Inhoud

Voorwoord bij de zesde druk 3

1 Introductie 17
- 1.1 Inleiding 17
- 1.2 De preprocessor, compiler en linker 17
- 1.3 Eerste voorbeeld 19
  - 1.3.1 Toelichting bij het eerste voorbeeld 19
  - 1.3.2 Declaratie van variabelen 21
  - 1.3.3 Getaltypen in C++ 23
  - 1.3.4 Expressies: de operatoren +, -, *, / en % 25
  - 1.3.5 Typeconversie 26
  - 1.3.6 De typecast 27
  - 1.3.7 Assignment-operatoren 28
  - 1.3.8 De increment-operator ++ en de decrement-operator -- 29
  - 1.3.9 De naam C++ 30
  - 1.3.10 Postfix en prefix 30
  - 1.3.11 Prioriteiten 31
- 1.4 Literals en constanten 31
  - 1.4.1 Constanten 33
- 1.5 De scope van variabelen en constanten 35
- 1.6 Het type char 35
  - 1.6.1 ASCII-code 36
  - 1.6.2 Escape sequences 37
  - 1.6.3 Literals van het type char 39
- 1.7 C++11: de typeaanduiding auto 40
  - 1.7.1 De opmaak van de broncode 41
- 1.8 Samenvatting 41
- 1.9 Vragen 42
- 1.10 Opgaven 43

2 Selecties en herhalingen 45
- 2.1 Inleiding 45
  - 2.1.1 Relationele operatoren 45
  - 2.1.2 Logische operatoren: de en-operator && 46
  - 2.1.3 Een bool-variabele 48
- 2.2 Het if-statement 49
- 2.3 Het if-else-statement 51
  - 2.3.1 Over cin 52
- 2.4 Het switch-statement 52
  - 2.4.1 Wat niet kan met een switch-statement 55

2.5 Het for-statement 55
2.5.1 Controlegedeelte van het for-statement 56
2.5.2 Zet geen puntkomma na het controlegedeelte 58
2.5.3 De scope van de controlevariabele 59
2.6 Recht onder elkaar zetten van gehele getallen 59
2.6.1 De manipulator setfill() 61
2.6.2 De manipulatoren hex, oct en dec 62
2.6.3 Het gebruik van setw() met tekst 63
2.6.4 Links of rechts uitlijnen 64
2.6.5 Het uitvoerformaat van floating-pointgetallen 65
2.7 Variaties met een for-statement 66
2.7.1 For-statement waarvan de body niet wordt uitgevoerd 69
2.8 Het enumerated type 70
2.8.1 C++11: strongly typed (scoped) enumerations 72
2.9 Het while-statement 72
2.9.1 Syntax van het while-statement 73
2.10 Het do-while-statement 75
2.10.1 De invoerbuffer 76
2.11 De oneindige loop 77
2.12 Het break-statement 79
2.13 Controleren van de invoer 80
2.14 Het continue-statement 82
2.15 Algoritmen: een loop binnen een loop 83
2.15.1 Genest for-statement 83
2.15.2 Het omgekeerde probleem 84
2.16 Samenvatting 88
2.17 Vragen 89
2.18 Opgaven 90

3 Functies 95
3.1 Inleiding 95
3.2 Een simpele functie 95
3.2.1 Prototype 96
3.2.2 Implementatie van de functie 96
3.2.3 Functieaanroep 96
3.3 Een functie met argumenten 97
3.3.1 Voordeel van een functie met argumenten 100
3.3.2 Actuele en formele argumenten 100
3.3.3 Nog een functie 100
3.3.4 Defaultargumenten 102
3.4 Wiskundige functies 103
3.4.1 C++-library 103
3.5 Functies die een waarde afleveren 104
3.6 Gestructureerd programmeren en functies 105
3.7 Functies die geen waarde afleveren 108
3.8 Richtlijnen bij het schrijven van functies 108

3.9 Prototyping, aanroep, implementatie en volgorde 109
3.9.1 Het prototype of de declaratie van de functie 109
3.9.2 De functieaanroep 110
3.9.3 De definitie of implementatie van de functie 111
3.9.4 De volgorde van functies 111
3.9.5 Splitsen van header en implementatie 111
3.10 Lokale variabelen 112
3.10.1 De argumenten van een functie 114
3.11 Statische en globale variabelen 114
3.11.1 Statische variabelen 114
3.11.2 Initialisatie van statische variabelen 116
3.11.3 Globale variabelen 116
3.11.4 Globale en lokale variabelen met dezelfde naam 117
3.12 Reference-argument 118
3.12.1 Wanneer gebruik je een reference-argument? 120
3.13 Uitbreiding van de richtlijnen voor het schrijven van een functie 120
3.13.1 De functie verwissel() 122
3.14 Een functie die een referentie aflevert 123
3.15 Functie-overlading 126
3.15.1 Overladen op grond van aantal argumenten 127
3.15.2 Pre- en postcondities 128
3.15.3 Overladen op grond van type van de argumenten 129
3.16 Samenvatting 129
3.17 Vragen 130
3.18 Opgaven 131

4 C-arrays en pointers 133
4.1 Inleiding 133
4.2 Het geheugen 133
4.2.1 Bytes 133
4.2.2 Bits 134
4.2.3 Adressen 135
4.3 Arrays 137
4.3.1 Een C-array van doubles 138
4.3.2 De bovengrens van een array 140
4.3.3 Initialisatie van een C-array 140
4.3.4 De grootte van een C-array bepalen met sizeof 141
4.3.5 C-arrays en andere variabelen tegelijk declareren 142
4.4 Algoritme: zoeken van de grootste waarde in een C-array 142
4.5 C-array als argument van een functie 143
4.5.1 Nogmaals: zoeken van de grootste waarde in C-array 145
4.6 Een const arrayargument 146
4.6.1 Algoritme: zoeken van een getal in een C-array 146
4.7 C++11: range-based for 148

4.8	Tweevoudige arrays	149
	4.8.1 Initialiseren van een tweevoudige C-array	150
	4.8.2 Een tweevoudige C-array als argument van een functie	151
	4.8.3 Een speelbord voor boter, kaas en eieren	153
4.9	Voor- en nadelen van C-arrays	156
4.10	Pointers	158
	4.10.1 Opmerkingen over notaties bij het gebruik van pointers	160
	4.11.1 Opschuiven van een pointer naar double	164
	4.11.2 Rekenen met pointers	164
	4.11.3 De betekenis van *p++	165
4.12	Pointers en arraynotatie	165
4.13	C++11: nullptr	167
4.14	Een adres als functiewaarde	167
	4.14.1 Pointers naar een constante	169
	4.14.2 Pointer-constante	170
4.15	Een const-pointer als argument	171
4.16	Pointer naar een functie	173
4.17	Typedef	176
	4.17.1 Typedef voor functiepointer	177
4.18	Samenvatting	178
4.19	Vragen	179
4.20	Opgaven	179
5	Strings en vectoren	183
5.1	Inleiding	183
5.2	C-strings	183
5.3	Een string-object	184
	5.3.1 Invoer van strings vanaf het toetsenbord	185
	5.3.2 Combineren van cin>> en getline()	187
	5.3.3 Invoerbuffer leegmaken	188
5.4	Samenvoegen en conversie	189
	5.4.1 Uitvoer naar een string met een stream	190
5.5	Een paar functies van de klasse string	191
	5.5.1 De lidfuncties length() en size()	191
	5.5.2 De lidfunctie substr()	192
	5.5.3 De lidfunctie replace()	192
	5.5.4 De lidfunctie find()	192
	5.5.5 De lidfunctie c_str()	193
5.6	Vergelijken van strings	194
5.7	Een iterator voor een string	195
	5.7.1 C++11: range-based for-statement en string	197
	5.7.2 Omzetten van een string in hoofdletters	197
	5.7.3 String omkeren	199
5.8	Vectoren	202
	5.8.1 Een vector declareren en vullen	202
	5.8.2 Iterator voor een vector	203

5.8.3 C++11: auto, range-based for-statement en vector 204
5.8.4 Een vector initialiseren met de waarden uit een array 205
5.8.5 C++11: een vector initialiseren met een initialisatielijst 206
5.8.6 Kopiëren van een vector 206
5.8.7 Een printfunctie voor een int-vector 206
5.8.8 De indexoperator van een vector 207
5.8.9 Het groeien van een vector 209
5.9 Functies van de klasse vector 210
5.10 C++11: de containerklasse array 212
5.11 Samenvatting 213
5.12 Vragen 213
5.13 Opgaven 214

6 Klassen maken 217
6.1 Inleiding 217
6.2 Een klasse voor bankrekeningen 217
6.2.1 Klassendiagram 218
6.2.2 De broncode van de klasse Bankrekening 218
6.2.3 Een constructor 219
6.2.4 De constructor toevoegen en aanroepen 220
6.2.5 Bankrekening-objecten maken 220
6.2.6 C++11: uniforme initialisatie 221
6.2.7 De functie toString() 222
6.2.8 Meer functies voor de klasse Bankrekening 223
6.2.9 De functie stort() 223
6.2.10 De functie neemOp() 224
6.2.11 De functie getSaldo() 224
6.2.12 Het nieuwe klassendiagram van Bankrekening 224
6.2.13 Nieuwe broncode van Bankrekening 224
6.3 Een klasse voor studenten 226
6.3.1 De attributen 226
6.3.2 Getters en setters 227
6.3.3 De broncode van de klasse Student 228
6.4 Data hiding 230
6.5 Een klasse voor datums 231
6.5.1 Implementatie van Datum 232
6.5.2 Const lidfuncties 233
Defaultwaarden voor constructor 234
6.6 Relatie tussen klassen 234
6.6.1 Een initialisatielijst 237
6.6.2 Een const reference-argument 238
6.7 Objecten 239
6.8 De kassa 240
6.8.1 Automatische defaultconstructor 240
6.8.2 Attribuut voor de kassa 240
6.8.3 Zelfgemaakte defaultconstructor 241

		6.8.4 Een paar functies voor de kassa	241
		6.8.5 Lidfuncties buiten de klasse definiëren	242
		6.8.6 De scope-operator ::	243
		6.8.7 Over inline functies	244
	6.9	Over constructors	245
		6.9.1 Constructor-overloading	245
		6.9.2 Constructor met defaultargumenten	246
		6.9.3 Constructor-overloading en defaultargumenten	247
		6.9.4 C++11: delegerende (delegating) constructor	247
		6.9.5 Initialiseren van een constante in een klasse	248
		6.9.6 C++11: directe initialisatie van attributen	248
		6.9.7 Het keyword struct	248
	6.10	Samenvatting	249
	6.11	Vragen	249
	6.12	Opgaven	250
7	Objectgeoriënteerd ontwerpen		253
	7.1	Inleiding	253
	7.2	Teams van studenten	253
		7.2.1 De copy-constructor	256
		7.2.2 Twee kopieën of niet?	257
	7.3	De klasse Team met een vector	258
		7.3.1 Klassendiagram met een collectie	260
		7.3.2 Compositie en aggregatie	261
		7.3.3 Multipliciteiten in UML	262
	7.4	Documentatie maken	263
		7.4.1 Het schrijven van tekst voor doxygen	263
		7.4.2 Genereren van de documentatie	265
	7.5	Een winkel	266
		7.5.1 Analyse	267
		7.5.2 Een lijst van de zelfstandige naamwoorden	267
		7.5.3 De klassen	268
		7.5.4 De associaties	269
		7.5.5 Navigeerbaarheid	269
		7.5.6 Het type van de attributen	269
		7.5.7 De functies	269
		7.5.8 De broncode van de klasse Artikel	270
		7.5.9 De broncode van de klasse Catalogus	271
		7.5.10 Nogmaals over navigeerbaarheid	271
		7.5.11 De broncode van de klasse Bestelling	272
	7.6	Sms-dienst	272
		7.6.1 Analyse van de sms-dienst	272
		7.6.2 De lidfuncties	274
		7.6.3 De broncode van SMS	274
		7.6.4 De broncode van Provider	275
		7.6.5 De pijloperator	276

7.6.6 De broncode van Mobiel 276
7.6.7 Forward declaratie 277
7.7 Samenvatting 278
7.8 Vragen 278
7.9 Opgaven 279

8 Conversie en operatoren 281
8.1 Inleiding 281
8.2 Constructors en conversie 281
8.2.1 Het woord explicit 282
8.2.2 Initialisatie met een string 283
8.2.3 Conversie na de initialisatie 283
8.2.4 Voorwaardelijke compilatie 286
8.2.5 Expliciet aanroepen van een constructor 286
8.3 Operator overloading 287
8.3.1 Unaire, binaire en ternaire operatoren 287
8.3.2 Een somfunctie 289
8.3.3 Tijdelijk object door een constructor laten maken 291
8.3.4 Een operator + in plaats van een somfunctie 291
8.3.5 Welke operatoren mag je overladen? 294
8.3.6 Overladen van unaire en binaire operatoren 294
8.3.7 Overladen van een toekenningsoperator 295
8.3.8 De pointer this 296
8.4 Globale en friend-operatoren 298
8.4.1 De friend-operator*() 299
8.4.2 Implementatie van friend buiten de klasse 301
8.4.3 Een globale operatorfunctie 301
8.4.4 Een insertion- of uitvoeroperator << 302
8.5 Conversie van klasse naar een standaardtype 303
8.6 Conversie tussen klassen 305
8.7 Vragen 308
8.8 Opgaven 309

9 Overerving 313
9.1 Inleiding 313
9.2 Een basisklasse 313
9.3 Afgeleide klasse 315
9.3.1 Private, public en protected 315
9.3.2 Defaultconstructor van de basisklasse 317
9.3.3 Eigen constructor voor afgeleide klasse 318
9.4 Functie-overriding 319
9.5 Generalisatie 322
9.5.1 Aanroepen van functie in basisklasse 329
9.6 Afgeleide klasse van een afgeleide klasse 329
9.6.1 Initialisatielijst en indirecte basisklasse 332
9.6.2 Wat erft een afgeleide klasse niet? 332
9.6.3 C++11: geërfde constructor (inherited constructor) 332

	9.7	Toegangsregels	333
		9.7.1 Wanneer gebruik je wat?	334
	9.8	Multiple inheritance	335
		9.8.1 Ambiguïteit bij multiple inheritance	338
	9.9	Virtuele basisklasse	339
		9.9.1 Oplossing via virtuele basisklasse	342
		9.9.2 Initialisatielijst en virtuele basisklassen	345
	9.10	Samenvatting	345
	9.11	Vragen	345
	9.12	Opgaven	346
10	Dynamisch geheugen		349
	10.1	Inleiding	349
	10.2	Dynamische arrays	349
	10.3	Een destructor	351
		10.3.1 Wanneer wordt een destructor aangeroepen?	354
		10.3.2 Wanneer moet je zelf een destructor schrijven?	355
		10.3.3 Memory leakage	356
	10.4	C++11: smart pointers	356
		10.4.1 De klasse weak_ptr	358
	10.5	Geheugentekort en new	359
	10.6	Copy-constructor en toekenningsoperator	361
		10.6.1 De copy-constructor	361
		10.6.2 Een diepe kopie met de copy-constructor	362
		10.6.3 De toekenningsoperator	364
		10.6.4 Herdefinitie van de indexoperator [ ]	367
	10.7	Objecten en dynamisch geheugen	369
	10.8	Een lineaire lijst	370
		10.8.1 De opbouw van de lijst	371
		10.8.2 Het maken van de lijst	372
		10.8.3 De lijst langslopen met een pointer	373
	10.9	Een verbeterde lijst	375
		10.9.1 Een friend-klasse	377
		10.9.2 Geen friend, maar public get-functies	378
		10.9.3 Een destructor voor de lijst	379
	10.10	Een iterator voor een lijst	381
	10.11	Samenvatting	386
	10.12	Vragen	386
	10.13	Opgaven	387
11	Templates		389
	11.1	Inleiding	389
	11.2	Functietemplates	389
		11.2.1 Een template voor de functie maximum	390
		11.2.2 De essentie van het templatemechanisme	392
		11.2.3 Functietemplate en zelfgedefinieerde klasse	392

	11.2.4 Een functietemplate met twee template-argumenten	395
	11.2.5 Overladen van functietemplate	396
	11.2.6 Schrijven van een functietemplate	396
11.3	Klassentemplates	396
	11.3.1 Implementatie van lidfunctie van een templateklasse	399
11.4	Template voor een lineaire lijst	399
	11.4.1 Generieke lineaire lijst	400
	11.4.2 Lidfuncties buiten de templateklasse implementeren	403
11.5	Afgeleide klasse van een klassentemplate	403
	11.5.1 Afgeleide klasse zonder genericiteit	403
	11.5.2 Afgeleide klasse met behoud van genericiteit	406
11.6	De klasse list uit de standaardbibliotheek	406
	11.6.1 De functie merge()	413
	11.6.2 Een templatefunctie voor print()	414
11.7	De klasse stack	416
	11.7.1 Nut van een stack	418
11.8	De klasse queue	419
11.9	De klasse deque	421
11.10	Samenvatting	422
11.11	Vragen	423
11.12	Opgaven	423

12	Algoritmen	427
12.1	Inleiding	427
12.2	Soorten iteratoren	427
	12.2.1 Speciale iteratoren	428
12.3	Een algoritme	429
	12.3.1 Een const-iterator	430
	12.3.2 Een algemenere versie van zoek()	431
	12.3.3 Een functie zoek() voor een vector	431
	12.3.4 Een templateversie van zoek()	432
	12.3.5 De laatste versie van zoek()	434
	12.3.6 Het algoritme find()	436
12.4	Het algoritme for_each()	437
12.5	Predicaten	438
	12.5.1 Unair predicaat en find_if()	439
	12.5.2 Binair predicaat en sort()	440
12.6	Functieobjecten	442
	12.6.1 Functieobject en algoritme	444
	12.6.2 Functieobject met attribuut	446
12.7	C++11: lambdafuncties	447
	12.7.1 Lambdafunctie met capture list	449
	12.7.2 Capture by reference	450
	12.7.3 Mogelijkheden voor de capture list	451
	12.7.4 De lay-out van de code van een lambdafunctie	452

12.8	Het algoritme copy()	453
	12.8.1 Een back inserter	454
	12.8.2 Een front inserter	454
	12.8.3 Een inserter die gebruikmaakt van insert()	454
	12.8.4 Een ostream__iterator	455
12.9	treams en copy()	456
	12.9.1 Een bestand maken met copy()	456
	12.9.2 Een bestand lezen met copy()	457
	12.9.3 Bestand met gehele getallen direct op het scherm zetten	459
	12.9.4 Alle karakters uit een bestand lezen	459
12.10	Zelfgedefinieerd type en copy()	460
12.11	Over algoritmen, iteratoren, containers en streams	463
12.12	Samenvatting	463
12.13	Vragen	464
12.14	Opgaven	464
online 13	Polymorfie en virtuele functies	467
13.1	Inleiding	467
13.2	Drie klassen met een functie die toString() heet	467
13.3	Een virtuele functie	471
	13.3.1 Polymorfie	472
13.4	Polymorfie met rechthoeken en driehoeken	472
	13.4.1 Dynamische of late binding	475
13.5	Abstracte klasse	477
	13.5.1 Abstracte klasse Figuur	478
	13.5.2 Een vector en een lijst met figuren	480
13.6	Virtuele functies in een keten van afgeleide klassen	482
13.7	Polymorfie via een referentie	482
13.8	Samenvatting	484
13.9	Vragen	484
13.10	Opgaven	484
online 14	Streams	485
14.1	Inleiding	485
14.2	Standaardstream-objecten	485
	14.2.1 De standaardstreams cerr en clog	486
14.3	Een bestand maken en lezen met behulp van een stream	487
	14.3.1 Een bestand lezen met behulp van een stream	489
	14.3.2 Strings en bestanden	490
14.4	Het controleren van een stream	492
	14.4.1 De functies eof(), fail(), bad() en good()	494
14.5	Drie manieren om een tekstbestand te lezen	496
14.6	I/O met objecten	497
	14.6.1 I/O van objecten met behulp van operatoren	498
	14.6.2 Minder foutgevoelige invoer	501

14.7 Objecten in een bestand 503
14.8 Mogelijkheden bij het openen van bestanden 505
14.8.1 Bestand later aan een stream koppelen 505
14.8.2 Bestand openen om aan het einde toe te voegen 505
14.8.3 Bestaand bestand openen en leegmaken 505
14.8.4 Per se werken met een bestaand bestand 506
14.8.5 Per se niet werken met een bestaand bestand 506
14.8.6 Lezen én schrijven uit een bestand: fstream 507
14.9 Random file access 508
14.9.1 Bestand openen met de file-pointer aan het eind 510
14.9.2 Opnieuw lezen uit zelfde bestand: de functie clear() 510
14.10 Binaire bestanden 510
14.10.1 Maken van een binair bestand 511
14.10.2 Lezen van een object uit een binair bestand 512
14.11 Kopiëren van een bestand 514
14.11.1 Kopiëren met behulp van rdbuf() 514
14.11.2 Kopiëren van een bestand naar het beeldscherm 515
14.12 Schrijven naar een string 515
14.12.1 Een ostreamstring leegmaken 517
14.13 Vragen 517
14.14 Opgaven 517

online 15 Excepties 519
15.1 Inleiding 519
15.2 Try, throw en catch 519
15.3 Verschillende excepties van hetzelfde type 522
15.4 Excepties van een verschillend type 525
15.4.1 Exceptie definiëren binnen een klasse 526
15.4.2 Alle excepties opvangen 527
15.4.3 Informatie in de exceptie 527
15.5 Re-throw 529
15.6 Excepties als afgeleide klassen 530
15.7 Standaardexcepties 533
15.7.1 Zelf een standaardexceptie opwerpen 535
15.8 Excepties specificeren in de declaratie van een functie 536
15.8.1 Functie die geen enkele exceptie aflevert 536
15.8.2 Afleveren van een niet-gespecificeerde exceptie 536
15.9 Invoer van bepaald soort getallen 537
15.10 Vragen en oefeningen 539
15.11 Opgaven 539

online	Bijlage A	Header en implementatie	541
online	Bijlage B	Namespaces	547
online	Bijlage C	Tabel van operatoren	551
online	Bijlage D	Gereserveerde woorden en voorgedefinieerde identifiers	555
online	Bijlage E	ASCII-Tabel	557
online	Bijlage F	Literatuur	559
	Register		i

Hoofdstuk 1

Introductie

1.1 Inleiding

Deze editie is geschreven met een lezer in gedachten die enigszins bekend is met elementaire begrippen en principes van veelgebruikte programmeertalen. Die kennis heb je bijvoorbeeld opgedaan door code te schrijven in Java, HTML en JavaScript, of Python. Hoewel veel programmeertalen in de basis op elkaar lijken, kunnen ze in die basis ook op subtiele wijze van elkaar verschillen. Subtiliteiten in een programmeertaal kunnen essentieel blijken. Ik geef daarom in dit hoofdstuk een overzicht van de elementaire voorzieningen en eigenaardigheden van C++. Daarbij zal ik steeds proberen de eigenschappen te demonstreren aan de hand van een geschikt werkend voorbeeld.

1.2 De preprocessor, compiler en linker

Elk C++-programma dat je intikt, de broncode, moet eerst worden vertaald voor het kan worden uitgevoerd. Voor elk platform, zoals Linux, Android, iOS en OS X, Windows voor desktop en tablets en Windows Mobile, zijn compilers in omloop die ervoor zorgen dat de vertaalde versie van je programma geschikt is voor dat platform. Het hele vertaalproces verloopt in fasen waarbij achtereenvolgens de volgende onderdelen een rol spelen: de preprocessor, de compiler en de linker.

De preprocessor leest de broncode en gaat al lezend op zoek naar preprocessoropdrachten, de zogeheten *preprocessor directives*. Een preprocessor directive kun je herkennen aan het hekje `#` dat ervoor staat. Een voorbeeld van een preprocessor directive is:

```
#include <iostream>
```

`<iostream>` is de naam van een zogeheten headerbestand (*header file*), dat onderdeel is van het C++-systeem. De complete inhoud van dit bestand moet door de preprocessor worden ingevoegd (geïnclude) in de broncode. In het headerbestand staan definities die noodzakelijk zijn om de compiler zijn werk goed te laten doen. Deze definities behelzen bijvoorbeeld constanten, prototypen en klassen. Hoe die definities eruitzien wordt in de rest van dit boek duidelijk. Het resultaat van het werk van de preprocessor is een zogeheten translation unit (vertalingseenheid).

Na de preprocessor is het de beurt aan de compiler om de translation unit te vertalen. De compiler heeft twee belangrijke taken:

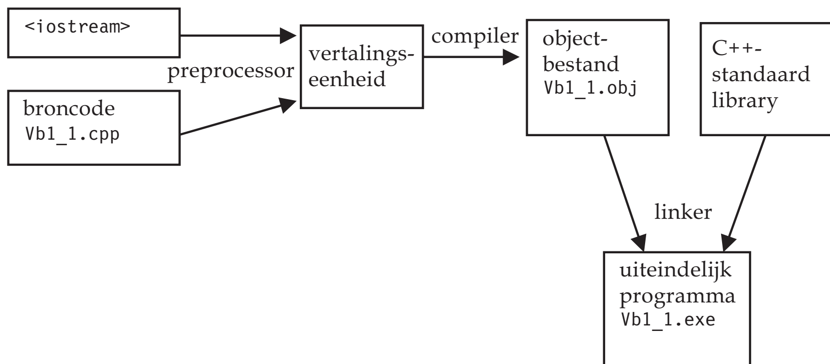
1. Het programma controleren op fouten en daar melding van maken.
2. Als er geen fouten gevonden zijn, het programma vertalen naar machinecode, dat wil zeggen: vertalen naar code die hoort bij de microprocessor die zich in de computer of het apparaat bevindt.

Laten we even aannemen dat het programma taalkundig correct is, zodat er een vertaling tot stand komt. De vertaling wordt opgeslagen in een bestand waarvan de naam het achtervoegsel `.obj` heeft. Deze code heet wel *objectcode*, of doelcode (als tegenhanger van broncode).

De rol van de compiler is nu uitgespeeld. Vervolgens komt de linker aan de beurt.

C++ wordt geleverd met honderden voorgedefinieerde functies, klaar voor gebruik. Deze functies zitten opgeborgen in bestanden die *libraries* heten. Het woord *library* betekent letterlijk bibliotheek, maar het is duidelijk dat het hier niet om boeken gaat. De overeenkomst met een echte bibliotheek is dat er een voorraad functies is (in plaats van boeken), waar elk C++-programma naar believen kopieën van kan maken. Alle bibliotheken die standaard bij C++ worden geleverd heten de *C++ standard library*, of C++-standaardbibliotheek.

Een belangrijke taak van de linker is het uit de bibliotheek halen van een kopie van de functies die het vertaalde C++-programma nodig heeft en deze toe te voegen aan het C++-programma. Het resultaat daarvan wordt weggeschreven naar een nieuw bestand, vaak met het achtervoegsel `.exe`. Het `.exe`-bestand is het bestand waar het allemaal om draait: dit bevat in machinecode alle opdrachten en informatie die het betreffende apparaat en het bijbehorende besturingssysteem nodig hebben om datgene uit te voeren wat je in de broncode van het C++-programma hebt ingetikt. In figuur 1.1 is alles nog eens in beeld gebracht.



Figuur 1.1

1.3 Eerste voorbeeld

Laten we eens kijken naar een eenvoudig, maar goed werkend C++-programma:

```
// Voorbeeld 1.1
#include <iostream>                 // preprocessor directive
using namespace std;                // gebruik van namespace, zie toelichting

int main()                          // start van de functie main(), zie toelichting
{
  int a;                            // declaratie van variabele a
  int b;                            // declaratie van variabele b
  int antwoord;                     // declaratie van variabele antwoord
  a = 17;                           // a krijgt de waarde 17
  b = 24;                           // b krijgt de waarde 24
  antwoord = a + b;                 // antwoord krijgt de waarde van a + b

  cout << "Het resultaat is: "; // tekst naar uitvoerscherm
  cout << antwoord << endl;         // antwoord naar uitvoerscherm
  cin.get();                        // wacht op indrukken van de Entertoets
  return 0;                         // zie toelichting
}
```

De uitvoer van dit programma ziet er zo uit:

```
Het resultaat is: 41
```

1.3.1 Toelichting bij het eerste voorbeeld

Het belangrijkste gedeelte van voorbeeld 1.1 is het gedeelte dat begint met de regel `int main()` en dat helemaal onderaan eindigt met de sluitaccolade. Dit gedeelte is een functie die `main()` heet. Elk C++-programma moet een functie hebben met de naam `main()`. Als het programma wordt uitgevoerd, komt altijd als eerste de functie `main()` aan bod. Dit wil zeggen dat de opdrachten die tussen de openings- en sluitaccolades achter `main()` staan, als eerste worden uitgevoerd. Deze opdrachten heten ook wel *statements*. Zoals je ziet eindigt elk statement met een puntkomma.

Als eerste worden in de functie `main()` drie variabelen gedefinieerd: `a`, `b` en `antwoord`. C++ kent veel verschillende typen voor variabelen, deze drie zijn van het type `int`. Het woord `int` is een afkorting van *integer*, wat *geheel getal* betekent. Het benoemen van het type en de naam van een variabele heet ook wel het declareren van die variabele.

In de volgende drie regels krijgen de variabelen een waarde. Dit soort opdrachten heten *assignment-statements* of toekenningsopdrachten. Het teken = heet de *assignment-operator* of toekenningsoperator. Met behulp van de assignment-operator geef je een waarde aan

een variabele. De variabele staat altijd aan de linkerkant van het teken =. Zo'n variabele wordt ook wel een *modifiable lvalue* genoemd. De letter l van lvalue staat voor *left*.

De in- en uitvoer van gegevens wordt in C++ geregeld via *streams*, ofwel stromen van informatie. Er is bijvoorbeeld een stream van je programma naar het beeldscherm. In voorbeeld 1.1 staan twee opdrachten die met `cout` (spreek uit: cie-out) beginnen. Deze regels zorgen voor de uitvoer naar het beeldscherm. Het symbool `<<`, dat uit twee kleinerdantekens bestaat, moet je opvatten als één symbool. Dit symbool heet de insertion operator of uitvoeroperator. Met behulp van de uitvoeroperator stuur je iets naar `cout`. Als je meerdere dingen tegelijk naar `cout` wilt sturen, kun je dat doen door meerdere uitvoeroperatoren achter elkaar te zetten:

```
cout << antwoord << endl;
```

Met `endl`, wat een afkorting is van *endline*, zorg je ervoor dat de cursor op het beeldscherm op een nieuwe regel begint. In dit geval wordt eerst `antwoord` op het scherm gezet en daarna gaat de cursor naar de volgende regel. `endl` is een voorbeeld van een zogeheten *manipulator.* Een *manipulator* is een uitdrukking die van invloed is op de indeling (format) van de uitvoer (of invoer). In het volgende hoofdstuk vind je meer voorbeelden van manipulatoren.

Het een-na-laatste statement van het programma is:

```
cin.get();
```

Deze opdracht wacht tot je op de Enter-toets drukt. Sommige ontwikkelomgevingen hebben de gewoonte om het uitvoerscherm in een flits te tonen, te kort om iets van de uitvoer te kunnen zien. Door aan het eind van je programma de opdracht `cin.get()` neer te zetten, blijft het uitvoerscherm zichtbaar tot je op Enter drukt. Om ruimte te sparen zal ik vanaf het volgende hoofdstuk deze opdracht niet steeds aan het eind van `main()` neerzetten.

In C++ kan een functie een waarde afleveren en de betekenis van `return 0` is dat de functie `main()` de waarde `0` aflevert. In principe kun je deze waarde opvragen vanuit het besturingssysteem. Na afloop van het programma kun je aan de waarde `0` zien dat het programma succesvol is geëindigd. Als je een programma draait met daarin een fout, kun je eventueel een andere waarde dan `0` laten afleveren. Die waarde geeft dan aan dat er iets mis is.

Volgens standaard-C++ is het niet noodzakelijk `main()` af te sluiten met `return 0`. Om ruimte te sparen in de voorbeelden zal ik deze opdracht vanaf het volgende hoofdstuk dan ook weglaten. Sommige compilers, die zich niet helemaal aan de standaard houden, klagen als je `return 0` weglaat.

Alle namen in C++ zijn opgeborgen in een zogeheten namespace. Een namespace is een verzameling namen die zelf weer een naam heeft. Dit is een manier om naamconflicten

te vermijden, die zich snel kunnen voordoen als je met meerdere programmeurs aan een groot project werkt. Een namespace is wat dat betreft vergelijkbaar met een directory. Met de opdracht `using namespace std` zeg je dat je gebruik wilt maken van namen uit de namespace die `std` (afkorting van *standard*) heet. De in voorbeeld 1.1 gebruikte namen `cin` en `cout` maken deel uit van deze namespace. Meer over namespaces kun je lezen in een van de bijlagen.

Alles wat in de broncode achter twee slashes op dezelfde regel staat, is commentaar of een toelichting voor de menselijke lezer. Het commentaar wordt door de compiler overgeslagen en heeft dus geen invloed op het vertaalde resultaat. Als je meer dan een regel commentaar hebt, kun je dit beginnen met slash sterretje (`/*`) en eindigen met sterretje slash (`*/`). Bijvoorbeeld:

```
/* Dit is ook
   commentaar
   dat 3 regels beslaat */
```

1.3.2 Declaratie van variabelen

In C++ moet je variabelen declareren voordat je ze kunt gebruiken. In plaats van

```
int a;
int b;
int antwoord;
```

mag je ook schrijven:

```
int a, b, antwoord;
```

De typeaanduiding `int` geldt voor alle variabelen die erachter komen, tot aan de eerstvolgende puntkomma. De variabelen moeten van elkaar gescheiden worden door een komma. Als je zo veel variabelen hebt dat ze niet meer op een regel passen, kun je gewoon op de volgende regel doorgaan. De compiler begrijpt ook een schrijfwijze als:

```
int a, b,
    antwoord;
```

De naam van een variabele heet een *identifier*. Niet elke identifier is geschikt als naam voor een variabele. Namen van variabelen in C++ moeten voldoen aan de volgende regels:

- Een naam mag je opbouwen uit kleine letters, hoofdletters, cijfers en de *underscore* _ .
- Een naam mag nooit beginnen met een cijfer.
- Een naam mag niet een gereserveerd woord (*reserved word*) of een voorgedefinieerde identifier (*predefined identifier*) zijn.

Een gereserveerd woord is een woord dat in de taal C++ een speciale betekenis heeft. Een voorgedefinieerde identifier is een naam die al eens gebruikt is, maar niet tot de taal zelf behoort. Een lijst van gereserveerde woorden vind je in bijlage D.

Namen die aan de genoemde regels voldoen zijn bijvoorbeeld:

```
X           x                 a23
Hoogte      ANNA              _a23
Hoogte      breedte1          x_1
x1          breedte_1         x_2
x2          Dit_is_een_naam   aantalPersonen
```

Namen die niet aan deze regels voldoen zijn bijvoorbeeld:

```
2a                   // begint met een cijfer
Dit is geen naam     // bevat spaties
d'66                 // bevat apostrof
4711                 // begint met een cijfer
C&A                  // bevat & (ampersand)
C++                  // bevat plustekens
```

C++ is *case sensitive* en maakt dus verschil tussen hoofdletters en kleine letters: `hoogte` is een andere variabele dan `Hoogte` (en `HOOGTE` is weer een andere). Je mag namen zo lang maken als je wilt, maar er zijn compilers die onderscheid maken tussen (bijvoorbeeld) alleen de eerste 32 tekens.

Voor de zelfgekozen namen van variabelen in dit boek houd ik mij aan de volgende regels:

- een identifier begint met een kleine letter;
- in een identifier staat geen underscore, behalve als het een constante betreft, zie paragraaf 1.4.1;
- als een identifier uit twee of meer woorden bestaat, begint het eerste woord met een kleine letter en het volgende woord met een hoofdletter vast aan het voorafgaande woord. Bijvoorbeeld: `aantalEuros`, `nieuwSaldo`, `percentageEersteSchijf`;
- identifiers geven bij voorkeur zo precies mogelijk aan wat de betekenis van de variabele is. De naam `temperatuur` of `tijd` is veel duidelijker dan de naam `t`. En `nieuwSaldo` en `oudSaldo` zijn duidelijker dan `saldo1` en `saldo2`.

Er zijn andere goede regels denkbaar dan deze. Zo zijn er veel programmeurs die juist wel underscores gebruiken, `nieuwSaldo` wordt dan `nieuw_saldo`. In elk geval raad ik elke beginnende C++-programmeur sterk aan een keuze te maken en de regels consequent in eigen programma's toe te passen. Hierdoor worden ze – voor jezelf en voor anderen – leesbaarder en begrijpelijker.

1.3.3 Getaltypen in C++

Naast het type `int` kent C++ nog drie typen voor gehele getallen: `short`, `long` en `long long`. Op veel systemen wordt een getal van het type `short` in twee bytes opgeslagen en heeft dan een bereik van -32768 tot en met 32767. Een `int` wordt doorgaans in vier bytes opgeslagen en heeft een bereik van -2147483648 tot en met 2147483647. Het bereik van `short` is dus veel kleiner dan dat van `int`, maar daar staat tegenover dat het geheugengebruik slechts de helft is. Op veel systemen wordt een `long` in acht bytes opgeslagen, waardoor het bereik van `long` aanzienlijk groter is dan dat van `int`. Je zou verwachten dat een `long long` dan in zestien bytes wordt opgeslagen, maar dat hoeft niet het geval te zijn. De standaard eist slechts dat voor het bereik geldt:

```
short ≤ int ≤ long ≤ long long
```

Het is dus van belang na te gaan welke waarden worden gehanteerd door de compiler die je gebruikt .

De integer-getaltypen kennen ook een `unsigned` variant: gehele getallen zonder teken, dus met uitsluitend niet-negatieve waarden. Je krijgt dergelijke typen door het woord `unsigned` voor het getaltype te zetten. Een dergelijk woord heet wel een *type modifier* of kortweg *modifier*. Bijvoorbeeld:

```
unsigned short x;
unsigned int xx;
unsigned long xxx;
unsigned long long xxxx;
```

Als het bereik van een `short` loopt van -32768 tot en met 32767, dan omvat het bereik van `unsigned short` de getallen van 0 tot en met 65535. Wanneer je het negatieve getal -1 opbergt in een `unsigned short`, wordt de waarde omgezet in 65535. Het getal -2 wordt omgezet in 65534 et cetera.

Voor de andere `unsigned`-typen geldt iets dergelijks. Zie voor een overzicht van het bereik van verschillende getaltypen de tabel aan het eind van deze paragraaf.

Voor gebroken getallen (getallen met een decimale punt) kent C++ de typen `float`, `double` en `long double`.

In veel C++-implementaties kun je in een `float` getallen opbergen van 3.4 10-38 tot 3.4 10+38, zowel positief als negatief. Een `float` wordt weergegeven met een precisie van zeven cijfers. Dat wil zeggen dat de overige cijfers achter de decimale punt worden weggelaten door afronding.

Het type `double` is met 15 cijfers een stuk nauwkeuriger dan `float`.

Naam	Mogelijk aantal bytes	Kleinste waarde	Grootste waarde	
integer-type				
short	2	-32768	32767	
unsigned short	2	0	65535	
int	4	-2147483648	2147483647	
unsigned int	4	0	4294967295	
long	8	-9223372036854775808	9223372036854775807	
unsigned long	8	0	18446744073709551615	
long long	8	-9223372036854775808	9223372036854775807	
unsigned long long	8	0	18446744073709551615	
floating point-type				**Precisie**
float	4	3.4e-38	3.4e+38	7 cijfers
double	8	1.7e-308	1.7e+308	15 cijfers
long double	10	3.4e-4932	1.1e+4932	19 cijfers

In deze tabel staat een overzicht van het aantal bytes en het bereik van getaltypen zoals die in een specifieke C++-implementatie kunnen voorkomen. Bij het floating-pointtype zijn in de tabel bij de kleinste en grootste waarde alleen positieve getallen aangegeven, maar dergelijke grenzen gelden ook in het negatieve gebied. Houd er rekening mee dat de implementatie die je gebruikt kan afwijken van de gegevens in deze tabel. Met de operator `sizeof` kun je het aantal bytes van elk type opvragen, zie voorbeeld 1.2.

```
// Voorbeeld 1.2  Aantal bytes opvragen met sizeof
#include <iostream>
using namespace std;

int main() {
  cout << "short: "       << sizeof( short )       << " bytes" << endl;
  cout << "int: "         << sizeof( int )         << " bytes" << endl;
  cout << "long: "        << sizeof( long )        << " bytes" << endl;
  cout << "long long: "   << sizeof( long long )   << " bytes" << endl;
  cout << "float: "       << sizeof( float )       << " bytes" << endl;
  cout << "double: "      << sizeof( double )      << " bytes" << endl;
```

```
  cout << "long double: " << sizeof( long double ) << " bytes" << endl;
  cin.get();
  return 0;
}
```

De C++-implementatie die ik nu gebruik (Microsoft Visual C++ 2015) geeft als uitvoer:

```
short: 2 bytes
int: 4 bytes
long: 4 bytes
long long: 8 bytes
float: 4 bytes
double: 8 bytes
long double: 8 bytes
```

1.3.4 Expressies: de operatoren +, -, *, / en %

Om rekenkundige uitdrukkingen (expressies) met integers en floats te kunnen maken, heb je de volgende operatoren tot je beschikking:

- de operator `+` voor optellen;
- de operator `-` voor aftrekken;
- de operator `*` voor vermenigvuldigen;
- de operator `/` voor de deling, maar deze werkt bij integers anders dan bij float;
- de operator `%` voor de rest van de deling.

De operatoren `+`, `-` en `*` zullen weinig problemen geven. Bij de operatoren `/` en `%` geef ik een toelichting:

De operator `/` voert met twee integers de gehele deling uit, dat wil zeggen dat hij uitrekent hoeveel keer het tweede getal in het eerste gaat. De uitkomst is altijd een geheel getal, deze deling heet wel de *gehele deling*. Bijvoorbeeld:

```
30 / 7    // levert het gehele getal 4, omdat 7 vier keer in 30 gaat
125 / 60  // levert 2
10 / 8    // levert 1
```

De operator `%` heet ook wel de modulo-operator. Deze levert met twee integers de rest die overblijft na de gehele deling. Voorbeelden van de rest van de gehele deling:

```
30 % 7    // levert 2, omdat na deling van 30 door 7 er 2 overblijft
125 % 60  // levert 5
10 % 8    // levert 2
```

De operatoren `/` en `%` zijn handig bij klokrekenen: als je bijvoorbeeld wilt weten hoeveel uren en minuten er in 1412 minuten gaan, zou je daarvoor het volgende programma kunnen schrijven:

```
// Voorbeeld 1.3  Klokrekenen
#include <iostream>
using namespace std;

int main() {
  int minuten, uren, restMinuten;      // declaratie van 3 variabelen

  minuten = 1412;                      // assignment statement
  uren = minuten / 60;                 // bereken het aantal uur
  restMinuten = minuten % 60;          // bereken overblijvende minuten

  cout << minuten << " minuten = ";    // waarden en
  cout << uren << " uur en ";          // tekst naar
  cout << restMinuten << " minuten";   // uitvoerscherm
  cin.get();
  return 0;
}
```

De uitvoer van dit programma is:

```
1412 minuten = 23 uur en 32 minuten
```

Bij het toepassen op twee floats werkt de operator `/` als de gewone deling:

```
10.0 / 4.0    // levert 2.5
1.0 / 3.0     // levert 0.3333333
```

De vraag is wat er gebeurt als je twee verschillende typen in een expressie combineert, bijvoorbeeld een integer en een float. Zie daarvoor de volgende paragraaf.

1.3.5 Typeconversie

Als je een uitdrukking hebt met twee (of meer) van de typen `short`, `int`, `long`, `long long`, `float`, `double` of `long double` hebt, vind er *typeconversie* plaats. Bekijk eens het volgende fragment:

```
  int aantal = 3;
  float prijs = 2.75;
  double totaal;
  totaal = aantal * prijs;
  cout << "Totaal: " << totaal;
```

De uitvoer is:

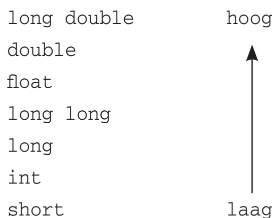
```
Totaal: 8.25
```

In het statement

```
totaal = aantal * prijs;
```

komen drie verschillende typen voor: `int`, `float` en `double`. De compiler heeft hier geen problemen mee. De `int`-waarde van `aantal` wordt geconverteerd naar `float`, alvorens hem te vermenigvuldigen met de `float`-waarde van `prijs`. Het resultaat is een `float` die geconverteerd wordt naar `double`, om ten slotte in de variabele `totaal` te worden opgeborgen.

Deze conversie gebeurt automatisch en heet dan ook *automatische typeconversie*.
De regels voor automatische typeconversie zijn tamelijk eenvoudig. Onder de getaltypen bestaat een ordening van laag naar hoog, zie figuur 1.2.



Figuur 1.2

Als de compiler een (deel van een) expressie tegenkomt met daarin twee verschillende typen, wordt het laagste type opgewaardeerd naar het hoogste. Het type van het resultaat is dat van het hoogste type in de expressie.

1.3.6 De typecast

Soms gebeurt een conversie niet automatisch, terwijl je die wel wilt. Je kunt in een aantal gevallen conversie afdwingen met een zogeheten *typecast* of *cast*. Het Engelse werkwoord *to cast* betekent zoiets als 'in een andere vorm gieten'. Een voorbeeld waarin de cast nuttig is, is de deling van twee `int`-getallen, zie ook paragraaf 1.3.3. Als je simpelweg de operator `/` gebruikt, krijg je de gehele deling:

`3/4` levert `0` en niet `0.75`

Als je toch `0.75` als antwoord wilt, kun je dat voor elkaar krijgen met een cast. Hiervoor heeft C++ de operator `static_cast`. Zie voorbeeld 1.4.

```
// Voorbeeld 1.4  Cast
#include <iostream>
using namespace std;

int main() {
  int teller = 3, noemer = 4;
  double uitkomst1, uitkomst2;
```

```
  uitkomst1 = teller / noemer;                         // gehele deling
  uitkomst2 = static_cast<double>( teller ) / noemer;  // gewone deling
  cout << "Zonder cast: "  << uitkomst1 << endl;
  cout << "Met cast:    "  << uitkomst2 << endl;
  cin.get();
  return 0;
}
```

De uitvoer is:

```
Zonder cast: 0
Met cast:    0.75
```

De uitdrukking `static_cast<double>(teller)` zorgt ervoor dat de waarde van de variabele `teller` (die zelf van het type `int` is) geconverteerd (*gecast*) wordt naar `double`. Daarmee wordt de waarde `3` in dit voorbeeld in feite geconverteerd naar de waarde `3.000000000000000`. Omdat in de uitdrukking:

```
static_cast<double>( teller ) / noemer
```

een `double` door een `int` gedeeld wordt, zal de `int` in de noemer automatisch naar een `double` worden geconverteerd. De deling is dus uiteindelijk `double` / `double`, met als resultaat een `double`. Dit is in overeenstemming met de regel dat het type van het resultaat dat van het hoogste type in de expressie is.

1.3.7 Assignment-operatoren

Het isgelijkteken `=` heet de assignment-operator, of toekenningsoperator. Je kunt een variabele meteen bij de declaratie een waarde geven, dat heet dan initialisatie. Je kunt een variabele na de declaratie een (andere) waarde geven, dat heet assignment of toekenning:

```
int teller = 1;         // declaratie en initialisatie
teller = teller + 1;    // toekenning (assignment)
```

De laatste opdracht moet je als volgt lezen:

```
nieuwe waarde van teller = oude waarde van teller plus een
```

Hiermee heeft `teller` de waarde 2 gekregen.

Met de assignment-operator `+=` kun je hetzelfde bereiken. Het statement

```
teller += 1;
```

doet precies hetzelfde als

```
teller = teller + 1;
```

Het voordeel van het gebruik van de operator `+=` is dat je de naam van de variabele maar één keer hoeft te noemen. Op dezelfde manier kun je andere waarden dan 1 optellen met behulp van deze operator:

```
teller += 3;
teller += toename;
```

De assignment-operator `+=` staat niet op zichzelf. Ook voor de operatoren `-`, `*`, `/` en `%` zijn er overeenkomstige toekenningsoperatoren. Zo hebben bijvoorbeeld de statements in de linkerkolom dezelfde betekenis als die in de rechterkolom:

```
hoeveelheid = hoeveelheid - 10;    hoeveelheid -= 10;
aantal = aantal * 2;               aantal *= 2;
lengte = lengte / 3;               lengte /= 3;
rest = rest % 12                   rest %= 12;
```

Al deze bewerkingen gelden zowel voor gehele getallen als voor floating-pointgetallen, behalve `%` en `%=` die je alleen met integer-typen kunt gebruiken.

1.3.8 De increment-operator ++ en de decrement-operator --

De waarde `1` optellen bij een variabele komt zo vaak voor, dat C++ nog een manier heeft om dat te doen: met behulp van de *increment-operator* `++`.

```
  int teller = 0;
  teller++;
  cout << teller << endl;
  teller++;
  cout << teller << endl;
```

De uitvoer van dit fragment is:

```
1
2
```

Het statement

```
teller++;
```

heeft hetzelfde effect als

```
teller += 1;
```

De increment-operator `++` wordt veel gebruikt in *for-statements*, zie het volgende hoofdstuk.

Als tegenhanger van `++` bestaat de *decrement-operator* `--`, die de waarde van de variabele met 1 vermindert, zie paragraaf 2.7 voor een voorbeeld.

1.3.9 De naam C++

De operator `++` staat in C++ voor 'één optellen bij', of, wat ruimer opgevat: 'iets toevoegen aan'. De taal met de naam C++ is ontstaan door iets toe te voegen (namelijk klassen) aan de taal C.

1.3.10 Postfix en prefix

De operatoren `++` en `--` kun je achter of voor de variabele zetten. Stel dat aantal de waarde 5 heeft en prijs de waarde 10. Dan maakt het verschil of je schrijft

```
bedrag = ++aantal * prijs;
```

of

```
bedrag = aantal++ * prijs;
```

Als de `++` voor de variabele in kwestie staat, heet dit een *prefix*-operator. Bij het gebruik van `++` als *prefix*-operator wordt eerst de variabele verhoogd, en daarna de waarde van de variabele gebruikt in de expressie. Dus in het eerste geval krijgt `aantal` de waarde 6 en die waarde wordt gebruikt in de berekening, waardoor bedrag de waarde 60 krijgt.

Staat de `++` na de variabele, dan heet dit een *postfix*-operator. Als je de `++` als *postfix*-operator gebruikt, wordt eerst de waarde van de variabele gebruikt om de expressie te berekenen en vervolgens wordt de variabele met `1` verhoogd. In het tweede geval wordt de waarde `5` van `aantal` gebruikt voor de berekening van `bedrag`, die dus de waarde 50 krijgt. Als dat gebeurd is, wordt `aantal` verhoogd van 5 naar 6.

Ook in het volgende geval is er verschil tussen het gebruik van de prefix- of postfix-operator:

```
int i = 10;           int i = 10;
int n = ++i;          int m = i++;
```

Bij prefix, `int n=++i`, wordt eerst de waarde van `i` verhoogd en wordt daarna deze verhoogde waarde toegekend aan `n`.

Bij postfix, `int m=i++`, wordt eerst de waarde van `i` toegekend aan `m` en daarna wordt de waarde van `i` verhoogd.

Na afloop heeft `i` in beide gevallen de waarde `11`, evenals `n`, en `m` heeft de waarde `10`.

1.3.11 Prioriteiten

Elke operator heeft een prioriteit (precedence). Dit wil zeggen dat er afspraken zijn over de voorrang die de ene operator heeft op de andere. Als een operator een hogere prioriteit heeft, wordt hij eerder uitgevoerd.

- De operatoren `*`, `/`, en `%` hebben gelijke prioriteit.
- De operatoren `+` en `-` hebben gelijke prioriteit.
- De operatoren `*`, `/`, en `%` hebben hogere prioriteit dan de operatoren `+` en `-`.

Als in een uitdrukking rekenkundige operatoren met gelijke prioriteit voorkomen, wordt de berekening van links naar rechts uitgevoerd. Dit heet links associatief (zie de bijlage met een tabel van operatoren achter in dit boek).

Heel kort en slordig samengevat kun je zeggen dat vermenigvuldigen en delen voor optellen en aftrekken gaan. Als beginnend programmeur moet je oppassen voor het volgende:

```
int a;
a = 10 / 2 * 5;
```

Wat is de waarde van de variabele `a`? Is het `25` of `1`? De operatoren `/` en `*` hebben gelijke prioriteit, waardoor de uitdrukking van links naar rechts wordt uitgevoerd. Dus eerst wordt `10/2` uitgerekend en het resultaat daarvan wordt met `5` vermenigvuldigd. Het juiste antwoord is dan ook `25`.

1.4 Literals en constanten

Getallen zoals `23` of `-7.68`, heten ook wel *literals*. Afhankelijk van de notatie behoort een literal tot het ene of tot het andere type. In figuur 1.3 op pagina 33 staat een opsomming van mogelijke notaties van literals.

Een literal als `52` wordt geïnterpreteerd als een `int`, en dus niet als `short`, `long` of `long long`. Een literal als `3.14` wordt geïnterpreteerd als `double`, en niet als `float`.

Gehele getallen kennen veel verschillende notatiemogelijkheden.

In computertoepassingen spelen hexadecimale (zestientallige) getallen vaak een rol. De notatie van hexadecimale getallen begint met `0x` of met `0X`. De notatie van octale (achttallige) getallen begint met een nul. Vanaf C++14 is het mogelijk binaire literals aan te geven door een getal te beginnen met `0b` of `0B`.

Zoals je in figuur 1.3 (pagina 33) kunt zien, kun je de compiler dwingen sommige literals te interpreteren als behorend tot een bepaald type door achter de literal een zogeheten suffix te zetten. Een suffix maak je met behulp van de letters F, L en U (of met kleine let-

ters f, l en u). De suffixen L en U kunnen ook samen voorkomen bij gehele getallen. De betekenis van deze suffixen is als volgt:

- Met `F` dwing je af dat een `double` als `float` wordt geïnterpreteerd.
- Met `L` dwing je af dat een geheel getal als `long` wordt geïnterpreteerd, of dat een `double` als `long double` wordt geïnterpreteerd.
- Met `LL` dwing je af dat een geheel getal als `long long` wordt geïnterpreteerd.
- Met `U` dwing je af dat een geheel getal `int` als `unsigned int` wordt geïnterpreteerd.
- Met `LU` of `UL` dwing je af dat een geheel getal als `unsigned long` wordt geïnterpreteerd.
- Met `LLU` of `ULL` dwing je af dat een geheel getal als `unsigned long long` wordt geïnterpreteerd.

Een voorbeeld: sommige compilers geven een waarschuwing of zelfs een foutmelding als je schrijft:

```
long getal = 12000000000;
```

Een dergelijke melding kun je voorkomen door te schrijven:

```
long getal = 12000000000L;    // long met suffix L
```

Vanaf C++14 kun je een apostrof gebruiken om (grote) gehele getallen makkelijker leesbaar te maken:

```
long long x = 10000000000LL;
long long y = 10'000'000'000LL; // C++14 apostrof als scheidingsteken
```

of:

```
int z = 0B10'000'000'000;       // C++14 binaire literal
```

In figuur 1.3 zie je nog een paar voorbeelden van literals met en zonder suffix.

literal	decimale waarde	soort	type

`120`
`+120`
`-120`
	`120`
	`120`
	`-120`
		decimaal geheel getal	`int`

`32L`
`231l`
`2000000000L`
`2'000'000'000L`
	`32`
	`231`
	`2 miljard`
	idem, vanaf C++14
		decimaal geheel getal	`long`

`0x1A` `0x1b`	`26` `27`	hexadecimaal getal	`int`
`0xaabbccddeeLL` `0xaa'bb'cc'dd'eeLL`	`733295205870` idem, vanaf C++14	hexadecimaal getal	`long long`

`07`
`0B1000`
`0b1000`
	`7`
	`8`
	`idem`
		octaal getal
		binair getal (vanaf C++14)
			`int`

`32768U` `32768u`	`32768` `32768`	unsigned decimaal geheel getal	`unsigned int`

`32UL`
`32LU`
`32ul`
`32lu`
	`32`
	`32`
	`32`
	`32`
		unsigned decimaal geheel getal	`unsigned long`

`3.14159`
`3.14159e3`
`3.14159e-2`
	`3.14159`
	`3141.59`
	`0.0314159`
		floating-pointgetal	`double`

`3.14159F` `3.14159f`	`3.14159`	floating-pointgetal	`float`
`3.14159L` `3.14159l`	`3.14159`	floating-pointgetal	`long double`

Figuur 1.3

1.4.1 Constanten

Een formule als

```
omtrek = pi * middellijn;
```

is gemakkelijker te lezen en te onthouden dan

```
omtrek = 3.14159265358979 * middellijn;
```

hoewel de betekenis precies hetzelfde is.

Het kan daarom verstandig zijn constanten in je programma een naam te geven. Dat doe je door het woord `const` voor de declaratie te zetten. Net als `unsigned` is een woord als `const` een *type modifier*. Door de modifier `const` in een declaratie weet de compiler dat het om een constante gaat en niet om een variabele. Het gevolg is dat er een waarschuwing zal komen als je toch probeert de waarde van zo'n constante te veranderen. In het volgende voorbeeld zijn twee constanten gedefinieerd. Het is gebruikelijk de namen van constanten met uitsluitend hoofdletters te spellen, en een underscore als scheidingsteken te gebruiken als de naam van de constante uit meerdere woorden bestaat.

```
// Voorbeeld 1.5  De modifier const
#include <iostream>
using namespace std;

int main() {
  const double PI = 3.14159265358979;
  double middellijn = 2.5;
  cout << "Omtrek = " << PI * middellijn << endl;

  const int UUR_PER_ETMAAL = 24;
  int aantalDagen = 43;
  cout << "Totaal aantal uur is: " << aantalDagen * UUR_PER_ETMAAL;
  cin.get();
  return 0;
}
```

De uitvoer van dit programma is:

```
Omtrek = 7.853982
Totaal aantal uur is: 1032
```

Een constante *moet* meteen bij de declaratie een waarde krijgen (geïnitialiseerd worden). Je mag dus niet schrijven:

```
const int UUR_PER_ETMAAL;
// ...
UUR_PER_ETMAAL = 24;
```

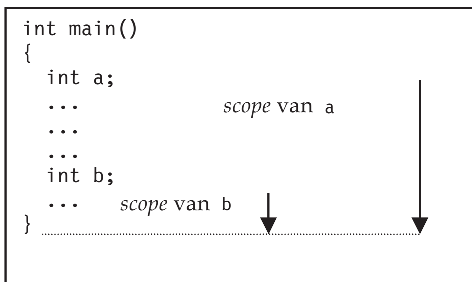
Het moet zo:

```
const int UUR_PER_ETMAAL = 24;
```

Een dergelijke constante wordt in het C++-jargon ook wel een *lvalue* genoemd, in tegenstelling tot een variabele die een *modifiable lvalue* is.

1.5 De scope van variabelen en constanten

Bij elke variabele (voor constanten gelden dezelfde regels) die je in een programma definieert hoort een *scope*, dat is het gedeelte in het programma waarbinnen je deze variabele kunt gebruiken. De scope van de variabelen die je in de hoofdfunctie `main()` definieert, loopt vanaf de plaats van de declaratie tot aan de eerstvolgende sluitaccolade na de declaratie. Een stel accolades met daartussen statements heet ook wel een *blok* (Engels: block). De scope van een variabele begint waar hij is gedeclareerd, en loopt door tot het einde van het blok waarin hij is gedefinieerd. Figuur 1.4 laat hiervan een voorbeeld zien.



Figuur 1.4

In de voorbeelden in dit hoofdstuk is er steeds maar één blok en loopt de scope door tot de laatste sluitaccolade van `main()`. In de volgende hoofdstukken zie je dat accolades op veel meer plaatsen in een programma kunnen voorkomen, waardoor er ook een grotere variatie aan blokken en dus aan scopes kan ontstaan.

1.6 Het type char

In variabelen van het type `char` kun je één letter of één ander teken opslaan. Nooit meer dan één. Een dergelijk teken moet je in het programma tussen apostrofs zetten, bijvoorbeeld de letter a als `'a'`, een punt als `'.'` en een spatie als `' '`. De reden dat je apostrofs moet gebruiken is dat deze aan de compiler duidelijk maken dat het om een letter gaat, en niet om bijvoorbeeld een variabele die de naam `a` heeft.

In de praktijk heb je het type `char` waarschijnlijk vooral nodig in programma's waarin de gebruiker op een bepaalde toets moet drukken om verder te gaan. Bijvoorbeeld als je programma een menu op het scherm zet zoals dit:

```
Druk op:
 S  voor Spreadsheet
 T  voor Tekstverwerken
 Q  om het programma te verlaten
```

In zo'n geval is het duidelijk dat de gebruiker op een van de letters `S`, `T` of `Q` moet drukken. Deze letter moet je aan het programma doorgeven. Dat kan met `cin.get()`. De functie

`get()` wacht tot je op een toets en daarna op Enter drukt en levert de bij de toets horende letter af, zodat je de letter kunt opbergen in een variabele van het type `char`. Zie voorbeeld 1.6, dat vertelt op welke letter je hebt gedrukt:

```
// Voorbeeld 1.6  Wacht op het indrukken van een toets
#include <iostream>
using namespace std;

int main() {
  char ch;
  cout << "Druk op een lettertoets" << endl;
  ch = cin.get();    // cin.get() wacht op indrukken van een
                     // toets en Enter bergt de letter op in ch
  cin.get();         // Vangt Enter op
  cout << "U heeft op de letter " << ch << " gedrukt.";
  cin.get();
  return 0;
}
```

Mogelijke uitvoer:

```
Druk op een lettertoets
s
U heeft op de letter s gedrukt.
```

De belangrijkste regel in dit programma is

```
ch = cin.get();
```

In deze opdracht wordt de ingedrukte letter (nadat je op Enter hebt gedrukt), opgeborgen in de variabele `ch`. Alle tekens die je in een variabele van het type `char` kunt opslaan, zal ik *karakters* noemen. Een karakter kan dus een kleine letter, een hoofdletter, een cijfer of een leesteken zoals een spatie of een punt zijn. De ingedrukte Enter vang je op met de tweede `cin.get()`.

1.6.1 ASCII-code

Letters en andere tekens worden in computers gecodeerd als een getal. Deze wijze van coderen van letters en andere tekens is al meer dan honderd jaar oud en stamt nog uit de tijd van telexverbindingen. Via een kabel werden letters gecodeerd verzonden en aan de andere kant opgevangen door een telex, een soort elektrische typemachine die de letters automatisch uittikte. Een codering die nu veel in computers toegepast wordt, is de *ASCII-code* (American Standard Code for Information Interchange, vertaald: Amerikaanse standaardcode voor het uitwisselen van informatie). In zijn eenvoudigste vorm is ASCII een codering voor 128 verschillende tekens. Een tabel met ASCII-codes vind je in een

bijlage bij dit boek. ASCII-codering is een deelverzameling van Unicode, een codering die een veel grotere verzameling tekens toelaat.

Omdat de code voor elk karakter een getal is, kun je verrassend genoeg met karakters rekenen, zie voorbeeld 1.7.

```
// Voorbeeld 1.7  ASCII-code
#include <iostream>
using namespace std;

int main() {
  char letter1 = 'A', letter2;
  cout << letter1 << endl;
  letter2 = letter1 + 1;  // tel 1 op bij de ASCII-code van 'A'
  cout << letter2 << endl;
  cin.get();
  return 0;
}
```

De uitvoer van dit programma is:

```
A
B
```

Waarom is dat zo? De verklaring zit in de ASCII-codering: de variabele `letter1` is geïnitialiseerd met `'A'`. Deze `'A'` wordt als het getal `65` opgeslagen. In het statement:

```
letter2 = letter1 + 1;
```

wordt `1` opgeteld bij `65`, dit levert `66` en dat is de ASCII-code van de letter `B`.

1.6.2 Escape sequences

In C++ is een aantal karakter-literals gedefinieerd met een speciale betekenis. Meestal worden deze karakters *escape sequences* genoemd. Een escape sequence bestaat uit een backslash, gevolgd door een ander teken.

De achtergrond van escape sequences is de volgende: zoals je weet moet je een string in C++ tussen aanhalingstekens zetten, bijvoorbeeld:

```
cout << "Druk op een lettertoets" << endl;
```

De aanhalingstekens geven het begin en het einde van de string aan. Maar wat als je het aanhalingsteken zelf op het scherm wilt zetten? Zoals in de tekst *Druk op een "lettertoets"*.

Je zou het volgende kunnen proberen:

```
cout << "Druk op een "lettertoets"" << endl;  // werkt niet
```

Dit werkt niet omdat de compiler het tweede aanhalingsteken interpreteert als het einde van de string, en een foutmelding geeft omdat na dat einde nog meer tekens komen. Doormiddel van een backslash kun je de compiler vertellen dat het aanhalingsteken erna niet het einde van de string is, maar dat het geïnterpreteerd moet worden als een teken van de string zelf:

```
cout << "Druk op een \"lettertoets\"" << endl;  // werkt wel
```

De uitvoer is:

```
Druk op een "lettertoets"
```

Er zijn meer combinaties met een backslash die de compiler vertellen: let op, hier komt een teken met een speciale of afwijkende betekenis. Zie figuur 1.5.

escape sequence	**betekenis**
`\a`	piepje (alert)
`\b`	Backspace
`\f`	nieuwe pagina (form feed)
`\n`	nieuwe regel (newline), zelfde effect als `endl`
`\r`	naar begin van regel (carriage return)
`\t`	horizontale tab
`\v`	verticale tab
`\\`	backslash
`\'`	apostrof
`\"`	aanhalingsteken
`\0`	null karakter

Figuur 1.5

De meeste van deze escape sequences gebruik je waarschijnlijk maar heel af en toe. Als je ze gebruikt, moet je ze óf tussen apostrofs zetten, óf in een tekst opnemen die tussen aanhalingstekens staat. Voorbeelden:

```
cout << '\a';
```

levert een piepje. En

```
cout << "aap" << '\n' << "noot";
```

levert:

```
aap
noot
```

Precies hetzelfde effect kun je bereiken met:

```
cout << "aap\nnoot";
```

Of met:

```
cout << "aap" << endl << "noot";
```

Misschien is de dubbele backslash `\\` de meest gebruikte escape sequence. Dat komt doordat de backslash veel computersystemen gebruikt wordt om de namen van subdirectory's van elkaar te scheiden. Stel dat je een subdirectory hebt die `c:\nieuw` heet, en je wilt de naam van deze subdirectory door een C++-programma op het scherm laten zetten, dan kun je schrijven:

```
cout << "c:\nieuw";       // fout!
```

Dit heeft de volgende merkwaardige uitvoer:

```
c:
ieuw
```

Dat komt doordat de combinatie `\n` een escape sequence is met de betekenis van een nieuwe regel. Als je een backslash op het scherm wilt, moet je een dubbele backslash gebruiken, dus zo:

```
cout << "c:\\nieuw";    // zo is het wel goed
```

Dit heeft als uitvoer:

```
c:\nieuw
```

1.6.3 Literals van het type char

Een literal van het type char staat altijd tussen apostrofs. Een dergelijke literal kan verder bestaan uit een enkel teken, uit de hexadecimale ASCII-code van het karakter, of uit een escape sequence. Zie de voorbeelden in figuur 1.6.

Literal	betekenis	soort	type
'a' '8'	letter a cijfer 8	karakter	char char
'\x41'	letter A	karakter met hexadecimale code 41	char
'\\' '\n'	backslash newline	karakter in de vorm van escape sequence	char char

Figuur 1.6

1.7 C++11: de typeaanduiding auto

In C++11 kun je de typeaanduiding `auto` gebruiken[1] als je een variabele bij de declaratie initialiseert (een waarde geeft), en de compiler op grond van die initialisatie (automatisch) zelf het type kan bepalen voor de variabele. Een paar voorbeelden:

```
auto a = 5;                          // a heeft type int
auto b = 3.14;                       // b heeft type double
auto c = 'z';                        // c heeft type char
const auto D = 31;                   // D heeft type const int
```

Je kunt in een opdracht meerdere variabelen tegelijk declareren, ze moeten dan wel hetzelfde type opleveren:

```
const auto UREN = 24, MAANDEN = 12;       // beide type const int
```

Het volgende is fout (resulteert niet in hetzelfde type):

```
auto a = 5, b = 3.14;            // fout: int en double
```

Het is misschien verleidelijk alle variabelen met behulp van `auto` te declareren. Dat is echter een minder goed idee, omdat het voor jezelf en voor anderen die je programma lezen duidelijker is als je het type van variabelen expliciet aangeeft. In hoofdstuk 6 en volgende hoofdstukken blijkt dat je in C++ zelf typen kunt definiëren met behulp van klassen en dat de standaardbibliotheek over veel voorgedefinieerde klassen beschikt. De typeaanduidingen van dergelijke klassen kunnen behoorlijk gecompliceerd worden, en `auto` geeft de programmeur de mogelijkheid zich niet te hoeven bekommeren om de precieze notatie van het type.

[1] Voorheen bestond in C++ de storage class `auto` voor zogeheten *automatic* variabelen. Deze betekenis van `auto` is met C++11 komen te vervallen.

1.7.1 De opmaak van de broncode

Om programma's overzichtelijk en goed leesbaar te maken, is het verstandig je te houden aan een paar richtlijnen voor de opmaak van de broncode (de tekst van het programma). Die richtlijnen hebben vooral betrekking op het inspringen (en weer terugspringen) van de programmaregels. De stijl die ik in dit boek hanteer is de volgende:

- Zet een openingsaccolade aan het einde van een regel en spring op de volgende regel twee spaties in.
- Zorg dat de regels daarna recht onder elkaar staan.
- Spring bij elke sluitaccolade weer twee spaties terug.

De opmaak van een programma krijgt dan het volgende aanzien:

```
int main() {                      // openingsaccolade
  statement;                      // inspringen
  statement;
  for( i = 1; i <= 10; i++ ) {   // openingsaccolade
    statement;                    // inspringen
    statement;
  }                               // sluitaccolade springt terug
  statement;
  statement;
}                                 // sluitaccolade springt terug
```

Er zijn veel andere stijlen in omloop. Welke stijl je kiest maakt niet erg veel verschil en voor de compiler maakt het helemaal niets uit, maar probeer de eenmaal gekozen stijl vol te houden, zodat er een overzichtelijke lay-out ontstaat.

1.8 Samenvatting

- Het maken van een programma in C++ begint met het schrijven van broncode.
- De broncode wordt gelezen door een programma dat de preprocessor heet.
- De preprocessor heeft als belangrijkste taak de zogeheten headerbestanden die achter de opdracht `#include` staan in de broncode in te voegen.
- Het resultaat is een translation unit. Deze unit wordt door de compiler vertaald naar voor de processor begrijpelijke machinecode, de zogeheten doelcode (object code .obj).
- Vervolgens koppelt de linker functies uit de standaardbibliotheek aan de objectcode en ontstaat een uitvoerbaar programma (.exe).
- De uitvoering van een programma begint altijd in een functie met de naam `main()`.
- Commentaar kun je in de broncode aangeven achter `//` of tussen `/*` en `*/`.
- In een C++-programma kun je variabelen gebruiken. Een variabele heeft een naam en een type, en je kunt er een waarde van het betreffende type in opbergen.
- Een variabele moet je declareren voor je hem kunt gebruiken, dat wil zeggen dat je zijn naam en type moet noemen.
- Standaardtypen voor gehele getallen zijn `short`, `int`, `long` en `long long`.

- De typen voor gehele getallen komen ook `unsigned` voor.
- Standaardtypen voor gebroken getallen (floating point) zijn `float`, `double` en `long double`.
- Elk getaltype heeft zijn eigen bereik dat per implementatie kan verschillen.
- Met een typecast kun je van het ene naar het andere getaltype converteren.
- Voor getaltypen zijn een groot aantal operatoren gedefinieerd: rekenkundige, toekennings-, increment- en decrement-operatoren. De increment- en decrement-operator kun je prefix of postfix gebruiken.
- Elke operator heeft een prioriteit en een associativiteit (zie bijlage C).
- Een constante krijg je door voor de declaratie van een variabele de modifier `const` te zetten. De identifier van een constante noteer je bij voorkeur met hoofdletters.
- Een ander standaardtype is `char` voor letters, leestekens, cijfers en andere karakters. Intern wordt een waarde van het type `char` omgezet in een geheel getal.
- Escape sequences zijn speciale tekens die voorafgegaan worden door een backslash.

1.9 Vragen

1. Wat is de functie van een compiler?
2. Wat doet een linker?
3. Wat is `main()` voor een speciale functie?
4. Waartoe dienen headerbestanden?
5. Met welk symbool begint commentaar dat één regel beslaat?
6. Hoe kun je een preprocessor directive herkennen?
7. Met behulp van welk object kun je gegevens vanuit je programma naar het beeldscherm sturen?
8. Welke include-directive moet je in het programma zetten om `cout` en `cin` te kunnen gebruiken?
9. Wat is het verschil in resultaat tussen de volgende twee statements?
   ```
   cout << "21 + 43";
   ```
   en
   ```
   cout << (21 + 43);
   ```
10. Hoe ziet de insertion-operator eruit en wat doet hij?
11. Wat is het verschil tussen een constante en een variabele?
12. Welke basistypen voor gehele getallen bestaan er in C++? En welke voor gebroken getallen?
13. Wat zal de waarde van de variabelen `resultaat1` en `resultaat2` in het volgende stukje programma zijn?

    ```
    int x = 13, y = 7;
    double resultaat1, resultaat2;
     resultaat1 = x / y;
     resultaat2 = x % y;
    ```

14. Wat is een cast?
15. Wat zal de waarde zijn van de variabele `resultaat`?

```
int x = 13, y = 7;
double resultaat;
resultaat = static_cast<double>( x ) / y;
```

16. Welke typeconversies vinden er plaats in de laatste regel van het volgende fragment:

```
short s = 5;
int i   = 1000;
long lg = 12000000000L;
double totaal;
totaal = (s + i) * lg;
```

17. Wat is het Nederlandse woord voor assignment?
18. Welke assignment-operatoren zijn er in C++?
19. Wat is de scope van een variabele?
20. Wat is na afloop van het volgende fragment de waarde van `som`?

```
int x = 3, y = 5, som = 100;
x++;
y += 5;
som += x;
som += y;
```

21. Wat verandert er aan de waarde van `som` als je in het vorige fragment het statement `x++` vervangt door `++x`?

1.10 Opgaven

1. a. Probeer te voorspellen wat volgens C++ de antwoorden van de volgende drie uitdrukkingen zullen zijn:

```
10 / 2 * 5
4 / 5 * 2
2 * 4 / 5
```

   b. Controleer je voorspellingen door een programma te schrijven dat bovenstaande expressies uitrekent.
   c. Welke conclusies kun je trekken over de volgorde van de bewerkingen `/` en `*` in C++?
2. De grootst mogelijke waarde van een `int` is op een bepaald systeem gelijk aan `2147483647`. Als je er `1` bij optelt krijg je een negatieve waarde. Schrijf een programma dat controleert of dit ook voor je eigen systeem geldt.
3. Schrijf een programma waarin twee negatieve `int`-waarden worden opgeteld en dat er een positief getal uit komt.

4. Getallen van het type `unsigned int` zijn getallen zonder teken, dus getallen groter dan nul (of gelijk aan 0). Wat gebeurt er als je toch een negatieve gehele waarde in een variabele van het type `unsigned int` opbergt?
5. Schrijf een programma dat de twee float-getallen 123.456F en 654.321F bij elkaar optelt. Als er een onnauwkeurigheid in de uitvoer is, kijk dan of dit verandert als je `float` wijzigt in `double`.
6. Verander het programma van de vorige opgave zo, dat er twee floating-pointgetallen worden opgeteld die samen groter zijn dan de grootste `float`. Welke uitvoer levert dit programma?
7. Wat gebeurt er als je twee positieve getallen op elkaar deelt, waarvan de uitkomst kleiner is dan de kleinste `double` ongelijk aan nul?
8. Schrijf een programma met daarin een constante voor het aantal maanden per jaar, en twee variabelen voor het maandsalaris en het jaarsalaris. Laat het programma vragen om invoer van een maandsalaris. Doe dit als volgt:

   ```
   cout << "Voer maandsalaris in: ";
   cin >> maandsalaris;
   cin.get();
   ```

   De uitvoer van het programma moet het jaarsalaris zijn (zonder vakantiegeld).
9. Pas het programma van de vorige opgave zodanig aan dat ook het vakantiegeld berekend en uitgevoerd wordt. Neem aan dat het vakantiegeld 8% is van het jaarsalaris.
10. Schrijf een programma dat berekent hoeveel seconden er in een jaar van 365 dagen gaan. Definieer zoveel constanten als nodig zijn voor de berekening.
11. Definieer in een programma het percentage btw (21%) als constante. Laat het programma vragen om de prijs van een artikel zonder btw, en laat het programma de btw én de prijs inclusief btw op het scherm zetten.
12. Als de vorige opgave, maar nu vraagt het programma om een bedrag inclusief btw en de uitvoer bestaat uit de btw en het bedrag zonder btw.

Hoofdstuk 2

Selecties en herhalingen

2.1 Inleiding

Net als veel andere talen kent C++ beslissingsopdrachten als `if` en `if..else` en herhalingsopdrachten als `for`, `while` en `do-while`. Voordat ik deze statements bespreek, is het nuttig iets te weten over relationele en logische operatoren.

2.1.1 Relationele operatoren

Een *relationele operator* vergelijkt twee uitdrukkingen met elkaar, zoals de operator `>` (groter dan) die kijkt of het een groter is dan het ander. Het resultaat van een uitdrukking met een relationele operator kent altijd maar twee mogelijkheden: *true* of *false*. De waarden `true` en `false` behoren in C++ tot een apart basistype: het type `bool`. Het woord `bool` is afgeleid van de naam van de Engelse wiskundige George Boole, die rond 1850 een studie heeft gemaakt van de logica van `true` en `false`.

In figuur 2.1 staan de relationele operatoren waarmee je waarden van getallen in C++ kunt vergelijken.

Operator	Betekenis
<	kleiner dan
>	groter dan
<=	kleiner dan of gelijk aan
>=	groter dan of gelijk aan
==	is gelijk aan
!=	is ongelijk aan

Figuur 2.1

Merk op dat de operator *is gelijk aan* uit twee isgelijktekens bestaat. Een fout die veel beginnende C++-programmeurs maken is dat ze een enkel isgelijkteken (`=`) gebruiken in plaats van het dubbele isgelijkteken (`==`). Het enkele isgelijkteken is geen relationele operator, maar de toekenningsoperator.

Als de integervariabele `jaartal` de waarde `1850` heeft, dan geldt:

```
jaartal == 1850  is waar, dus levert de waarde true
jaartal != 1850  is niet waar, dus levert de waarde false
jaartal >= 1800  is waar, dus levert de waarde true
jaartal <= 1850  is waar, dus levert de waarde true
jaartal < 1900   is waar, dus levert de waarde true
```