'n statement is al geïntroduceerd in hoofdstuk 1 (de for-lus), maar hier worden deze statements meer uitgebreid behandeld.

2.1 Datatypen en typecasts

2.1.1 Verschillende soorten gehele getallen

De taal C kent verschillende datatypen voor het opslaan van gehele getallen. In tegenstelling tot de meeste programmeertalen, ligt de grootte van deze datatypen niet vast, maar is afhankelijk van het platform. Het meest standaard datatype voor gehele getallen is *int*, maar ook *char* en *long* kunnen worden gebruikt.

Platform-afhankelijke informatie over datatypen
Voor het schrijven van portable code, is het van belang te kunnen weten wat de maximale waarde is die in de verschillende datatypen kan worden opgeslagen. Deze informatie is opgenomen in een door de ANSI-standaard voorgeschreven header-file `limits.h`. Deze header-file wordt bijvoorbeeld beschreven in appendix A van [1].

Opdracht 2.1: Integers

Compileer het volgende programma en voer het tweemaal uit. Bekijk het resultaat:

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          int i;
6          short j = 20000;
7          int k = 20000;
8          int l = 2000000000;
9          long m = 2000000000;
10
11         printf ("i=%d\n", i);
12         printf ("  j=%d,   k=%d,    l=%d,   m=%ld\n", j, k, l, m);
13         j = 2*j, k=2*k, l=2*l, m=2*m;
14         printf ("2*j=%d, 2*k=%d, 2*l=%d, 2*m=%ld\n", j, k, l, m);
15
16         return 0;
17 }
```

Het gevonden resultaat zal afhankelijk zijn van welke compiler en welke PC gebruikt wordt. Het kan zijn dat de compiler de variabele `i` initialiseert op 0, maar vaak ook heeft `i` een willelekeurige, onvoorspelbare waarde. Verder zal de waarde `2 * 20000` meestal niet meer passen in een variabele van het type short, en zal waarschijnlijk een negatieve waarde geprint worden vanwege de overflow. De waarde `2 * 20000` zal waarschijnlijk wel passen in een variabele van het type int, maar de waarde `2 * 2000000000` zal waarschijnlijk niet meer passen in een type int. Op systemen waarbij voor het type long 64 bits gebruikt worden zal deze laatste waarde geen probleem geven.

Opdracht 2.2: Chars

Wat doet het volgende programma?

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          char c;
6
7          c = 'a';
8          printf ("%c=%d\n", c, c);
```

```
9
10          c = 'z';
11          printf ("%c=%d\n", c, c);
12
13          c = 'a';
14          printf ("%c %c %c\n", c, c+1, c+2);
15
16          for (c = 'a'; c <= 'z'; c++) {
17                  printf ("%c", c);
18          }
19          printf ("\n");
20
21          return 0;
22  }
```

In C wordt het type char gebruikt om een karakter te representeren. Het karakter wordt daarbij opgeslagen als een integer, meestal volgens de ASCII codering (zie Bijlage A). Naast dat de integer waarde geprint kan worden, kunnen er ook rekenkundige bewerkingen op een variabele van type char gedaan worden. Een variabele van het type char kan dan ook op dezelfde manier gebruikt worden als een variabele van het type int, met dat verschil dat de maximum waarde bij een char wel kleiner zal zijn omdat een char slechts in 1 byte (8 bits) wordt opgeslagen.

2.1.2 Strings

Strings in C worden gerepresenteerd als een rij chars achter elkaar, afgesloten door het speciale char '\0'. Dit afsluit-teken is echter niet te zien in de specificatie van de string. De string wordt gespecificeerd in de C code als een rij karakters tussen dubbele quotes (zoals "this_is_a_string"), waarbij de compiler voor de representatie in het programma de quotes verwijdert en een '\0' toevoegt aan het einde. Wil men tekens invoeren die een speciale betekenis hebben, zoals de dubbele quotes, dan dient men deze tekens vooraf te laten gaan door een 'escape' karakter. Voor veel karakters is dit het backslash teken (zie §3.3 van [1]). Het % karakter heeft binnen `printf()` nog een speciale betekenis, en het escape karakter hiervoor is het karakter zelf.

Opdracht 2.3: Speciale tekens

Het onderstaande programma

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          printf ("\"Let\'s program 100%% in C!\"\n");
6
7          return 0;
8  }
```

zal de volgende string printen (inclusief quotes en inclusief een newline):

```
"Let's program 100% in C!"
```

2.1.3 Floating point getallen

In de tot nu toe besproken datatypen is het alleen mogelijk om gehele getallen (integers) op te slaan. Vaak is het echter handig of noodzakelijk om met reële-getallen (floating point) te kunnen rekenen. In C zijn daarvoor de typen *float* en *double* beschikbaar. Om floating point getallen te kunnen representeren in een computer wordt gebruik gemaakt

van een codering [1]. Het gebruik van deze codering zorgt ervoor dat het rekenen met floating point getallen wat onverwachte eigenschappen heeft. De volgende opdrachten verduidelijken dit effect.

Opdracht 2.4: Floating point getallen: bereik

Het volgende programma dient de getallen af te drukken in de range 1.0e20 tot 1.0e20 + 1.0e4. Wat gebeurt er echter ?

```
#include <stdio.h>

int main (void)
{
        int s = 1;
        double a = 1.0e20;
        double b = 1.0e20 + 1.0e4;

        printf ("a = %f, b = %f\n", a, b);

        for (a = 1.0e20; a < b; a = a + 1)
                printf ("step %d: a = %f\n", s++, a);

        return 0;
}
```

De verklaring is te vinden in het feit dat een floating point getal maar een eindige precisie heeft om waarden te representeren. Hoewel de waarde 1 op zich natuurlijk gepresenteerd kan worden met een double, valt de waarde 1 in het niet wanneer hij wordt opgeteld bij de waarde 1.0e20.

Bereik van floating point getallen
Floating point getallen hebben een sterk niet-lineair bereik: rondom 0 is het mogelijk om zeer kleine getallen weer te geven (hoge nauwkeurigheid), richting oneindig is het mogelijk zeer grote getallen weer te geven (groot bereik).

Opdracht 2.5: Floating point getallen: nauwkeurigheid

Bij het volgende programma zou je wellicht verwachten dat het programma 90 iteraties uitvoert en vervolgens stopt. Waarom werkt het echter niet op deze manier? (Hint: vervang `%f` door `%.20f`).

```
#include <stdio.h>

int main (void)
{
        double one_third = 1.0 / 3;
        double t = 0;

        while (t != 30) {
                printf ("%f\n", t);
                t += one_third;
        }

        printf ("done!\n");

        return 0;
}
```

[1] Voor de meeste systemen is dit de IEEE standaard 754 (`http://en.wikipedia.org/wiki/IEEE_754`)

Nauwkeurigheid van floating point getallen
Rekenen met floating point getallen voldoet niet altijd aan de gebruikelijke algebraische regels als commutativiteit en associativiteit. Dit kan uiteraard leiden tot ongewenst gedrag van een programma. Vergelijken van floating point getallen kan dan ook beter niet met een gelijkheids-controle plaatsvinden, maar aan de hand van een maximale afwijking ten opzichte van het te vergelijken getal.

2.1.4 Typecasts

In C heeft elke variabele een type. Soms is het noodzakelijk om (tijdelijk) een type om te zetten in een ander type, daarvoor kunnen *typecasts* worden gebruikt. De taal schrijft ook een aantal automatische typecasts voor, bijvoorbeeld bij het uitvoeren van een berekening op twee verschillende datatypen.

Opdracht 2.6: Typecasts

Het resultaat van het volgende programma is 0 in plaats van 0.75. Maak gebruik van explicite typecasts om het juiste antwoord te krijgen.

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          int a = 3;
6          int b = 4;
7          double c = a / b;
8
9          printf ("double c = %f\n", c);
10
11         return 0;
12 }
```

2.2 Vergelijkingsoperatoren en conditionele statements

Met de tot nu toe gebruikte statements en expressies is het mogelijk om allerhande berekeningen uit te voeren, maar om bijvoorbeeld grenzen van ingevoerde getallen te kunnen controleren, zijn extra statements nodig. De *if*- en *switch*-statements kunnen worden gebruikt om een beslissing te nemen. Deze beslissing wordt gemaakt op basis van het resultaat van een conditioneel statement.

Opdracht 2.7: Vergelijkingsoperatoren en lazy-evaluation

Een belangrijke mogelijkheid in C is het samenvoegen van meerdere vergelijkingen in een enkel conditioneel statement. Dit kan met de logische operatoren && en ||. Hierbij wordt gebruik gemaakt van lazy-evaluation (in het boek aangeduid met “short-circuit evaluation”, zie §4.4 van [1]), wat wil zeggen dat de test wordt uitgevoerd van links naar rechts en dat de test wordt afgebroken zodra duidelijk is wat de uitkomst is. Bijvoorbeeld: bij een test && is het duidelijk dat de uitkomst 0 is wanneer de linkse expressie 0 is.

Het volgende programma test de waardes van twee ingevoerde getallen. De code maakt gebruik van verschillende if-statements. Herschrijf de code zó dat er slechts één if-statement nodig is. Maak hierbij gebruik van de effecten van lazy-evaluation.

```c
#include <stdio.h>

int main (void)
{
        int     i, j;
        int     do_print = 0;

        scanf ("%d%d", &i, &j);

        if (i != 0)
                if ((j / i) > 5)
                        do_print = 1;

        if (j < 10)
                do_print = 1;

        if (do_print)
                printf ("Input valid!\n");

        return 0;
}
```

2.3 Herhalings-statements

De taal C heeft verschillende lus-statements. In de meeste gevallen is het mogelijk om de ene constructie om te zetten in een andere constructie. Hoewel de lussen dus grotendeels functioneel hetzelfde zijn, zijn bepaalde lus-constructies veel logischer te gebruiken voor een bepaalde taak dan andere.

Opdracht 2.8: Van `while`-lus naar `for`-lus

De volgende code maakt gebruik van een `while`-lus, herschrijf deze lus tot een `for`-lus zonder de werking van het programma te veranderen.

```c
#include <stdio.h>

int main (void)
{
        int i = 0;

        while (i < 10)
        {
                printf ("loop %d\n", i);
                i++;
        }

        return 0;
}
```

Opdracht 2.9: Van `for`-lus naar `do ... while`-lus

De volgende code maakt gebruik van een `for`-lus, herschrijf deze lus tot een `do ... while`-lus zonder de werking van het programma te veranderen.

```c
#include <stdio.h>

int main (void)
{
        int i;

        for (i = 20; i >= 0; i -= 2)
                printf ("i = %d\n", i);

        return 0;
}
```

Opdracht 2.10: Van `do ... while`-lus naar `while`-lus

De volgende code maakt gebruik van een `do ... while`-lus, herschrijf deze lus tot een `while`-lus zonder de werking van het programma te veranderen.

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          int i = 10;
6
7          do
8          {
9                  printf ("%5d\n", i);
10         }
11         while (--i);
12
13         return 0;
14 }
```

Opdracht 2.11: Bijkomende gevolgen van lussen

Compileer het volgende programma:

```
1  int main (void)
2  {
3  }
```

Welke warning geeft de compiler?

Compleer nu een aangepaste versie van het programma:

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          while (1)
6          {
7                  int a;
8
9                  scanf ("%d", &a);
10                 printf ("a + 2 = %d\n", a + 2);
11         }
12 }
```

Waarom blijft de eerdere warning nu achterwege? Hint: Als je bedenkt dat de gebruikte compiler “slim” is, wat kan de compiler dan uit de while-lus opmaken?

Opdracht 2.12: Het programma `is_prime`

Als afsluiting van deze sessie volgt na de eerdere oefenopgaven nu een klein programma waarin gebruik gemaakt moet worden van datatypen, if-statements en lusconstructies. Het programma moet van een ingegeven getal uitzoeken of het een priemgetal is of niet. Zoals waarschijnlijk al bekend is, is een priemgetal een natuurlijk getal met precies twee verschillende natuurlijke delers. Het getal 1 is dus *geen* priemgetal, het is immers alleen deelbaar door 1. De getallen 2 en 3 zijn bijvoorbeeld wel priemgetallen, ze zijn enkel deelbaar door 1 en door zichzelf. Het getal 4 heeft drie delers: 1, 2 en 4, en is dus geen priemgetal. Met het volgende stappenplan (algoritme) is te bepalen of een getal een priemgetal is:

1. als het getal < 2, dan is het geen priemgetal
2. als het getal > 2, deel het dan herhaaldelijk door opeenvolgende getallen, beginnend bij 2 en eindigend bij de wortel van het getal. Indien zo’n deling een rest 0

oplevert, dan heeft het getal meer dan 2 delers en is dus geen priemgetal. Als het getal door geen van deze getallen deelbaar is, dan is het getal een priemgetal.

Een eerste stap in het oplossen van het probleem is het bepalen van de relevante datatypen. In dit geval is er slechts één invoer, het getal waarvan bepaald moet worden of het een priemgetal is. Uit de opdracht blijkt dat dit een natuurlijk getal is (geheel getal ≥ 0), het datatype kan dus een *unsigned int* zijn. In dit geval is gekozen voor het type *int* omdat de invoer door de gebruiker wordt ingegeven en dus ook negatieve getallen kan bevatten.

Om te bepalen of het getal een priemgetal is, wordt dus herhaaldelijk gedeeld door een oplopend getal *i* van het type *int*. Het boolean (true/false) resultaat verschijnt in de variabele *result*. In C is er geen boolean datatype, de waarde *false* wordt gerepresenteerd met 0, alle waarden $\neq 0$ hebben de betekenis *true*. Het is gebruikelijk voor booleans het type *int* te nemen.

De omschrijving van het algoritme in de vorm van een NSD (Nassi-Shneiderman Diagram, zie Bijlage D voor meer informatie) is afgebeeld in figuur 2.1. Zet dit diagram om in C-code. Houdt je strict aan de beschrijving m.b.v. het NSD. Breidt de code uit tot een volledig programma door voor het algoritme het getal in te lezen, en door na het algoritme de waarde van de variabele *result* te testen en te printen of het ingelezen getal wel of niet een priemgetal is. De operator mod in het NSD staat voor modulo en geeft de rest van een deling terug. Zo geldt bijvoorbeeld:

$10 \bmod 3 = 1$ want $10/3 = 3$ met een rest van 1.

Dit werkt uiteraard alleen voor integer delingen.

input:

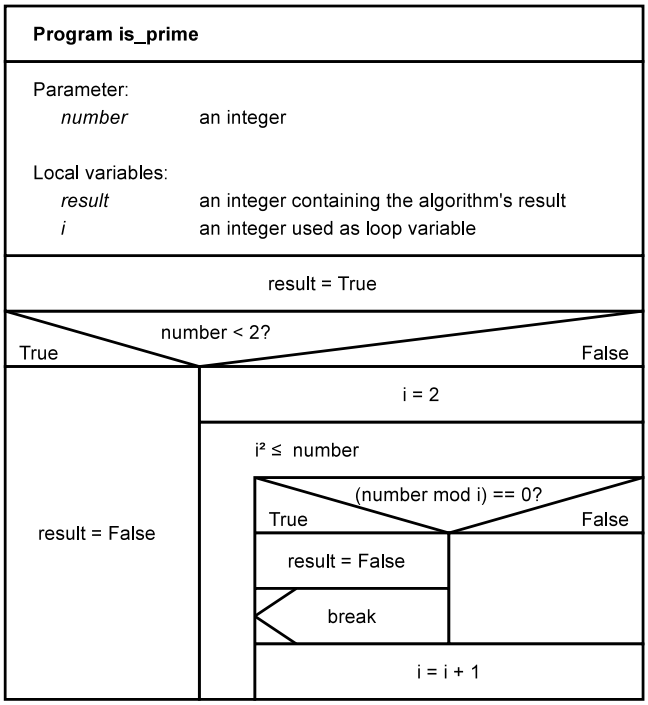
Een positief geheel getal:

```
4
```

output:

Herhaling van het getal met uitspraak "is a prime number" of "is not a prime number":

```
4 is not a prime number
```



Figuur 2.1: NSD van het algoritme van `is_prime`

Hoofdstuk 3

Functies en specifiers

Theorie

- Hoofdstuk 5 uit [1]
- Practicum Handleiding §3.2
- Hoorcollege 2

Leerdoelen

In deze sessie zal je:

- kennis maken met functies: declaratie, definitie en gebruik
- kennis maken met recursie
- leren werken met specifiers voor variabelen en functies.

Inleiding

Hoewel het met de huidige kennis mogelijk is om complexe programma's te schrijven, mist er nog een belangrijk programma-besturingselement: functies. In C worden functies gebruikt voor twee doeleinden:

- hergebruik van code
- abstractie

Hergebruik van code heeft vooral grote voordelen voor de programma grootte en voor het onderhoud van code (er hoeft immers slechts op één plek in de code gewerkt te worden), abstractie heeft vooral voordelen voor het ontwerp en de leesbaarheid van programma's. Beide doelen komen in de voorbeelden naar voren.

3.1 Functies

3.1.1 Declaratie, definitie en gebruik

In C is het noodzakelijk om variabelen vóór gebruik te declareren. Ook functies moeten voor gebruik gedeclareerd worden, maar C heeft ook een ingebouwde standaard (default) declaratie voor functies. Voor een functie genaamd `f` zal deze er als volgt uitzien:

```
int f ();
```

Opdracht 3.1: Functies zonder correcte declaratie

Het volgende programma maakt duidelijk dat incorrecte resultaten kunnen worden verkregen wanneer een functie niet gedeclareerd of gedefinieerd is voordat hij wordt gebruikt. In het programma wordt een functie `f` pas gedefinieerd nadat hij gebruikt (aangroepen) is, terwijl ook een declaratie voor het gebruik van `f` ontbreekt.

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          printf ("%d\n", f (1));
6
7          return 0;
8  }
9
10 int f (double x)
11 {
12         return (x + 1) - 1;
13 }
```

De functie `f` wordt aangeroepen met een argument van het type int in plaats van met een argument van het type double. Omdat het de compiler tijdens de aanroep van `f` niet bekend is dat de actuele parameter 1 van het type int moet worden geconverteerd naar de formele parameter x van het type double, vindt de juist conversie niet plaats en wordt er een verkeerde waarde doorgegeven. Pas het programma zodanig aan dat het wel naar behoren werkt. Dit kan op verschillende manieren.

3.1.2 Hergebruik van code

Een voordeel van functies is het hergebruik van code. De code die in een functie staat kan eenvoudig op meerdere plekken in een programma worden gebruikt, zonder dat die code even zoveel malen voorkomt in het programma. Dit betekent dat de grootte van het programma gereduceerd kan worden. Een misschien nog wel belangrijker voordeel van het hergebruiken van code, is dat eventuele aanpassingen slechts op één plek in de code hoeven te gebeuren. Een laatste groot voordeel is dat het makkelijk is om functies te delen met meerdere programma's, dit gebeurt meestal via een zogenaamde *library*. Ook in de programma's die je tot nu toe hebt geschreven, is hier al gebruik van gemaakt: de functies voor in- en uitvoer `printf` en `scanf` zijn hier voorbeelden van.

Opdracht 3.2: Hergebruik van code

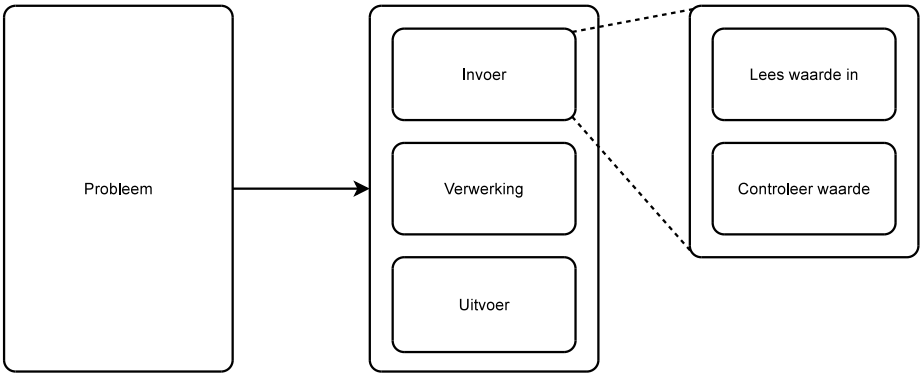
Het volgende programma laat de gebruiker een aantal getallen invoeren en controleert vervolgens of de getallen aan bepaalde voorwaarden voldoen.

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

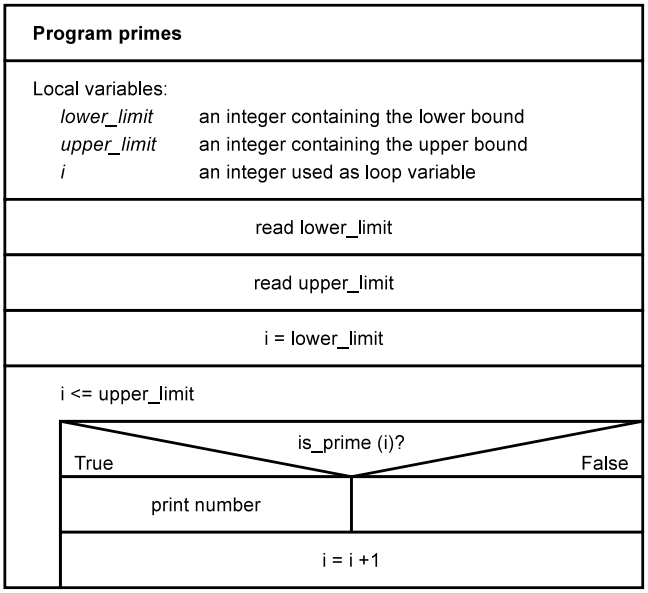
```
{
        int a = 0, b = 0, c = 0, d = 0;

        /* Read in a     */
        scanf ("%d", &a);

        if (a != 0 && a < 10)
                printf ("Input valid!\n");
        else
                printf ("Input invalid!\n");

        /* Read in b     */
        scanf ("%d", &b);

        if (b != 0 && b < 14)
                printf ("Input valid!\n");
        else
                printf ("Input invalid!\n");

        /* Read in c     */
        scanf ("%d", &c);

        if (c != 0 && c < 6)
                printf ("Input valid!\n");
        else
                printf ("Input invalid!\n");

        /* Read in d     */
        scanf ("%d", &d);

        if (d != 0 && d < 22)
                printf ("Input valid!\n");
        else
                printf ("Input invalid!\n");

        printf ("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

        return 0;
}
```

Het programma heeft alleen de main-functie en is daardoor niet erg handig opgeschreven. Veel code wordt, met iets andere getalwaarden, herhaald. Hieronder is een aanzet gegeven voor een nieuwe versie van het programma waarbij gebruik wordt gemaakt van een functie `check_input` welke de ingevoerde getallen controleert. Maak de main-functie nu verder af, gebruik makende van aanroepen naar de functie `check_input`, zodanig dat hetzelfde gedrag wordt verkregen als bij het oorspronkelijke programma.

```
#include <stdio.h>

void check_input (int number, int upper_limit)
{
        if (number != 0 && number < upper_limit)
                printf ("Input valid!\n");
        else
                printf ("Input invalid!\n");
}

int main (void)
{
        /* FIXME
         * use function check_input to implement
         * same behavior as original program.
         */

        return 0;
}
```



Figuur 3.1: Voorbeeld van een top-down ontwerp



Figuur 3.2: NSD van het programma `primes`

3.1.3 Abstractie

De mogelijkheid tot het invoeren van abstractie is de belangrijkste reden voor het bestaan van functies. Met behulp van abstractie is het mogelijk om een top-down ontwerp te maken. Top-down ontwerpen betekent dat er van boven naar beneden ontworpen wordt. Er wordt gestart met een overzicht van het gehele probleem dat wordt opgedeeld in logische deel-problemen. Op hun beurt worden deze deel-problemen weer opgedeeld in kleinere deel-problemen, net zolang als nodig is om eenvoudig implementeerbare blokken over te houden. Een eenvoudig voorbeeld is weergegeven in figuur 3.1. Het is erg eenvoudig om een dergelijke abstractie weer te geven in NSD's, zoals wordt aangetoond in de volgende opdracht.

Opdracht 3.3: Abstractie: het programma `primes`

Hoofdstuk 2 eindigde met het programma `is_prime` waarmee van een getal bepaald kon worden of het een priemgetal was of niet. Nu wordt gevraagd een programma `primes` te schrijven dat van een reeks getallen bepaalt welke getallen priemgetallen zijn. De code van `is_prime` wordt hierbij gebruikt in de vorm van een functie `is_prime`. Het NSD van `primes` kan er dan uitzien zoals afgebeeld in figuur 3.2. Hierin is te zien dat er een aanroep wordt gedaan naar de functie `is_prime`. Doordat de desbetreffende code is ondergebracht in een functie en niet in het hoofdprogramma zelf, is het erg eenvoudig te begrijpen wat het programma `primes` moet doen. In de meeste gevallen van het gebruik van functies wordt gebruik gemaakt van zowel abstractie als hergebruik van code.

Maak van het algoritme van `is_prime` een functie, implementeer het hoofdprogramma en test het programma.

input:

Een ondergrens *lower_limit* (integer) en een bovengrens *upper_limit* (integer)

```
10
30
```

output:

Een lijst van alle priemgetallen in de range van *upper_limit* t/m *upper_limit* (dus inclusief de grenzen)

```
11
13
17
19
23
29
```

3.1.4 Recursie

Zoals in [1] is vermeld, is een functie recursief als deze zichzelf, direct of indirect aanroept. Een aantal problemen uit de wiskunde kunnen worden geformuleerd als functies die zichzelf direct aanroepen: faculteit, Fibonacci en GCD.

Opdracht 3.4: GCD recursief en iteratief (assignment 18, H5 van [1])

De grootste gemene deler (greatest common divider, GCD) van twee positieve gehele getallen is het grootste getal dat deler is van beide getallen. Zo is 3 de GCD van 6 en 15, terwijl de GCD van 15 en 11 gelijk is aan 1. De volgende code is een recursieve functie die de GCD van twee positieve gehele getallen berekent:

```
1 int gcd (int p, int q)
2 {
3         int r;
4
5         if ((r = p % q) == 0)
6                 return q;
7         else
8                 return gcd (q, r);
9 }
```

Test deze functie uit in een programma door een main-functie te schrijven die de waardes van p en q opvraagt en de functie aanroept. Schrijf dan een iterative versie van de

functie `gcd` en test ook die uit. Bedenk dat de iteratieve versie dezelfde operaties moet uitvoeren als de recursieve versie. De recursieve functie doet dit door herhaald zichzelf aan te roepen, de iteratieve versie doet dit door een lus te gebruiken.
Van deze opdracht moet de iteratieve versie ingeleverd worden.

input:

Twee getallen: *p* (integer) en *q* (integer)

```
6
15
```

output:

De GCD van *p* en *q*

```
3
```

3.2 Specifiers

3.2.1 `static` variabelen in functies

Een variabele in een functie bestaat normaal slechts binnen die functie. In de meeste gevallen is dit ook precies wat de bedoeling is, maar soms kan het handig zijn om informatie over functie-aanroepen heen te bewaren. Een mogelijkheid om dit te doen, is gebruik te maken van globale variabelen, welke buiten een functie gedeclareerd worden. Het nadeel hiervan is echter dat deze variabelen vanuit elke plaats in het programma te wijzigen zijn. Een andere mogelijkheid is om een variabele binnen een functie `static` te declareren. Op deze wijze zal de variabele alleen binnen de functie bekend zijn, maar zal de waarde van de variabele tevens over functie-aanroepen heen bewaard blijven..

Opdracht 3.5: Gemiddelde berekening met behulp van een `static` variable

Schrijf een functie `calculate_average` die bij elke aanroep het gemiddelde berekent van alle getallen waarmee de functie eerder is aangeroepen. Het programma stopt als de invoer gelijk wordt aan 0. De main-functie en het prototype van het programma zijn al gegeven. Implementeer de functie met behulp van `static` gedeclareerde variabelen binnen de functie. Er mag dus geen gebruik worden gemaakt van globale variabelen (buiten de functie).

```
1  #include <stdio.h>
2
3  double calculate_average (int number)
4  {
5          /* FIXME
6           * calculate and return average so far.
7           */
8  }
9
10 int main (void)
11 {
12         double average;
13
14         while (1)
15         {
16                 int number;
17
18                 scanf ("%d", &number);
19
20                 if (number == 0)
21                         break;
22                 else
23                         average = calculate_average (number);
```

```
24          }
25          printf ("%.1f\n", average);
26
27          return 0;
28  }
```

input:

Een rij getallen (integers), afgesloten met het cijfer 0

```
1
2
3
0
```

output:

Het gemiddelde van de gegeven rij getallen, afgerond op 1 cijfer achter de komma

```
2.0
```

3.2.2 `extern` en `static` bij gebruik van meerdere C-bestanden

In §3.2.1 is de specifier `static` gebruikt voor variabelen om de waarde van een variabele te kunnen bewaren over de grenzen van een functie heen. Deze specifier kan echter ook, binnen een andere context, een andere betekenis hebben. Tezamen met de specifier `extern` kan de specificier `static` gebruikt worden wanneer gewerkt wordt met meerdere C-bestanden. De tekst die hierover in het boek (zie §5.11 van [1]) staat, specifiek over `extern`, is niet helemaal correct.

Indien een programma groot wordt, dan is het verstandig zo'n programma op te delen in verschillende C-files. Deze files bevatten dan functies die logischerwijs bij elkaar horen. Er zijn dan wel manieren nodig om functies en variabelen uit de ene C-file te kunnen gebruiken in de andere C-file. Tegelijkertijd is het ook handig als het juist mogelijk is bepaalde variabelen en functies te kunnen verbergen voor andere C-files. In C worden daartoe de specifiers `extern` en `static` gebruikt.

De specifier `extern`

De specifier `extern` vertelt de compiler dat bepaalde functies of variabelen ergens anders gedefinieerd zijn. Voor variabelen levert dit twee mogelijkheden op:

1. Een verwijzing vanuit een functie naar een globale variable in diezelfde file
2. Een verwijzing naar een globale variabele in een andere file

De volgende code geeft een voorbeeld van het eerste punt:

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          extern int a;
6
7          printf ("%d\n", a);
8
9          return 0;
10 }
11
12 int a = 2;
```

Als in de declaratie de specifier `extern` wordt weggelaten, dan wordt een nieuwe (lokale) variabele *a* gebruikt in plaats van de globale variabele *a*.

Voor het tweede punt zijn uiteraard twee C-files nodig:

`var.c`:

```
1 int a;
```

`main.c`:

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          extern int a;
6
7          printf ("%d\n", a);
8
9          return 0;
10 }
```

Zoals uitgelegd in bijlage B kunnen verschillende C-files onafhankelijk van elkaar worden gecompileerd. De linker zorgt er dan voor dat de resulterende object-files aan elkaar worden gelinked tot een enkele executable. De specifier `extern` geeft hier aan de compiler door dat de definitie van de variabele ergens anders plaatsvindt. Dit betekent dus ook dat een fout hiermee (als bijvoorbeeld de variabele nergens wordt gedefinieerd) pas door de linker wordt opgemerkt.

Om in de ene C-file gebruik te kunnen maken van functies in een andere C-file moeten deze functies ook als `extern` worden gedeclareerd. In C zijn functies echter standaard als `extern` gedefinieerd, het is dus niet noodzakelijk om hierbij de specifier `extern` te gebruiken.

Initialisaties van `extern` gedeclareerde variabelen
Een `extern` gedeclareerde variabele mag niet worden geïnitialiseerd! Met `extern` wordt aangegeven dat de variabele zich ergens anders bevindt. Als zo'n variabele echter een initialisatiewaarde heeft, dan kan de compiler niets anders doen dan de variabele terplekke aanmaken. Indien echter in twee verschillende C-files dezelfde `extern` gedeclareerde variabele twee verschillende initialisatiewaarden krijgt toegekend, dan worden er dus twee verschillende variabelen aangemaakt. Tijdens het compileren wordt hier met een warning aandacht voor gevraagd, bij het linken geeft dit echter een error.
Uiteraard mag de variabele op de plek van de *definitie* (daar waar er geen `extern` voor staat wél een initialisatiewaarde hebben.

De specifier `static`

Het gebruik van de specifier `extern` maakt het mogelijk om variabelen en functies in de ene C-file, te benaderen vanuit een andere C-file. In sommige gevallen is dat echter juist ongewenst. Dit kan bijvoorbeeld tot problemen leiden indien functies in een library zijn opgenomen. Aangezien in C elke globale variabele en elke functie een unieke naam moet hebben, zou dit betekenen dat elke naam die in een library wordt gebruikt, niet meer te gebruiken is voor het programma. Dat is uiteraard de bedoeling voor de interface-functies van de library, maar als in die code gebruik wordt gemaakt van interne hulp-functies, dan is het wenselijk als die functies niet zichtbaar zijn. Eenzelfde situatie kan ook optreden voor variabelen.

De specifier `static` kan gebruikt worden om dit soort problemen te vermijden. Door een functie of een globale variabele als `static` te declareren, is die variabele of functie enkel zichtbaar vanuit de C-file waarin deze voorkomt.

Voorbeeld van het gebruik van `extern` en `static`

Het volgende voorbeeld bestaat uit twee C-files waarin gebruik gemaakt wordt van de specifiers `extern` en `static`. Bestudeer dit voorbeeld goed en bekijk het gevolg van het aanpassen of weglaten van de specifiers.

`main.c`:

```
1  #include <stdio.h>
2
3  /* Prototype of extern_print     */
4  void extern_print (void);
5
6  /* Local function print          */
7  void print (void)
8  {
9          extern_print ();
10 }
11
12
13 /* Global variable a             */
14 int a = 10;
15
16 int main (void)
17 {
18         printf ("a = %d\n", a);
19         print ();
20
21         return 0;
22 }
```

`print.c`:

```
1  #include <stdio.h>
2
3  /* Global variable a, only visible from within this file                */
4  static int a = 20;
5
6
7  /* Local function print, only visible from within this file             */
8  static void print (void)
9  {
10         printf ("a = %d\n", a);
11 }
12
13 /* Interface-function, this function can be called from other files     */
14 void extern_print (void)
15 {
16         print ();
17 }
```

output:

```
a = 10
a = 20
```

Samengevat geldt het volgende voor `static` en `extern` bij het gebruik van meerdere C-files:

- Een globale variabele en een functie zijn (zonder extra specifiers) bruikbaar in andere C-files

- Om vanuit een C-file een globale variabele in een andere C-file te kunnen benaderen, is het noodzakelijk deze als `extern` te declareren in eerstgenoemde C-file
- Om een globale variabele of een functie te verbergen voor andere C-files, kan deze als `static` worden gedefinieerd

Bijkomende gevolgen van de specifier `static` voor functies
Het gebruikt van de specifier `static` voor functies heeft nog een tweetal extra voordelen:

- De compiler weet dat static functies niet kunnen worden aangeroepen vanuit een andere C-file. Als een C-file een static functie bevat die niet binnen die file wordt aangeroepen, dan wordt hier een warning over gegeven. Dit leidt dus tot betere foutcontrole.
- Indien gecompileerd wordt met optimalisaties aan, kan de compiler ervoor kiezen een static functie inline te gebruiken. Dit wil zeggen dat de functie niet als functie wordt aangeroepen, maar dat de code van de functie direct op de plek van de aanroep wordt geplaatst. Dit verwijdert de aanroep-overhead en kan de code sneller maken.

3.2.3 ABC-formule met interface

Om zelf te oefenen met de verschillende specifiers en werken met meerdere C-bestanden voor een enkel programma, volgt een opdracht waarbij het de bedoeling is een programma te schrijven waarmee de wortels van een willekeurig tweede orde polynoom te vinden zijn. Zoals bekend is uit de analyse, zijn deze wortels te berekenen met de ABC-formule.

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

Indien de discriminant (het deel onder de wortel) negatief is, zijn er geen reële oplossingen maar zijn er twee imaginaire oplossingen. Indien de discriminant nul is, is er één oplossing.

Opdracht 3.6: ABC-formule

Implementeer een programma dat de ABC-formule berekent op basis van via een eenvoudige interface ingevoerde getallen. Maak gebruik van twee C-bestanden: (1) `abc.c` waarin de functies komen die te maken hebben met de ABC-formule, en (2) `interface.c` waarin de functies komen die te maken hebben met de interface van het programma. De bestanden dienen apart gecompileerd te worden. Doe dus niet het ene bestand in het andere bestand includen! Het bestand `abc.c` moet tenminste de volgende functies bevatten:

- `void abc (double a, double b, double c)`
- `double calculate_discriminant (double a, double b, double c)`

Het bestand `interface.c` moet tenminste de volgende functie bevatten:

- `void get_parameters (void)`
- `int main (void)`

De functie `get_parameters` moet de waarden van a, b en c inlezen. De aanroep van `abc (...)` dient vanuit `main` te gebeuren. De functie `abc (...)` moet de berekening uitvoeren (o.a. door de functie `calculate_discriminant` daarvoor te gebruiken) en de resultaten direct naar standaard uitvoer printen.

BELANGRIJK: maak bij deze opdracht op de juiste wijze gebruik van de specifiers `static` en `extern` zodat het niet mogelijk is om functies en variabelen die enkel van belang zijn voor de ene C-file, te benaderen vanuit de andere C-file. Een oplossing die hieraan niet voldoet, zal worden afgekeurd!

input:

Het aantal testcases (integer), daarna één regel per testcase, de waarden van *a*, *b* en *c* (doubles). Bijvoorbeeld:

```
3
2 0 0
1 3 2
3.0 4.5 9
```

output:

```
The roots of 2.0000x^2 + 0.0000x + 0.0000 are:
x = -0.0000
The roots of 1.0000x^2 + 3.0000x + 2.0000 are:
x1 = -1.0000, x2 = -2.0000
The roots of 3.0000x^2 + 4.5000x + 9.0000 are:
x1 = -0.7500+1.5612i, x2 = -0.7500-1.5612i
```

De bedoeling is om uitvoer te geven na elke regel invoer. In en uitvoer komt dus in de praktijk door elkaar op het scherm te staan.

Verpak beide bestanden `abc.c` en `interface.c` tezamen in één zip file (zie bijv. `http://www.7-zip.org`) en lever deze file in.

Functionele scheiding

In dit programma wordt het resultaat van de abc-formule geprint vanuit de C-file `abc.c`. Voor een mooiere en striktere scheiding zou de functie `abc` de uitkomsten eigenlijk moeten teruggeven aan de aanroeper, die vervolgens zelf de resultaten print. Het probleem is dat de resultaten van de `abc`-functie verschillend zijn en meer dan één variabele vereisen. Het is in C goed mogelijk om die resultaten terug te geven aan de aanroeper, maar dat vereist het gebruik van structures of pointers en die worden pas later behandeld.

Hoofdstuk 4

Arrays, pointers en strings

Theorie

- Hoofdstuk 6 uit [1]

Leerdoelen

In deze sessie zal je:

- leren werken met arrays, pointers en strings

Inleiding

Om efficient met data te kunnen omgaan kent C de begrippen arrays en pointers. Het werken met arrays wordt door de meesten niet als moeilijk ervaren maar het begrip pointer en het werken met pointers wordt over het algemeeen lastiger gevonden door mensen voor wie C nieuw is. In dit hoofdstuk zullen we oefeningen doen met arrays en pointers. Verder komen ook strings aan de orde. Een string is in feite equivalent aan een array van characters. Bij het werken met strings wordt ook vaak gebruik gemaakt van pointers.

Opdracht 4.1: Bespaar tijd

Zorg ervoor dat je de belangrijkste begrippen uit hoofdstuk 6 van [1] begrijpt.

4.1 Arrays

Een array variabele is in feite een verzameling enkelvoudige variabelen waarbij toegang tot de enkelvoudige variabelen wordt verkregen m.b.v. indicering. Een array variabele kan meer dan één index hebben, oftewel meer dan één dimensie hebben. Een 1 dimensionaal array komt bijvoorbeeld overeen met een vector en een 2 dimensionaal array met een matrix. Voordeel van het gebruik van een array is dat heel veel variabelen van hetzelfde type met één statement zijn te declareren, en dat de variabelen m.b.v. indicering op een geordende, systematische wijze, toegankelijk zijn. Een eigenschap van een array variabele in C is dat de range van elke index altijd begint bij 0.

Opdracht 4.2:

Bekijk en test het volgende programma is C.

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5          int a[2][3] = {{11, 12, 13}, {21, 22, 23}};
6          int i, j;
7
8          for (i = 0; i < 2; i++)
9          {
10                 for (j = 0; j < 3; j++)
11                 {
12                         printf ("%d ", a[i][j]);
13                 }
14                 printf ("\n");
15         }
16
17         return 0;
18 }
```

De specificatie van de grootte (dimensies) van de array (namelijk 2 en 3) is hier in feite overbodig omdat de compiler dit ook zelf kan bepalen aan de hand van de initialisatie.

Opdracht 4.3: Uitgaven berekening

Stel een bedrijf wil een flexibel programma waarmee het kan berekenen hoeveel elke klant uitgeeft. Stel verder dat het bedrijf N producten verkoopt en dat het aantal klanten M is. Maak een programma dat als invoer heeft de prijzen van de producten en de hoeveelheden van de producten die elke klant koopt. De uitvoer is het bestede bedrag per klant. De waarden van N en M kunnen variëren en worden bij de invoer gespecificeerd. Er geldt echter wel dat N en M nooit groter zijn dan 100.

input:

De waarde van N gevolgd door de prijzen van de N producten. Daarna de waarde van M, gevolgd door (per regel) de aantallen van de producten die elke klant heeft gekocht.

```
4
5 7.50 10 3
2
10 20 10 15
40 15 8 10
```

output:

Het bedrag dat elke klant besteed heeft, nauwkeurig tot op 2 cijfers achter de komma.

```
345.00
422.50
```

De uitvoer dient pas geprint te worden nadat alle invoer gelezen is !

Omdat de maximum waarden van N en M bekend zijn, nl. 100, kan worden volstaan met arrays te gebruiken die gedeclareerd worden met een vaste grootte, bijv. `double price[100]` en `double numbers[100][100]`. De arrays worden bij kleinere waarden van N en M slechts gedeeltelijk gevuld. Merk verder op dat de berekening in feite gelijk staat aan het vermenigvuldigen van een matrix met dimensie M x N, met een vector met dimensie N. Denk verder aan initialisaties, waar nodig.

4.2 Pointers

Pointers zijn variabelen die het adres van een andere variabele kunnen bevatten. Een pointer naar een int kan bijvoorbeeld het adres van een variabele van het type int bevatten. M.a.w. ze bevatten niet zelf een waarde die van belang is, maar ze wijzen (pointen) naar een andere variabele die een waarde heeft welke van belang is. De waarde van de pointer zelf (het adres) is een getal waar de gebruiker niet veel mee kan, maar waarmee de computer de betreffende plaats in het geheugen kan vinden. Door te werken met pointers kan bijvoorbeeld op efficiente wijze toegang tot data worden verkregen, of kan de toegang tot de data worden aangepast zonder dat de data zelf in het geheugen verplaatst hoeft te worden (bijvoorbeeld bij sorteren).

Opdracht 4.4:

Bekijk en test het volgende programma is C.

```
#include <stdio.h>

int main(void)
{
        int a, b;
        int *ptr;

        a = 1;
        b = 2;
        ptr = &a;

        printf ("a has value %d and is stored at address %x\n",
                                                a, (int)&a);
        printf ("b has value %d and is stored at address %x\n",
                                                b, (int)&b);
        printf ("ptr has value %x and is stored at address %x\n",
                                                (int)ptr, (int)&ptr);
        printf ("The value of the integer pointed to by ptr is %d\n", *ptr);

        return 0;
}
```

Merk op dat de adressen plaatsen in het geheugen van de computer aanduiden en dat het onvoorspelbaar is welke waarde zij hebben wanneer je het programma opstart. Dit is o.a. afhankelijk van hoe de compiler de variabelen in het geheugen plaatst en van hoe het operating system het programma opstart.

Opdracht 4.5:

Bekijk en test het volgende programma is C.

```
#include <stdio.h>

int main(void)
{
        int a[5] = {10, 20, 30, 40, 50};
        int *ptr = a;

        printf ("a[0] has value %d and is stored at address %x\n",
                                            a[0], (int)&a[0]);
        printf ("a[1] has value %d and is stored at address %x\n",
                                            a[1], (int)&a[1]);
        printf ("a[2] has value %d and is stored at address %x\n",
                                            a[2], (int)&a[2]);
        printf ("*a has value %d and is stored at address %x\n",
                                            *a, (int)a);
        printf ("*(a + 2) has value %d and is stored at address %x\n",
                                            *(a + 2), (int)(a + 2));
```

```
18          printf ("*ptr has value %d and is stored at address %x\n",
19                                                  *ptr, (int)ptr);
20          printf ("*(ptr + 2) has value %d and is stored at address %x\n",
21                                                  *(ptr + 2), (int)(ptr + 2));
22
23          return 0;
24  }
```

Het bovenstaande programma toont aan (aan de hand van de waarden van de pointers) dat de elementen van een array achter elkaar in het geheugen worden geplaatst: Het werkgeheugen van een programma is georganiseerd als een array van bytes; Een element van het type int neemt meestal 4 bytes in beslag (afhankelijk van het type computer); Daarom zullen de adressen van de achtereenvolgende elementen uit de array meestal met een waarde 4 toenemen.
Het is verder mogelijk om een integer waarde i bij een pointer op te tellen. De compiler zal dan de waarde van de pointer verhogen met een waarde zodanig dat het resultaat het adres geeft van een variabele die i plaatsen verder staat. Voor een pointer naar een variabele van het type int zal de pointer dan typisch naar een adres i * 4 bytes verder wijzen.

Opdracht 4.6: Vectors sorteren

Schrijf een programma dat vectors sorteert op hun lengte. De lengte $|v|$ van een N dimensionale vector $v = (x_0, x_2, x_3, ...x_{N-1})$ wordt gegeven door

$$|v| = \sqrt{\sum_{n=0}^{N-1} x_i^2} \tag{4.1}$$

Sorteer de vectors door een array bij te houden met pointers naar de verschillende vectors, en door alleen de pointerwaarden aan te passen tijdens het sorteren. Een opzet voor het programma is hieronder gegeven, waarbij het inleesgedeelte al geschreven is.

```
1  /*
2   * Student   : FIXME
3   * Nummer    : FIXME
4   * Opdracht : 4.6
5   */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <math.h>
10
11 int main (void)
12 {
13          int dim, num;
14          int i, j;
15          double **w;
16
17          scanf ("%d %d", &dim, &num);            /* read N and M */
18          w = calloc (num, sizeof (double *));  /* allocate array of M pointers */
19          for (i = 0; i < num; i++) {
20                  /* allocate space for N dimensional vector */
21                  w[i] = calloc (dim, sizeof (double));
22                  /* read the vector */
23                  for (j = 0; j < dim; j++) {
24                          scanf ("%le", &w[i][j]);
25                  }
26          }
27
28          /* FIXME
29           * Sort the vectors and print them.
30           */
31
```

```
32          return 0;
33  }
```

Te zien is dat er eerst een array met pointers naar doubles wordt gealloceerd, m.b.v. de library functie `calloc`. De groote van het array is M, waarbij M het totaal aantal te sorteren vectors is. Vervolgens wordt voor elke vector die ingelezen wordt een array van doubles ter grootte N gealloceerd, waarbij N de dimensie van de vector is. De pointers naar deze arrays (teruggegeven door `calloc`) worden opgeslagen in de array `w`. De vectoren kunnen nu gesorteerd worden met een sorteer algoritme zoals het bubble sort algoritme.

input:

De invoer bestaat uit eerst de dimensie N, vervolgens het aantal vectors M, en tenslotte - per regel - de coordinaten van elke vector.

```
3 5
4 4 4
2 4 2
1 1 9
-1 -2 5
1 1 1
```

output:

De vectors, per regel, en gesorteerd naar oplopende lengte. Gebruik het %e formaat om de coordinaten te printen.

```
1.000000e+00 1.000000e+00 1.000000e+00
2.000000e+00 4.000000e+00 2.000000e+00
-1.000000e+00 -2.000000e+00 5.000000e+00
4.000000e+00 4.000000e+00 4.000000e+00
1.000000e+00 1.000000e+00 9.000000e+00
```

Voorbeelden van hoe gesorteerd kan worden m.b.v. een bubble sort algoritme staan beschreven op de slides van het 3e college, en in §6.7 en §6.13 van [1], waar dit algoritme gebruikt wordt om respectievelijk integers en strings te sorteren. De `swap` functie voor het omwisselen van 2 vectors zou er bijv. zo kunnen uitzien:

```
1  void swap (double **p, double **q)
2  {
3          double *tmp;
4
5          tmp = *p;
6          *p = *q;
7          *q = tmp;
8  }
```

waarbij bij de aanroep de adressen van de 2 vectors worden meegegeven. Het verdient verder aanbeveling om een functie te maken welke de lengte van een vector retourneert.

4.3 Strings

Een string is niets anders dan een array van characters waarbij het array wordt afgesloten met een '\0' character. Op soortgelijke wijze als hierboven bij int arrays of double arrays kan naar een string gerefereerd worden met de naam van een character array of met een pointer naar een character (`char *`).

Opdracht 4.7:

Bekijk en test het volgende programma is C.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5          char s[7] = "abcdef";
6          char *ptr = s;
7
8          printf ("s[0] has value %c and is stored at address %x\n",
9                                                          s[0], (int)&s[0]);
10         printf ("s[1] has value %c and is stored at address %x\n",
11                                                          s[1], (int)&s[1]);
12         printf ("s[2] has value %c and is stored at address %x\n",
13                                                          s[2], (int)&s[2]);
14         printf ("s has value %s (%x)\n", s, (int)s);
15         printf ("s + 2 has value %s (%x)\n", s + 2, (int)(s + 2));
16         printf ("ptr has value %s (%x)\n", ptr, (int)ptr);
17         printf ("ptr + 2 has value %s (%x)\n", ptr + 2, (int)(ptr + 2));
18
19         return 0;
20 }
```

Merk op dat er een belangrijk verschil is tussen de declaratie van een een character array zoals `char s[7];` en een pointer naar een character zoals `char *ptr;`. In het eerste geval wordt tevens geheugen voor het opslaan van de characters gealloceerd (7 bytes voor 7 characters, inclusief het '\0' character), terwijl in het tweede geval alleen maar een pointer naar een char wordt gealloceerd.

Opdracht 4.8: Woorden tellen

Schrijf een programma dat telt hoe vaak een woord voorkomt in de programma invoer. Het te tellen woord dient als programma argument te worden meegegeven. Wanneer geen programma argument wordt meegegeven dient een foutmelding te worden gegeven. Het programma leest de tekst waarin het woord geteld moet worden van de standaard invoer. Er mag vanuit gegaan worden dat regels in de invoer niet meer dan 1024 karakters hebben. De string #EOF aan het begin van een regel geeft het einde van de invoer aan. Komt het woord overlappend voor, dan telt dat ook. Is het woord bijvoorbeeld aa, dan zou bij de invoer aaaa het woord 3 maal geteld moeten worden.

Onder Code::Blocks kunnen programma argumenten worden meegegeven via Project → Set program's arguments....

input:

Een aantal regels tekst, afgesloten met een regel met #EOF

```
Let it be, let it be, let it be, let it be.
Whisper words of wisdom, let it be.
#EOF
```

output:

Indien programma argument is: let

```
4
```

Library functies

Om regels van standaard invoer te lezen kan het beste gebruikt worden gemaakt van de library functie `fgets()`. Bij een aanroep van `fgets()` zal het programma wachten net zolang totdat iets is ingetypt en een Enter is gegeven. Wanneer `buf` een char array is met grootte 1026, dan zal een aanroep van `fgets()` op de volgende manier een regel van maximaal 1024 karakters, plus een newline karakter, van standaard invoer lezen:

```
fgets (buf, 1025, stdin);
```

Door `fgets()` in een while lus aan te roepen, net zolang totdat #EOF wordt gelezen, kunnen alle regels van de invoer gelezen worden. Na elke aanroep van `fgets()` kan vervolgens de regel met de functie `strncmp()` worden langsgelopen om te tellen hoe vaak het te zoeken woord voorkomt. De bibliotheek functie `strlen()` kan daarbij dan weer van pas komen om het aantal karakters van het te tellen woord te bepalen.

Hoofdstuk 5

Structures en lists

Theorie

- Hoofdstuk 9 – 11
- Hoorcollege 4

Leerdoelen

In deze sessie zal je:

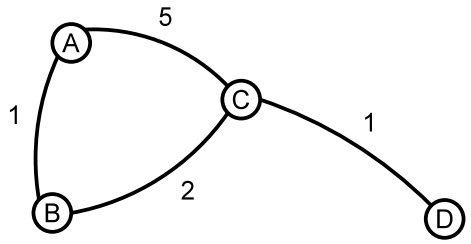
- leren werken met het openen en lezen van bestanden
- leren werken met geavanceerde structuren
- leren werken met dynamisch geheugenmanagement

Inleiding

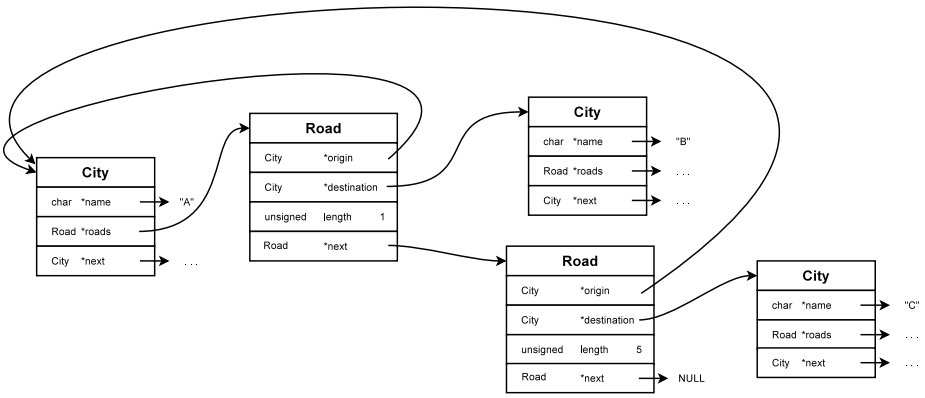
Bij deze opdracht zal worden kennis gemaakt met hoe geavanceerde datastructuren kunnen worden opgebouwd en hoe (eenvoudige) bewerkingen op deze datastructuren kunnen worden uitgevoerd. Dit zal gebeuren aan de hand van het opbouwen van een datastructuur welke een kaart van steden met tussenliggende wegen representeerd. Later zal deze opdracht kunnen worden uitgebreid (zie appendix E) om een routeplanner te maken voor de robot welke binnen het project ”Smart Robot Challange” gerealiseerd zal worden.

5.1 Datastructuren

Om een kaart met steden en hun verbindingen (wegen) bij te kunnen houden, moeten verschillende datastructuren worden gebruikt. Deze datastructuren zijn beschreven in het bestand `EE1400assignments.zip` dat van Blackboard gedownload kan worden. In dit bestand staan in de directory `roadplan` een aantal .c en .h bestanden die deze datastructuren implementeren:



Figuur 5.1: Een voorbeeld van een kaart met steden A, B, C en D. Elke lijn representeert in dit geval 2 wegen (beide richtingen)



Figuur 5.2: Schematische weergave in datastructuren van een deel van de voorbeeld-kaart. Dit deel toont alle wegen vertrekkend uit A, inclusief de daarmee verbonden steden.

`city.c`, `city.h`	Aanmaken en verwijderen van `City` datastructuren
`road.c`, `road.h`	Aanmaken en verwijderen van `Road` datastructuren
`roadplan.c`	Het eigenlijke programma
`salloc.c`, `salloc.h`	Memory-allocatie met NULL-pointer controle

In de volgende paragrafen wordt dieper ingegaan op deze datastructuren.

5.1.1 Een voorbeeld van een kaart

Om de onderlinge samenhang van de verschillende structuren te verduidelijken, wordt gebruik gemaakt van een eenvoudige kaart met vier steden en vier tweerichtingswegen. Deze kaart is afgebeeld in figuur 5.1. In figuur 5.2 is een deel van deze kaart schematisch weergegeven in de gebruikte datastructuren. Hieronder zal dit verder worden toegelicht.

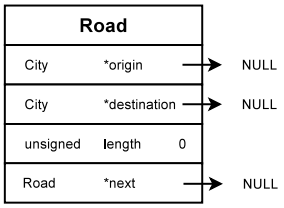
5.1.2 Wegen: de `Road` datastructuur

Voor elke weg dienen drie zaken worden opgeslagen:

- de stad waar de weg begint
- de stad waar de weg eindigt

- de lengte van de weg

De twee steden worden opgeslagen als pointers naar `City` datastructuren (zie §5.1.3), de lengte van de weg wordt opgeslagen in een unsigned integer. Een weg is éénrichtingsverkeer: als er een weg van A naar B gaat en ook weer terug, dan moet dit worden opgegeven in de datastructuur als twee wegen. Een schematische weergave van een lege `Road` datastructuur is weergegeven in figuur 5.3. Naast de 3 bovengenoemde data omvat de Road datastructuur ook nog een pointer naar een volgende Road datastructuur, om op die manier een lijst te kunnen vormen van wegen die uit een stad vertrekken. De datastructuur `Road` en de bijbehorende functies zijn beschreven in de C-files `road.h` en `road.c`.



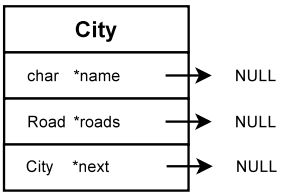
Figuur 5.3: Schematische weergave van de `Road` datastructuur

5.1.3 Steden: de `City` datastructuur

Voor elke stad dienen twee soorten informatie te worden opgeslagen:

- de naam van de stad
- de wegen die vanuit de stad vertrekken

De naam van de stad is een string en wordt dus opgeslagen als een `char *`. Aangezien er meerdere wegen kunnen zijn die vanuit de stad vertrekken, wordt er een verzameling wegen gekoppeld aan de stad. Hiervoor wordt gebruik gemaakt van een linear linked list (zie [1]) van Road structures. Een pointer binnen de City structure naar een Road structure verwijst naar het eerste element van deze lijst. Een schematisch weergave van een lege `City` datastructuur is weergegeven in figuur 5.4. Naast de 2 bovengenoemde data omvat de City datastructuur ook nog een pointer naar een volgende City datastructuur om op die manier een lijst van alle steden op de kaart te kunnen bijhouden. De datastructuur `City` en de bijbehorende functies zijn beschreven in de C-files `city.h` en `city.c`.



Figuur 5.4: Schematische weergave van de `City` datastructuur

De verschillende datastructuren worden dynamisch in het geheugen aangemaakt met functie `malloc ()`. Het is belangrijk om dit geheugen weer vrij te geven als het niet langer nodig is. Dat laatste gebeurt met de functie `free`. Als een programma steeds geheugen opvraagt, maar niets vrijgeeft, dan wordt gesproken van een *memory leak*.

5.1.4 De map-data

De data voor het maken van de kaart staat in een bestand met de volgende specificaties:

- Een integer *n* die het aantal steden op de kaart aangeeft
- Daarna *n* namen van steden, de naam is maximaal 100 tekens lang
- Daarna een integer *m* die het aantal wegen op de kaart aangeeft
- Daarna *m* wegen in de vorm: vertrekstad bestemmingsstad lengte. De steden worden aangegeven met de naam (maximaal 100 tekens). De lengte is een unsigned integer die de lengte in kilometers representeert.

Het databestand van de kaart van figuur 5.1 heeft de volgende inhoud:

```
4
A
B
C
D
8
A B 1
A C 5
B A 1
B C 2
C A 5
C B 2
C D 1
D C 1
```

5.2 Opzet van het programma

Een groot deel van het programma is al gegeven. De code is hieronder weergegeven.

```
1  /*
2   * Student   : FIXME
3   * Nummer    : FIXME
4   * Opdracht : 5.1 + 5.2
5   */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <limits.h>
11
12 #include "salloc.h"
13 #include "city.h"
14 #include "road.h"
```

```
15
16 #define MAX_STRING_LENGTH 100
17
18
19 /*
20  * Locate city with name city_name on the map
21  */
22 City *find_city (City *list_of_cities, char *city_name)
23 {
24         City *c = list_of_cities;
25         while (c != NULL)
26         {
27                 if (strcmp (c->name, city_name) == 0)
28                         return c;
29
30                 c = c->next;
31         }
32         return NULL;
33 }
34
35
36 /*
37  * Delete a map
38  */
39 static void delete_map (City *map)
40 {
41         City *map_copy;
42
43         while (map != NULL)
44         {
45                 map_copy = map;
46                 map = map->next;
47                 delete_city (map_copy);
48         }
49 }
50
51
52 /*
53  * Build the map datastructure
54  */
55 static City *create_map (FILE *data_file)
56 {
57         int num_of_cities, i;
58
59         City *map = NULL;
60
61         /* Read in city-names   */
62         fscanf (data_file, "%d", &num_of_cities);
63
64         for (i = 0; i < num_of_cities; i++)
65         {
66                 char city_name[MAX_STRING_LENGTH + 1];
67                 City *city;
68                 City *c;
69
70                 fscanf (data_file, "%s", city_name);
71
72                 if (find_city (map, city_name) != NULL)
73                 {
74                         fprintf (stderr, "City %s already on the map\n", city_name
                                );
75                         delete_map (map);
76                         exit (EXIT_FAILURE);
77                 }
78
79                 city = new_city (safe_strdup (city_name));
80
81                 if (map == NULL)
82                         /* This is the first city of the map */
83                         map = city;
84                 else {
85                         /* Find last of city list */
86                         c = map;
87                         while (c->next) c = c->next;
```

```
88                          /* And append new city there */
89                          c -> next = city;
90                  }
91          }
92
93          /* FIXME (Assignment 5.1)
94           * Read number of roads
95           * For each road:
96           *    Read origin city name, destination city name and length
97           *    Find pointers to orgin city structure and destination city structure
98           *    using the function find_city()
99           *    Create new road structure using new_road() function
100          *    Add it to the list of roads of the origin city.
101          */
102
103
104         return map;
105 }
106
107
108 static void print_city_roads (City *map)
109 {
110         City *city;
111         Road *road;
112
113         for (city = map; city != NULL; city = city->next)
114         {
115                 printf ("Roads from city %s:\n", city->name);
116
117                 for (road = city->roads; road != NULL; road=road->next) {
118                         printf ("   to city %s", road->destination->name);
119                         printf (" (length %d)\n", road->length);
120                 }
121         }
122 }
123
124
125 static void find_shortest_roads (City *map)
126 {
127         /* FIXME  (Assignment 5.2)
128          * While input and input != "0"
129          *    Find pointer to specified city using find_city()
130          *    Visit the roads originating from this city,
131          *    and find the road that has the shortest length
132          *    Print the destination city and the road length.
133          */
134 }
135
136
137 int main (int argc, char *argv[])
138 {
139         FILE *data_file = NULL;
140         City *map = NULL;
141
142         /* Check arguments              */
143         if (argc != 2)
144         {
145                 fprintf (stderr, "Usage: %s <datafile>\n", argv[0]);
146                 exit (EXIT_FAILURE);
147         }
148
149         /* Open data-file               */
150         if ((data_file = fopen (argv[1], "r")) == NULL)
151         {
152                 fprintf (stderr, "Error opening file <%s>\n", argv[1]);
153                 exit (EXIT_FAILURE);
154         }
155
156         /* Create map-data-structure    */
157         if ((map = create_map (data_file)) == NULL)
158         {
159                 fprintf (stderr, "No map data found\n");
160                 exit (EXIT_FAILURE);
161         }
```

```
162
163         print_city_roads (map);
164
165         find_shortest_roads (map);
166
167         /* Outit                          */
168         if (data_file != NULL)
169                 fclose (data_file);
170         if (map != NULL)
171                 delete_map (map);
172
173         return EXIT_SUCCESS;
174 }
```

De informatie voor het maken van de kaart komt uit een databestand. De naam van dit databestand wordt meegegeven als programma argument. Te zien is dat in de main functie allereerst gecontroleerd wordt of er een programma argument is meegegeven (in dat geval is argc gelijk aan 2). Indien dit niet het geval is wordt er een foutmelding geprint en wordt het programma afgebroken. Daarna wordt geprobeerd het opgegeven bestand te openen. Indien niet lukt wordt ook een foutmelding gegeven en het programma afgebroken. Vervolgens wordt de de routine `create_map` aangeroepen waarin het bestand wordt ingelezen en de datastructuur wordt opgebouwd.

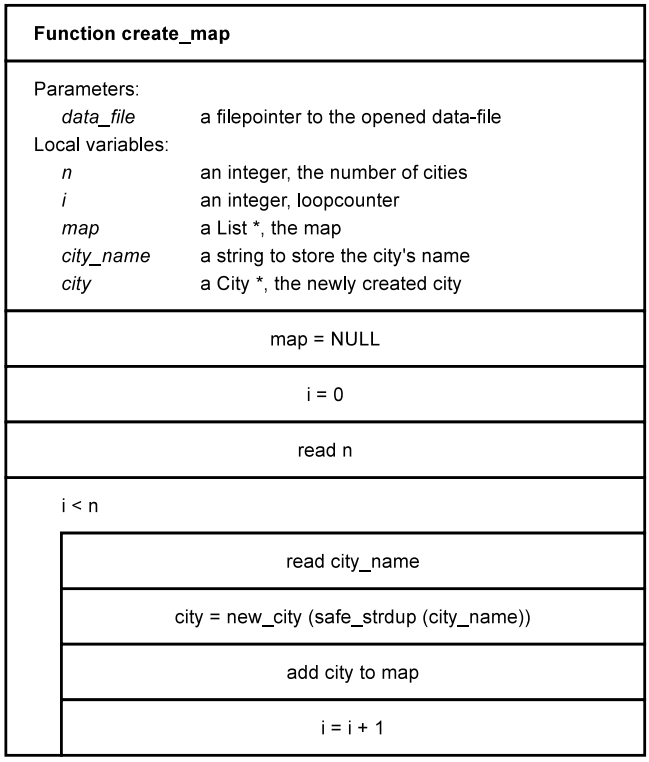
5.3 Lees het databestand in en bouw de kaart op

In de functie `City *create_map (FILE *data_file)` dient de data uit het databestand te worden verwerkt en dient de kaart te worden opgebouwd met de `City` en `Road` datastructuren. De code voor het inlezen van de steden is al gegeven. Zoals beschreven in §5.1.4, bevat het databestand eerst een integer die aangeeft hoeveel steden er op de kaart staan, gevolgd door de namen van de steden zelf. De hoofdstructuur van dit deel van de functie is weergegeven in het NSD van figuur 5.5. Wanneer de naam van een stad is ingelezen wordt eerst gekeken, m.b.v. de functie `find_city ()`, of de naam van de stad al eerder is gelezen. Zo ja dan wordt er een foutmelding gegeven. Bekijk ook hoe in de functie `find_city ()` de lijst met tot dan toe ingelezen steden wordt langsgelopen om te zien of de naam al voorkomt in de lijst. Bij het aanmaken van de `City *city` wordt gebruik gemaakt van de functie `safe_strdup ()` (uit `salloc.c`). Dit is noodzakelijk omdat in de city-structuur enkel een pointer naar de string wordt bewaard. Bij de volgende lus-doorgang zal de waarde van de string veranderen en zonder de aanroep van `safe_strdup` zou dus ook de naam van alle reeds aangemaakte cities veranderen. De functie `safe_strdup` maakt een kopie van de string en geeft het adres van deze kopie terug. Vervolgens wordt de nieuwe city structure toegevoegd aan de lijst. Is de lijst nog leeg dan wordt het eerste element uit de lijst, de nieuwe city structure. Anders wordt de nieuwe city structure toegevoegd aan het einde van de huidige lijst met steden.

Opdracht 5.1: Inlezen van de wegen.

Het tweede deel van het databestand bevat de informatie van de verschillende wegen op de kaart. De wegen bevatten een vertrekstad en een bestemmingsstad die beiden worden opgegeven met de naam van de stad. Schrijf nu de code om de wegen in te lezen en de informatie over de wegen te koppelen aan de steden. Laat je daarbij inspireren door de code die gebruikt wordt om de steden in te lezen.

Allereerst zal voor elke weg moeten worden bepaald welke structures de vertrekstad en bestemmingsstad representeren. Indien deze structures niet gevonden kunnen



Figuur 5.5: NSD van het deel van de functie `create_map` dat de steden inleest

worden aan de hand van de namen van deze steden dient uiteraard een foutmelding te worden gegeven en het programma te worden afgebroken.

Gebruik de functie `new_road ()` om een `Road` structure aan te maken en kijk goed wat voor argumenten deze functie nodig heeft. De structure welke door `new_road ()` wordt geretouneerd moet vervolgens worden toegevoegd aan de `roads` lijst van de stad waaruit de weg vetrekt, op een soortgelijke wijze als waarop een `City` structure wordt toegevoegd aan de lijst van cities.

Als er een weg wordt opgegeven waarvan de begin- of eind-stad niet bestaat, toon dan de volgende foutmelding en sluit het programma af:

```
Cannot find city ??  on the map
```

waarbij op de plaats van `??` uiteraard de naam van de desbetreffende stad komt te staan.

Deze opdracht moet ingeleverd worden, maar wacht hiermee totdat je ook de volgende opdracht hebt gedaan en lever het werk dan in één keer in.

5.4 Lees een naam van een stad en print de kortste weg.

Opdracht 5.2: Zoeken van de kortste weg vanuit een bepaalde stad.

Implementeer de functie `find_shortest_roads ()`. Deze functie dient van de standaard invoer een stad te lezen en vervolgens de korste weg te printen die vertrekt vanaf deze stad. De functie dient dit herhaaldelijk uit te voeren totdat het cijfer 0 wordt gelezen i.p.v. de naam van een stad.

Bij het uitvoeren van het programma dient de naam van het bestand dat de kaart beschrijft te worden gespecificeerd als een programma argument (zie ook opdracht 4.8). Zet dit bestand in de Code::Blocks project map.

input:

Namen van steden, afgesloten met een cijfer 0

```
A
C
X
D
0
```

output:

De wegen vertrekkende vanuit elke stad, gevolgd door de korste wegen die vanuit de gespecificieerde steden vetrekken

```
Roads from city A:
   to city B (length 1)
   to city C (length 5)
Roads from city B:
   to city A (length 1)
   to city C (length 2)
Roads from city C:
   to city A (length 5)
   to city B (length 2)
   to city D (length 1)
Roads from city D:
   to city C (length 1)
Shortest road from city A is to city B: 1
Shortest road from city C is to city D: 1
City X not on map
Shortest road from city D is to city C: 1
```

Na de invoer van een stad dienen onmiddelijk de wegen vanuit die stad te worden geprint, zodat in het commando window de invoer en de uitvoer, die hierboven gescheiden zijn weergeven, in de praktijk door elkaar komen te staan.

Lever bij deze opdracht alleen het bestand `roadplan.c` in, waarin deze opdracht en de voorafgaande opdracht uitgewerkt is.

Bijlage A

ASCII-tabel

Onderstaande tabel bevat de ASCII-codes die op de meeste computersystemen in gebruik zijn.

Dec	Symbol		Dec	Symbol	Dec	Symbol	Dec	Symbol	
000	␀	NUL (’\0’)	032		064	@	096	‘	
001	☺	SOH	033	!	065	A	097	a	
002	☻	STX	034	"	066	B	098	b	
003	♥	ETX	035	#	067	C	099	c	
004	♦	EOT	036	$	068	D	100	d	
005	♣	ENQ	037	%	069	E	101	e	
006	♠	ACK	038	&	070	F	102	f	
007	•	BEL (’\a’)	039	'	071	G	103	g	
008	◘	BS (’\b’)	040	(	072	H	104	h	
009	○	HT (’\t’)	041	)	073	I	105	i	
010	◙	LF (’\n’)	042	*	074	J	106	j	
011	♂	VT (’\v’)	043	+	075	K	107	k	
012	♀	FF (’\f’)	044	`	076	L	108	l	
013	♪	CR (’\r’)	045	-	077	M	109	m	
014	♫	SO	046	.	078	N	110	n	
015	☼	SI	047	/	079	O	111	o	
016	►	DLE	048	0	080	P	112	p	
017	◄	DC1	049	1	081	Q	113	q	
018	↕	DC2	050	2	082	R	114	r	
019	‼	DC3	051	3	083	S	115	s	
020	¶	DC4	052	4	084	T	116	t	
021	§	NAK	053	5	085	U	117	u	
022	▬	SYN	054	6	086	V	118	v	
023	↨	ETB	055	7	087	W	119	w	
024	↑	CAN	056	8	088	X	120	x	
025	↓	EM	057	9	089	Y	121	y	
026	→	SUB	058	:	090	Z	122	z	
027	←	ESC	059	;	091	[	123	{	
028	└	FS	060	<	092	\	124	|	
029	↔	GS	061	=	093	]	125	}	
030	▲	RS	062	>	094	^	126	~	
031	▼	US	063	?	095	_	127	⌂	DEL

Bijlage B

Compilatie en executie

Voordat een programma dat geschreven is in C kan worden uitgevoerd, moet het eerst worden vertaald naar machinecode. Dit proces wordt het buildproces genoemd. In deze appendix wordt het buildproces globaal beschreven. Ook de benodigde systeemcommando's voor de verschillende stappen zijn hier te vinden.

Normaal worden deze stappen automatisch uitgevoerd door Code::Blocks m.b.v. de Build en Run commando's. Voor het begrip van de mogelijkheden en voor de interpratie van eventuele foutmeldingen is het echter handig wanneer men enig inzicht heeft in dit proces.

De stappen zoals hieronder beschreven kunnen eventueel "met de hand" worden uitgevoerd in een commando window. Zo'n window kan worden geopend onder Windows met het commando "Command Prompt" of "cmd", en onder Linux met het commando "Terminal" of "XTerm". Belangrijk is daarbij verder dat in de environment variabele PATH het pad naar de `gcc` compiler is opgenomen. Bij Windows kan dit bijvoorbeeld zijn C:\Program Files\CodeBlocks\MinGW\bin, bij Linux: /usr/bin.

B.1 Het buildproces

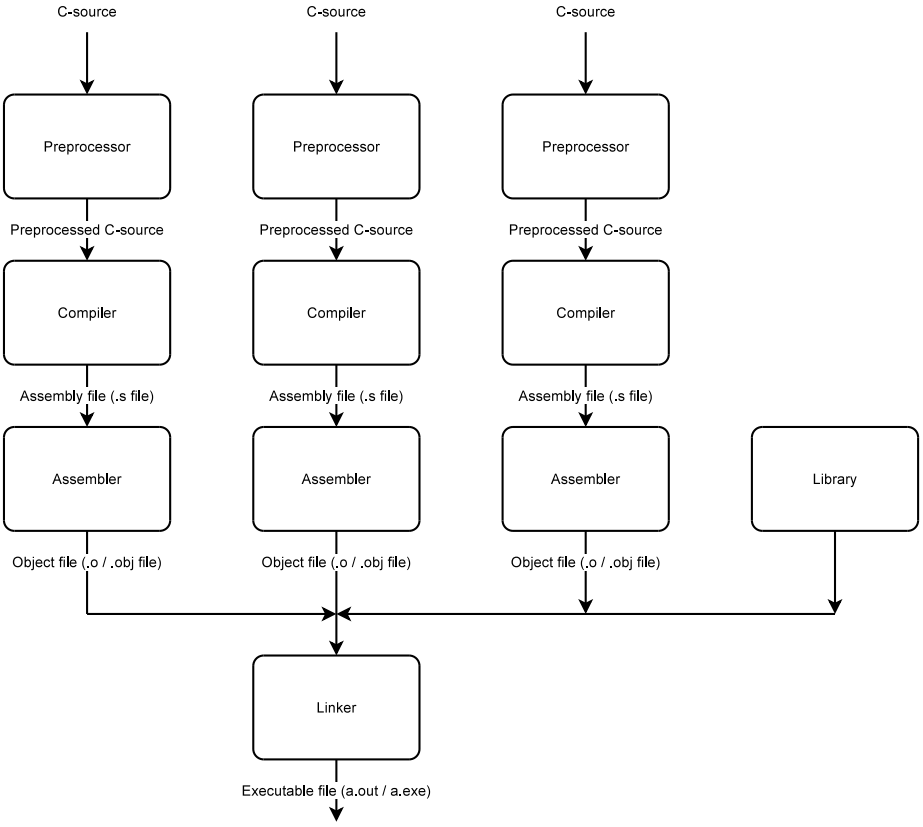
Het omzetten van C-code in machinecode gaat in verschillende stappen. Iedere C-file wordt eerst *preprocessed*, vervolgens *compiled* en *assembled*. Tot slot worden alle bestanden samen *gelinked* tot een *executable*. Dit proces is overzichtelijk weergegeven in figuur B.1.

B.1.1 De preprocessor

De preprocessor verwerkt de *preprocessor directives*. Dat zijn alle regels die beginnen met een `#`, zoals `#include<>`, `#define` en `#ifdef`. De invoer van de preprocessor bestaat uit C-files (.c). De uitvoer van de preprocessor bestaat uit preprocessed C-files (.c) en bevat ook gewoon C-code.

B.1.2 De compiler

De compiler vertaalt de high-level C-code via tokens in platformafhankelijke assemblycode. Daarnaast kan de compiler bepaalde optimalisaties uitvoeren op de code. De



Figuur B.1: Overzicht van het build proces

invoer van de compiler bestaat uit preprocessed C-files (.c). De uitvoer van de compiler bestaat uit assembly (.s) files.

B.1.3 De assembler

De assembler zet de assemblyinstructies om in machinecode. De invoer van de assembler bestaat uit assembly files (.s). De uitvoer van de assembler bestaat uit object files (.o/.obj). Om van de C-file main.c een objectfile te maken kun je het volgende commando gebruiken: `gcc -c main.c`.

B.1.4 De linker

De linker zorgt ervoor dat alle objectfiles samen een executable vormen. De invoer van de linker bestaat uit objectfiles (.o/.obj), de uitvoer van de linker bestaat uit een executable. Om van de objectfile main.o de executable main te maken kun je het volgende commando gebruiken in een commando window: `gcc -o main main.o`

Het is ook mogelijk om het assembleren en het linken met één commando uit te voeren: `gcc -o main main.c`

Meerdere files
Als er meerdere C- of objectfiles tegelijk gebouwd moeten worden dan kan dat op deze manier:

```
gcc -c file1.c file2.c file3.c
gcc -o main file1.o file2.o file3.o
```

Ook hier is het mogelijk om alle stappen in één commando uit te voeren:

```
gcc -o main file1.c file2.c file3.c
```

Regels voor dit practicum
Programma's die bij dit practicum moeten worden ingeleverd, moeten geschreven zijn in ANSI C en mogen geen warnings of errors meer bevatten. Om die reden moeten de opties `-ansi`, `-pedantic` en `-Wall` worden meegegeven aan de compiler. Verder worden er geen optimalisaties meegegeven. Om de juistheid van uw code te controleren zal uw code dan ook met het volgende commando worden gecompileerd, zoals hieronder weergegeven:

```
gcc -Wall -ansi -pedantic -o main main.c
```

B.2 Het uitvoeren van een programma

Het bouwen van een programma resulteert in een executable. Om de executable "main" uit te voeren kan het volgende commando gebruikt worden (de executable moet dan wel executierechten hebben en zich in de working directory bevinden):

```
./main
```

Bijlage C

Code::Blocks tutorial

De *Integrated Development Environment (IDE)* die tijdens dit practicum gebruikt wordt is *Code::Blocks* 10.05. Code::Blocks is open source, zodat iedereen het kosteloos kan downloaden en gebruiken. Meer informatie is te vinden op `http://www.codeblocks.org`. Hier zijn onder andere een uitgebreide handleiding en een aantal uitgebreide tutorials te vinden.

Alhoewel het gebruik van Code::Blocks niet verplicht is, is het gebruik ervan, zeker voor beginnend programmeurs, sterk aan te raden. Als er problemen zijn met de IDE zullen de assistenten alleen bij Code::Blocks helpen met het oplossen ervan.

In de volgende tutorial worden de elementen van Code::Blocks die voor dit practicum belangrijk zijn stap voor stap uitgelegd. Eerst wordt beschreven hoe een nieuw project aangemaakt wordt. Daarna wordt besproken hoe nieuwe bestanden aan het project kunnen worden toegevoegd. Vervolgens wordt er een overzicht gegeven van de Code::Blocks workspace. Dan worden de belangrijkste instellingen voor dit practicum besproken. Daarna volgt een stukje over de automatische source code formatter en ten slotte wordt het bouwen en uitvoeren van een programma in Code::Blocks besproken.

Sommige secties zijn zo vanzelfsprekend dat ze niet per se stap voor stap gevolgd hoeven te worden. Ze zijn slechts voor de volledigheid in de tutorial opgenomen. Met name de secties “Een nieuw project aanmaken”, “Een bestand toevoegen”, “Overzicht van de Code::Blocks workspace” en “Bouwen en uitvoeren van een Code::Blocks project” kunnen snel doorlopen worden.

- Start de Code::Blocks IDE.

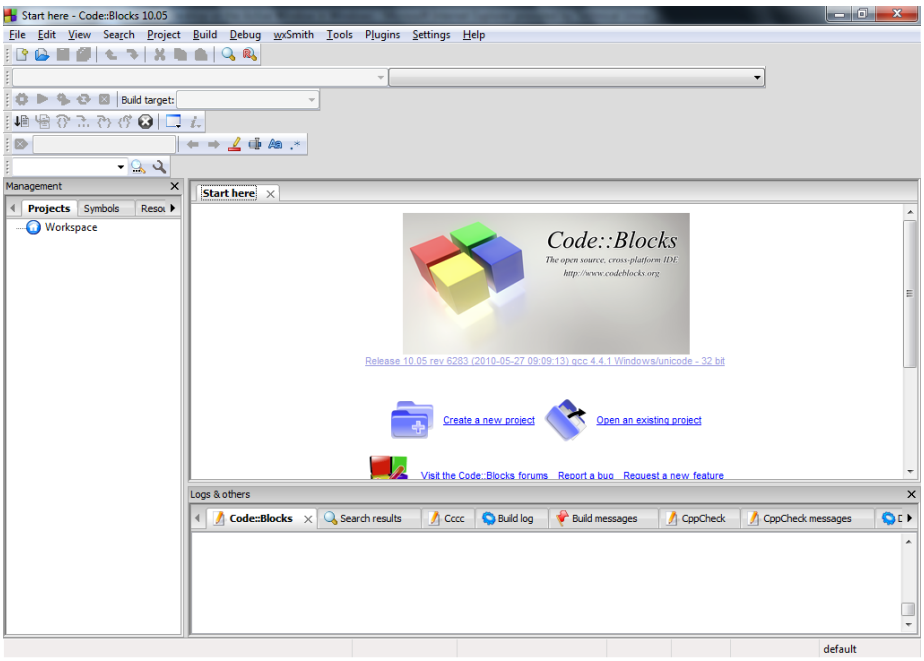
Als het goed is opent nu het scherm uit figuur C.1.

Wanneer er wordt gevraagd om een compiler te selecteren, kies dan voor MinGW/gcc (Windows) of gcc (Linux).

C.1 Een nieuw project aanmaken

Code::blocks werkt, zoals de meeste IDE’s, op basis van projecten. Binnen een project bevindt zich o.a. informatie over de source files van een programma en kan er, normaal gesproken, één executable worden aangemaakt. Gebruik de volgende stappen om een nieuw project aan te maken:

- Kies in het menu File → new → project of klik in het startscherm op Create a new project



Figuur C.1: Code::Blocks startscherm

De wizard *New from template* opent. Voer de volgende stappen uit:

- Kies Console application
- Klik op Go

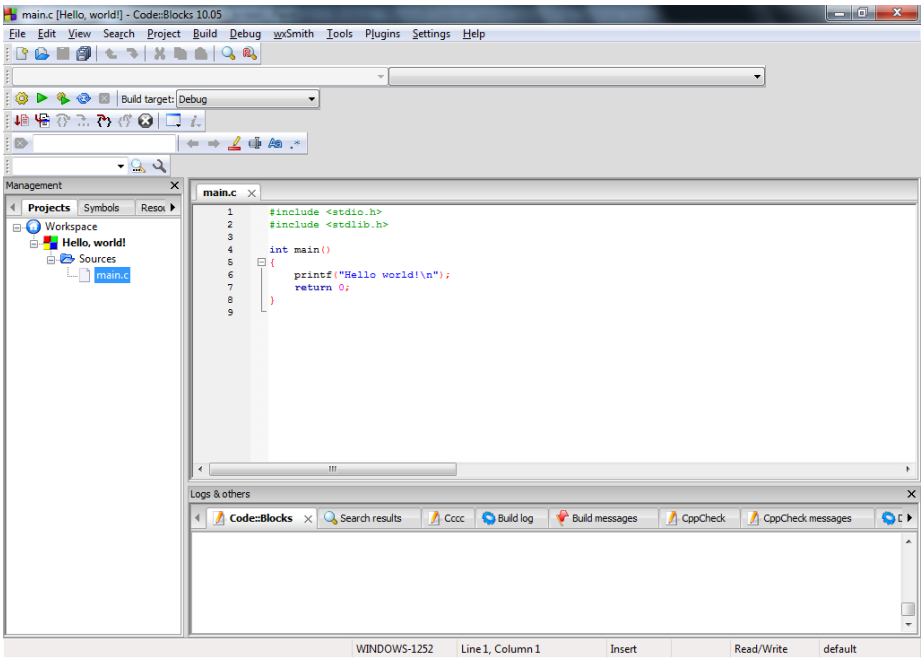
De wizard *Console applictation* opent. Voer de volgende stappen uit:

- Klik op Next >
- Kies C
- Klik op Next >
- Kies Hello, world! als project title
- Kies een geschikte directory (dus een directory met schrijfrechten) om je project in op te slaan
- Klik op Next >
- Klik op Finish

Als alles goed gegaan is zie je nu de Code::Blocks workspace met een open project. In de volgende sectie wordt hiervan een overzicht gegeven.

C.2 Overzicht van de Code::Blocks workspace

De Code::Blocks workspace bestaat uit 3 delen: *Management*, *editor* en *Logs & others*. In het kader *Management* kun je dingen vinden die met project management te maken hebben, zoals bijvoorbeeld een overzicht van alle files die bij het project horen. De *editor* is de plaats waar de broncode geschreven en bewerkt wordt. In *Logs & errors* zijn alle meldingen, zoals errors en warnings, te vinden die tijdens het bouwproces optreden. Een screenshot van de Code::Blocks workspace is te vinden in afbeelding C.2.



Figuur C.2: Code::Blocks workspace met het bestand `main.c` geopend

C.3 Bouwen en uitvoeren van een Code::Blocks project

Alle acties die nodig zijn om een programma te bouwen en/of uit te voeren, zijn te vinden in het menu Build (zie ook de desbetreffende knoppen in één van de werkbalken). De meest belangrijke zijn *Build (* Ctrl + F9 *)*, *Run (* Ctrl + F10 *)*, *Build and run (* F9 *)* en *Rebuild (* Ctrl + F11 *)*.

Wanneer een programma wordt uitgevoerd met het commando *Run* dan zal zich een console window openen waar uitvoer van het programma te zien is en waar eventueel invoer ingetypt kan worden.

Opdracht C.1: Hello world!

Voer de stappen ”build” en ”run” uit voor het programma `main.c` dat hierboven is aangemaakt en verifieer of de melding ”Hello world!” in het console window verschijnt.

Opdracht C.2: Studienummer

Modificeer het bestand `main.c` zodanig dat het programma achter de melding “Hello, world!” je naam en studienummer op het scherm print. Bouw vervolgens het programma en voer het uit.

C.4 Belangrijke instellingen voor dit practicum

Programma's die voor dit practicum zijn geschreven worden bij de beoordeling gebouwd met de opties `-Wall`, `-ansi` en `-pedantic`. In Code::Blocks dien je bij het bouwproces ook deze opties mee te geven. Dat gaat op de volgende manier:

- Kies in het menu Settings → Compiler and debugger...
- Klik in het tabbladen Compiler settings en Compiler flags de volgende vinkjes aan:
  - In C mode, support all ISO C90 programs. In C++ mode, remove GNU extensions that conflict with ISO C++ [-ansi]
  - Enable all compiler warnings (overrides every other setting) [-Wall]
  - Enable warnings demanded by strict ISO C and ISO C++ [-pedantic]

C.5 Gebruikmaken van de automatische sourcecode formatter

Code::Blocks biedt de mogelijkheid om automatisch de broncode op te maken. Dat gaat op de volgende manier:

- Kies in het menu Plugins → Source code formatter (AStyle)

Voorkeuren voor de automatische source code formatter kun je op de volgende manier instellen:

- Kies in het menu Settings → Editor ...

De wizard *Configure editor* opent.

- Kies in het linker menu Source formatter
- De instellingen kunnen worden aangepast in het rechter menu
- Klik op OK

C.6 Een nieuw bestand aan het project toevoegen

Zodra een nieuw project aangemaakt wordt, bevat dit project het bestand `main.c`. Voor een Console application zal hier default code in staan om de regel Hello world! te printen. Het kan uiteraard nuttig zijn om binnen een project meerdere bestanden te gebruiken. Deze sectie beschrijft hoe twee bestanden, `functions.h` en `functions.c`, aan het project toegevoegd kunnen worden.

C.6.1 Het bestand `functions.h` aan het project toevoegen

Voer de volgende stappen uit om het bestand `functions.h` aan het project toe te voegen:

- Kies in het menu File → New → File...

De wizard *New from template* opent. Voer de volgende stappen uit:

- Kies C/C++ header
- Klik op Go

De wizard *C/C++ header* opent. Voer de volgende stappen uit:

- Klik op Next >
- Klik op ... onder *Filename with full path*

De wizard *Select filename* opent. Voer de volgende stappen uit:

- Vul in het textveld achter name *functions.h* in
- Klik op save

Voer in de wizard *C/C++ header* de volgende stappen uit:

- Klik op All
- Klik op Finish

Voeg alle nieuwe files toe aan de buildtargets!
Als er tijdens het buildproces onverklaarbare fouten optreden kan het zijn dat bepaalde bestanden niet aan de buildtargets zijn toegevoegd. Voer de volgende stappen uit om het bestand `functions.h` aan de buildtargets toe te voegen:

- Klik met rechts op het bestand functions.h en selecteer Properties...
- Ga naar het tabblad Build
- Selecteer de buildtargets
- Klik op Ok

C.6.2 Het bestand `functions.c` aan het project toevoegen

Voer, om het bestand `functies.c` aan het project toe te voegen, de volgende stappen uit:

- Kies in het menu File → New → File...

De wizard *New from template* opent. Voer de volgende stappen uit:

- Kies C/C++ source
- Klik op Go

De wizard *C/C++ source* opent. Voer de volgende stappen uit:

- Selecteer C
- Klik op Next >
- Klik op ... onder *Filename with full path*

De wizard *Select filename* opent. Voer de volgende stappen uit:

- Vul in het textveld achter name *functions.c* in
- Klik op save

Voer in de wizard *C/C++ source* de volgende stappen uit:

- Klik op All
- Klik op Finish

Rebuild de Code::Blocks workspace
Zodra Code::Blocks een programma bouwt, worden alleen die bestanden gecompileerd die sinds de vorige build gewijzigd zijn. *Warnings in eerder gebouwde C-files worden dan dus niet meer weergegeven!* Om er zeker van te zijn dat een programma zonder warnings en errors bouwt moet de Code::Blocks workspace herbouwd worden! De codeblocks workspace kan herbouwd worden via Build → Rebuild workspace.

Bijlage D

Nassi-Shneiderman diagrammen

Inleiding

Programmeren is een vaardigheid die niet gebonden is aan een specifieke taal, het kunnen opdelen in kleinere delen, het kiezen van de juiste datatypen en algoritmes staat (redelijk) los van een programmeertaal. Een programmeertaal biedt de mogelijkheid een programma te implementeren. Uiteraard heeft de keuze van de programmeertaal wel enige invloed, maar vaak is het eenvoudig om een structuur of een algoritme in een andere taal te implementeren. Er zijn verschillende hulpmiddelen om een programma te ontwerpen, onafhankelijk van de uiteindelijke implementatie-taal, bijvoorbeeld flowcharts en Nassi-Shneiderman diagrammen en pseudo-code. Elk van deze systemen heeft voor- en nadelen.

Flowcharts zijn waarschijnlijk het meest bekend en worden behalve voor programmeren ook gebruikt voor allerhande andere toepassingen. In een flowchart wordt de flow van een programma weergegeven, verschillende type blokken geven stappen in een proces aan en worden verbonden met pijlen die de flow-richting aanduiden. Een groot voordeel van een flowchart is dat het relatief eenvoudig is om een bestaand systeem te beschrijven. Een belangrijk nadeel is dat een flowchart geen structuur heeft, er is een enkel blok om een keuze aan te duiden. Een herhaling wordt aangegeven met zo'n dergelijk keuze-blok en een pijl die naar een eerder deel in de flowchart wijst. Het is hierdoor lastiger om een ingewikkelde flowchart om te zetten naar een programma in een gegeven programmeertaal.

Nassi-Shneiderman diagrammen bieden een mogelijkheid om een gestructureerd ontwerp te maken. Er zijn eenvoudige constructies voor het aangeven van herhalingsstatements en keuzes. Door de gebruikte notaties is het bijvoorbeeld direct duidelijk welke stappen tot een lus behoren en of de lus de controle aan het begin of aan het eind heeft. Door de nadruk op gestructureerde elementen is het niet mogelijk (of in elk geval lastig) om in NSDs spaghetti-code te ontwerpen. NSDs zijn bijna triviaal te implementeren in een programmeertaal met gestructureerde elementen. Nadelen van NSDs zijn dat het maken ervan meer werk is dan flowcharts, ook kan het aanpassen ervan meer werk kosten.

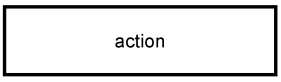
Pesudo-code biedt een manier om in een niet-bestaande programmeertaal een algoritme te beschrijven. Omdat pseudo-code vaak gebaseerd is op talen als Pascal en C, is het implemeren in een dergelijke taal vaak erg eenvoudig. Een nadeel van pseudo-code is dat het geen structuur weergeeft, de beschrijving is enkel in tekst. Het opstellen van een programma in pseudo-code biedt de mogelijkheid om taal-specifieke opmaak elementen (zoals bijvoorbeeld de afsluitende ';' van C) weg te laten, waardoor meer nadruk ligt op het beschreven probleem. Om een programma in pseudo-code te kunnen schrijven, is het echter al bijna noodzakelijk reeds te kunnen programmeren.

Met deze gegeven voor- en nadelen en het doel van deze cursus, is gekozen voor het gebruik van NSDs voor het beschrijven van programma's. NSDs zijn voor het eerst beschreven in [2] en beschikbaar als Nederlande norm NEN 1422 (zie [3]).

D.1 Elementen van NSD's in relatie tot C

D.1.1 Actie

Stappen van een programma worden in een NSD weergegeven met een blok zoals afgebeeld figuur D.1.

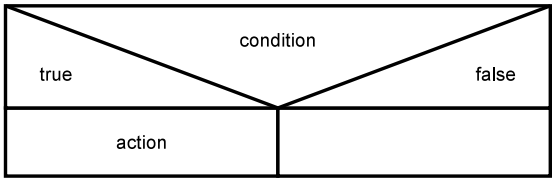


Figuur D.1: Een actie-blok in een NSD

Zo'n dergelijke actie (statement) kan van alles inhouden: invoer of uitvoer, toekenning, functie-aanroep etc. Het blok kan zelfs leeg blijven, waarmee aangegeven wordt dat er niets in gebeurt. Hoewel dit op zich niet zo zinnig is, kan dit wel gebruikt worden voor bijvoorbeeld de keuze-blokken. De afmetingen van het blok liggen niet vast, die keuze is geheel aan de gebruiker. Het mag duidelijk zijn dat het omzetten van zo'n actie-blok naar C-code niet moeilijk is, maar wel afhangt van wat het blok precies representeert.

D.1.2 Keuzes

Om een keuze te kunnen maken in een NSD kan gebruik gemaakt worden van het blok uit figuur D.2.

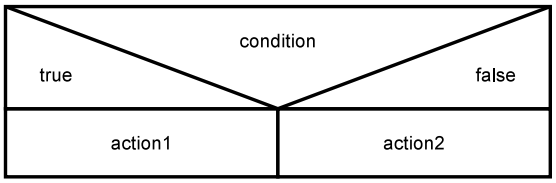


Figuur D.2: Een if-then-blok in een NSD

De conditie waarop getest wordt, staat bovenin het blok. Afhankelijk van de uitkomst van deze conditie, wordt de linker- (hier true) of rechtertak (hier false) genomen. In C is dit als volgt op te schrijven:

```
1 if (condition)
2         action;
```

Dit blok is ook uit te breiden met een action in het geval de conditie false oplevert. Dit leidt tot een if-then-else-blok dat is afgebeeld in figuur D.3.

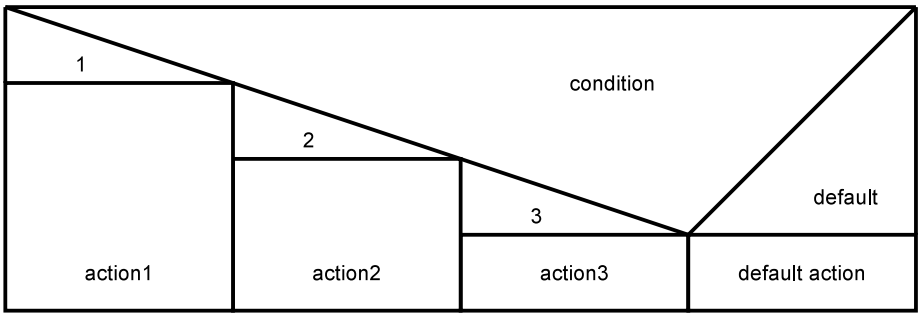


Figuur D.3: Een if-then-else-blok in een NSD

Ook dit is eenvoudig om te schrijven naar C-code:

```
1 if (condition)
2         action1;
3 else
4         action2;
```

NSD's hebben ook een mogelijkheid om een keuze uit een lijst mogelijkheden weer te geven. Hiervoor wordt het blok van figuur D.4 gebruikt.



Figuur D.4: Een switch-blok in een NSD

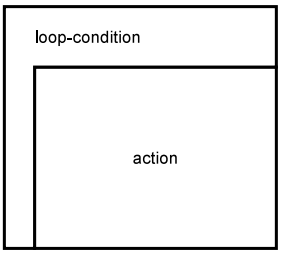
De cijfers 1, 2 en 3 geven hier een keuze aan. Indien de uitkomst van condition verschilt van deze keuzes, wordt het blok onder default uitgevoerd. Dit komt overeen met de volgende C-code:

```
1  switch (condition)
2  {
3          case 1:
4                  ...
5                  break;
6
7          case 2:
8                  ...
9                  break;
10
11         case 3:
12                 ...
13                 break;
```

```
14
15          default:
16                  ...
17  }
```

D.1.3 Herhalingen

Behalve keuzes, is het ook belangrijk om herhalingen te kunnen implementeren. Deze herhalingen zijn in NSD's zeer eenvoudig en overzichtelijk weer te geven. Een lus waarin de lus-conditie vooraf wordt gecontroleerd, kan worden weergegeven als in figuur D.5.

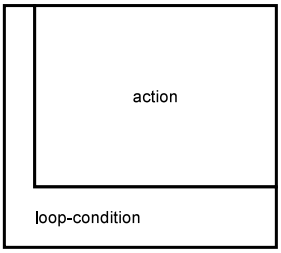


Figuur D.5: Een lus met de controle aan het begin

Door deze structuur is het zeer eenvoudig om te zien welke statements tot de lus behoren. Het diagram correspondeert uiteraard met de volgende C-code:

```
1  while (loop-condition)
2  {
3          action;
4  }
```

Het is ook mogelijk om de lus-conditie aan het eind van de lus te controleren. Het bijbehorende NSD is afgebeeld in figuur D.6.



Figuur D.6: Een lus met de controle aan het eind

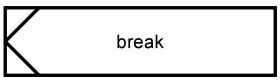
De corresponderende C-code is dan als volgt:

```
1  do
2  {
3          action;
4  }
5  while (loop-condition);
```

`do ... while` of `do ... until`
In dit geval is voor beide lus-constructies een herhaal-zolang-als lus aangenomen. In principe is er niets op tegen om een herhaal-totdat lus te implementeren, al heeft C hier niet direct een lus-constructie voor (in Pascal bestaat bijvoorbeeld een `repeat ... until`).
Indien er een mogelijkheid tot verwarring is, is het verstandig expliciet aan te geven wat de lus aangeeft: een herhaling zolang de conditie geldt, of een herhaling zolang de conditie niet geldt. Dit kan bijvoorbeeld door het toevoegen van de woorden while en until waar nodig.

D.1.4 Het `break`-statement

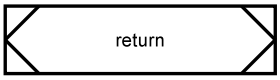
Het `break`-statement maakt het mogelijk de binnenste lus te verlaten. Het is mogelijk een `break`-statement weer te geven via een normaal statement-blok, in dit geval is er echter gekozen voor een wat opvallender weergave. Het hier gebruikte blok voor een `break`-statement is afgebeeld in figuur D.7.



Figuur D.7: Een `break`-statement

D.1.5 Ontwerp abstractie

Abstractie is eenvoudig weer te geven in NSD's door verschillende NSD's te maken voor functies. Deze functies kunnen vervolgens worden aangeroepen door een statement-blok met de functie-naam te plaatsen. Hoewel het mogelijk is om in een statement-blok een `return`-statement op te geven, is er in deze handleiding voor gekozen om dit aan te geven met een speciaal blok. Dit is afgebeeld in figuur D.8.



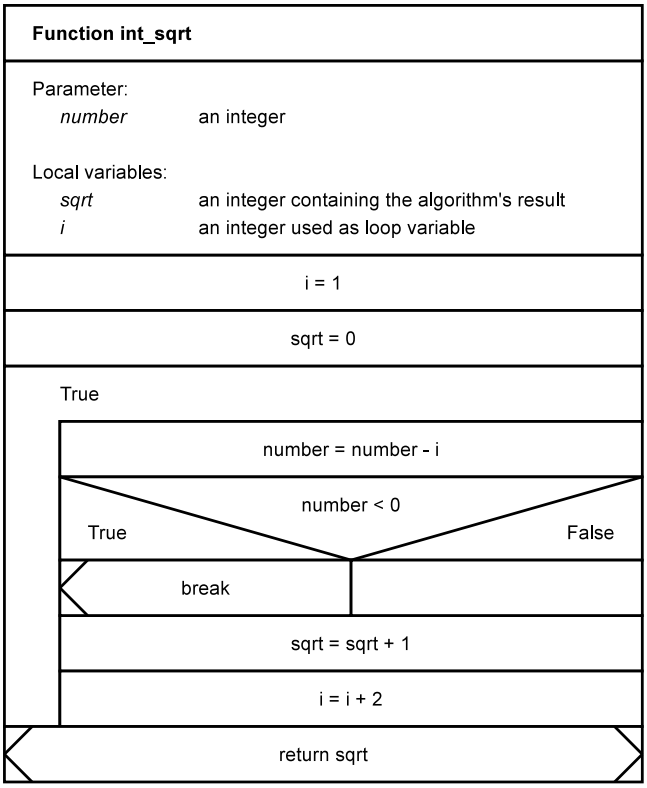
Figuur D.8: Een `return`-statement

D.2 Eenvoudig voorbeeld

Tot slot volgt een eenvoudig voorbeeld van een algoritme waarmee de (integer) wortel van een getal kan worden bepaald. Het algoritme is erop gebaseerd dat de wortel van een getal gelijk is aan het aantal malen dat een opeenvolgend oneven getal van het te onderzoeken getal kan worden afgetrokken, zonder dat het resultaat daarvan negatief wordt. Bijvoorbeeld:

$$10-1=9,9-3=6,6-5=1,1-7=-6 \rightarrow \sqrt{10}=3$$

Er zijn dus drie opeenvolgende oneven getallen van 10 afgetrokken voordat het resultaat negatief werd. De wortel van 10 is dus 3. Het NSD dat bij dit algoritme hoort, is weergegeven in figuur D.9.



Figuur D.9: Het NSD voor de functie `int_sqrt`

De resulterende C-code volgt daar dan eenvoudig uit tot:

```
1  int int_sqrt (int number)
2  {
3          int i    = 1;
4          int sqrt= 0;
5
6          while (1)
7          {
8                  number -= i;
9                  if (number < 0)
10                         break;
11                 sqrt++;
12                 i += 2;
13         }
14         return sqrt;
15 }
```

Bijlage E

Routeplanner

E.1 Inleiding

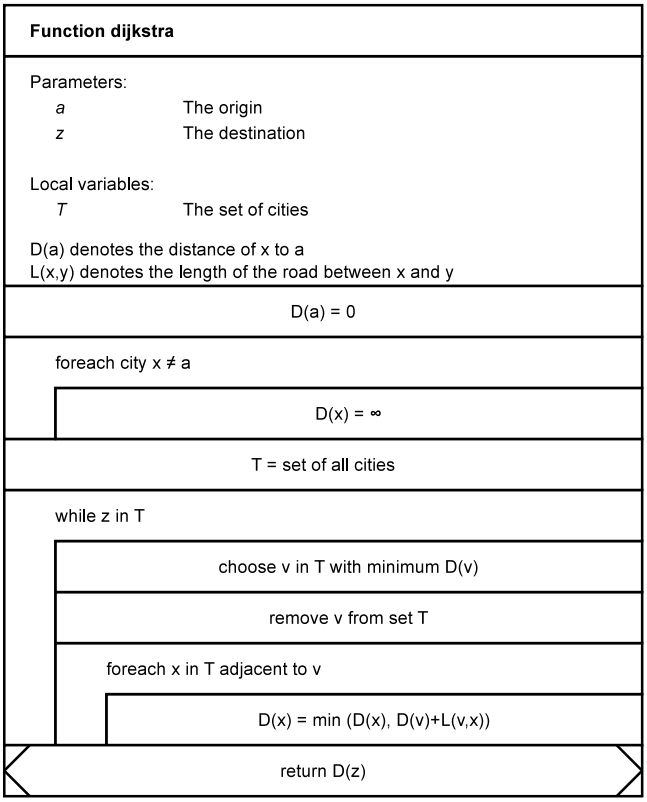
Deze appendix beschrijft hoe het programma uit hoofdstuk 5 kan worden uitgebreid om een routeplanner te implementeren m.b.v. Dijkstra's kortste pad algoritme. Deze informatie komt van pas wanneer later tijdens het project "Smart Robot Challenge" (in het tweede semester) een robot dient te worden te worden ontworpen welke zelfstandig zijn weg zoekt in een doolhof.

E.2 Dijkstra's korste pad algoritme

Om de kortste route tussen twee steden op de kaart uit hoofdstuk 5 te kunnen bepalen, wordt gebruik gemaakt van Dijkstra's kortste pad algoritme. Dit algoritme is eenvoudig en erg snel. Een NSD van het algoritme is afgebeeld in figuur E.1. Om de werking van het algoritme te verduidelijken, is in figuur E.2 een uitgewerkt voorbeeld van de route van A naar D op de kaart van afbeelding 5.1 weergegeven. Zoals al uit het NSD is op te maken, is het algoritme eenvoudig, de volgende stappen beschrijven de werking:

1. Geef alle steden een initiële distance van ∞, behalve de beginstad, die krijgt een distance van 0.
2. Maak een set T waarin alle steden zijn opgenomen.
3. Zoek de stad met de kleinste afstand uit en verwijder deze uit de set T. Deze stad heeft een minimale afstand en wordt verder niet meer in het algoritme gecontroleerd.
4. Bezoek vanaf deze huidige stad alle naburige steden die nog in de set T zitten en bepaal de minimum afstand tot die steden.
5. Herhaal vanaf stap 3 zolang de doel-stad nog in de set T zit.

In het voorbeeld van figuur E.2 is te zien dat na de tweede ronde de afstand van stad C gelijk geworden is aan 5 via een weg met lengte 5 van stad A naar stad C. In de derde ronde wordt stad B als huidige stad gekozen. Nu blijkt dat de afstand naar stad C via stad B korter is: namelijk 1 (van A naar B) + 2 (van B naar C) = 3.



Figuur E.1: NSD van Dijkstra's korste pad algoritme

Het algoritme geeft in principe alleen de lengte van het korste pad terug, maar door in het algoritme bij te houden via welke weg de kortste afstand tot een stad wordt bereikt, is het mogelijk het korste pad weer te geven.

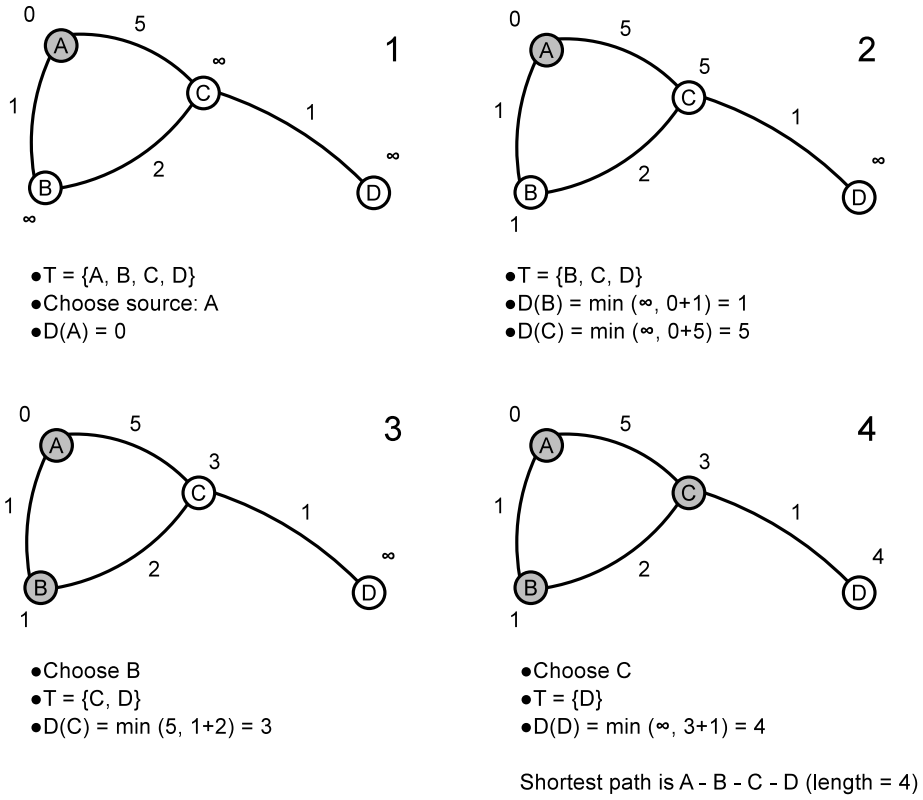
E.2.1 Implementeer Dijkstra's korste pad algoritme

Nu de map-data is ingelezen en de kaart is opgebouwd, kan er een route op de kaart bepaald worden. Voordat het algoritme van Dijkstra kan worden geïmplementeerd, moeten er eerst een aantal toevoegingen worden aangebracht in de `City` datastructuur.

Opdracht E.1: Breid de `City` datastructuur uit

Hoewel de `City` datastructuur voldoende informatie bevat om de kaart op te bouwen, zijn extra datavelden nodig om Dijkstra's algoritme te kunnen gebruiken. Het algoritme houdt bij wat de huidige afstand is van een gegeven stad tot de vertrekstad. Dit kan worden opgeslagen in een unsigned int. Verder houdt het algoritme een set van steden bij. Deze set kan hier heel eenvoudig op een iets andere manier worden bijgehouden: in plaats van een set steden, wordt per stad bijgehouden of die stad nog in de set zit. Dit kan dan worden bijgehouden met een boolean variabele, in C is dat dan een int.

Zoals eerder gezegd, geeft Dijkstra's algoritme de lengte van het kortste pad terug. In het geval van de routeplanner is korste route zelf natuurlijk veel interessanter. Het



Figuur E.2: Uitgewerkt voorbeeld van de werking van Dijkstra's kortste pad algoritme

algoritme van Dijkstra bepaalt in elke ronde welke stad het meest dichtbij is, vervolgens wordt van alle naburige steden bepaald wat dan de afstand is. Als nu bij deze stap niet alleen de korste afstand wordt bijgehouden, maar ook de weg waarmee deze korste afstand realiseerbaar is, dan is het mogelijk de korste route terug te halen. Om deze weg op slaan, is uiteraard een pointer naar een `Road` structuur nodig.

Om deze extra informatie op te kunnen slaan, moet de `City` structuur worden uitgebreid. Voeg de volgende velden toe aan de structuur:

- `unsigned int distance` waarin de huidige kleinste distance wordt opgeslagen
- `Road *shortest_path` waarin de weg wordt opgeslagen die tot de huidige kleinste distance leidt
- `int is_still_in_set` waarin wordt bijgehouden of de huidige stad nog deel uitmaakt van de set T

Uiteraard moet niet alleen de datastructuur worden aangepast, ook de functie waarmee een nieuwe stad wordt aangemaakt moet de nieuwe velden initialiseren. Pas dus ook de desbetreffende code aan.

Opdracht E.2: Schrijf de hulp-functie `find_minimum_distance_city`

Eén van de stappen in Dijkstra's algoritme is het bepalen van de stad met de kleinste waarde van `distance`. Schrijf een functie `City *find_minimum_distance_city (List *map)` waarmee deze stad kan worden bepaald.

Opdracht E.3: Implementeer het algoritme van Dijkstra

Met de hulpfunctie `find_minimum_distance_city` en de extra velden in de `City` datastructuur kan het algoritme van Dijkstra worden geïmplementeerd volgens het NSD van figuur E.1.

Opdracht E.4: Bepaal de kortste route via backtracking

Als het algoritme van Dijkstra is geëindigd, dan zijn in de verschillende `City` datastructuren de velden `distance` en `shortest_path` bijgewerkt. Via backtracking is het mogelijk om te achterhalen wat het kortste pad is. In het voorbeeld van figuur E.2 wijst na het uitvoeren van het algoritme van Dijkstra het veld `shortest_path` van stad D naar de weg tussen D en C. In de structuur van stad C wijst dit veld naar de weg van C naar B enzovoorts. Gebruik deze informatie om de route van begin- naar eindpunt af te drukken. Dit kan het meest eenvoudig worden geïmplementeerd met een recursieve functie.

Opdracht E.5: Geef het gebruikte geheugen vrij

Zorg ervoor dat aan het eind van het programma al het dynamisch gealloceerde geheugen weer wordt vrijgegeven. Maak hiervoor gebruik van de `delete`-functies, zoals `delete_city ()` en `delete_road ()`. Een aantal van deze functies zal je moeten aanpassen of zelf moeten schrijven.

Opdracht E.6: Lees gebruikersinvoer in

Als laatste opgave moet de gebruikersinvoer vanuit standaard invoer worden ingelezen. De invoer kan meerdere routes opgeven, dit kan gevolgen hebben voor de initialisatiestap van het algoritme van Dijkstra.

Als er een ongeldige stad is ingevoerd, dan moet de volgende melding worden weergegeven (het programma moet *niet* worden afgesloten):

```
cannot find city ??  on the map
```

waarbij op de plaats van `??` uiteraard de naam van de desbetreffende stad komt te staan. Als er geen route te vinden is tussen twee opgegeven steden, dan moet de volgende melding worden weergegeven (het programma moet *niet* worden afgesloten):

```
no route found between city ??  and city ??
```

waarbij op de plaats van de twee `??` uiteraard de namen van de desbetreffende steden komen te staan.

De voorbeeld-invoer hieronder is bedoeld voor gebruik met de kaart van figuur 5.1. De map-data van deze kaart is terug te vinden in het bestand `example_map`.

input:

Het aantal testcases (integer), daarna één regel per testcase, de namen van twee steden (maximaal 100 tekens) waartussen een route moet worden gezocht. Bijvoorbeeld:

```
4
a f
g a
a c
a d
```

output:

```
Cannot find city f on the map
Cannot find city g on the map
Total distance of the route between a and c is 3km
a - b (1km)
b - c (2km)
Total distance of the route between a and d is 4km
a - b (1km)
b - c (2km)
c - d (1km)
```

Bibliografie

[1] A. Kelley and I. Pohl, *A Book on C*, fourth edition, Pearson Education, september 2008

[2] I. Nassi and B. Shneiderman, *Flowchart techniques for structured programming*, SIGPLAN Notices XII, August 1973 (`http://www.cs.umd.edu/hcil/members/bshneiderman/nsd/1973.pdf`)

[3] NEN norm 1422, *Programmastructuurdiagrammen*, november 1979